EICoM
ELECTRICAL COMPUTER MECHATRONICS

خوارزميات

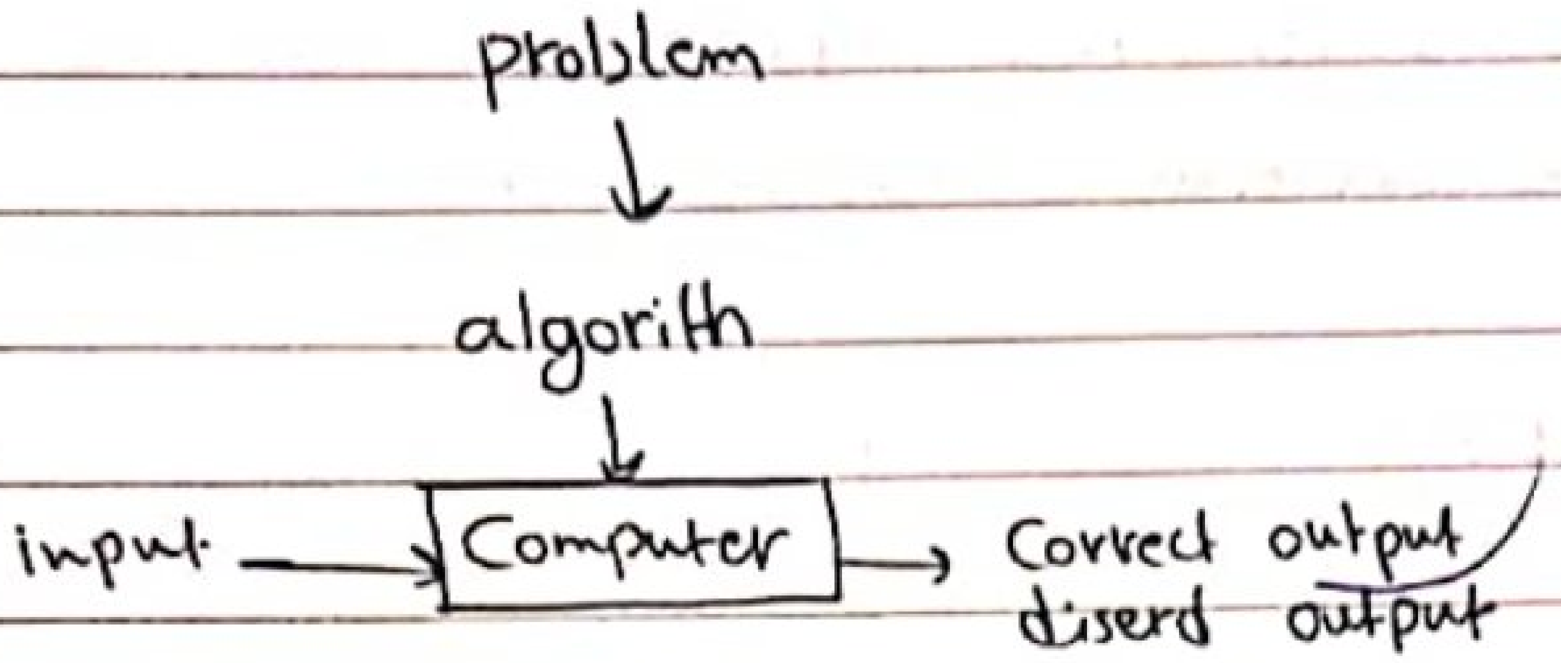
د. خليل يوسف

للطالبتان المبدعتان
فلسطين حمدان
فيحاء الحديدي

إرادة - ثقة - تغيير

Algorithm

Algorithm: is a sequence of unambiguous instructions for a solving a problem



efficient

other defined: a precisely defined sequence of (computational) steps that transfer a given input into a desired output. [Sequence of precise and clear step to solve a problem]

a method for solving problem: ① Correctness the method
② performance the method

* Not all problems can be defined by an algorithm, Ex: halting problem

the computer can't take the program is solve or not

* A problem is said to be solvable ⇒ If a computer program can be written to produce the correct output for any input.

→ what is an algorithm?

The way to write an algorithm ⇒ ALGORITHM NAME {
inputs)
output(S)
steps
}

→ Title of the course → Design and analysis of Algorithms
↳ - Time efficiency of Complexity
- Space " "

Approaches
① theoretical analysis
② Empirical "

→ Dose there exist a better algorithm? lower bounds, optimality analyzing Algorithms → To asymptotically compare the performance of multiple algorithms used to solve the same problem.

How to compare with one is better (time complexity, space complexity)

Comparison based on Time complexity (basic operation)

* Asymptotic performance Analysis. [For the n is size of the input]

$f(n) = n$ ③ , $f(n) = n^2$ ④ $o(1) < o(\log n) < o(n) < o(n^2)$

$f(n) = \log n$ ② , $f(n) = n^3$ بالنسبة لل growth

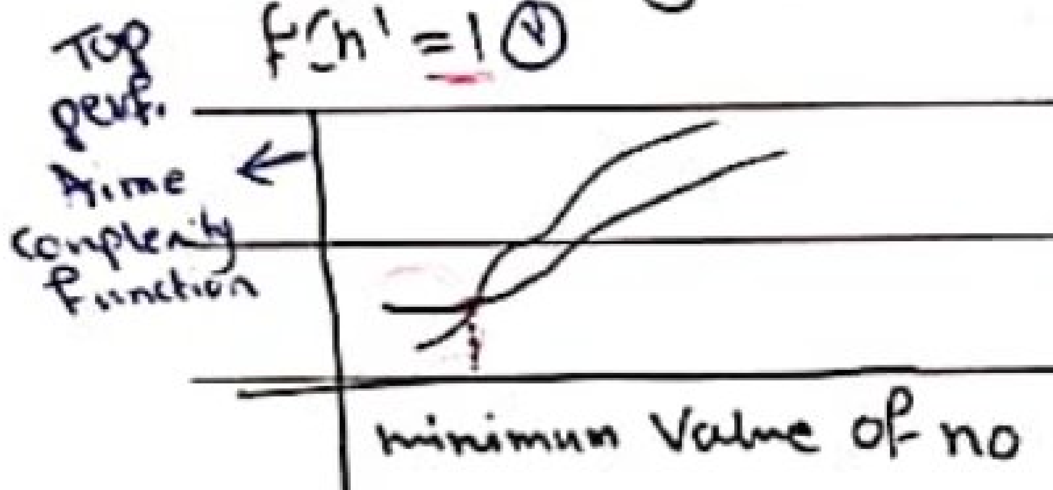
$f(n) = n \log n$, $f(n) = 2^n$

$f(n) = 1$ ①

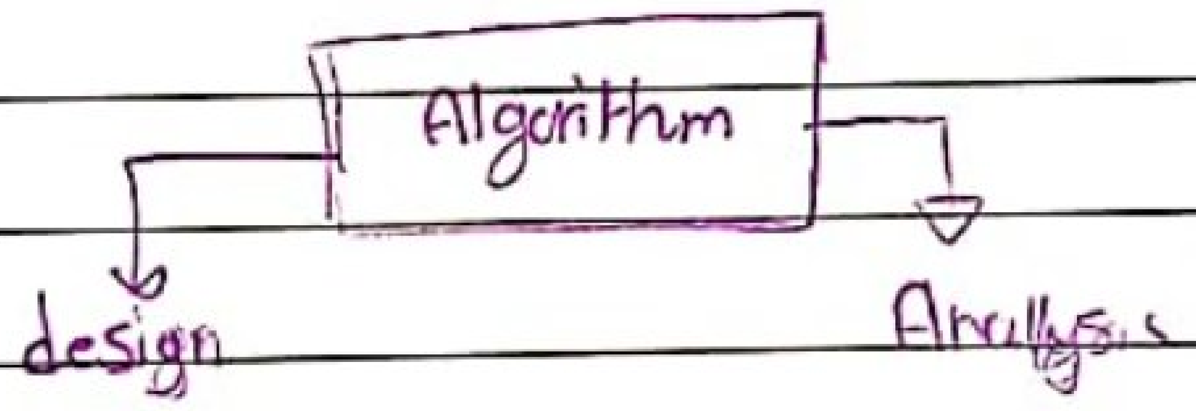
لحتى اعرف لازم اقارن ع عدد n كبير

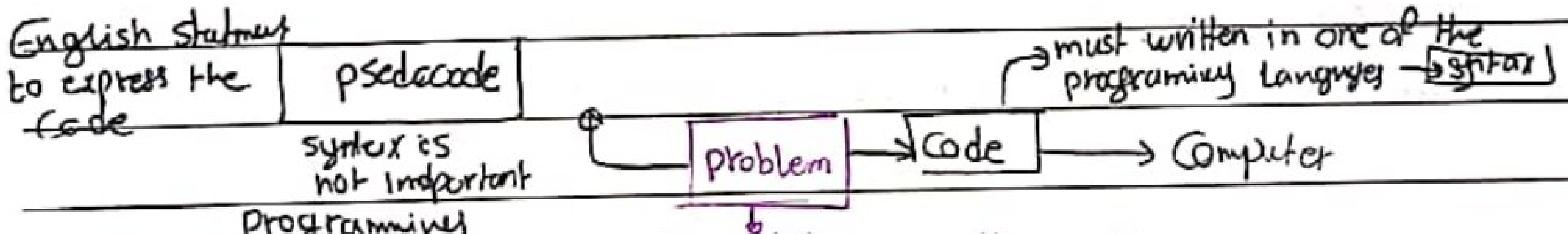
to finding order of Growth

Count # of Basic operand as a function of the size of the input



$n > n_0$
↳ knorn





English statmeus to express the ~~code~~
psedocode
problem
Code
Computer
must written in one of the programming langiges → syntax

syntax is not important
Programming languigs independent.

we desired to solve this problem using computer

⇒ we need to write a code or program.

problem → we can have Multiple Algorithms to solve the same problem ~~but~~ we desire the algorithms that provides the best performance

but what do we mean By the word performance?

↳ we will Quantify this word — time complexity / space complexity

time ↑ speed ↓ time needed to sploit the input and get corred and desird output

→ Asymptotic performance → order of growth Notation

Big-oh O( )

Big-theta Θ( )

Big-omega Ω( )

little-oh o( )

little-omage ω( )

lowest growth → # of Basic operands ↓ → Least of time ↓ → performance or speed ↑

↓time ↑ performance or speed

# of Basic operands

lowest growth

Design techniques. ① Brute force ② Divide and ..

Sorting problem

. lets assume each Algorithm step will take one clock cycle

Example. lets Consider an Algorithm for Searching a list of element for a partical one (key)

→ linear serch $\overset{input}{(A, n, k)}$

A → array; n → size of array; k → particular key that we are looking for

input: linear list of element A, $\overset{index}{A = \{a_1, a_2, \dots a_n\}}$

output: return the index of the forrd key otherwise, return "-1" or "null"

```
linear Search (A, n, k)
{
  " for i=1 to n (int i=1, i<=n; i++)
  { if (k == A[i])  → Basic operation
        return i;
  }
  return -1
}
```

A = {5, 10, 3, 4, 20}

n = Size of input

	# of comparison
~~k=30~~	
k=30	5 → worst case O(n)
k=5	1 → Best case O(1)

small order terms and Constants are Ignored

2n → o(n)

3n + 1000000 → o(n)

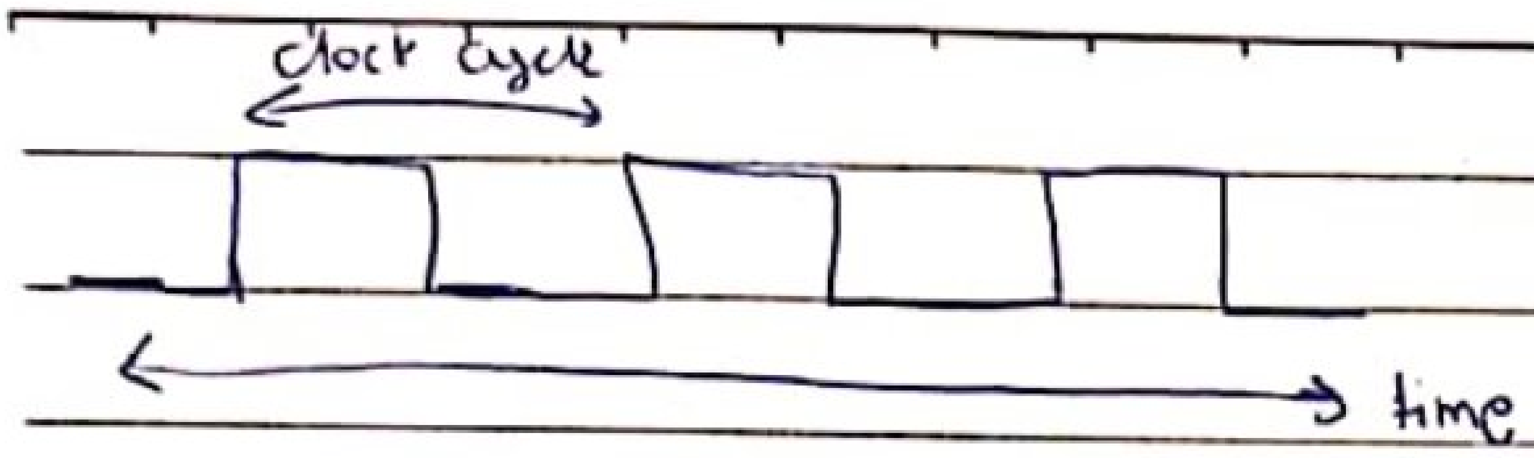
n → O(n)

CR = 2GHZ

clock cycle time $= \frac{1}{CR}$

$= \frac{1}{2\times10^{9} \, HZ} = 0.5 \, nsec$

Independently from hardware and programme



Empirical

clock Rate = $\frac{\text{\# of clock cycle}}{1\,sec}$ = # of clock cycle HZ , $HZ = \frac{1}{sec}$

4 GHZ → 4×10^9 HZ

clock cycle Time = $\frac{1}{CR} = \frac{1}{4\times10^9}$ sec = 0.25 nsec.

Array of n element ← A[a₁,a₂,…an] ; → Key to be searched for

Linear search $\underbrace{(A, K)}_{\text{input}}$

```
for i=1 : n
    if ( A[i] == K)
        return i ;
return -1 ;
```

basic operation

[مقارنة] steps

هنا احنا بنحل theoretically مابنعرفش أوقات

. we count the basic operation [best case / worst "]

. problem 2 :

$T(n) = \sum_{i=1}^{n-1} 1 = n-1$ ↘ one unit of time

: MAXval ← A[0]

for (1 ← i) للوصول لل MAX

for j

problem 3 :

# of n → last index - first index + 1

two for loop واحد اللي بتنقل وواحد اللي بتقارن

$n-1-1+1 \rightarrow \boxed{n-1}$

$T(n) = \sum_{i=1}^{n-1} i = (n-1)(n-2)+ \dots +1$ بجمع عدد المقارنات

$\Theta(n)$

$\rightarrow \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)(n)}{2} = \frac{n^2-n}{2}$

$\Theta(n^2)$

* Arithmatic Series = $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

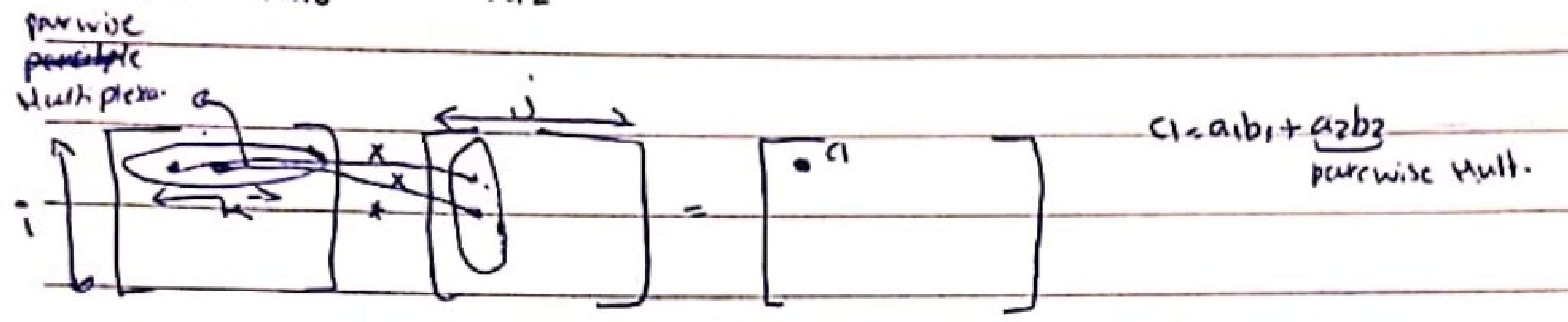
$\sum_{i=1}^{n} i = 1+2+3+4+5+ \dots n$

$\sum_{i=1}^{n} = n+n-1+n-2+n-3+ \dots 1$ +

$2\sum_{i=1}^{n} i = (n+1)+(n+1)+(n+1)+(n+1) \dots \rightarrow \frac{2\sum_{i=1}^{n} i}{2} = \frac{n(n+1)}{2}$ نفسها كلها

for i

$A_{n\times m} * B_{m\times L} = C_{n\times L}$



$c_1 = a_1b_1 + \underbrace{a_2b_2}_{\text{parwise Mult.}}$

* online Lec.

Sorting problem

↳ Insertion Algorithm [Design / analysis

↳ input: array and n element $A = \{\overset{\text{index}}{a_1}, \dots, \overset{i}{a_n}\}$ n

output: sorted array → ascending ✓ s.t. $a'_1 \le a'_2 \le \dots \le a'_n$ wher $\{a'_1, \dots, a'_n\} \in A$
↳ decending x

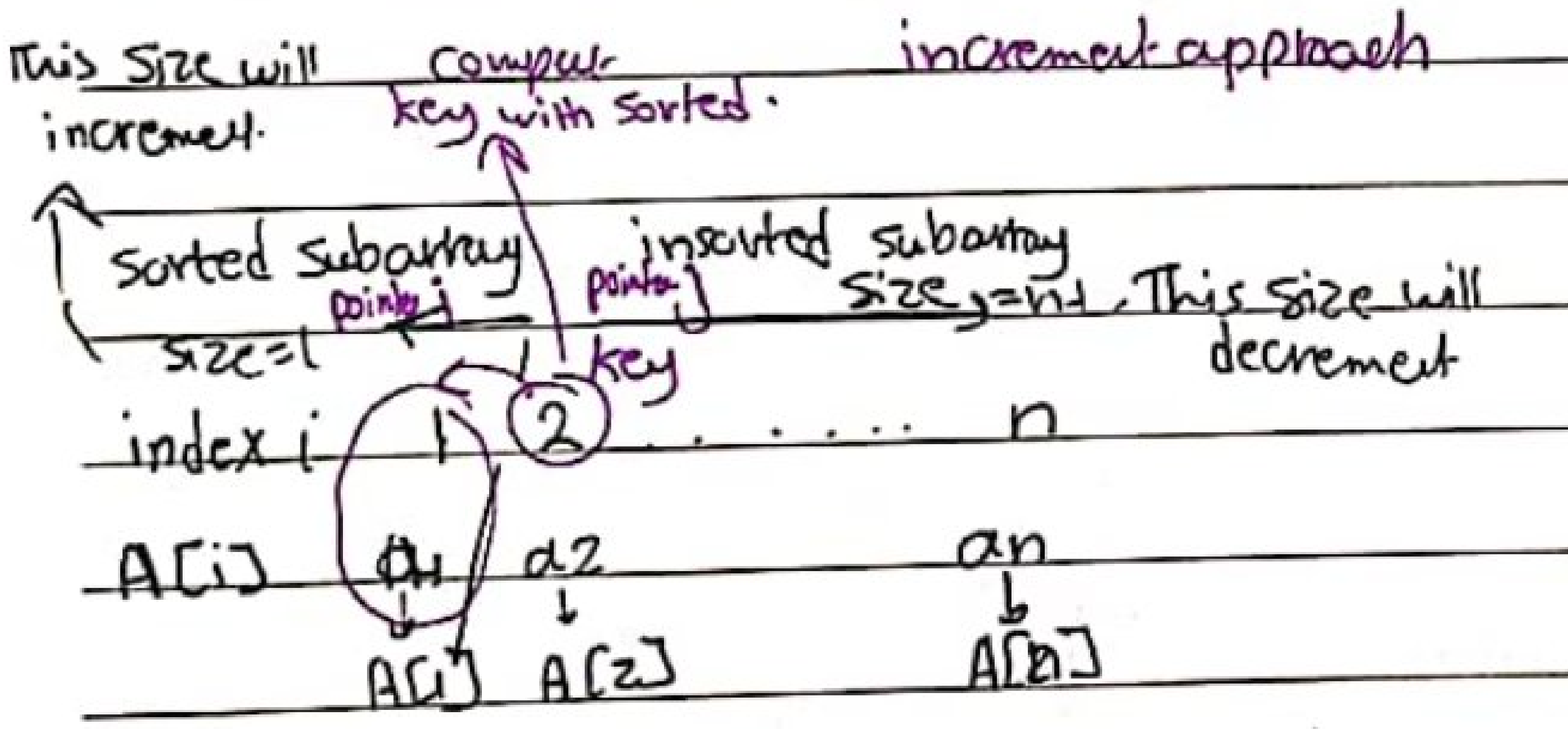
How → need to write the set of steps that will take input to an desired output

↳ Design startigy: Decrese and canque → ① Reduce problem instance to small inst. of the same problems
→ ② Solve small instance
↳ ③ Extand solution of smaller instance to obtain solution to original //.

implementation

Top-down or Bottom up

↳ another name: increment approach



Let's assume the first element is already sorted.

```
Insertion sort (A,n)
[ for (i=2 to n)
    [ key=A[i]
      j=i-1                j محكوم بـ i
```

```
While j>0 && A[j] > key
{ A[j+1] = A[j]
    j = j-1;
}
```

'ec

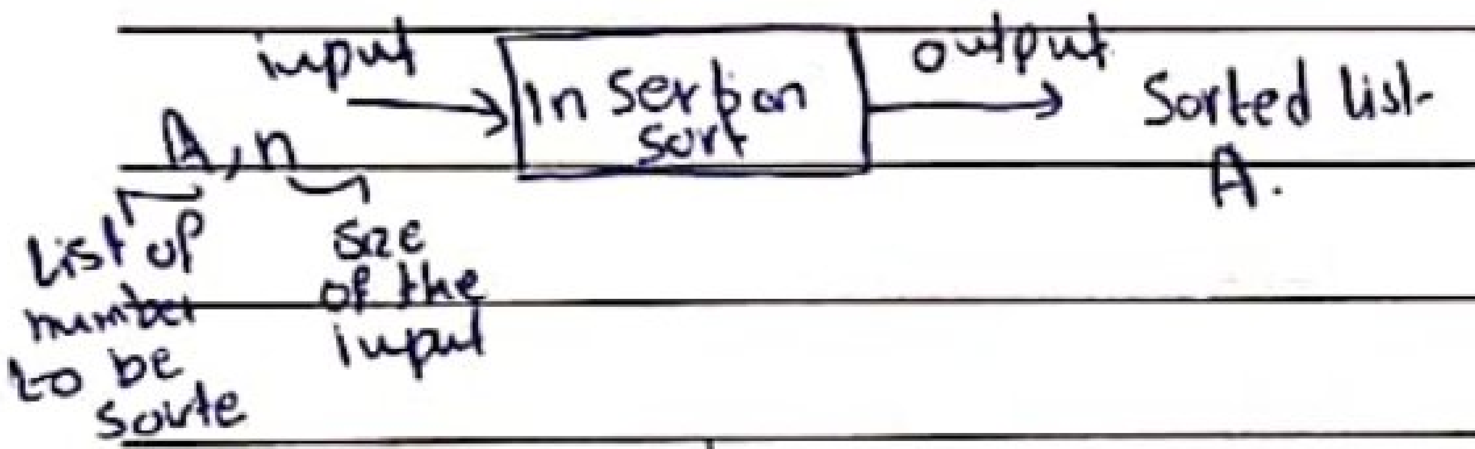
Insertion Sort.

→ Consider as one example of the sorting problem when we have a list of numbers $A = \{a_1 ... a_n\}$ of a goal to find the correct ^اول المتتالي^ peratation $\{\bar{a}_1, \bar{a}_2 ... \bar{a}_n\}$ of the list A s.t:

$a_1' \leq a_2' ... \leq a_n'$ (asending order)

e.g $A = \{3,2,1\} = \{a_1, a_2, a_3\}$, n=3

sol: $\{1,2,3\} = \{a_1', a_2', a_3'\}$



Insertion sort analysis:

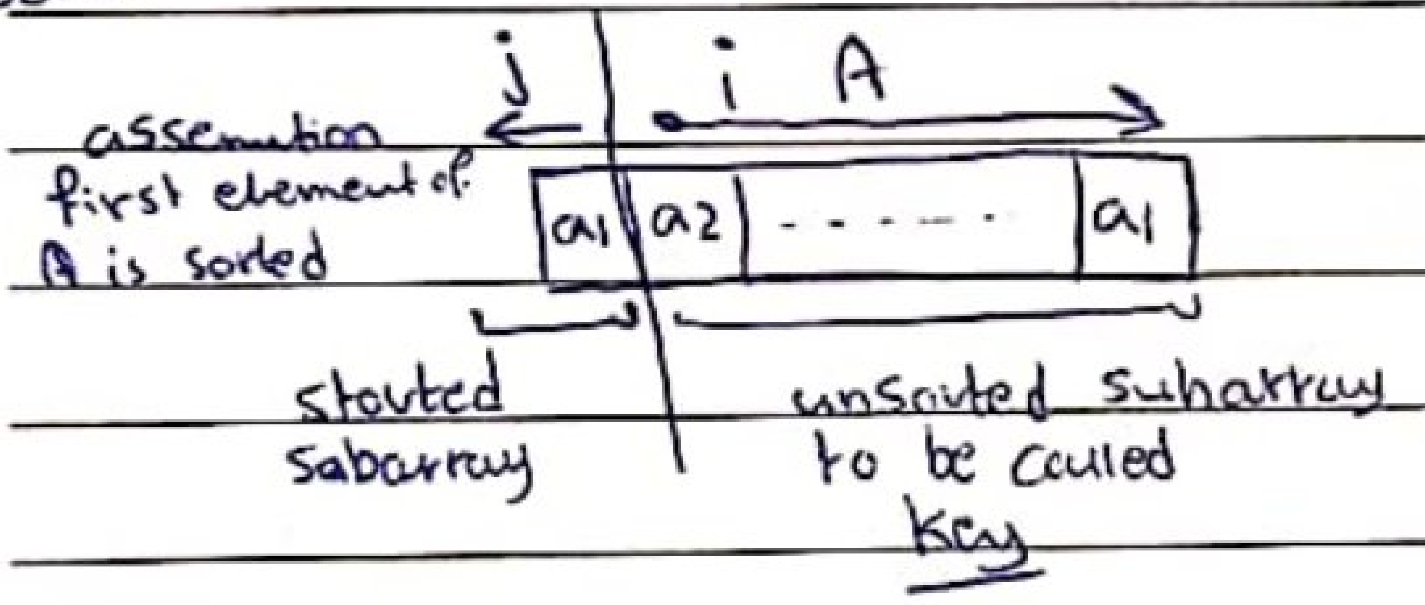
① time complexity: count the basic operation as of a function the size of the input. [T(n)]

↳ Best case:

↳ worst case.

↳ average case: usually requried prubalistic analysis (not focus)

② space complexity:


j
i A
assumtion first element of A is sorted
a1
a2
- - - - -
an
storted subarray
unsorted subarray to be called key

ادا a2 > a1 بعكس

بيكون عندي 2 points واحد بعين وواحد بقارن

line

- Insertion sort (A,n)

```
1.  for i = 2 to n /
2.      key = A[i];                    // 2  // 1
3.      j = i-1;       j بيكون قبل i     // 1  // 2
                       // j>0
4.  while j>0 & A[j] > key       Basic operation.
5.      A[j+1] = A[j];
6.      j = j-1;                       // 1
7.  A[j+1] = key;
```

$\boxed{t_i = 5}$

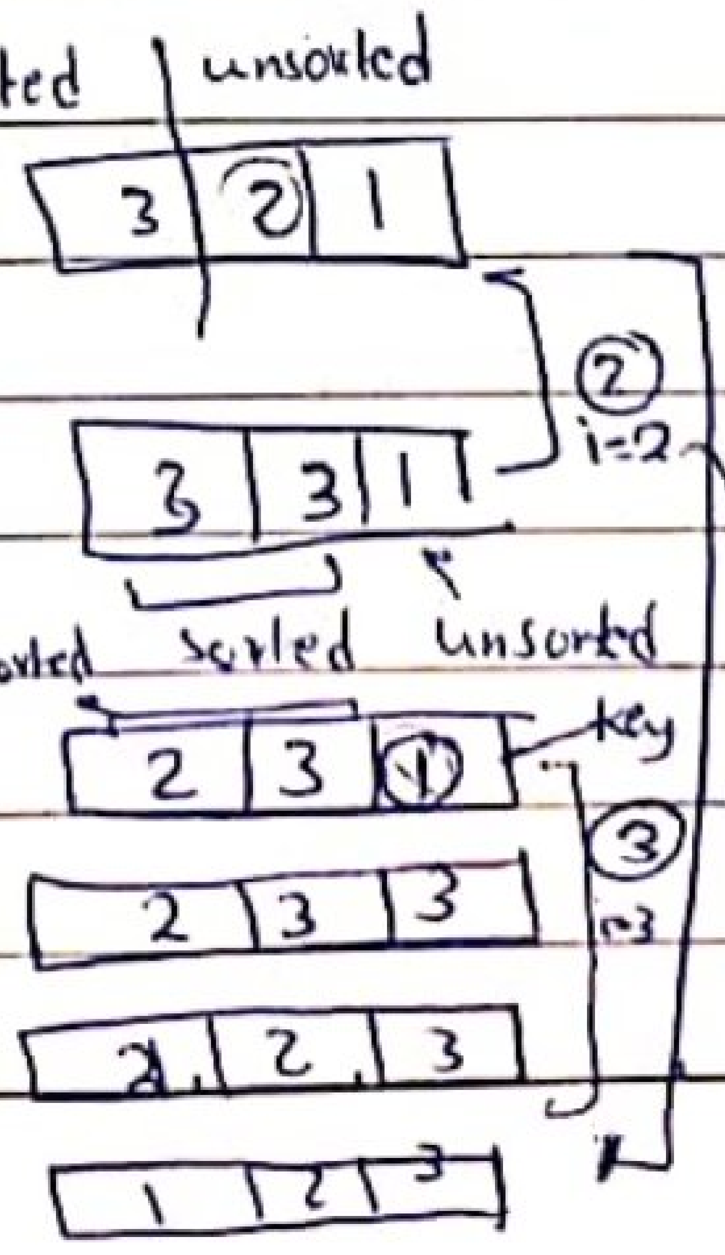
worst case

2+3

لازم j > 0

و key > A[j] نمشي

لازم اكتب الخطوات ←



# of Comparisiont

index for key

عدد المرات باقارن بعدد ③ كانوا كلهم كانوا معكوسات

- Time complexity function: T(n) lets assume t_j = # of time the inner loop while is executed for the ith For loop (outer loop)

$T(n) = \sum_{i=2}^{n} t_i = \begin{cases} \sum_{i=2}^{n} 1, \text{best} \\ \sum_{i=2}^{n} i, \text{worst} \end{cases}$ $\qquad t_j = \begin{cases} 1, \text{Best case} \\ i, \text{worst case} \end{cases}$

هنا بال سم مابنوا للعوال بنجمع ال index

i =	1	2	3	4	5
	{ 6	5	3,	2,	1 }
# of comparison	sorted	2	3	4	5

$\sum_{i}^{n} i = \frac{n(n+1)}{2}$ $\left[\begin{array}{l} \text{let } j = i-1 \\ \text{when } i=2, j=1 \\ i=n, j\ n=n-1 \end{array}\right]$ renaming

$2+3+4+5 = \boxed{14}$ only valied in the (worst case) seniros when the input is reverz sorted

$\boxed{t_i = i}$ quadric $O(n^2)$

1	2	3	5	6
	1	1	1	1

$T_i = 4 = 1+1+1+1$ $\quad \boxed{4}$ مو بس ح نقارن

$T(n) = \boxed{n-1}$ → 4 Input is already sorted (Best case)

$\boxed{t_i = 1}$ Linear $O(n)$

$\boxed{14}$? $\rightarrow \sum_{i=2}^{n} i = \underbrace{\sum_{j=1}^{n-1} j+1}$ $= \left\{\sum_{j=1}^{n-1} j\right\} + \left\{\sum_{j=1}^{n-1} 1\right\} = \frac{(n-1)(n-1+1)}{2} + n-1$ ← عشان الا بدأنا نعمل فيها $\boxed{n-1-1+1}$ n-1

عشان نخليها شبه اللاولى نعمل rename

$= \frac{(n-1)(n)+2n-2}{2} = \underbrace{\frac{n^2+n-2}{2}}_{T(n)}$ $\quad \frac{25+30-2}{2}$ (14)

closed Form Solution

$O(n^2)$ → order of Basic Asemptotic

line	time	count
1	C_1	n ← مفيتوح عملوات تاني نعملوها
2	C_2	n-1
3	C_3	n-1
4	C_4	$m = \sum_{i=2}^{n} t_i$ → # of times to execute this line (while) at the i iteration loop ال مرات عدد
5	C_5	m-1 → لاخر ام ببدا [sufficient كافية]
6	C_6	m-1
7	C_7	n-1

This is Design : ~~Design~~ Decrease & Conquer or increase ~~and~~ & ~~Conquer~~ Approche

Ex: A = { 7 | 2 | 1 | 6 | 4 }

7	7	1	6	4
2	7	1	6	4
2	7	7	6	4
2	2	7	6	4
1	2	7	6	4

efficeit For this algorith → n small or n in Best Case

* عبرنا عن بال Time ، اما بال Space ما احتجنا storage زيادة

* Best Case → # of Comparison (basic operation) = $\underbrace{n-1}$: $O(n)$
The inner loop "while" is only exeuted once

* worst case → when the input is reverse sorted (In this case) the inner loop loop "while" ~~is only~~ is executed number of time = the index value of outer loop For

i k

4 (3) 2 1 $\boxed{\text{key = 3}}$

امثلة من عندي

(4) 4 2 1

3 4 2 1 key = 2

3 4 4 1

3 3 4 1

2 3 4 1 key = 1

2 3 4 4

2 3 3 4 $\boxed{2+3+4 = 9}$

2 2 3 4

1 2 3 4

i k

7 2 1 6 4 key = 2

7 7 1 6 4

2 7 1 6 4 key = 1

2 7 7 6 4

2 2 7 6 4

1 2 7 6 4 key 6

1 2 7 7 4

1 2 6 7 4 key

1 2 6 7 7

1 2 6 6 7

1 2 4 6 7

30 10 40 20

(2) 30 30 40 20

10 30 40 20

(1) 10 30 40 20

10 30 40 40

(3) 10 30 30 40

10 20 30 40

(6) → average

Best worst

(3) (9)

اذا عكسنا يكون non-stable

stability . For dubicating $A.\{\underset{i}{\overset{b}{x}}, x, x, \quad \underset{k}{x}\}$

① nc change of variable $\sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1} 1$

② $\sum_{k=1}^{n-1} k+1$ let $k = j-1$ s1 $j=2 \rightarrow k=1$ $j=n \rightarrow k=n-1$ $\hookrightarrow j=k+1$ $= \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$

③ $\underbrace{\sum_{j=2}^{n} j}_{\frac{n^2+n-2}{2}} - \sum_{j=2}^{n} 1 = (\quad)(n-1)x_1$ $+(n-1+1)(1)$

$+ (n-1)$ $\frac{(n)(n-1)}{2} + (n-1)$

Best Case → The input is already sorted or we have array of duplicates

worst Case → when the input is reverse sorted

$$\sum_{k=0}^{10} 2^k = \frac{2^{\overset{n(+1)}{(10+1)}} - 1}{2-1}$$ closed form solution الحل بلا فاة

$$\sum_{k=0}^{\infty} \frac{d}{dx}\left[x^k\right] = \frac{d}{dx}\left(\frac{1}{1-k}\right)$$ اشتقاق الـ Sum يعني توزيع الكل

$$= \sum_{k=0}^{\infty} k x^{k-1} = \frac{-1 * -1}{(1-x)^2}$$

$$x\left[\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}\right]$$!الأس

$$\boxed{\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}}$$ $+O(1)$ لجل اي Constant

Ex 1 for
nested loop [do for.
do

$$T(n) = \sum_{i=1}^{n}\left[\sum_{j=i}^{n} 2*1\right] = \sum_{i=1}^{n} 2(n-i+1) - 2\sum_{i=1}^{n} n - 2\sum_{i=1}^{n} i + 2\sum_{i=1}^{n} 1$$

$$= 2n^2 - 2\left(\frac{n(n+1)}{2}\right) + 2n$$

$$= 2n^2 - n^2 - n + 2n$$

$$= n^2 + n$$

$O(n^2)$ أكبر قو   closed form solution

Ex 2:

ال ∑ للـ nested lop اكو بس نستخدم

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\underbrace{\sum_{k=1}^{j}2} = \sum_{i=1}^{n}\sum_{j=1}^{i}2(j-1+1) = \sum_{i=1}^{n}\underbrace{\sum_{j=1}^{i}2j} = \sum_{i=1}^{n}2\left(\frac{(i)(i+1)}{2}\right)$$

$$= \sum_{i=1}^{n} i^2+i$$

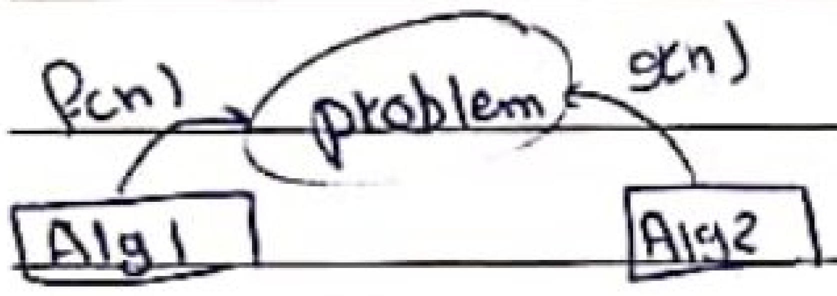
$$= \sum_{i=1}^{n} i^2 + \underbrace{\sum_{i=1}^{n} i}$$

$$= \frac{n(n+1)(2n+1)}{2} + \frac{n(n+1)}{2}$$

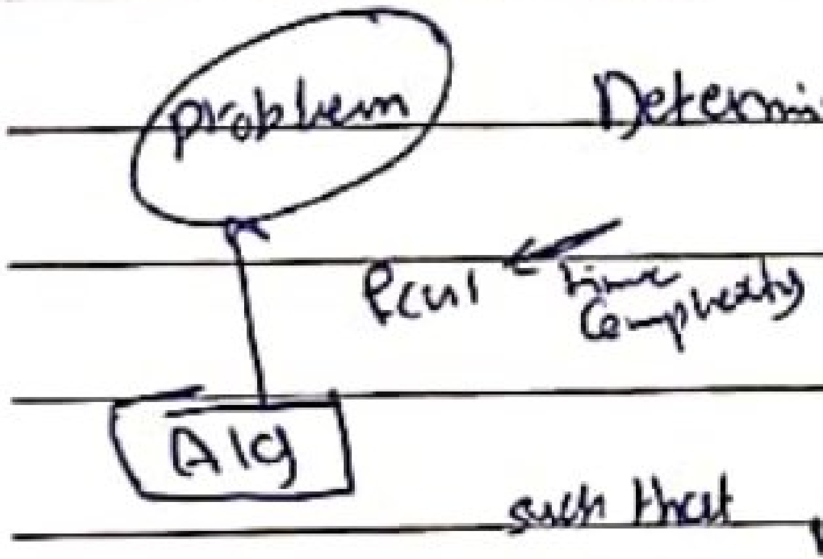
lect 2.

order of growth in relation

$f(n) = \square\,(g(n))$



f(n) and g(n) are time Complexity functions



Determine an ① upper bound of the given function
② Lower "
③

such that there exists
All f(n)} s.t. ∃ Constant

$c, n_0 > 0$, where

$0 \le f(n) \le c\,g(n)$

prove that $f(n) = O(g(n))$, when $f(n) = n^3+8n^2-4$

$g(n) = n^3$

→ find $c, n_0 > 0$ s.t. $\frac{n^3+8n^2-4}{n^3} \le \frac{cn^3}{n^3}$ $\forall n \ge n_0$

$1 + \frac{8}{n} - \frac{4}{n^3} \leq c$ g(n) اكبر ممكن نوصله

↳ let $n_0 = 2$

$1 + \frac{8}{2} - \frac{4}{8} \leq c$

$4.5 \leq c$ then chose $\boxed{c = 5}$

مثل ال O بالعكس

$\cdot\ f(n) = \Omega\ g(n)$

prove that $f(n) = \Omega(g(n))$

when $f(n) = n^3 + 8n^2 - 4$

$g(n) = n^3$

$cn^3 \leq n^3 + 8n^2 - 4$

$c \leq 1 + \frac{8}{n} - \frac{4}{n^3}$

$c \leq 1 + \frac{8}{2} - \frac{4}{8} \longrightarrow c \leq 4.4$

$c = 4$

$f(n) = \boxed{\Theta}\ g(n)$

All $f(n)$'s st $c_1, c_2, n_0 > 0$

when Ω

$0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall n \geq n_0$

little

* غلطا اشيل الاوله بم يكتبكا o, ω

$\cdot \lim\limits_{n \to \infty} \frac{f(n)}{g(n)}$ = constant $\boxed{\Theta}$

$\cdot \lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = 0$ $\boxed{o}$

inverse

$\omega \Longleftrightarrow o$

$\cdot \lim\limits_{n \to \infty} \frac{f(n)}{g(m)} = \infty$ $\boxed{\omega}$

~~another prog using the~~ file

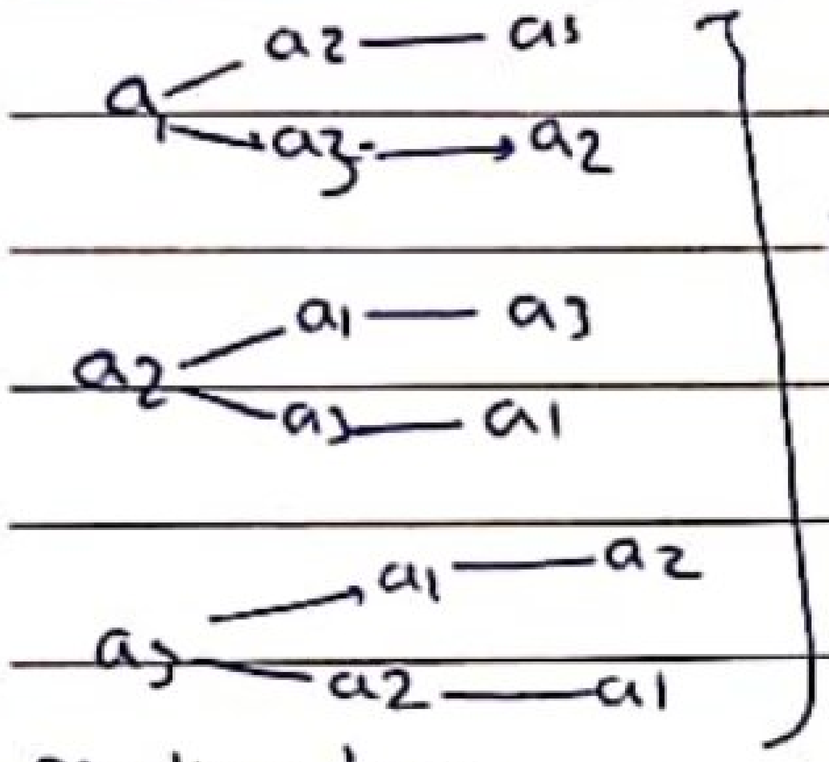
denied

~~attempt to~~ find ~~the file backstrap~~

another istall-on is alread in ~~progress~~

[illegible] Ex: a1, a2, a3 n=3

$3! = 3\times2\times1 = \underline{\underline{6}}$



pemetion tree

For sure ~~of~~ one of thes permution will be our disred solution

① list all the permutation as shown above for an input for zise n

② check the condition

(Brute force
Historical
sorting

O(n!) time complexity

- while

() → space (whileلل تبقى

for

Useful Summation Formulas

- **Arithmetic Series:** Constant differences $a_k - a_{k-1}$.

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2) \qquad \sum_{k=1}^{n} k^2 = \frac{n(2n+1)(n+1)}{6}.$$

- **Geometric Series:** Constant ratio $\frac{a_k}{a_{k-1}}$. For real $x \neq 1$,

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

Infinite decreasing geometric series if $|x| < 1$:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

- **Harmonic Series:** For positive integers n,

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$$

- **Integration and Differentiation of Series:** By differentiating both sides of the infinite geometric series formula and multiplying by x, we get:

$$\sum_{k=0}^{\infty} k * x^{k-1} * x = \sum_{k=0}^{\infty} k * x^k = \frac{x}{(1-x)^2}$$

Analyzing Code: Example 1

Analyze the running time of the following code segment, assuming that the time to perform the assignment on line 3 is 2 units of time. Provide a summation and solve it in closed form.

```
1. for $i \leftarrow 1$ to $n$
2.     do for $j \leftarrow i$ to $n$
3.         do $k \leftarrow k + j$
```

Analyzing Code: Example 2

Analyze the running time of the following code segment, assuming that the time to perform the assignment on line 4 is 2 units of time. Provide a summation and solve it in closed form.

```
1. for $i \leftarrow 1$ to $n$
2.     do for $j \leftarrow 1$ to $i$
3.         do for $k \leftarrow 1$ to $j$
4.             do $x \leftarrow x + 1$
```

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 2 \qquad \sum_{i=1}^{n}\sum_{j=1}^{i} 2(j-1+1)$$

$$\sum_{i=1}^{n} \frac{2\, i(i+1)}{2}$$

$$\sum_{i=1}^{n} i^2 + i$$

$$\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i$$

$$\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}$$

Discussion of the INSERTION-SORT Analysis

- What is the best-case running time for insertion sort? When does it occur?
  - Best case -- inner loop body never executed
  - The input array is already sorted
    - T(n) is a linear function

$$T(n)= c_1 n + c_2(n-1) + c_4(n-1) + \xcancel{c_5 \sum_{j=2}^{n} t_j} + c_6 \sum_{j=2}^{n} \xcancel{(t_j - 1)}$$
$$+ c_7 \xcancel{\sum_{j=2}^{n} (t_j - 1)} + c_8(n-1)$$

$$T(n) = c_9 n + c_{10}$$

Discussion of the INSERTION-SORT Analysis continued

- **What is the worst-case running time for insertion sort? When does it occur?**
  - Worst case -- inner loop body executed for all previous elements
    - **T(n) is a quadratic function**

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad -(c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

$$T(n) = c_9 n^2 + c_{10} n + c_{11}$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

Discussion of the INSERTION-SORT Analysis continued

- What is the **average-case running time** for insertion sort? (Refer to the book)
  - The "average case" is often roughly as bad as the worst case.
    - Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray A[1 .. *j*-1] to insert element A[*j*]
    - On average, half the elements in A[1 .. *j*-1] are less than A[*j*], and half the elements are greater.
    - On average, therefore, we check half of the subarray A[1 .. *j*-1] , and so t_j is about *j*/2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.
  - It should be mentioned that the average-case running time of an algorithm is usually being computed by applying the technique of probabilistic analysis (See Chapter 5)

- How much **space** is needed in (best-case, average-case, worst-case)?

- Is Insertion sort considered as a stable algorithm?

Algorithms

Dr. Khalil Yousef

Lecture 2:

Reading Assignment:
Read Chapter 3 of *the book*

Course Learning Outcomes

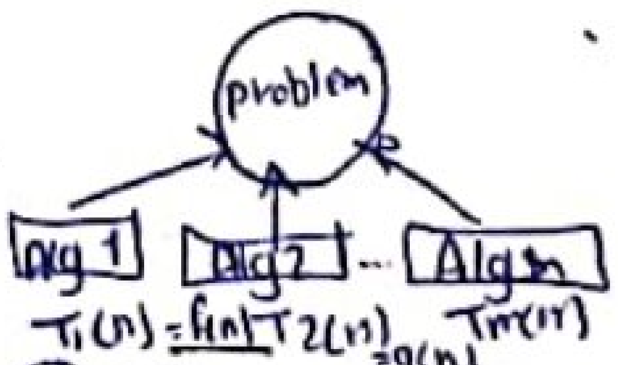
Analyze numerical computations algorithms

(e.g. matrix multiplication).

order of growth function and Notation → it is an important tool to compare the performance of sevival algorithme that solve a certain problem.

Time Complexity — Space Comp.

→ we will use five order of growth functions Notation.



order of growth fun. expression

① Big-oh $O(\,)$ ② Big-omega $\Omega(\,)$ ③ Big-theta ④ Little-oh $o(\,)$ ⑤ little-omega $\omega(\,)$

$f(n) = O(g(n))$

this operator isn't intendded to reflect an assignment relationship, but we use it for simplicity to main $\in$ (Belong to)

In other words, $O(g(n))$ is a class of functions, it isn't a single function

⇒ by writing the above expression we mean

Outline

- Big-Oh and Other Notations in Algorithm Analysis
  - Classifying Functions by Their Asymptotic Growth
  - Theta, Little oh, Little omega
  - Big Oh, Big Omega
  - Rules to manipulate Big-Oh expressions
  - Typical Growth Rates

Constant. $O(1)$ → top performane
linear $O(n)$
quadbric $O(n^2)$
:
exponitul $O(2^n)$
:
→ worst performance

Classifying Functions by Their Asymptotic Growth

- Asymptotic growth:
  - The rate of growth of a function as a size of input [term of n]

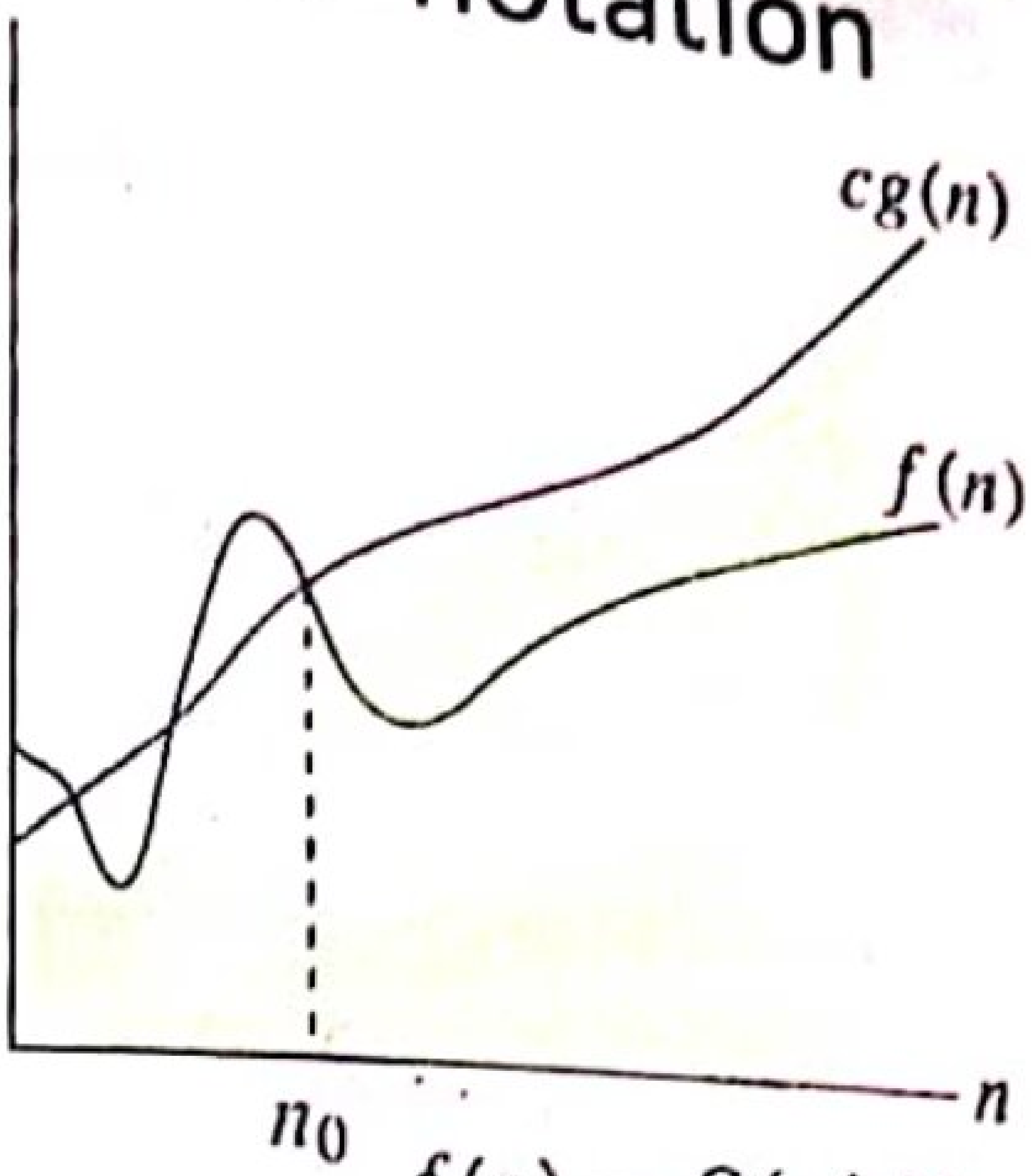
- Given a particular differentiable function *f(n)*, all other differentiable functions fall into three classes:
  - Growing with the same rate $\Theta(\ )$
  - Growing faster $\Omega(\ )$ or $\omega(\ )$
  - Growing slower $O(\ )$ or $o(\ )$

The Big-Oh Notation: O-notation

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions
- $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq cg(n),$$

$$\text{for all } n \geq n_0\}$$



$f(n) = O(g(n))$

We say $g(n)$ is an *asymptotic upper bound* for $f(n)$
Or we say $f(n)$ grows with **same rate or slower** than $g(n)$.

Important Notes

- A very useful aspect of this asymptotic notation (*Big-Oh* and others) is that constants and lower-order terms can be ignored.
  - Example
    - $5n^2+100n+22 = O(n^2)$ and $n = O(n^2)$
- The worst-case running time of INSERTION-SORT is $O(n^2)$, but this does not imply that there is a $O(n^2)$ bound on every input. For example, on sorted input INSERTION-SORT runs in linear time.
- Though we write $f(n) = O(g(n))$, technically this means $f(n) \in O(g(n))$.

Asymptotic Notation in Equations

- When asymptotic notation appears alone on the right-hand side of an equation, as in $f(n) = O(n^2)$, we are indicating that $f(n) \in O(n^2)$.

- When it appears in a formula, we interpret it as standing for *some anonymous function that shall remain nameless.*

- For example, $2n^2 + 3n + 1 \overset{\text{equal operator.}}{=} 2n^2 + \overset{\text{any function from the linear class [highest order n]}}{\Theta(n)}$ means: $2n^2 + 3n + 1 = 2n^2 + \underline{f(n)}$ where $f(n)$ is some function in the set $\Theta(n)$.

  $n+3$
  $\frac{n}{5} - 100 \ldots$

- By using this mechanism, we can eliminate clutter in equations.

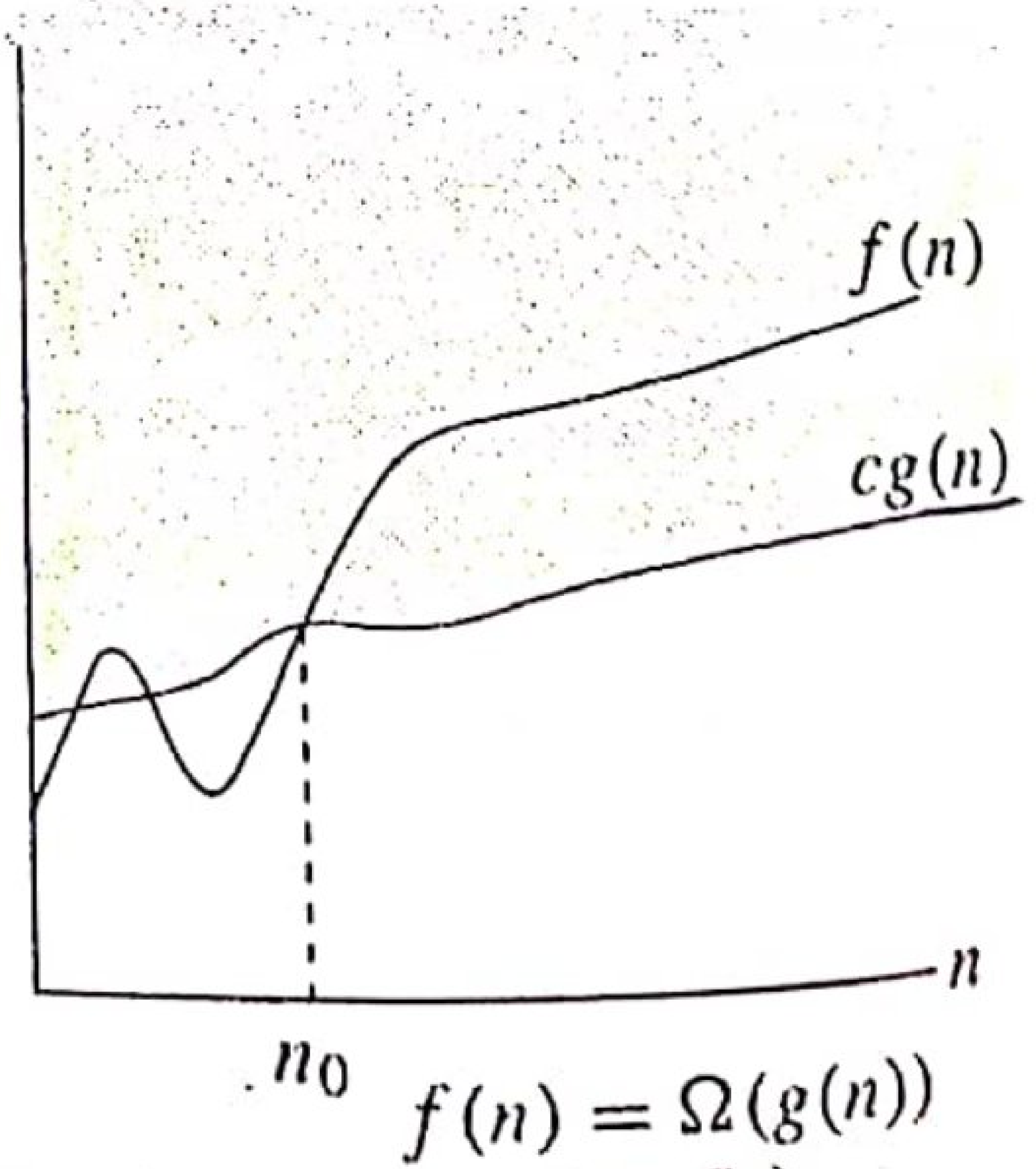
- What if it occurs on both sides? As in $2n^2 + 3n + \overset{\text{constant.}}{\Theta(1)} = 2n^2 + \overset{\text{linear}}{\Theta(n)}$. We take this to mean for any $f(n) \in \Theta(1)$ there is some $g(n) \in \Theta(n)$ such that $2n^2 + 3n + f(n) = 2n^2 + g(n)$.

The Big-Omega Notation: Ω-notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions
- $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

  $0 \leq cg(n) \leq f(n)$

  for all $n \geq n_0\}$



We say $g(n)$ is an asymptotic lower bound for $f(n)$

Or we say *f(n)* grows with **same rate or faster** than $g(n)$.

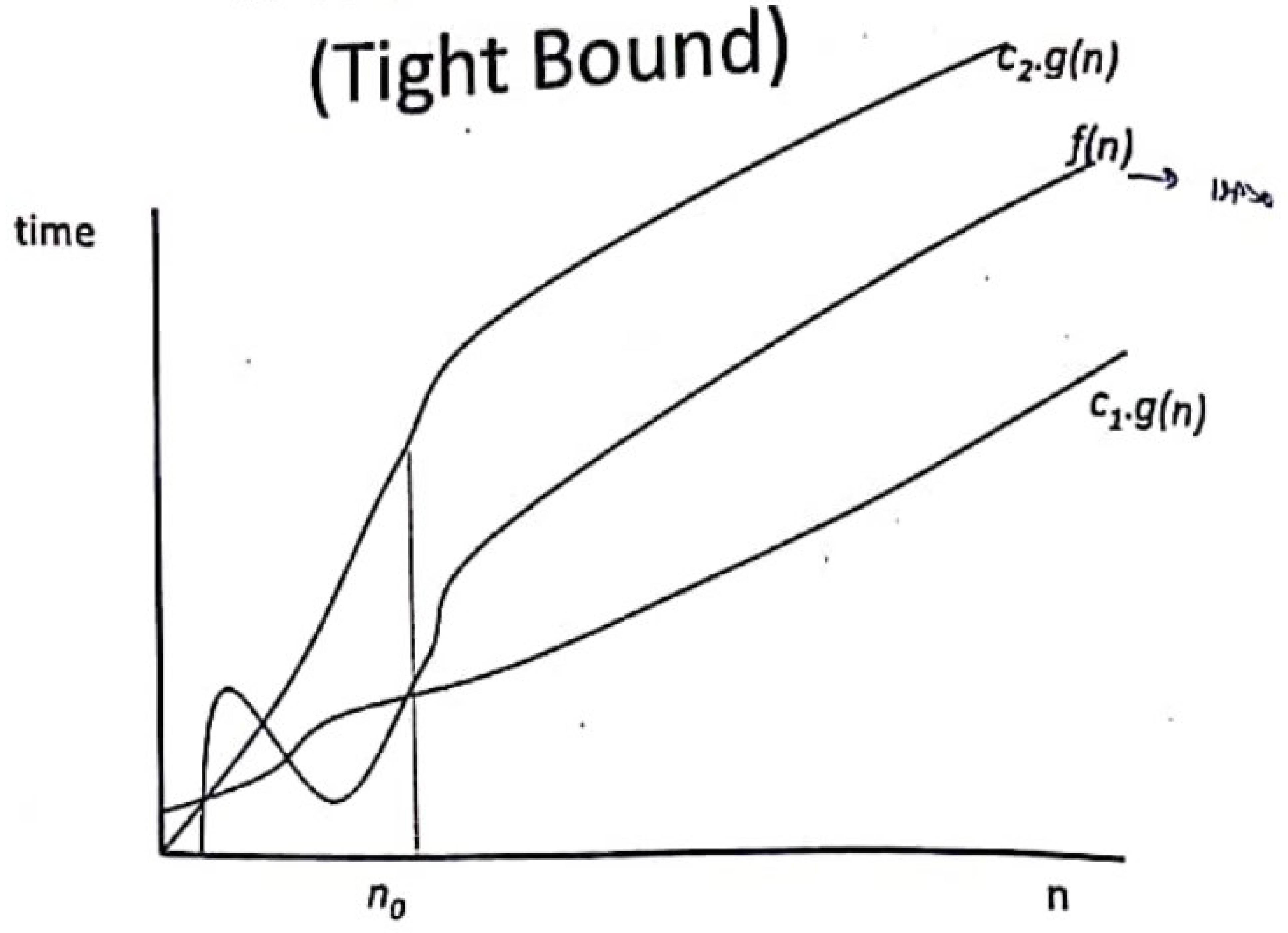
The Big-Omega Notation: Ω-notation

- Example:
  - $5n^2+100n+22 = \Omega(n^2)$ and $n^2 = \Omega(n)$.

Θ notation (Theta) (Tight Bound)

- In some cases,
  - If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - This means, that the worst and best cases require the same amount of time *t* within a constant factor
  - In this case we use a new notation called "theta Θ"
  - "theta Θ" represents an asymptotically tight bound
- For a given function *g(n)*, we denote by $\Theta(g(n))$ the set of functions
  - $\Theta(g(n))$ = {f(n): there exist positive constants $c_1>0$, $c_2>0$ and $n_0>0$ such that
    - $c_1\, g(n) \leq f(n) \leq c_2\, g(n)\ \forall\ n \geq n_0$}

Θ notation (Theta)
(Tight Bound)



$f(n) = \Theta(g(n))$

Discussion of the Asymptotic Notation

- Example 1: prove that $6n^3 \neq \Theta(n^2)$.

Proof by contradiction: i.e., assume $\overset{f(n)}{6n^3} = \Theta(\overset{g(n)}{n^2})$. $\exists c_1, c_2, n_0$

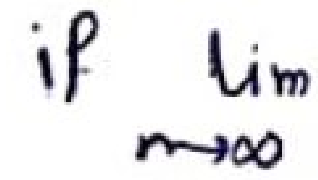
لـ حتى اطبق ال definition

$0 \le \underset{n2}{c_1 n^2} \le \underset{n^2}{6n^3} \le \underset{n^2}{c_2 n^2}, \forall n \ge n_0$ $\to \cancel{\div} n^2$

ear Const رح يكون اكبر من

هل $0 \le c_1 \le \underset{\text{Linear}}{6n} \le \underset{\text{Const}}{c_2}, \forall n \ge n_0$

خطأ



This implies that $n \le \frac{c_2}{6}, \forall n \overset{\text{large size of } n}{\ge} n_0$, a contradiction. $\overset{\text{خطأ}}{}$

Linear faster than Constant for n_0 - - -

if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = Constant $\overset{\neq 0}{}$, then $f(n) = \Theta(g(n))$
= 0 , then $f(n) = O(g(n))$
= ∞ ; then $f(n) = w(g(n))$

$\lim_{n\to\infty} \frac{6n^3}{n2} = \lim_{n\to\infty} 6n = \infty$

بنشتغل ع ال def اذا انطلب

Discussion of the Asymptotic Notation

- Example 2: $f(n) = 5n^2 + 1000n$

✓ Claim: $\underline{f(n)} = \Theta(\underset{}{\overset{g(n)}{\underline{n^2}}})$

① الزم احدد ال no
② احسب c1 و c2

Needed: c_1, c_2, and n_0, such that:

$$0 \le \frac{c_1 \overset{}{\underset{g(n)}{\underline{n^2}}}}{n^2} \le \frac{\underbrace{5n^2 + 1000n}_{f(n)}}{n^2} \le \frac{c_2 \underset{g(n)}{\underline{n^2}}}{n^2}$$

$$0 \le c_1 \le 5 + \frac{1000}{n} \le c_2$$

not unique

$5 + \frac{1000}{1000} \le c_2$

$c = 5$ او اقل ← $\underbrace{c_1 \le 6 \le c_2}$ → $c_2 = 6$ او اكثر

we need to pick n_0

لازم تكون
$c_1, c_2, (n_0) > 0$
قيم موجبة

Than sel compute vave for c_1 and c_2

One choice: $n_0 = 1000, c_1 = 5, c_2 = 6$

we assume no — افضل الحالات

$\lim \frac{f(n)}{g(n)} = \frac{5n^2 + 1000n}{n^2}$

$\lim 5 + \frac{100}{n} \to \boxed{5}$

Constant

So it's theta

Discussion of the Asymptotic Notation

- Example 3: Let us try to prove that $\underline{n \ne \Theta(n^2)}$.

Proof by con<u>tradiction</u>: i.e., assume $n = \Theta(n^2)$.

افرض انه العلاقة صحيحة وطبق ال condition

$$0 \le c_1 \frac{n^2}{n^2} \le \frac{n}{n^2} \le \frac{c_2 n^2}{n^2}, \forall n \ge n_0$$

✓

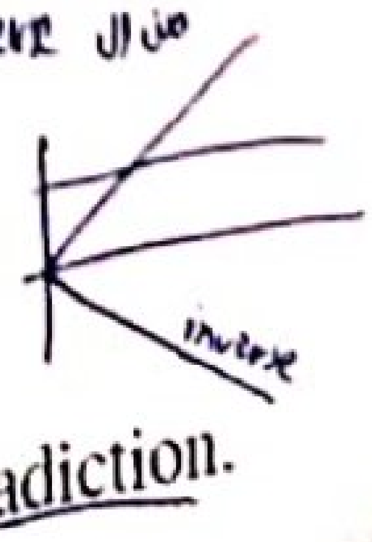
x

$$0 \le \underbrace{c_1 \le \frac{1}{n}}\underbrace{\le c_2}, \forall n \ge n_0$$

inven linear

inver line من > Constant لأن

بـ لأن ال Constant اعلى ال growth من ال linear inverse



This implies that $n \le \frac{1}{c_1}, \forall n \ge n_0$, a <u>contradiction</u>.

Little oh Notation: *o*-notation

← نفس ال O( ) ولكن بدون المساواة

- *o*-notation denotes an upper bound that is not asymptotically tight (it is certain to grow faster). In contrast, O may or may not be asymptotically tight.

- For a given function $g(n)$, we denote by $o(g(n))$ the set of functions:
- $o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$
- $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity: i.e. $f(n) = o(g(n))$ *iff* $\lim_{n\to\infty}[f(n) / g(n)] = 0$
- We say $g(n)$ is an *upper bound* for $f(n)$ that is *not* asymptotically tight.
  - For example, $\underbrace{2n}_{\text{slower than } n^2} = o(n^2)$, but $2n^2 \neq o(n^2)$.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{2n}{n^2} = \lim_{n\to\infty}\frac{2}{n} = \frac{2}{\infty} = \underline{\underline{0}} \checkmark$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{2n^2}{n^2} = \lim_{n\to\infty} 2 = 2 \times$$

↳ constant = Θ theta

O(*) versus o(*)

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$, for all $n \geq n_0\}$.

$o(g(n)) = \{f(n)$: for ***any*** positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$ }.

Thus $o(f(n))$ is a weakened $O(f(n))$.

For example: $n^2 = O(n^2)$

$n^2 \neq o(n^2)$ ✗

$n^2 = O(n^3)$ ✓ little o subset of litile omega

$n^2 = o(n^3)$ ✓

كل شي هو o little هو bigO

وليس كل bigo هو littleo

أي شي هو littleo هو big omega

Little omega Notation: ω-notation

it is the inverse of little O

- For a given function $g(n)$, we denote by $w(g(n))$ the set of functions:
- $\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$
- $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity: $f(n) = \omega(g(n)$ *iff* $\lim_{n\to\infty} [f(n) / g(n)] = \infty$
- We say $g(n)$ is an *lower bound* for $f(n)$ that is *not* asymptotically tight.

same rate

For example, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$

↳ لا يطلع

(Θ) constant

$\lim_{n\to\infty} \frac{\frac{n^2}{2}}{n} = \lim_{n\to\infty} \frac{n}{2} = \frac{\infty}{2} = \boxed{\infty}$

$f(n) = \boxed{17} \; g(n)$

Θ =

O ≤

Ω ≥

o <

ω >

Comparison of Asymptotic Functions

Transitivity:

Conjunction

Means AND

and

$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \to f(n) = \Theta(h(n))$ ⟶ apply كلا

Comment Function

$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \to f(n) = O(h(n))$

$f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \to f(n) = \Omega(h(n))$

$f(n) = o(g(n)) \wedge g(n) = o(h(n)) \to f(n) = o(h(n))$

$f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \to f(n) = \omega(h(n))$

Reflexivity:

Fun نفسها

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Symmetry:

$f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

تبديل ال f(n)
مع ال g(n)

Comparison of Asymptotic Functions (Cont...)

Transpose Symmetry:

O inverse Ω
o inverse ω

اذا قلبت (عكست)
بأخذ ال Inverse

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ iff } g(n) = \omega(f(n))$$

These properties allow us to draw an analogy between the asymptotic comparison of functions f and g and the comparison of real numbers a and b:

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = o(g(n)) \approx a < b$$

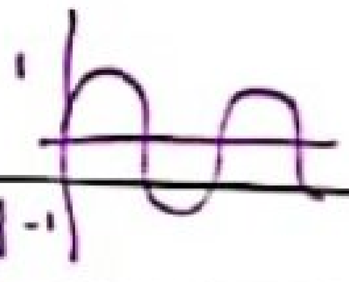
$$f(n) = \omega(g(n)) \approx a > b$$

Although two real numbers can be compared (using <, =, or >), not all functions are asymptotically comparable (e.g., n and n^{1+sinn} cannot be compared; the exponent of the second oscillates between 0 and 2).

static $\underline{\underline{n}}$ $\quad$ $\underline{n^{\overline{1+sinn}}}$

متغير

oscilate between 1 and -1



-1

we can't compare

$n^{1+0} = n$
$n^{1+1} = n^2$
$n^{1-1} = n^0 = 1$

غيرثابت

SUMMARY

The Big-Oh Notation

f(n) = O(g(n))

o( )

if f(n) grows with same rate or slower than g(n). or asymptotic bound

Means f(n) = Θ(g(n)) or
f(n) = o(g(n))

The Big-Omega Notation

f(n) = Ω(g(n))

ω( )

if *f*(n) grows with same rate or faster than g(n).

Means f(n) = Θ(g(n)) or
f(n) = ω(g(n))

The Big-Omega Notation

- **The inverse** of Big-Oh is **Ω**

- If $\quad g(n) = O(f(n))$,
- then $\quad f(n) = \Omega(g(n))$

Theta Θ

- *f(n)* and *g(n)* have same rate of growth, if
  - $\boxed{\lim(f(n) / g(n)) = c,}$ constant
  - $\overset{\infty}{\cancel{0}} < c < \infty, \quad n \to \infty$

    $c \neq 0$

- Notation: $\quad f(n) = \Theta(g(n))$
- Pronounced "theta"

Theta: Relation of Equivalence

- **Θ: "having the same rate of growth":**
  - Relation of equivalence
  - Gives a partition over the set of all differentiable functions - classes of equivalence.

class of functions not one function...

- Functions in one and the same class are equivalent with respect to their growth.

Little oh

$f(n)$ grows slower than $g(n)$
(or $g(n)$ grows faster than $f(n)$)

if

$\lim(f(n) / g(n)) = 0,\ n \to \infty$

Notation: $f(n) = o(g(n))$ pronounced "little oh"

Little omega

$f(n)$ grows faster than $g(n)$
(or $g(n)$ grows slower than $f(n)$)
if

$\lim(f(n) / g(n)) = \infty, \quad n \to \infty$

Notation: $f(n) = \omega\ (g(n))$ pronounced "little omega"

Little omega and Little oh

inverse

- if $g(n) = \mathbf{o}(f(n))$
- then $f(n) = \boldsymbol{\omega}(g(n))$

- **Examples:** Compare $\overset{f}{\boldsymbol{n}}$ and $\overset{g}{\boldsymbol{n^2}}$
  - $\lim(n/n^2) = 0, n \to \infty, n = o(n^2)$ $n = O(n^2)$
  - $\lim(n^2/n) = \infty, n \to \infty, n^2 = \omega(n)$ $n^2 = \Omega(n)$

Examples

Recall: Algorithms with Same Complexity

- Two algorithms have same complexity, if the functions representing the number of basic operations have same rate of growth.

- Among all functions with same rate of growth we choose the simplest one to represent the complexity.

$a^{\log_b n} = n^{\log_b a}$

$* \; 2^{\log_2 n} = n^{\log_2 2} = n^1 = \boxed{n}$

Example

$\boxed{\log_2 n \equiv \lg n}$

$* \; 8^{\log n} = n^{\log_2 8} = n^3$

- Compare ***n*** and ***(n+1)/2***
  - lim(n / ((n+1)/2)) = 2,
  - same rate of growth

- (n+1)/2 = **Θ(n)**
  - Rate of growth of a linear function

اسالة الامتحان

Right side   B function

left side
A function

	n^2	n	$2^{\log n}$	$8^{\log n}$
n^1	O	Θ	Θ	O

A = □ B

Example

- Compare ***n²*** and ***n²+ 6n***
  - lim(n² / (n²+ 6n))= 1
  - same rate of growth.

- n²+6n = **Θ(n²)**
- Rate of growth of a quadratic function

الافضل O, Ω

لازم Θ

Example

- Compare **log *n*** and **log *n*²**
  - lim(log n / log n^2) = $\underline{½}$
  - Same rate of growth.
  - log n^2 = **Θ(log n)**
  - **logarithmic** rate of growth

نفس الشي اذا كان log

$\log n^2 = \underbrace{2}_{\text{ignored}}\log n = \log n$

اذا القوى تغيرت او اذا ال base تغير

بضل نفس class log

$\log_a n^b \quad \log_a n^b$

$\left.\begin{matrix} b \neq b \\ a \neq a \end{matrix}\right\}$ بضل Θ

Example

Θ(n³):
- n^3
- $5n^3 + 4n$
- $105n^3 + 4n^2 + 6n$

Θ(n²):
- n^2
- $5n^2 + 4n + 6$
- $n^2 + 5$

Θ(log n):
- log $\underline{n}$
- log $\underline{n^2}$
- log (n + n³) → $\log n^3 \rightarrow \boxed{3\log n}$

$\log n2 \rightarrow \boxed{2\log n}$

نفس الشي

ال log كله مرفوع لقوة

$(\log n)^{③} \neq \log n^3$

$m3 \neq m$

Example

$n+5 = \Theta(n) = O(n) = O(n^2)$
$= O(n^3) = O(n^5)$, $O(1)$, $O(\frac{1}{n})$

The closest estimation: $n+5 = \Theta(n)$

The general practice is to use

The Big-Oh notation:

$n+5 = O(n)$ will pick the tigure bound

Techniques to show that *f*(*n*) is (not) Θ(*n*), O(*n*), or Ω(*n*)

to prove!

1. Use the definition. For example, to show $f(n) = O(n)$, find positive constants c, n_0 to solve $0 \le f(n) \le cg(n)$.

Content بطلع قيم ال valid ←

2. Proof by contradiction (e.g., to show $f(n) \neq O(n)$).

3a. If $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$ and $f(n)$, $g(n)$ are asymptotically non-negative (≥ 0 for large n), then $f(n) = o(g(n))$. This is a special case for o: $g(n)$ is growing *much faster than* $f(n)$.

3b. If $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = c, c \neq 0$ and $f(n)$, $g(n)$ are asymptotically non-negative (≥ 0 for large n), then $f(n) = \Theta(g(n))$.

Techniques to show that $f(n)$ is (not) $\Theta(n)$, $O(n)$, or $\Omega(n)$

3c. If $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \underline{\infty}$ and $f(n)$, $g(n)$ are asymptotically non-negative (≥ 0 for large n), then $f(n) = \omega(g(n))$. This is a special case for ω: $f(n)$ is growing *much faster than* $g(n)$.

4. L'Hôpital's Rule: If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, then $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)}$.

مشتقة البسط

مشتقة المقام

Rules to manipulate Big-Oh expressions

Rule 1:

a. If

$T_1(N) = O(f(N))$

and

$T_2(N) = O(g(N))$

then

$T_1(N) + T_2(N) =$

max(O(f (N)), O(g(N)))

$\rightarrow n^2+n = O(n^2)$

$+$

$\rightarrow n^3+n^2 = O(n^3)$

$O(n^3)$

eg. n n^2 n^2 n^3 max

في حالة الجمع باخذ الاكبر لل big oh

Rules to manipulate Big-Oh expressions

b. $f(n)+g(n) \neq \Theta(\min(f(n),g(n)))$

$n^4+n^2 \neq \Theta(n^2)$

$\underline{f(n)} \neq O\left((f(n))^2\right)$ → صح في حال كان
positive f(n) فقط

~~[illegible]~~
decreasing ←

$f(n)=\frac{1}{\underline{\underline{n^2}}} \neq O\left(\frac{1}{\underline{\underline{n^4}}}\right)$ → grows in negative $\underline{y}$

$f(n) \neq \Theta(f(n/2))$ decreas the size of input

$f(n)=4^n \neq \Theta\left(4^{n/2}\right)$ في حال exponential

$\underbrace{\boxed{4^n} \quad (4^{\frac{1}{2}})^n = \boxed{2^n}}_{\text{differens classis}}$

Rules to manipulate Big-Oh expressions

C. If

$T_1(N) = O(\ f(N)\ )$

and

$T_2(N) = O(\ g(N)\ )$

then

$T_1(N) \underline{\underline{*}}\ T_2(N) = O(\ f(N) \overset{ضرب}{\underline{\circledast}} g(N)\ )$

لذا بدي اضرب بضرب
ال O

Rules to manipulate Big-Oh expressions

Rule 2:

If T(N) is a polynomial of degree **k**, then

$$T(N) = \Theta(N^k)$$

اي poly يكون
ال Θ highest
order term

Rule 3:

the power k
less than or
same rate
the linear

growth of a log
function regard to

$\log^k N = O(N)$ for any constant **k**. $= (\log N)^k < \text{linear}$

ال log المرفوع لقوة بكون slower
than linear

$$(\log n)^{1000} = \log^{1000} n = O(n)$$

$n^{1+\log n} \boxed{\;} n^1 \to \Omega$

$n^{1+\log n} \boxed{\bullet} n^2 \to \Omega$

any const. $= O(\log n)$

$\to n^{1+\log n} \boxed{\Omega} n^{1000}$

$n \lg n \; \boxed{\;} \; n^2$

$n \log n \; \boxed{O} \; n*n$

$\log n < n$

Examples

- $n^2 + n = O(n^2)$
  - we disregard any lower-order term
- $n\log(n) = O(n\log(n))$ and Θ
- $n^2 + n\log(n) = O(n^2)$ and Θ

linear أقل من
order ال

Standard Notations: Monotonicity

Monotonicity of a function:

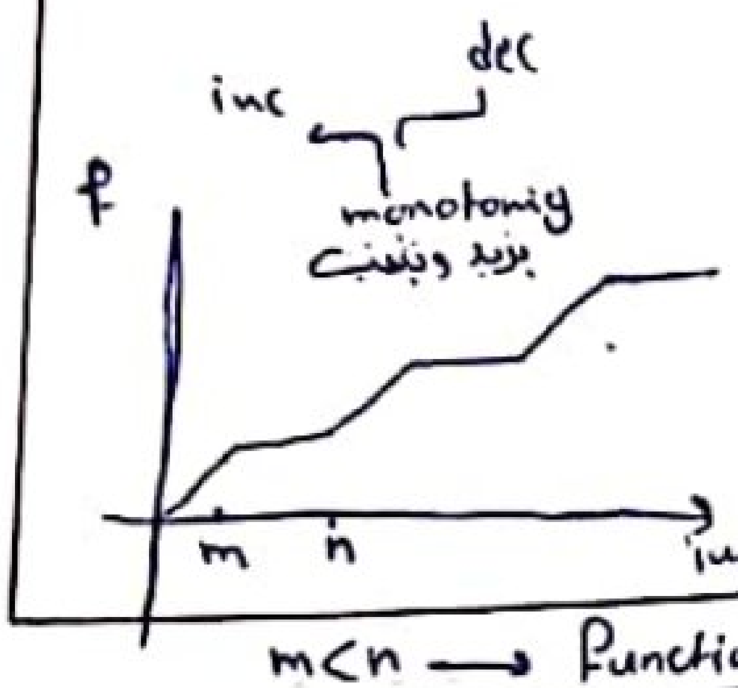
A function is monotonically increasing if $m \le n \to f(m) \le f(n)$.

A function is monotonically decreasing if $m \le n \to f(m) \ge f(n)$.

A function is strictly increasing if $m < n \to f(m) < f(n)$.

A function is strictly decreasing if $m < n \to f(m) > f(n)$.

الdecreas بالعكس
> ← <
< ← >



دائماً متزايد بيكون
strictly
dec inc

$m<n \to \text{function}_m \le \text{function}_n$

Standard Notations: Floors and Ceilings

For any real number x, the greatest integer less than or equal to x is called the floor of x, denoted $\lfloor x \rfloor$.

For any real number x, the least integer greater than or equal to x is called the ceiling of x, denoted $\lceil x \rceil$.

مهم For all real x, $x-1 < \lfloor x \rfloor \le x \le \overset{\text{ceiling}}{\lceil x \rceil} < x+1$.

For any integer n, $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$.

For any integer n and integers $a \ne 0$, $b \ne 0$:

Ceiling مع ceiling
بناخذ ceiling للكل

$$\left\lceil \frac{\lceil \frac{n}{a} \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil$$

floor مع floor
بناخذ floor للكل

$$\left\lfloor \frac{\lfloor \frac{n}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor$$

Polynomials

Given a positive integer d, a polynomial in n of degree d is a function $p(n)$ in the form:

$$p(n) = \sum_{i=0}^{d} \overset{\text{factor}}{a_i} n^i$$

الجبر فوق ←

$a_0 n^0 + a_1 n^1 + \dots a_d n^d$

$O(n^d)$ بناخذ دايمًا ال highest order

$(n^d$

where $a_0, a_1 \dots a_n$ are called coefficients, and $a_d \neq 0$.

For an asymptotically positive polynomial of degree d (i.e., $a_d > 0$), $p(n) = \Theta(n^d)$.

We say that $f(n)$ is polynomially bounded if $f(n) = n^{O(1)}$, which is equivalent to $f(n) = O(n^k)$ for constant k.

Exponentials

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^{-1} &= \tfrac{1}{a} \\ (a^m)^n &= a^{mn} \\ (a^m)^n &= (a^n)^m \\ a^m a^n &= a^{m+n} \end{aligned}$$

It is useful to know some properties of the special exponential e^x.

For example, for all real x, $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \boxed{\sum_{i=0}^{\infty} \frac{x^i}{i!}}$. Opti

For all real x, $e^x \geq 1 + x$, where equality holds only if $x = 0$. ← bound

لازم نتحقق لحتى اعمل هاي الطريقة

When $\boxed{|x| \leq 1}$ $\underbrace{1 + x \leq e^x \leq 1 + x + x^2}_{\text{from series}}$.

e^x faster than $1+x$ and $O(1+x+x^2)$

$\forall x, \lim_{n\to\infty}(1 + \frac{x}{n})^n = e^x$

Logarithms

$$\begin{aligned}
\lg n &= \log_2 n \text{ (binary logarithm)} \\
\ln n &= \log_e n \text{ (natural logarithm)} \\
\lg^k n &= (\lg n)^k \text{ (exponentiation)} \\
\lg \lg n &= \lg(\lg n) \text{ (composition)}
\end{aligned}$$

بدون شي بحول ع دايماً

For all real $a > 0, b > 0, c > 0$: exp + log

$$\begin{aligned}
a &= b^{\log_b a} \\
\log_c(ab) &= \log_c a + \log_c b \\
\log_b a^n &= n \log_b a \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b(\tfrac{1}{a}) &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b n} &= n^{\log_b a}
\end{aligned}$$

$a^{\log_b b} = a^1 = a$

$n^{\log_b a} = a^{\log_b n}$

$4^{\lg n^2} = 4^{2\log n} = 4^2 * 4^{\log n}$

$= 16 * n^{\log 4}$

$= \boxed{16n^2}$

quabritic function

~~= 8 for~~ $8^{\log n} = \boxed{n^3}$

Logarithms

- Show that $a^{\log_b n} = n^{\log_b a}$

$$\text{let } a = b^{\log_b a}$$

$$\therefore a^{\log_b n} = \left(b^{\log_b a}\right)^{\log_b n} = b^{(\log_b a)(\log_b n)} = b^{(\log_b n)(\log_b a)} = a^{\log_b n} = \left(b^{\log_b n}\right)^{\log_b a} = n^{\log_b a}$$

Logarithms (Cont...)

منفصلات

Note that $\underline{\lg n + k} = \underline{(\lg n) + k}$.

Since changing a log base only changes a value by a constant factor, we will usually use $\lg n$ when we don't care about constant factors.

Note, when $|x| < 1$:

series for Ln

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$$

bound for ln

For $\boxed{x > -1}$, $\frac{x}{1+x} \le \ln(1+x) \le x$, where the equality holds if $x = 0$.

A function is polylogarithmically bounded if $f(n) = \lg^{O(1)} n$.

Rates of Growth

ع

The rates of growth of any positive exponential function is $\boxed{\text{faster}}$ than any polynomial function, as the following shows.

For all real constants a, b, where $a > 1$, $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, hence we can conclude $n^b = o(a^n)$.

grow slower exp.

polynomials for b degree

little oh

expontion (a > b)

The rates of growth of any polynomial function is $\boxed{\text{faster}}$ than any polylogarithmic function, as the following shows.

+ Linear

By substituting $\lg n$ for n and 2^a for a in the above, we get:

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0$$

grow slower

Hence, we can conclude $\lg^b n = o(n^a)$, for any $a > 0$.

logrith power لِنْ

Polynomial little-oh

Factorials

$f(n) = \log n!$, $g(n) = (\log n)!$

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n(n-1)! & n > 0. \end{cases}$$

$$n! = \prod_{i=1}^{n} i$$

$2! = 2 \times 1$

$3! = 3 \times 2 \times 1$

$n! = n(n-1)(n-2)\cdots 1$

Intuition: $n!$ is the number possible permutations of a given input set with n members. This is fast-growing!

A weak upper bound on the factorial function is $n! \leq n^n$.

Stirling's Approximation provides us with tighter upper and lower bounds.

$$\hookrightarrow n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \underbrace{\left(1 + \Theta\left(\frac{1}{n}\right)\right)}_{\text{ignor}} = \sqrt{2\pi n}\left(\frac{n}{e}\right)$$

inverse linear

$$\text{lower bond } \underbrace{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}_{} \leq n! \leq \underbrace{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}}_{\text{upper bond}}, n \geq 8$$

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$\boxed{f(n)} \log(n!) = \lg\left[(2\pi n)^{\frac{1}{2}} \cdot \left(\frac{n}{e}\right)^n\right] = \frac{1}{2}\log(2\pi n) + n\log\left(\frac{n}{e}\right)$$

$$= \frac{1}{2}\left[\underbrace{\lg 2}_{\text{const}} + \underbrace{\log\pi}_{\text{const}} + \underbrace{\lg n}_{\log}\right] + \underbrace{n\log n}_{\text{highest}} - \underbrace{n\log e}_{\text{~~const~~ } n(\text{const})}$$

$$\boxed{g(n)} = (\lg n)!$$

$$= \sqrt{2\pi\log n}\left(\frac{\log n}{e}\right)^{\lg n}$$

$$= \sqrt{2\pi}(\log n)^{\frac{1}{2}} \boxed{\frac{\log n^{\lg n}}{e^{\lg n}}}$$

highest order $\boxed{= n^{\lg\log n}}$ ~~$\log e$~~

$f(n) = O(g(n))$ $= \Theta(n\lg n)$ $= O(n^2)$

$f(n) \leq g(n)$ $O(n^{\log\log n})$

The Iterated Logarithm Function

$\log^* n$ ↳ and recursive (زي لعادة

عمليات بكل بنفذها لحد ما اوصل لل 1 او اقل

The notation $\lg^* n$ is used to represent the iterated logarithm, which is an extremely slow growing function defined in terms of $\lg^{(i)} n$, which is a function defined on non-negative integers that applies the logarithm function i times in succession.

$$\lg^{(i)} n = \begin{cases} n & \text{if } i = 0, \\ \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0 \\ \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0 \text{ or } \lg^{(i-1)} \text{ is undefined} \end{cases}$$

$\log 4 = 2$
$\log 2 = 1$
$\boxed{2}$

$\log 16 = \log 4$
$\log 2 = 1$
$\boxed{3}$

Then:

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

For example:

$\lg^* n$, $\lg^k(\lg n)$ الاولى

$\lg^* 2 = 1$
$\lg^* 4 = 2$
$\lg^* 16 = 3$
$\lg^* 65536 = 4$, etc.

$$2^{16} = \underbrace{16 = 4 = 2 = 1}_{④}$$

Rank Ordering Functions by Order or Growth

To rank order a list of functions into an arrangement $f_1, f_2, \dots f_n$ and $f_1 = \Omega(f_2), f_2 = \Omega(f_3), \dots f_{n-1} = \Omega(f_n)$ (as well as identify those functions that belong to the same equivalence class, where $f_1(n)$ and $f_2(n)$ belong in the same equivalence class if and only if (iff) $f_1(n) = \Theta(f_2(n))$), we can use what we know about the functions themselves and asymptotic notation. Much of the ranking is based on:

- Exponential functions grow faster than polynomial functions, which grow faster than polylogarithmic functions.
- The base of a logarithm does not matter asymptotically (recall $\log_b a = \frac{\log_c a}{\log_c b}$), but the base of an exponential and the degree of a polynomial do matter.
- Identities can help, as can working with approximation formulas such as Stirling's approximation.

Rank Ordering Functions by Order or Growth: Useful Identities

1. $2^{\lg n} = n^{\lg 2} = n$

$[2^{\lg n}]^{1/\lg n} = [n]^{1/\log n}$

$2 = n^{1/\log n}$

2. $4^{\lg n} = n^{\lg 4} = n^2$

3. $(\lg n)^{\lg n} = n^{\lg \lg n}$

4. $2 = n^{\frac{1}{\lg n}}$ (raising identity 1 to the power $\frac{1}{\lg n}$)

5. $(\sqrt{2})^{\lg n} = 2^{\frac{1}{2}\lg n} = 2^{\lg \sqrt{n}} = \sqrt{n}$

$n^{\log\sqrt{2}} = n^{\log 2^{1/2}} =$ [illegible]

* rank the following fun

$n^{1/\lg n}$ Θ, Ω, ...

1000, $n^{1/\lg n}$

$1000 = \Theta(n^{1/\lg n})$

$1000 = \Theta(2)$ [consta

* n^3, $n^{\lg \lg n}$

O

Rank Ordering Functions by Order or Growth: More Useful Identities

1. $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+\Theta(\frac{1}{n}))$

2. $n! = \Theta(n^{n+\frac{1}{2}}e^{-n})$ (drop constants and low order term)

3. $(\lg n)! = \Theta((\lg n)^{\lg n+\frac{1}{2}}e^{-\lg n}) = \Theta((\lg n)^{\lg n+\frac{1}{2}}n^{-\lg e})$ (substitute $\lg n$ for n in 2) $\hookrightarrow$ $O(n^{\lg\lg n})$

4. $\lg(n!) = \Theta(n\lg n)$

5. $n! = o(n^n)$ $n(n-1)(n-2)\times\cdots 1$
 $n\times n * n\cdots\times n$

6. $n! = \omega(2^n)$
 أكبر

CLR 3-3: Rank Ordering Functions by Order or Growth: Examples

top performance ↓ Ranking by asymptotic growth rate, equivalent classes are enclosed by '[ ]'.

Θ نفس ال

Const. $[1, n^{1/\lg n}]$ $\Theta(1)$
$\lg(\lg^* n)$
$[\lg^*(\lg n), \lg^*(n)]$ Θ
$2^{\lg^* n}$
$\ln\ln n$
$\sqrt{\lg n}$
$[\ln n$, $\log n]$
$\lg^2 n$
$2^{\sqrt{2\lg n}}$
$(\sqrt{2})^{\lg n}$
linear → $2^{\lg n}$
$[n\lg n, \lg(n!)]$ Θ
$[4^{\lg n}, n^2]$
n^3,
$[(\lg n)!, n^{\lg\lg n}, (\lg n)^{\lg n}]$ Θ

$\log n$ | $\lg n^2$
↓ ↓
$m < m^2$
$m = O(m^2)$

… $(3/2)^n$
2^n
$n2^n$
e^n
$n!$
$(n+1)!$ ← زيادة Term so $n! = O(n+1)!$
2^{2^n}
$\boxed{2^{2^{n+1}}}$
slowest performance

Base ال[illegible] exponential
أعلى ال growth ال

From slowest to the fastest.

exp faster poly

Rank Ordering Functions by Order or Growth: Examples

$n^{\log\log n}$

قريب من nlogn

Order the following functions:

1. $(\lg n)^{\lg n}$ and n^3

$n^{\log\log n} = \Omega(n^3)$

prove [def. / Lim

2. $n^{\frac{1}{\lg n}}$ and n

$2 = O(n)$

3. 2^n and $(\frac{3}{2})^n$

$2^n = \Omega(\frac{3}{2})^n$

4. $\lg^2 n$ and $\ln n$ Ω

$(\lg n)^2$ $\log n$

5. $4^{\lg n}$ and $8^{\lg n}$ O

n^2 n^3

6. $(\lg n)!$ and $\lg(n!)$

Typical Growth Rates

order

C	constant, we write O(1)
logN ماعدي bas.	logarithmic
$\log^2 N$	log-squared
N	linear
NlogN	
N^2	quadratic
N^3	cubic
2^N	exponential
N!	factorial $O(\frac{n^m}{\text{اعلاهن}})$ 2^N

Problems

- $N^2 =$ $O(N^2)$
  - true
- $2N =$ $O(N^2)$
  - true
- $N =$ $O(N^2)$
  - true

- $N^2 =$ $O(N)$
  - Ω false
- $2N =$ $O(N)$
  - true
- $N =$ $O(N)$
  - true

Problems

- $N^2 = \Theta(N^2)$
  - true
- $2N = \Theta(N^2)$
  - O false
- $N = \Theta(N^2)$
  - O false

- $N^2 = \Theta(N)$
  - Ω false
- $2N = \Theta(N)$
  - true
- $N = \Theta(N)$
  - true

Course Learning Outcomes

Analyze numerical computations algorithms (e.g. **matrix multiplication**).

stop when size of 2x2

(recursion divide and conquer | ضرب مرات

and the same size ← n^2 ال مقسّمة

لكل مصفوفة 8 sub

A_{11} A_{12} A_{21} A_{22} · B_{11} B_{12} B_{21} B_{22} = C_{11} C_{12} C_{21} C_{22}

$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$

$= \Theta(n^2)$

Matrix Multiplication
Brute Force Alg.

Design methodology

Counting Scalar Multiplications in Matrix Multiplication

```
MATRIX-MULTIPLY(A, B)
if columns[A] ≠ rows[B]
    then error "incompatible dimensions"
    else for i ← 1 to rows[A]
            do for j ← 1 to columns[B]
                  do C[i, j] ← 0
                     for k ← 1 to columns[A]
                        do C[i, j] = C[i, j] + A[i, k] · B[k, j]
    return C
```

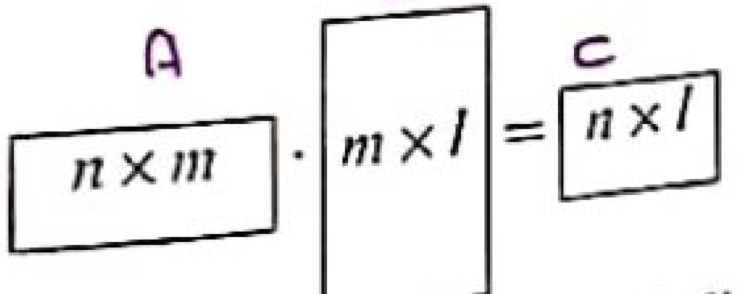
Basic operation

# of bit-wise Basic operation

For simplicity let n=m=l

we have same matric

$O(n^3)$



The number of scalar multiplications is: $n \times m \times l$

$(x-y)^2 = x^2 - 2xy + y^2$

$x \cdot x - 2xy + y \cdot y$ ③ mult

$(x-y)(x-y)$ ① mult.

اصغر حجم بقدر الحل

$A \times B = C$ 8 multiplication.
2x2 2x2 2x2

$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2 \times 2} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_{2 \times 2} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ 2x2

$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$

$c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot a_{22}$

$c_{21} = a_{21} \times b_{11} + a_{22} \cdot b_{21}$

$c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{21}$

8 multiplication

$2 \times 2 \times 2 = 8$

$O(n^3$

Matrix Multiplication Brute Force Alg.

- If ***m=l=n***
  - ***Then Time complexity is n^3***

? better بنقدر نعمل

اسم العالم

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right)_{2\times2} = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right)_{2\times2} * \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right)$$

$C_{00} = A_{00} * B_{00} + A_{01} * B_{10}$

$C_{10} = A_{10} * B_{00} + A_{11} * B_{10}$

$C_{01} = A_{01} * B_{01} + A_{11} * B_{11}$

$C_{11} = A_{01} * B_{01} + \cdots B_{11}$

2×2 (8) عمليات ضرب

$2\times2\times2 = 8$

تقلل عملية الضرب

7

Halsiey مرف

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

$$\left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right) = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) * \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right)$$

$$\begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

قلل عمليات الضرب

Time Complexity $= n^{2.807}$

$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$

$\boxed{\Theta(n^{\log_2^8}) = \Theta(n^3)}$

gusses for time complexity called

قلل ال recursive multiplication

Analysis of Strassen's Algorithm

- If *n* is not a power of 2, matrices can be padded with zeros.

  في حال ما كان نفس ال size ← zeros

- Number of multiplications:
  M(*n*) = 7M(*n*/2), M(1) = 1

- Solution: $\underline{M(n) = 7^{\log_2 n} = n^{\log_2 7}} \approx \overset{\text{perf افضل}}{n^{2.807}}$ vs. n^3 of brute-force alg. gusses

- Algorithms with better asymptotic efficiency are known but they are even more complex.

Course Learning Outcomes

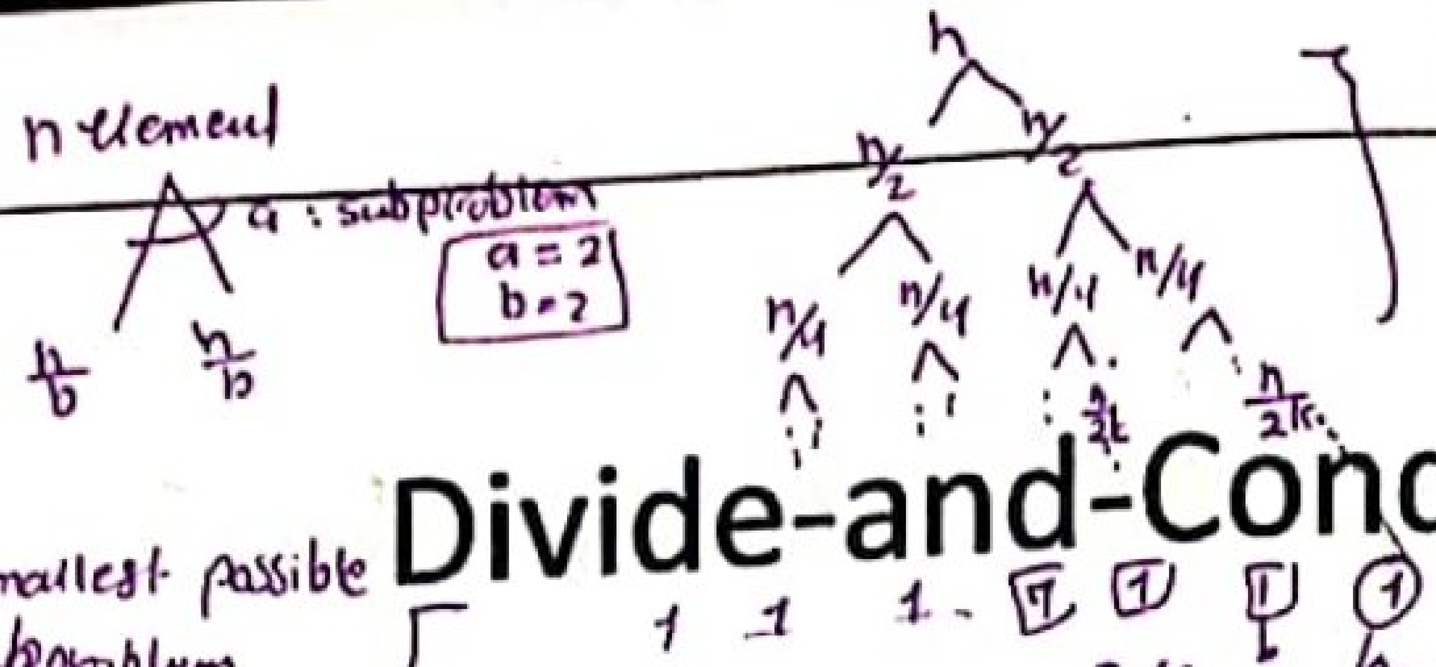
- **Use algorithm design methods**, such as exhaustive search, **divide-and-conquer** and dynamic programming, to develop efficient algorithms.

Designing Algorithms: Techniques/Strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

input = n element array

output array sorted



recursion tree

division is a recursion process

Divide-and-Conquer

In the context of sorting problem [smallest possible subproblem (leaves last level آخر

Base subproblem

لما يضل one element ← one element (sorted)

Base Case 1 element

$\frac{n}{2^k} = 1$

$n = 2^k$

$\boxed{k = \log n}$

# of division level

From root to leafs

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

	worst-case
Insertion Sort	$O(n^2)$
marge sort	$O(n\log n)$

[illegible]

$n \log n < n^2$

perf.
أفضل

up ↑ buttom

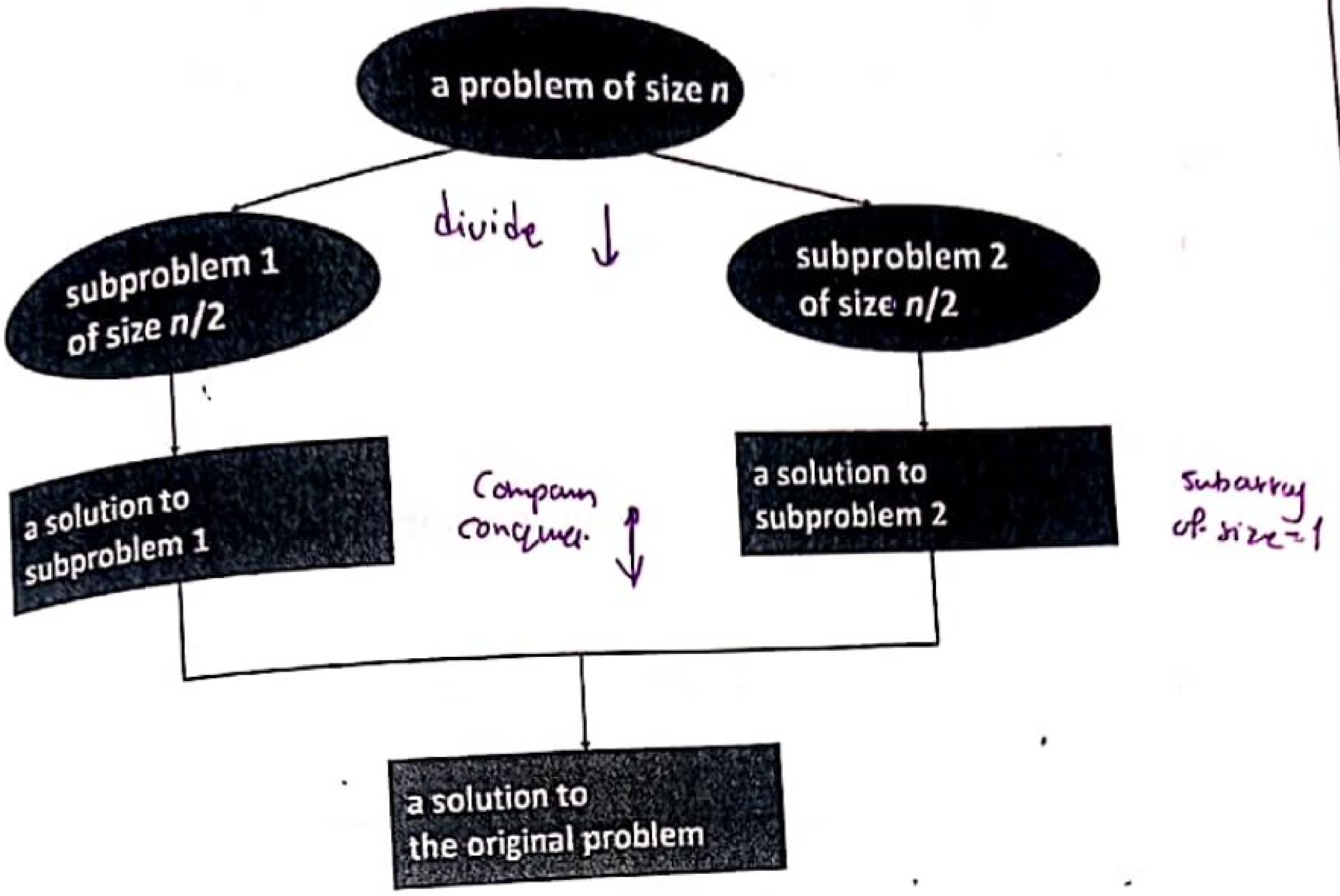
عاكس بنشتغل

top ↓ down

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms
- Binary search: decrease-by-half (or degenerate divide&conq.)

Divide-and-Conquer Technique (cont.)


a problem of size n
divide ↓
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
a solution to subproblem 1
Company conquer. ↕
a solution to subproblem 2
Subarray of size=1
a solution to the original problem

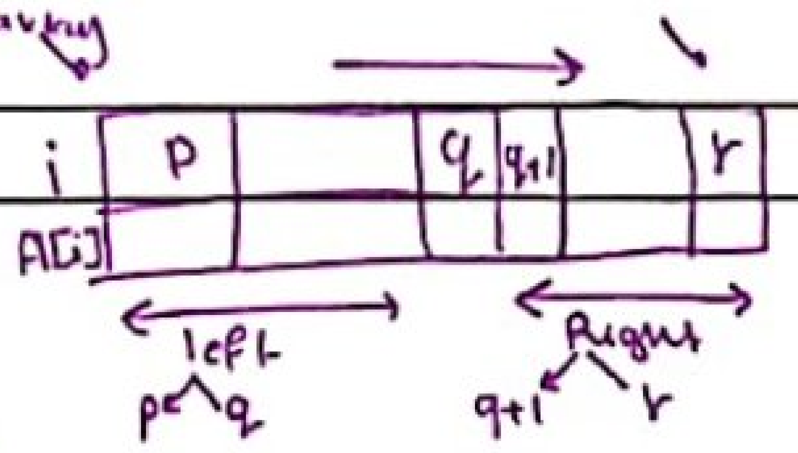
MERGE-SORT

- **MERGE-SORT** is an example of a divide-and-conquer algorithm.

Mergesort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays L and R في مرحلة Conquer
- Sort arrays L and R recursively
- Merge sorted arrays L and R into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

stop condition



can be any numbers.

p=1 , r=10

n = r-p+1 [may be even or odd

q = middle point.

$= \lfloor \frac{p+r}{2} \rfloor$ floor

~~input a array of n element~~

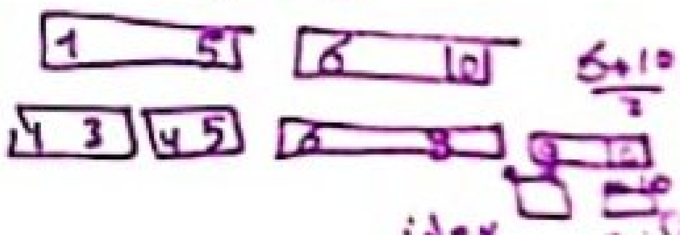
```
marge sort (A, p, r)
{ if p<r
  { q = ⌊(p+r)/2⌋
    merge sort (A, p, q), left
    merge sort (A, q+1, r), right
    merge (A, p, q, r)
  }
}
```

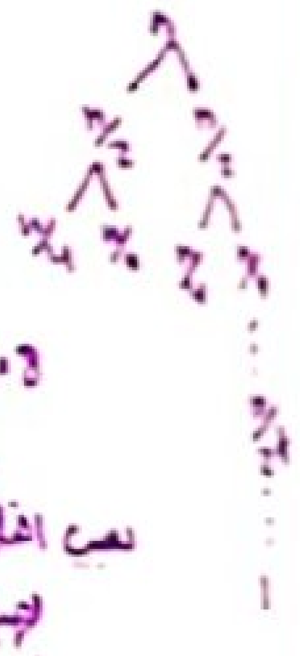
call

كده ترتيب

assume the right already sorted and left

MERGE-SORT $\lfloor \frac{1+10}{2} \rfloor = 5$





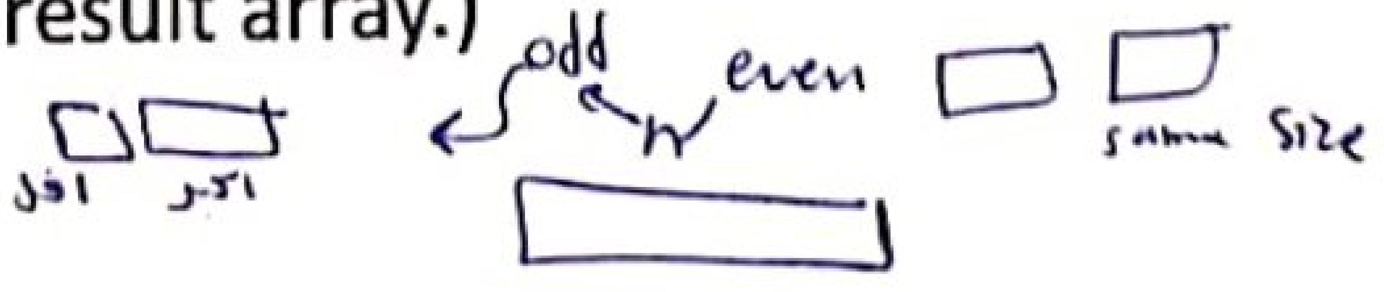
```
MERGE-SORT(A, p, r)
1. if p < r
2.     then q ← ⌊(p+r)/2⌋
3.         MERGE-SORT(A, p, q)
4.         MERGE-SORT(A, q + 1, r)
5.         MERGE(A, p, q, r)
```

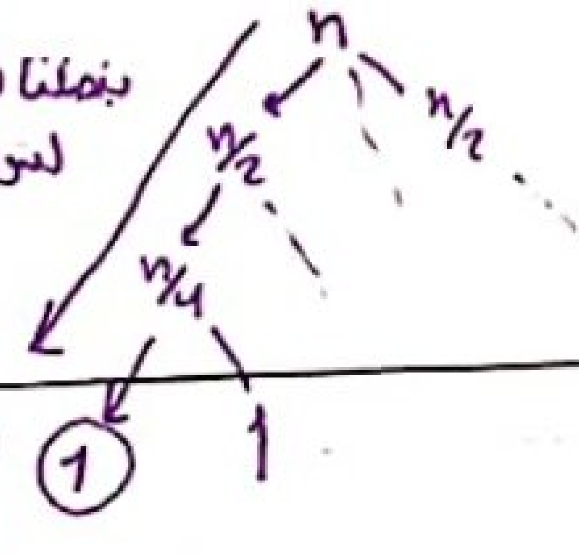
idex نص الفلان

first and last element بوصل بعضهم

P, q

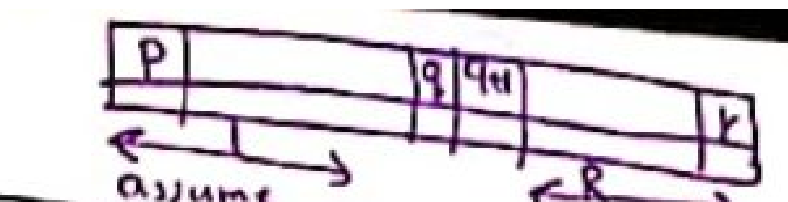
- **Divide:** Break the problem in half *D(n) = Θ(1).*
- **Conquer:** Recursively solve two problems of size *n/2* in 2*T(n/2*). (Keep dividing until length 1, which is sorted.)
- **Combine:** Merge two sorted arrays into one in *C(n) = Θ(n). (Keep moving* the smallest element of the two arrays into the result array.)





بنعملنا نفس الحركة لحد ما نوصل

لحد ما نوصل بعنصر واحد



assume already sorted

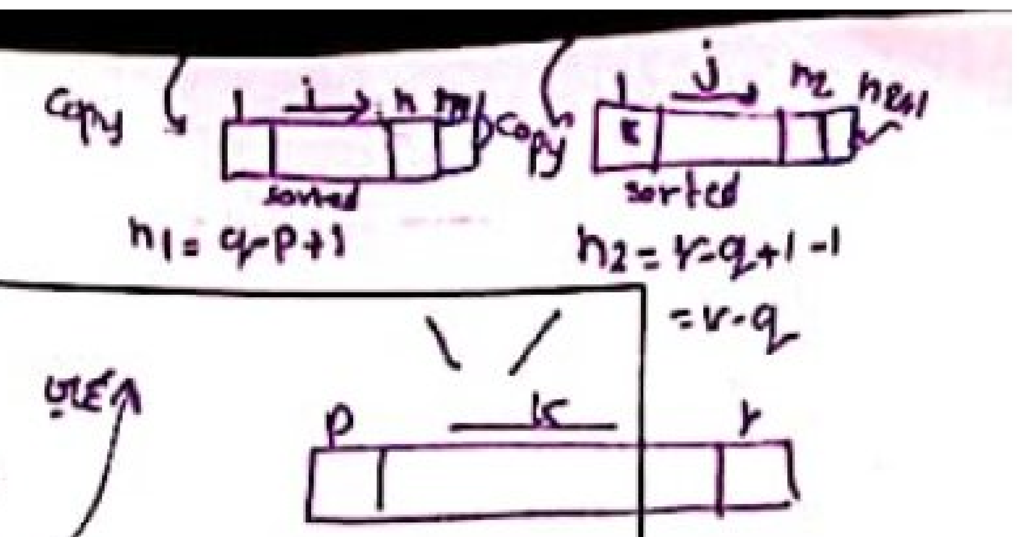
assume already sorted

Combing

defind auxiliary storage and name it L

defind auxiliary storage and name it R

copy ال original جو بنقل بنفس الـ array



$n_1 = q-p+1$

$n_2 = r-q+1-1$
$= r-q$

MERGE-SORT

```
MERGE(A, p, q, r)
    n_1 ← q − p + 1
    n_2 ← r − q
    create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1]
    for i ← 1 to n_1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n_2
        do R[j] ← A[q + j]
    L[n_1 + 1] ← ∞
    R[n_2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

```
merge(A,p,q,r)
{ i=1
  j=1
  n1=q-p+1,
  n2=r-q
  for i=1 to n1              // copy left
      L[i]=A[p+i-1]          // i=1, p+1-1→p
                             // i=n1, p+n1-1→p+(q-p+1)-1 = q
  for j=1 to n2              // copy Right
      R[j]=A[q+j]            // j=1→q+1
                             // j=n2→q+(n2)= q+r-q= r
  L[n1+1]=∞
  R[n2+1]=∞
  i=1
  j=1
  for k=p to r
      if L[i]≤R[j]
         A[k]= L[i]
         i =i+1
      else
         A[k]=R[j]
         j =j+1
}
```

n1

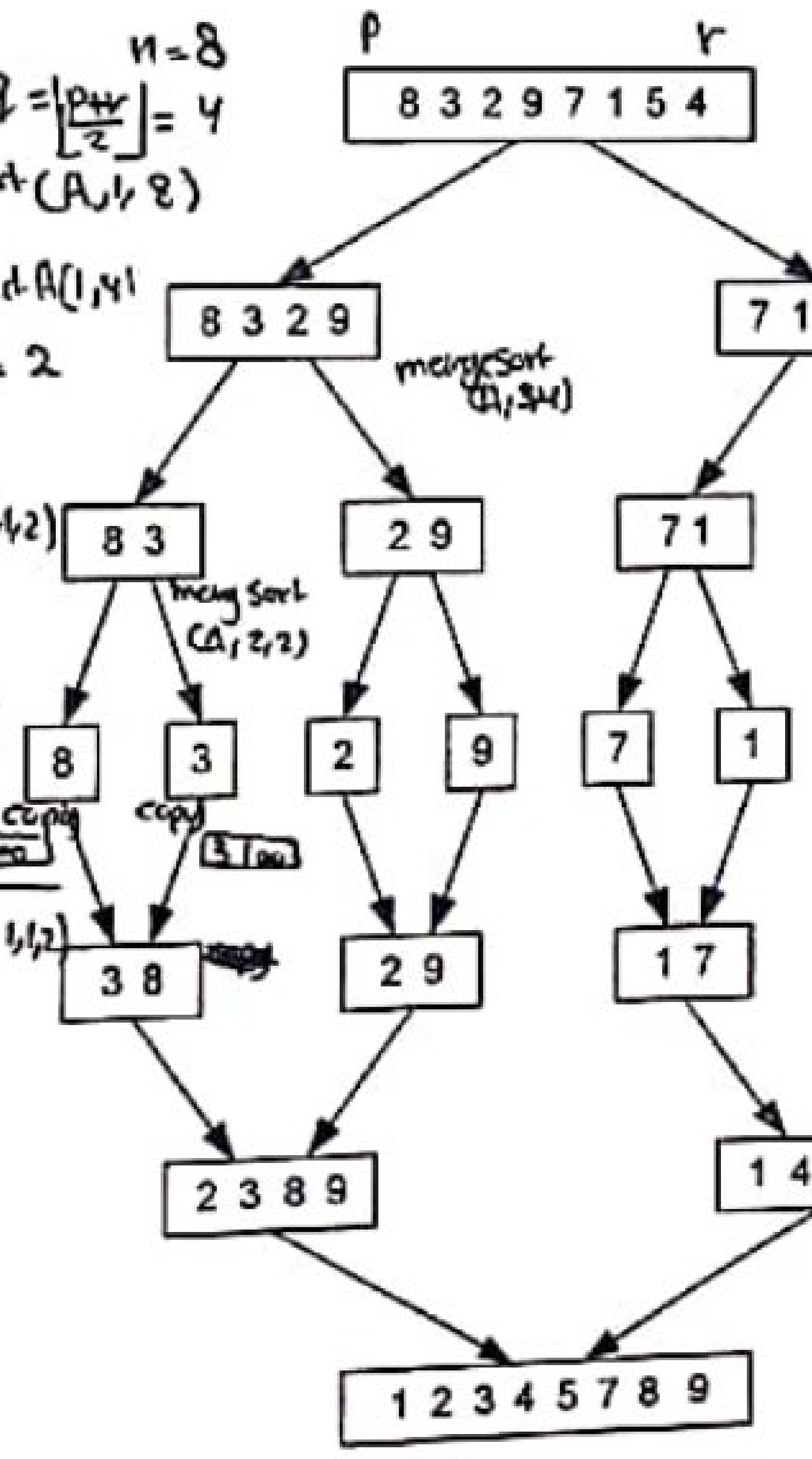
n2

يزيد one element

حتى نتوقف ال copy اذا كان الاول كلهم من ال first element بال R (او العكس)

بنرجع ال pointer للبداية

Mergesort Example 1



n=8

$q = \lfloor \frac{p+r}{2} \rfloor = 4$

mergSort (A,1,8)

بكل مره بغير ال q وال r

mergSort A(1,4)

(1<4) لذا

$q = \lfloor \frac{5}{2} \rfloor = 2$

mergSort (A,1,2)

(1<2) لذا

$q = \lfloor \frac{3}{2} \rfloor = 1$

mergSort (A,1,1)

1<1 (stop)

merg (A,1,1,2)

n1=1

n2=1

بقارن 8 ؟ 8<3

بنقل copy R

k=2

8<∞

بنقل copy L

i=2

$[n_3] = r-p+1$

Size of original subproblem when merge was called it it

- مستحيل نقارن ∞ مع ∞ لانه خلصنا k الخ

$\Theta([n])$

- how many level = logn

- total time complexity

= # of division level * work done at each level

= logn * Θ(n)

= $\underline{\Theta(n\lg n)}$

in all cases "

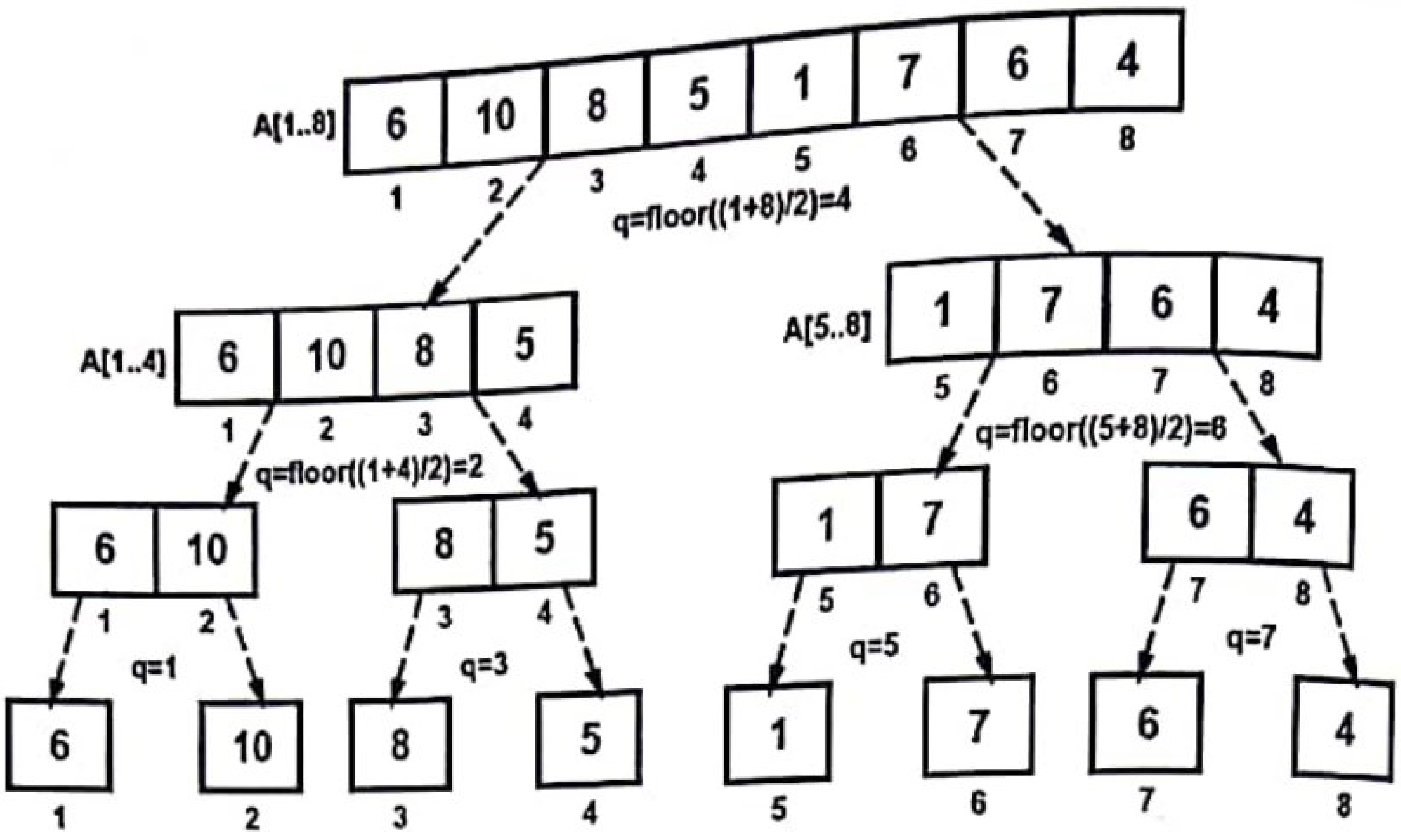
best and worst and avarge

		merge sort	insertion sort	

time complexit	best
	worst
	avr
		nlogn
		nlogn
		nlogn
			Θ(n)
			Θ(n²) Θ(n²)
				✓ insertion
				✓ merge

(extra storge other than need) space		Θ(n)	Θ(1)	✓ insertion

MERGE-SORT: Example 2: <6, 10, 8, 5, 1, 7, 6, 4>



A[1..8] 6 10 8 5 1 7 6 4

1 2 3 4 5 6 7 8

q=floor((1+8)/2)=4

A[1..4] 6 10 8 5

1 2 3 4

q=floor((1+4)/2)=2

A[5..8] 1 7 6 4

5 6 7 8

q=floor((5+8)/2)=6

6 10 | 1 2 | q=1

8 5 | 3 4 | q=3

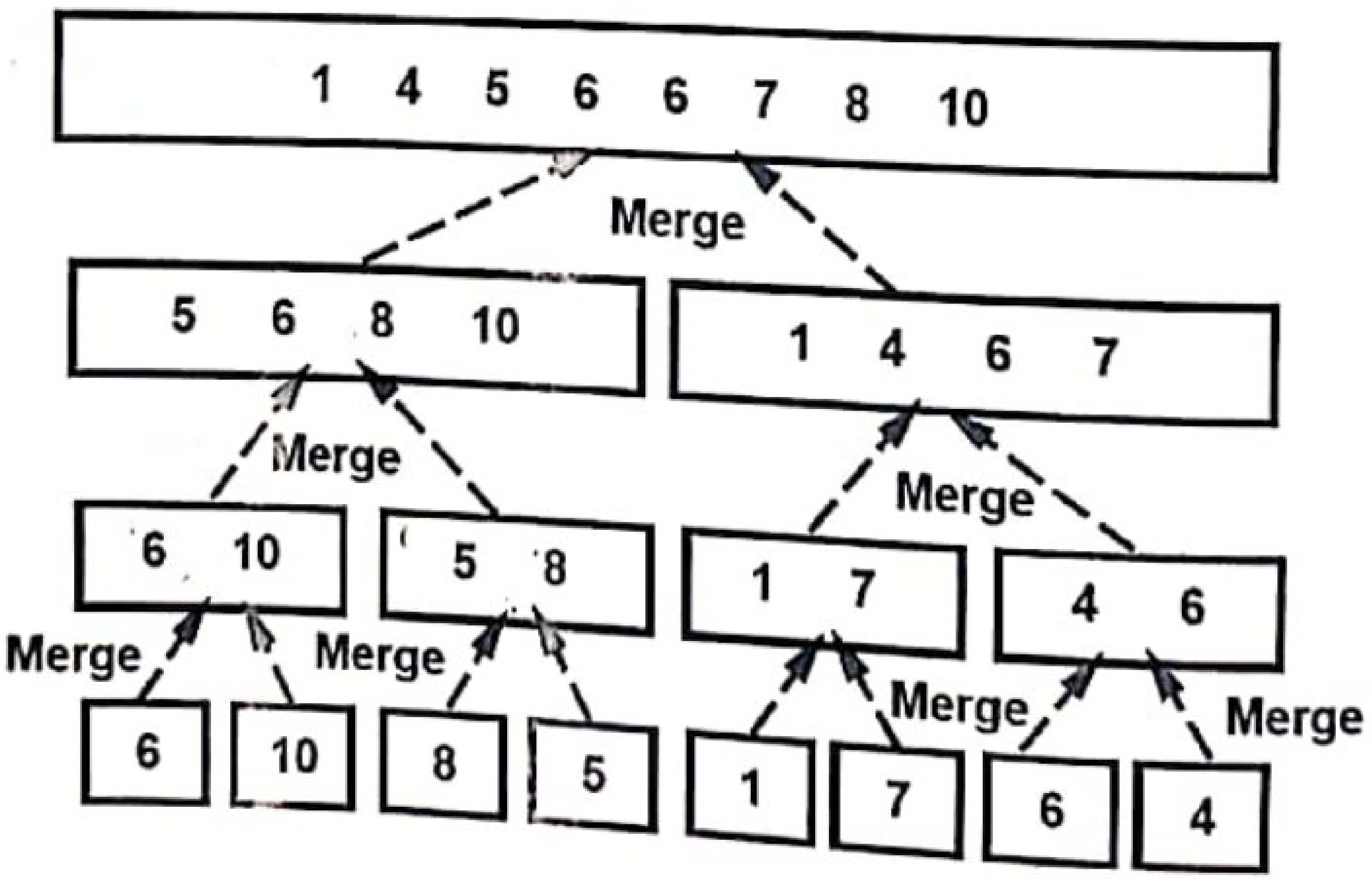
1 7 | 5 6 | q=5

6 4 | 7 8 | q=7

6 10 8 5 1 7 6 4

1 2 3 4 5 6 7 8

MERGE-SORT: Example2: <6, 10, 8, 5, 1, 7, 6, 4>



1 4 5 6 6 7 8 10

Merge

5 6 8 10

1 4 6 7

Merge

Merge

6 10

5 8

1 7

4 6

Merge

Merge

Merge

Merge

6 10 8 5 1 7 6 4

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Running Time of Divide and Conquer Algorithms

- We use recurrence equations to analyze the running time of a recursive algorithm.
- Let *T(n)* be the running time.
  - Divide into *a subproblems of size n/b → the running time of the subproblems* is *aT(n/b).*
  - *Division takes D(n) time.*
  - *Combining takes C(n) time.*

Base : Call

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ \underbrace{aT(\tfrac{n}{b})}_{\frac{n}{2}} + \underbrace{\overset{\text{divide}}{D(n)} + \overset{\text{Compayn}}{C(n)}}_{\Theta(n)} & \text{otherwise.} \end{cases}$$

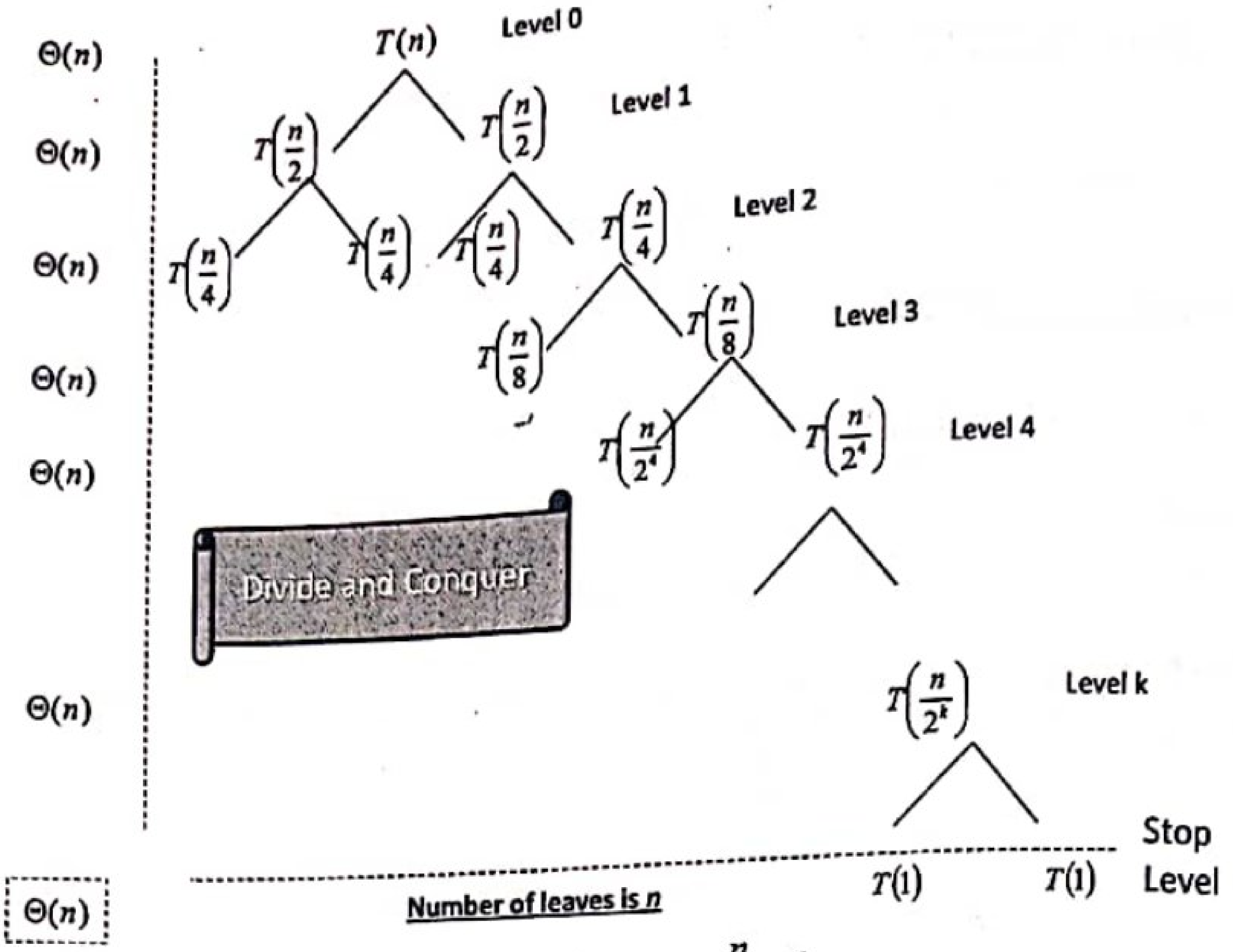
[illegible]

$$2T(\tfrac{n}{2}) = \underbrace{T(\lfloor \tfrac{n}{2} \rfloor)}_{\text{Floor}} + \underbrace{T(\lceil \tfrac{n}{2} \rceil)}_{\text{Celling}}$$

$\frac{11}{2} = 5.5$

$\lfloor 5.5 \rfloor = 5$

$\lceil 5.5 \rceil = 6$



Work Load = work done at each level * num of levels
$= \Theta(n\lg n)$

$\frac{n}{2^k}=1$

$\Rightarrow 2^k = n,$ $\therefore k = \lg n$ Number of levels

MERGE-SORT Analysis

For Merge-Sort, the running time is described by the following:

trivelal solution

$$T(n) = \begin{cases} \underline{\Theta(1)} & \text{if } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1. \end{cases}$$

We will show that T(n) is $\Theta(nlgn)$, and so it is asymptotically faster than INSERTION-SORT.

$\underbrace{\frac{1}{3}n}_{\text{subproblem}} \cdot \underbrace{\frac{2}{3}n}_{\text{subproblem}}$

$T(n) = T(\frac{n}{3}) + T(\frac{n}{3/2}) + \Theta(n)$

How to Generate a tight Band Cugs For Certain Recursion equation

① Substution method
② ↳ provide a Cugs and prove it use a mathmatical induction
① ↳ try to manipiulate the currenl recurtion equation to look Similer to well knoum recursion equation for which are already know their Time Complexity Function in order of growth notation

Lecture 4: Recurrences and Master Theorem

$T(n) = 2T(n/2) + n$ $\Theta(n\lg n)$

↳ $T(m) = 2T(m/2) + m$ $\Theta(m\log m)$

↳ $S(m) = 2S(m/2) + m$

where both $T(r)$ and $S(m)$ are time Compl. function

let $m = \log n$

$\boxed{n = 2^m}$

$T(2^m) = 2T((2^m)^{1/2}) + m$

$T(2^m) = 2T(2^{m/2}) + m$ let $S(m) = T(2^m)$

Ex: we know that recursion equation for merge sort $\boxed{T(n) = 2T(n/2) + \Theta(n)}$, $T(1)=1$, has order of growth notation $\Theta(n\log n)$

Know let, see how we can write, provide a tight bound Guess for the following recursion equation

$\boxed{T(n) = 2T(\sqrt{n}) + \lg n}$

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

$S(m) = 2S(m/2) + m$

so. = Θ $m\log m$, now we can substut back the original value of m

$\boxed{\log n \log\log n}$

Read Chapter 4 of *Introduction to Algorithms*

Recurrences

A recurrence is an equation or inequality that describes a function in terms of its value(s) on smaller inputs. For example, the following recurrence describes the worst-case running time of MERGE-SORT:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1. \end{cases}$$

The solution to this equation is $\Theta(n\lg n)$. An important question is how can we find such a closed-form expression (exact or asymptotic) for recurrences?

We will discuss 3 methods:

- **Substitution method:** Guess the bound and prove by $\overset{\text{Mathmatical}}{\uparrow}$ induction.
- **Iteration method:** Treat it as summation. construct a recursion tree
- **Master method:** A "cookbook" method for solving $\boxed{T(n) = aT(\frac{n}{b}) + f(n)},$ $a \geq 1, b > 1$, and $f(n)$ is a given function.

بنقدر نحل هدول الانواع

recursion tree [iteratively unroll the recursion equ. untell we see a pattern and we solve it ↳ provide a Cugs

1

Recurrences *continued*

In practice, we often neglect certain details when we state and solve recurrences. Sometimes we gloss over the assumption that the functions take integer arguments. For example, the MERGE-SORT worst-case running time is really:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \underbrace{\Theta(n)}_{c_1 n \le f(n) \le c_2 n} & n > 1. \end{cases}$$

(n بنعتبره)

We also typically don't state the boundary conditions. For sufficiently small n, generally $T(n) = \Theta(1)$. Hence, we often report the recurrence simply as:

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n)$$

We omit floors, ceilings, and boundary conditions when they don't matter. We also often solve for powers of 2 rather than all n (and in that case, we really haven't shown the bound for all n).

2

The Substitution Method

guess for time complexity

Example: Solve the recurrence: $T(n) = 4T(\frac{n}{2}) + n$, $\boxed{T(1) = 1.}$ Base case

فالحل ما اجى Base case بعتبره واحد

Guess to Verify by Induction: $\underbrace{T(n) = O(n^3)}$, i.e., $\underbrace{T(n) \le cn^3}_{\text{لازم تكون صح}}$.

اذا الفرض صح

① **Base Case:** $T(1) = 1 \le c1^3 = c$, if $\underbrace{c \ge 1}_{T(1) \le 1 n^3}$.

بحسب ال const. على الفرض تطبق ال Base case

② inductive hypothesis

Assume: $T(k) \le ck^3$, for $k \le n_{+1}$ $\boxed{\text{let } c = 1}$

اذا ما لازم بنفترض ↳ انا عندي مساواة بدل ما يساوي $k = n+1$, $k = n$

③ **Inductive Step:**

show that $T(k) \le ck^3$ when $k = n$

$$\begin{aligned} T(n) &= 4T(\tfrac{n}{2}) + n \\ &\le 4c(\tfrac{n}{2})^3 + n \\ &= \tfrac{c}{2}n^3 + n \\ &= \boxed{cn^3 - \tfrac{c}{2}n^3 + n} \\ &= cn^3 - (\tfrac{c}{2}n^3 - n) \\ &\le cn^3, \text{ if } c \ge 2, n \ge 1 \end{aligned}$$

$\frac{c}{2}k^3 = \frac{2}{2}ck^3 - \frac{c}{2}k^3$

$= ck^3 \boxed{- \frac{c}{2}k^3 + k}$

ع الاكبر اصغر كان مضروب فيها دلعب ب $O(n^2)$ بجرب

$\leftarrow T(k)$ بنحطها من ال Recursion الاصلية

$T(k) = \boxed{4T(\frac{k}{2})} + k$

use the inductive hypothesise $\boxed{\text{use } C=1}$

$\le 4c(\frac{k}{2})^3 + k$

$\le 4c\frac{k^3}{8} + k$

$\le \frac{ck^3}{2} + k \overset{?}{\le} ck^3$?

مع الاكبر اصغر لانه مضروب مضروب صح

The Substitution Method *continued*

Example on Renaming Variable

How to make a good guess:

- Similarity to known function, e.g., $T(n) = 2T(n/2) + n$
- Start with a loose upper and lower bound and try to narrow it down.

There are times when you can guess a bound but the math doesn't work out in the induction. Sometimes the inductive assumption isn't strong enough to prove a detailed bound; try subtracting a lower-order term. We will next show an example where this technique will help.

The Substitution Method *continued*

$O(n^3)$ is **not** a tight bound for $T(n) = 4T(\frac{n}{2}) + n$; we can show that $T(n) = O(n^2)$.

Check previous lecture (lecture 3).

$T(n) = O(n^2)$, $T(n) \leq cn^2$

$T(k) \leq ck^2$, $1 \leq c(1)^2 \rightarrow \boxed{c=1}$

$T(n) = 4\ T(n/2) + n$

$T(k) = 4\ T(k/2) + n$

$T(k) = 4\ c\frac{k^2}{4} + n$

$ck^2 = \frac{2c}{2}k^2 - \frac{c}{2}k^2$

$T(k) = ck^2 + n$

$ck^2 + n \leq ck^2$

$O(n^2)$

The Substitution Method*continued*

Sometimes a little algebraic manipulation can make an unknown recurrence into one we have seen before. For example,

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n$$

Renaming $n = 2^m$ (so $m = \lg n$) yields:

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m$$

Now we can rename $S(m) = T(2^m)$:

$$S(m) = 2S(\frac{m}{2}) + m$$

We know $S(m) = O(m\lg m)$. Changing back from $S(m)$ to $T(n)$ yields:

$$T(n) = T(2^m) = S(m) = O(m\lg m) = O(\lg n \lg\lg n)$$

$T(n) = 2T(n^{1/2}) + \lg n$

let $\lg n = m$,

نعوض $T(\frac{m}{2}) = S(m)$

وذلك لنتخلص من الجذر

The Iteration Method

Recursion tree

unroll the recursion equation several time untill we see a pattren (mathmatical seres)

The idea is to expand the recurrence and express the recurrence as a summation dependent only on n and the initial conditions. Then techniques for evaluating summations can be used to provide a bound on the solution. For example, recall the recurrence equation: $T(n) = 4T(\frac{n}{2}) + n$. Let us expand $T(n)$ a few times looking for patterns:

$T(1) = 1$ Base case

Iteration method

$$\begin{aligned}
T(n) &= n + 4T(\tfrac{n}{2}) \\
&= n + 4(\tfrac{n}{2} + 4T(\tfrac{n}{4})) \\
&= n + 4(\tfrac{n}{2} + 4(\tfrac{n}{4} + 4T(\tfrac{n}{8}))) = n + 4\tfrac{n}{2} + 4^2\tfrac{n}{4} + 4^3T(\tfrac{n}{8}) \\
&\vdots \\
&= n + 4\tfrac{n}{2} + 4^2\tfrac{n}{4} + 4^3\tfrac{n}{8} + \ldots + 4^kT(\tfrac{n}{2^k}) \\
&\vdots \triangleright \text{ Iterate until } \tfrac{n}{2^k} = 1 \text{ or } \lg n \text{ times.} \\
&= n + 4\tfrac{n}{2} + 4^2\tfrac{n}{4} + 4^3\tfrac{n}{8} + \ldots + 4^{\lg n}T(1), \text{ where } 4^{\lg n} = n^2 \\
&= \sum_{i=0}^{\lg n - 1} 4^i\tfrac{n}{2^i} + \Theta(n^2) \\
&= n(\sum_{i=0}^{\lg n - 1} 2^i) + \Theta(n^2) = n(\tfrac{2^{\lg n} - 1}{2 - 1}) + \Theta(n^2) \\
&= n(n-1) + \Theta(n^2) = \Theta(n^2) + \Theta(n^2) = \Theta(n^2)
\end{aligned}$$

$T(n) = 4T(\frac{n}{2}) + n$

$= n + 4\boxed{T(\frac{n}{2})}$

$= n + 4\boxed{4T(\frac{n}{2^2}) + \frac{n}{2}}$

$= n + \frac{4}{2}n + 4^2\boxed{T(\frac{n}{2^2})}$

$= n + \frac{4}{2}n + 4^2\left[4T(\frac{n}{2^3}) + \frac{n}{2^2}\right]$

$= n + \frac{4}{2}n + \frac{4^2}{2^2}n + 4^3T(\frac{n}{2^3})$

$\vdots$

$(\frac{4}{2})^0 n + (\frac{4}{2})^1 n + (\frac{4}{2})^2 n + (\frac{4}{2})^3 n + \ldots 4^k T(\frac{n}{2^k})$

stop when $\frac{n}{2^k} = 1 \rightarrow k = \lg n$

$T(n) = \left[\sum_{k=0}^{\log n - 1} (\frac{4}{2})^k n\right]^{2^k} + 4^{\lg n}T(1) \rightarrow$

$T(n) = n\sum_{k=0}^{\lg n - 1} 2^k + n^2$

$n\left\{\frac{2^{(\lg n - 1) + 1} - 1}{2 - 1}\right\} + n^2$

$= n[2^{\lg n} - 1] + n^2$

$= n(n-1) + n^2$

$= n^2 - n + n^2$

$= 2n^2 - n$

$= \Theta(n^2)$

الفرق مع ب

The Iteration Method *continued*

Let's obtain an exact solution for: $T(n)=2T(\frac{n}{2})+2$. Assume that n is a power of 2 and $\boxed{T(2)=1}$.

بدي مقارنة هدي

Min / MAX

$T(2)=1$

$$
\begin{aligned}
T(n) &= 2+2T(\tfrac{n}{2})\\
&= 2+2(2+2T(\tfrac{n}{4})) = 2+4+4T(\tfrac{n}{4})\\
&= 2+4+4(2+2T(\tfrac{n}{8})) = 2+4+8+8T(\tfrac{n}{8})\\
&\vdots\\
&= 2+4+8+16+\ldots+2^k+2^kT(\tfrac{n}{2^k})
\end{aligned}
$$

$2^1+2^2+2^3+2^3T(\frac{n}{2^3})$

$2^1+2^2+2^3+\ldots 2^k+2^k T(\frac{n}{2^k})$

$\vdots \rhd T(2)$ is the base case, so solve $\frac{n}{2^k}=2$.

$\frac{n}{2^k}=2$

$\boxed{\frac{n}{2^k}=2}$

$n=2^{k+1}$

$\boxed{2^k \cdot 2 = n}$

$k+1=\log n$

$2^{k+1}=n$

$k+1=\log n$

$\boxed{k=\log n-1}$

$\boxed{k=\log n-1 \quad T(2)=1}$

substution in $2^kT(\frac{n}{2^k})$

$$
\begin{aligned}
&\overset{k}{=} \sum_{i=1}^{\lg n-1} 2^i + \underbrace{2^{\lg n-1}T(\tfrac{n}{2^{\lg n-1}})}\\
&= \boxed{\sum_{i=0}^{\lg n-1} 2^i-1} + \tfrac{n}{2}T(\tfrac{n}{\frac{n}{2}})\\
&= \tfrac{2^{\lg n}-1}{2-1}-1+\boxed{\tfrac{n}{2}T(2)} \to \tfrac{n}{2}\\
&= n-2+\tfrac{n}{2} = \boxed{\tfrac{3}{2}n-2} \text{ since } T(2)=1.
\end{aligned}
$$

ادرزها بحسب لحال

لازم يبلش من 0

بلش من 0 وخلص 1

$\Theta(n)$

$\sum_{k=1}^{\log n-1} 2^k$

$2^{\log n-1} = 2^{\log n} \cdot 2^{-1} = \frac{2^{\log n}}{2} = \boxed{\frac{n}{2}}$

$= \sum_{k=0}^{\log n-1} 2^k - 2^0 \to \frac{2^{\log n}-1}{1} - \cancel{\log n}+1$

$(n-1) - \log n \to \boxed{n-2}$

$n-2+\frac{n}{2} \to \frac{3}{2}n-2$

$\Theta(n)$

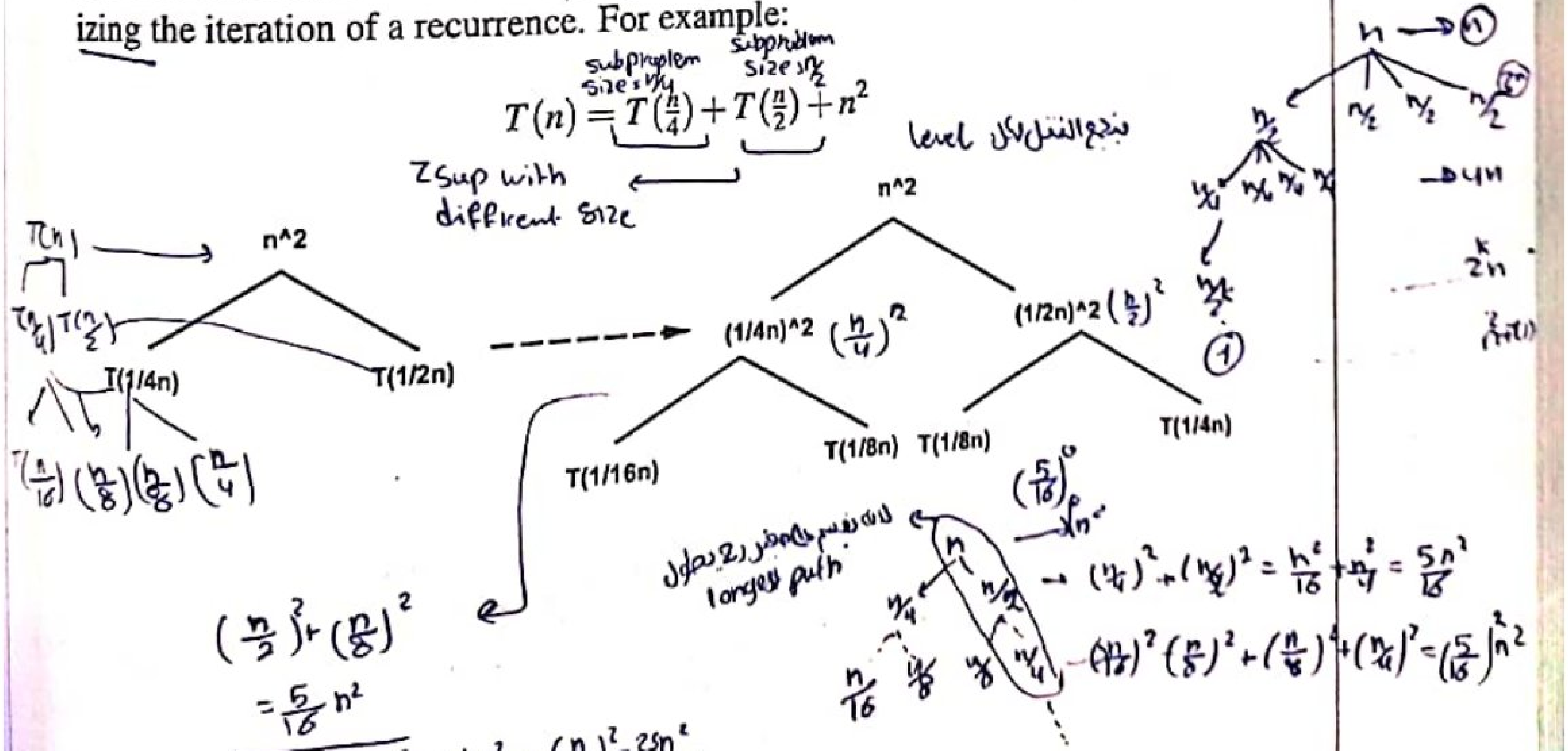
Recursion Trees

For certain forms of recursions a graphical representation is convenient for visualizing the iteration of a recurrence. For example:

$$T(n) = \underbrace{\overset{\text{subproblem size}=n/4}{T(\tfrac{n}{4})}} + \underbrace{\overset{\text{subproblem size}=n/2}{T(\tfrac{n}{2})}} + n^2$$

2sup with diffrent size

نجمع الكل لكل level



$\to (\frac{n}{4})^2 + (\frac{n}{2})^2 = \frac{n^2}{16} + \frac{n^2}{4} = \frac{5n^2}{16}$

$\to (\frac{n}{16})^2 (\frac{n}{8})^2 + (\frac{n}{8})^2 + (\frac{n}{4})^2 = (\frac{5}{16})^2 n^2$

$(\frac{n}{2})^2 + (\frac{n}{8})^2$

$= \frac{5}{16}n^2$

$(\frac{n}{16})^2 + (\frac{n}{8})^2 + (\frac{n}{8})^2 + (\frac{n}{4})^2 = \frac{25n^2}{256}$

$T(n) \leq \sum_{k=0}^{\log n} (\frac{5}{16})^k n^2 \to \leq \sum_{k=0}^{\infty} (\frac{5}{16})^k \cdot n^2$

$= [\frac{1}{1-\frac{5}{16}}] n^2 = \frac{16}{11} n^2 = \Theta(n^2)$

اذا نقسم ع ٢ اكبر ح انوقف اسرع

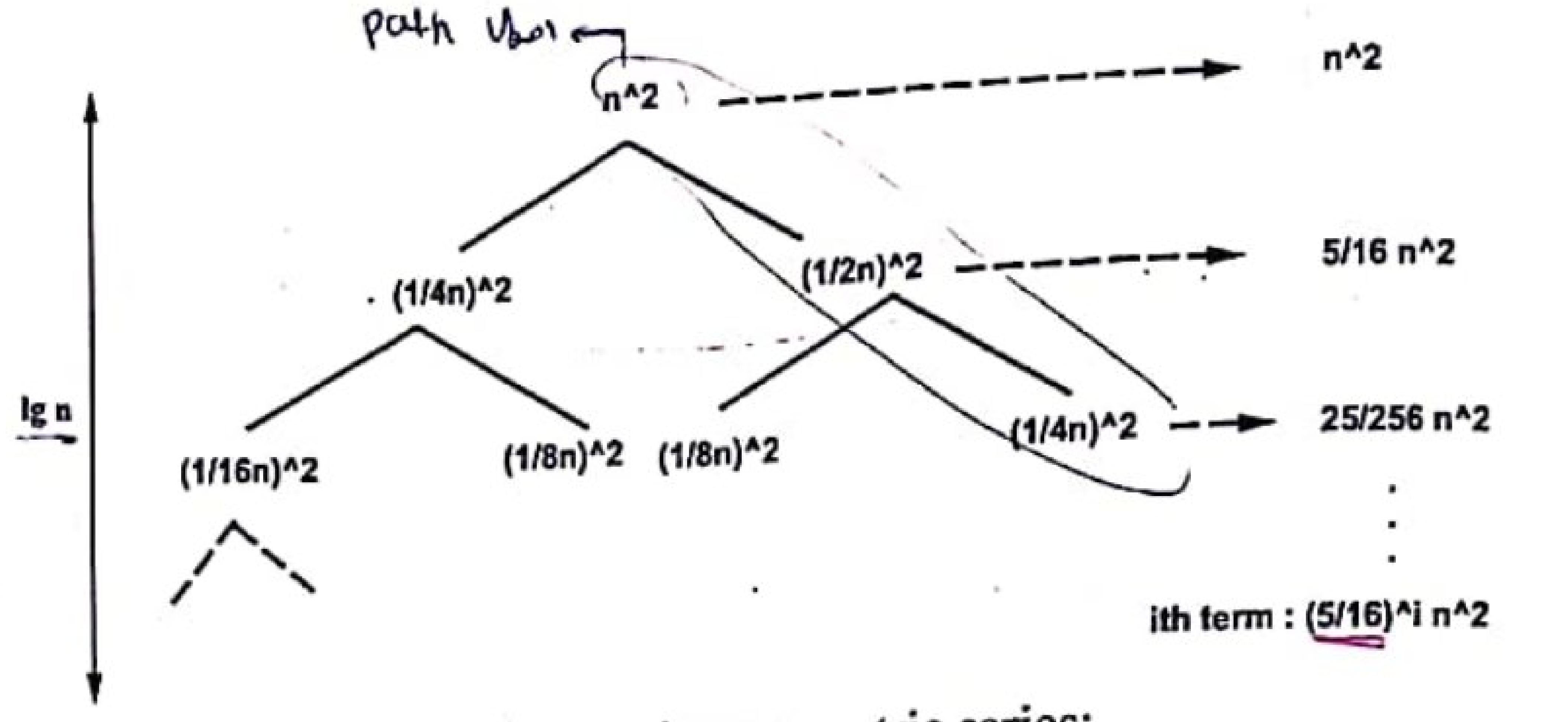
$T(1) = 1$ اذا نقسم ع ٢ نفس ال١٢ عمل

$\frac{n}{2^k} = 1 \rightarrow \boxed{k = \log n}$

Recursion trees *continued*

$T(n) = 1$

Recursion Tree for T(n) = T(n/4) + T(n/2) + n^2



ith term : (5/16)^i n^2

Summing over levels gives a decreasing geometric series:

ما قبل الاخير → لانه احنا رايحين ١

$$\sum_{i=0}^{\lg n-1}\left(\tfrac{5}{16}\right)^i n^2 \le n^2 \underbrace{\sum_{i=0}^{\overset{\text{decres series}}{\infty}}\left(\tfrac{5}{16}\right)^i}_{} = n^2\frac{1}{1-\frac{5}{16}} = O(n^2)$$

$$\frac{1}{1-x} = \left(\frac{1}{1-\frac{5}{16}}\right)n^2$$

root work

a^0 $\boxed{k=0}$ level zero 0 $\rightarrow$ n $\rightarrow$ $f(n) = a^0 f(\frac{n}{b^0})$ انحل لبرها

a^1 $\boxed{k=1}$ level 1 $\rightarrow$ $\frac{n}{b}$ $\frac{n}{b}$ $\rightarrow$ $f(\frac{n}{b}) + f(\frac{n}{b}) = a f(\frac{n}{b})$

a^2 level $k=2$ $\rightarrow$ n/b^2 n/b^2 — $a^2 f(n/b^2)$

level k

The Master Method

"Cookbook method" for solving recurrences of the type:

monotonic increas function

$$T(n) = aT\left(\frac{n}{b}\right) + \boxed{f(n),}$$

بنزل لما يكون نفس ال size الكل

اخر level

$\boxed{a^{\log_b n} * \Theta(1)}$

$T(1) = \Theta(1)$

$k = \log_b n$

$= \Theta(n^{\log_b a})$ leaves work

where $a \ge 1, b > 1$, and f is asymptotically positive.

- Divide the problem of size n into a subproblems of size $\frac{n}{b}$.
- Solve them recursively.
- Cost to divide the original problem and recombine the results is $f(n)$, which is asymptotically positive.

قبل الاخير

$$T(n) = \underbrace{\sum_{0}^{(\log_b n)-1} a^k f\left(\frac{n}{b^k}\right)}_{\text{let this be B}} + \underbrace{\Theta(n^{\log_b a})}_{A}$$

• 3cases

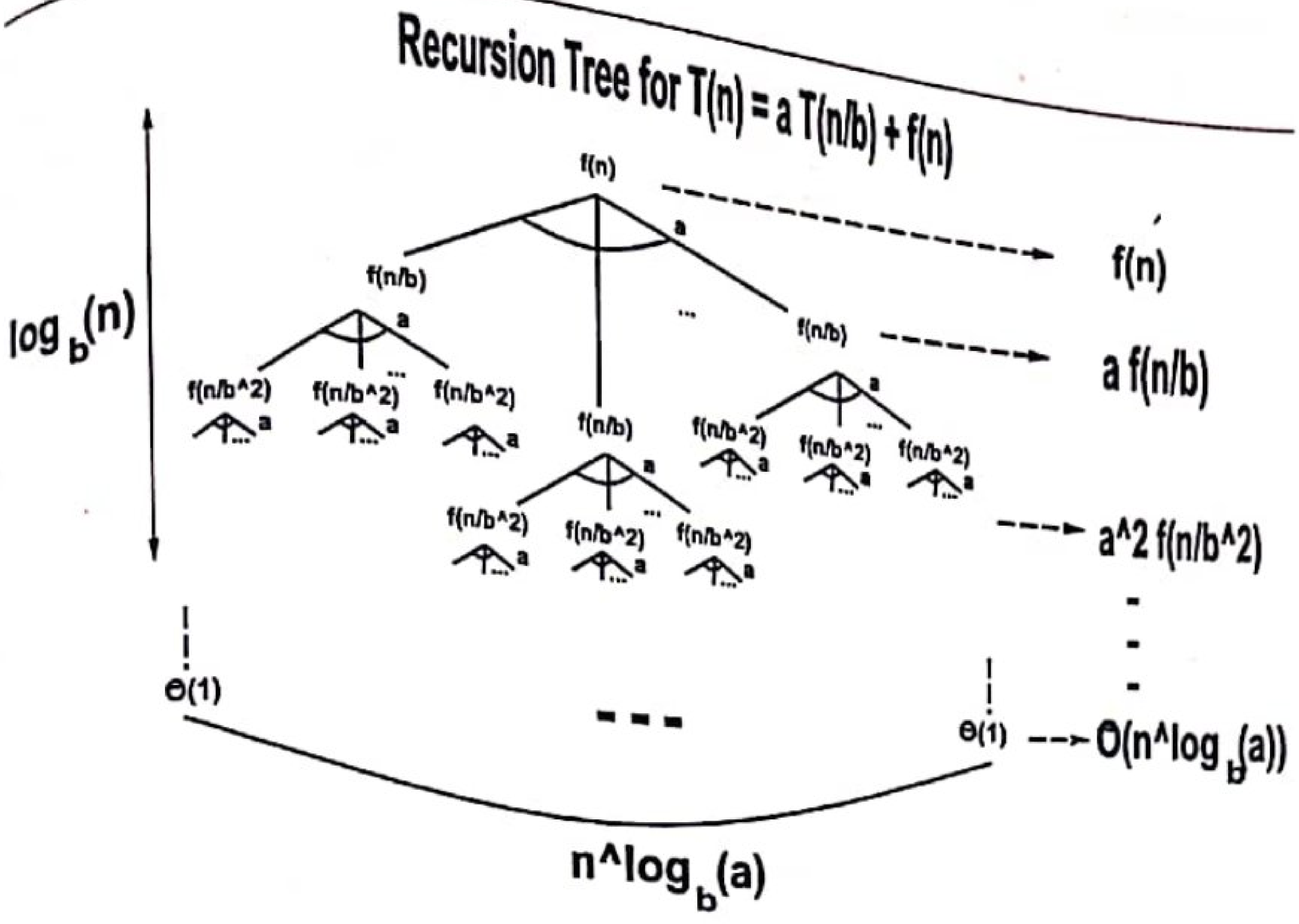
dominant of the root work $f(n)$

$A > B \rightarrow T(A)$ Case 1

$B > A \rightarrow T(B)$ Case 3

$B = A \rightarrow$ Case 2 up to log

The Master Method *continued*



The Master Theorem

Theorem 4.1 The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ be defined on the nonnegative integers by the recurrence:

$$T(n) = aT(\tfrac{n}{b}) + f(n), T(1) = \Theta(1)$$

where we interpret $\frac{n}{b}$ to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

poly فرق

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $\boxed{T(n) = \Theta(n^{\log_b a}).}$ Or alternatively, $\frac{n^{\log_b a}}{f(n)} = \Omega(n^{\varepsilon}), \varepsilon > 0$ (equivalently, $\frac{f(n)}{n^{\log_b a}} = O(n^{-\varepsilon}), \varepsilon > 0$). In other words, $n^{\log_b a}$ is polynomially larger than $f(n)$. The running time is dominated by the costs in the leaves.

$n^{\log_b a}$ (1). leaf work

The Master Theorem *continued*

مش لو مضروب ن

2. If $f(n) = \Theta(n^{\log_b a} \lg^k n)$, for $k \ge 0$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ (as in Exercise 4.4-2). Or alternatively, $\boxed{\frac{f(n)}{n^{\log_b a}}} = \Theta(\lg^k n)$ for some constant $k \ge 0$. In other words, $f(n)$ and $n^{\log_b a}$ are within a polylogarithmic factor. The cost is evenly distributed across levels.

3. If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if $a f(\frac{n}{b}) \le c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. Or alternatively, $\frac{f(n)}{n^{\log_b a}} = \Omega(n^{\varepsilon}), \varepsilon > 0$ and $\boxed{a f(\frac{n}{b}) \le c f(n)}$ for some constant $c < 1$ and all sufficiently large n. In other words, $f(n)$ is polynomially larger than $n^{\log_b a}$. Hence, the time is dominated by the cost of the root.

The Master Theorem *continued*

The following equation represents the running time for the class of problems for which the Master Method applies:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(\tfrac{n}{b^j})$$

Note that the number of small problems to solve (leaves in the tree) is $a^{\log_b n} = n^{\log_b a}$.

$$\begin{aligned} a^{\log_b n} &= (b^{\log_b a})^{\log_b n} \\ &= b^{\log_b a \log_b n} \\ &= b^{\log_b n \log_b a} \\ &= (b^{\log_b n})^{\log_b a} \\ &= n^{\log_b a} \end{aligned}$$

Applying the Master Method

Example 1: $T(n) = 9T(\frac{n}{3}) + n$

lower work

Test: $a = 9, b = 3, f(n) = n$ so $n^{\log_b a} = n^{\log_3 9} = \boxed{n^2.}$ $\cdot\epsilon$

Because $f(n) = n = O(n^{2-\varepsilon})$ for $\boxed{0 < \varepsilon \leq 1}$ (or $\frac{n^2}{n} = \Omega(n^{\varepsilon})$ for $\varepsilon > 0$), $\boxed{\text{case 1 applies}}$ and so $\boxed{T(n) = \Theta(n^2)}$. Confirm by induction: $T(n) \leq cn^2$.

Base Case: Assuming $T(1) = \Theta(1) \leq c1^2 = c$, if $c > 1$.

Assume: $T(k) = ck^2 - k$ for $k < n$

Inductive Step:

$$\begin{aligned}
T(n) &= 9T(\tfrac{n}{3}) + n \leq 9(c(\tfrac{n}{3})^2 - \tfrac{n}{3}) + n \\
&= cn^2 - 3n + n = cn^2 - 2n \\
&\leq cn^2 - n
\end{aligned}$$

Applying the Master Method *continued*

Example 2: $T(n) = 2T(\frac{n}{2}) \boxed{+2}$

Test: $a = 2, b = 2, \boxed{f(n) = 2}$ so $n^{\log_b a} = n^{\log_2 2} = \boxed{n.}$

Because $f(n) = \Theta(1) = O(n^{1-\varepsilon})$ for $0 < \varepsilon \leq 1$ (or $\frac{n}{2} = \Omega(n^{\varepsilon}), 0 < \varepsilon \leq 1$), case 1 applies and so $\boxed{T(n) = \Theta(n).}$

$T(n) \leq O(n)$

Example 3: $T(n) = 2T(\frac{n}{2}) + n \to n\log^2 \sim n\log^3$

Test: $a = 2, b = 2, f(n) = \boxed{n}$ so $n^{\log_b a} = n^{\log_2 2} = \boxed{n.}$

Because $f(n) = \boxed{\Theta(n^{\log_b a} \lg^0 n)}$ (or $\frac{n}{n} = \Theta(1) = \Theta(\lg^0 n)$), case 2 applies; hence, $T(n) = \Theta(n\lg n)$.

$+1$

$n^{\log 2} \log^{1} n \to \boxed{n\log n}$

Applying the Master Method *continued*

Example 4: $T(n) = 4T(\frac{n}{2}) + n^3$

Test: $a = 4, b = 2, f(n) = \boxed{n^3}$ so $n^{\log_b a} = n^{\log_2 4} = \boxed{n^2.}$

Because $f(n) = n^3 = \Omega(n^{\log_b a+\varepsilon}) = \Omega(n^{2+\varepsilon})$ (or $\frac{n^3}{n^{\log_2 4}} = n = \Omega(n^{\varepsilon})$) for $0 < \varepsilon \leq 1$, and $4f(\frac{n}{2}) \leq cf(n)$ for $c < 1$ (say $c = \frac{1}{2}$); hence case 3 applies and $T(n) = \Theta(n^3)$.

$a f(\frac{n}{b}) \leq c f(n)$

$4 f(\frac{n}{2}) \leq c f(n) \longrightarrow$ $4(\frac{n}{2})^3 \leq cn^3$

$\frac{4}{8} n^3 \leq cn^3 \rightarrow$ $\boxed{\frac{1}{2} \leq c}$ $\boxed{c = \frac{1}{2}}$ cf ✓

Example 5: $T(n) = 4T(\frac{n}{2}) + \frac{n^2}{\lg n}$

Test: $a = 4, b = 2, f(n) = \boxed{\frac{n^2}{\lg n}}$ so $n^{\log_b a} = n^{\log_2 4} \neq \boxed{n^2.}$

Because $\frac{\frac{n^2}{\lg n}}{n^2} = \frac{1}{\lg n}$, which is not $O(n^{-\varepsilon}), \varepsilon > 0$ (i.e., $\lg n \neq \Omega(n^{\varepsilon})$), $\Theta(\lg^k n), k \geq 0$, or $\Omega(n^{\varepsilon}), \varepsilon > 0$, no case applies (must use another method).

Recall the sorting Algorith we seen:

(1) selection sort.

(2) insertion sort

(3) Herg sort

name of algorith and type of data structure (similar to queu data but with special probaurity and charactrisic and operation)

Lecture 5: Heap Sort

dynamic allocated memory at run time

هون بنحكي عن شيء آخر

space complexity بتحسن ال

merge sort لل

	Algorith design	time complexity Best	Agv.	worst	space complexity	stability
(1)	Brut force	n^3	n^3	n^3	1	✓
(2)	decreasc and conqut	n	n^2	n^2	1	✓
(3)	divide and conque	nlogn	nlogn	nlogn	n	✓

Dr. Khalil Yousef

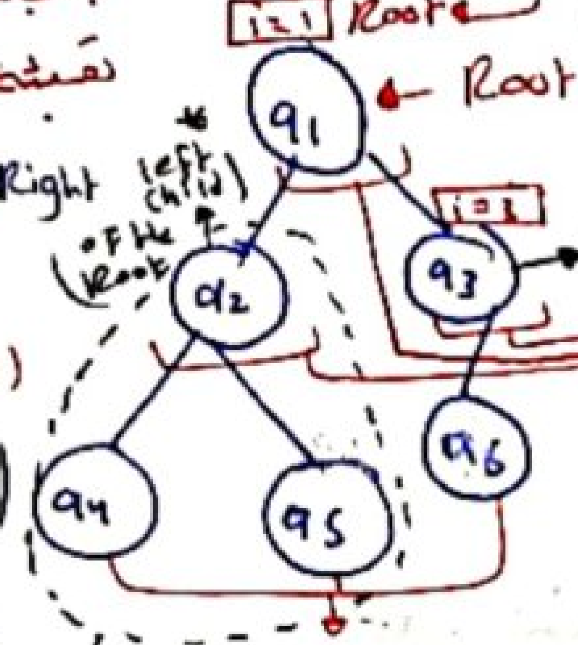
Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

Read Chapter 6 of *Introduction to Algorithms*

* لبيانات الـRoot وبنزل بالاتجاه (leaves)

نقطة الـelement ① Top ② left → Right ③ Down

A = {a1, a2, a3, a4, a5, a6, ...}



(two branches لا يوجد غير) (binary) لأننا (صحيح العدد element array)

(its a data struc.)

Lecture 5: Heap Sort

* تطبيق على كل node ⇒ إذا كان لديك index كيف أطلع تفاصيل الـRoot ?!

① Parent (i) = ⌊i/2⌋ (floor)

② left (i) = 2i

③ Right (i) = 2i+1

④ length (A) = n → # of element

أنا أطلع الـRight والـleft لأي node وإذا ما تجاوزت (n) بيكون الـ(n) أكبر not exist

Ex) 6: الـRight الها = 2i+1 = 13 أكبر من (6) مالها Right ; الـleft الها = 2i = 12 أكبر من (6) مالها left ⇒ الـ(6) ما الها child

⑤ indeces of the internal nodes = {1, ..., ⌊n/2⌋} (floor) (بيرتقي)

⑥ indeces of the External nodes = {⌊n/2⌋+1, ..., n} (floor) (بعد خلوص الـintern.)

* اذا كان node (2) branches / (1) branch → child (internal node); zero branch → External node OR leaves

الـleaves + External node مالهم child (يعني مالهم الـleft والـRight) (لا يشترط فيها حسب عدد الـelement من تكون الـlength)

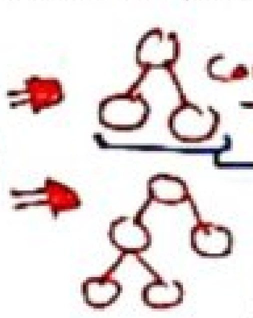
الـ(Branch / link) عالم وصل آخر level

⑦ # of node in a complet biny Tree = $\sum_{i=0}^{\text{depth of the tree}} 2^i = \sum_{i=0}^{\text{max hight of the Tree } h} 2^i = 2^{h+1}-1$

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

* Completness (internal nodes لازم كل الـ) (Complet Binary Tree) إذا تحقق (two child) يكون الهم بالـlevel لا يوجد one level



→ $2^{1+1}-1 = 3$ ☑

→ $(2^{2+1}-1)$ ☒

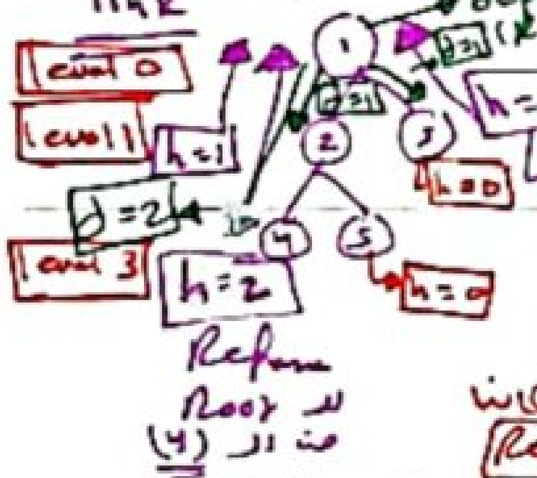
⑧ # of node in a biny complet the external node ⇒ $\sum_{i=0}^{h-1} 2^i = 2^h - 1$

(الـdepth علاقة عكسية مع Hight)

Read Chapter 6 of *Introduction to Algorithms*

* اي Tree يمكن المرسم (depth , Hight)

→ تقييس من GND level ويطلع طلوع
→ العكس من GND وينزل لتحت



(الـHight أو الـdepth max Hight of the Tree) → (Reface البعد وتحتسب)

(GND مستوى الـ Root الـمركز)

الـ(node الـhight is refered) لعينه الـsubtree بتنظر اليها .

* اذا بديا نتوقع عدد الـnode الـ(h = 0) ما بعد مكانا (Reface 0)

(كل الـinternal node اله two child) → (Completed Tree)

* لو ما كان يسمى (nearly complet biny Tree) ← يكون الـinternal node بالـlevel قبل الأخير الـlevel missing a child at least

(كل الـinternal عليه شرط one child)

The Sorting Problem

max hight of this = 2
max depth of this = 2 → The Tree not node ⚠ (في فرق بين الـTree أو الـnode high او deep)

The Sorting Problem (non-decreasing): (لما يحكي (0)(1) من الـlevel (0) للـlevel (1) بتكرار depth)

Input: a sequence of n numbers $< a_1, a_2, \ldots, a_n >$

Output: a permutation of the input sequence $< a'_1, a'_2, \ldots, a'_n >$ such that $a'_1 \leq a'_2 \leq \ldots \leq a_n$

(وحدة max + children node)

○ node

⑨ Heap size = length (A) ((decr. order) تساويا بالبداية فقط بعدين تصير heap size = ا اتوقف)

* Heap Data instr. its Complet (evry internal node have two child) OR nearly Complet biny Tree ⇒ heap Proparity)

Sorting algorithms:

كل element الـarray موصلين بـ (Biny Tree Complet OR nearly) + كل الـparent value تأكد أكبر من أو تساوي الـchildren → (parent(value) ≥ children)

Insertion Sort: $\Theta(n^2)$ worst case :

Merge Sort: $\Theta(n \lg n)$, (مشكلته space complexity أكبر)

a1 ≥ a2 / a3 , a3 ≥ a6
a2 ≥ a4 / a5

* from a Root to a leave (Path) مرتبين تصاعدي ; كل path مرتب (ترتيب تصاعدي) أقل أبو الـRoot

Same , but

Heap Sort: $\Theta(n \lg n)$ (large constant factor) , (بدون space) (space مشكلته bad !! التعويض فيه) ليس (max heap) الـRoot (max) ; أما الـ(min heap) الترتيب تنازلي لا كان الـRoot أكبر أو تساوي الـchild (Parent) Root ≤ child

Compair Time (complexity) (لكن الـconst من Heap أكثر merge)

Quick Sort: It has a worst case running time of $\Theta(n^2)$, but an average case running time of $\Theta(n \lg n)$ (small constant factor). (Best + Avg)

Linear sort algorithms will also be discussed which are not comparison sorts.

* يعني نستخدمه انه دائما الـ(Root) أصغر max heap ... Sorting أول اشي نشيل الـRoot (elements أكبر) عدد الـelement ح نقوم بتغيير (1) Array element بعدها ما نتبقى (n-1) ارجع اعمل (heap) ثانيا اشيل الـRoot صار مكانه ... دجحه مكانه (any leave) أقصى الـ(Key) نبحث أكبر اشي ونرجع ونقارن مع ... child لو بلقى أكبر العملية

⑨ (Heap size = length (A)) بالبداية بس

Some Sorting Terminology

A list or array of elements that is to be sorted often consists of elements that are records. Each record is sorted with respect to its **key**, which is some ordinal type. The record will also typically contain **satellite data**. The book does not focus on the satellite data.

→ (basic operation comparison)

Comparison Sort: Uses comparisons between elements in the input array (or list) of elements to produce a sorted output.

In-place Sort: Uses only a constant amount of storage in addition to the input data structure. (insertion)

→ heap ~~heap~~ (not stable)

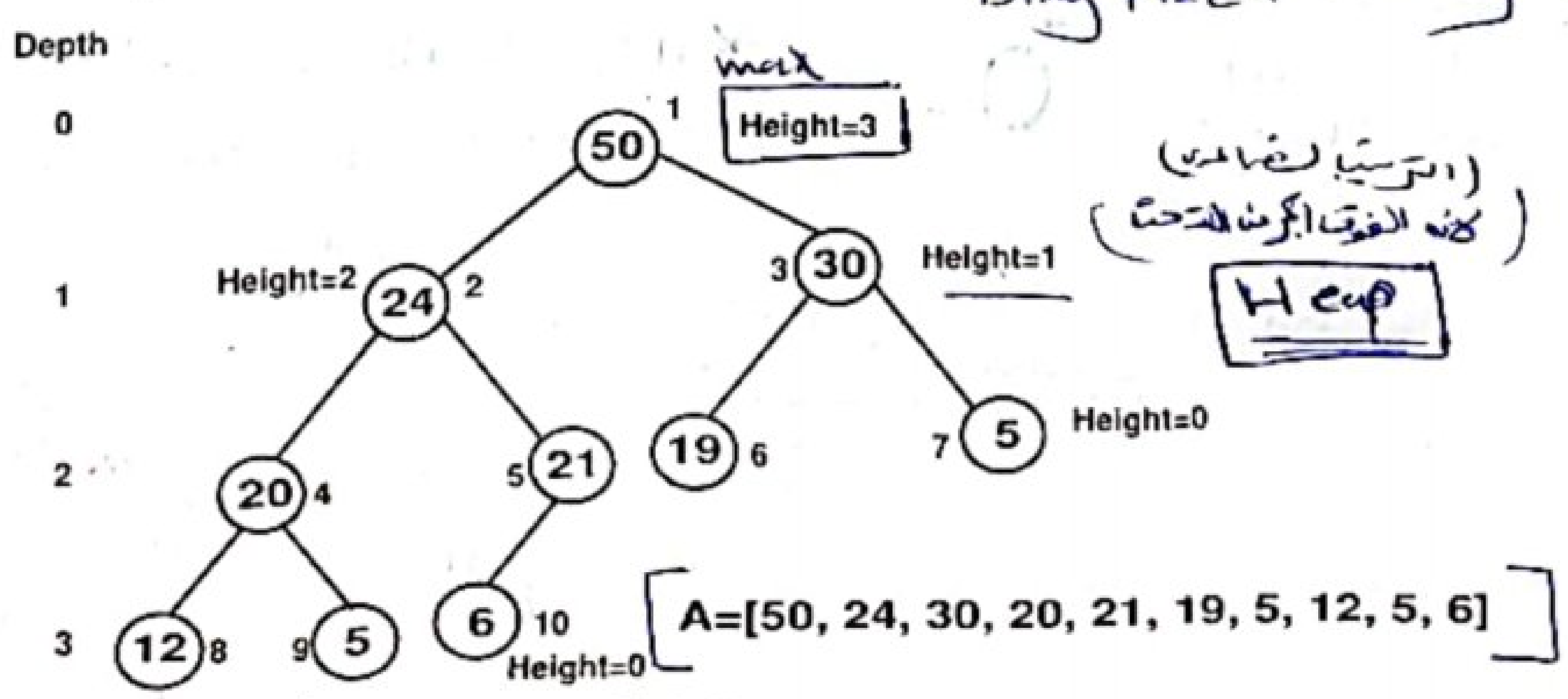
Stable Sort: Maintains the same relative ordering of elements with the same key in the sorted output as in the input.

The Heap Data Structure

We first define heaps and the various operations on them.

Definition: A (binary) **heap** is a nearly complete binary tree that satisfies the heap property. For example,

Biny tree + nearly Complet



(الترتيب تنازلي)
(لأنه الفوق أكبر من اللي تحت)

Heap

A=[50, 24, 30, 20, 21, 19, 5, 12, 5, 6]

تذكر الترتيب ال arry

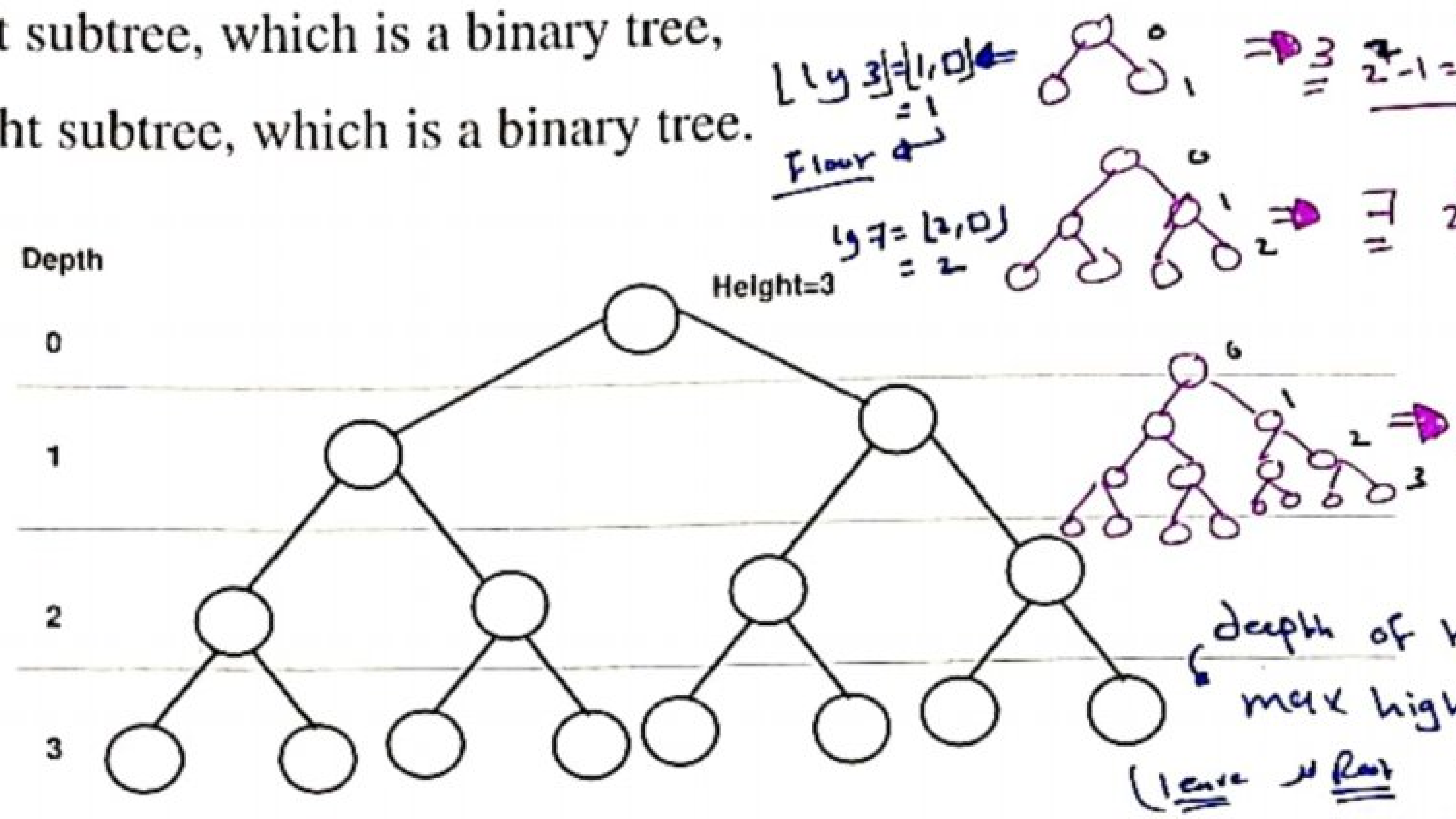
(nearly Complet)

Definition: The **heap property** is the key of the parent must be greater than or equal to the key of the child, i.e., $A[\text{PARENT}(i)] \geq A[i]$. $A[1]$ is the root of the heap.

Binary Tree Terminology

A **binary tree** T either contains no nodes or is comprised of three disjoint sets:

- the root node,
- the left subtree, which is a binary tree,
- the right subtree, which is a binary tree.



$\lfloor \lg 1 \rfloor = 0$
$\lfloor \lg 3 \rfloor = \lfloor 1.0 \rfloor = 1$
Floor
$\lg 7 = \lfloor 2.0 \rfloor = 2$
Depth
0
1
2
3
Height=3
0
1
2
3

$\Rightarrow \underline{1}$ $\quad 2^1 - 1 = 1$

$\Rightarrow \underline{3}$ $\quad 2^2 - 1 = 3$

$\Rightarrow \underline{7}$ $\quad 2^3 - 1 = 7$

$\Rightarrow \underline{15}$ $\quad 2^4 - 1 = 15$

Capket $\Rightarrow \boxed{2^{\#(\text{high})} - 1}$

depth of the tree = $\lfloor \lg n \rfloor$

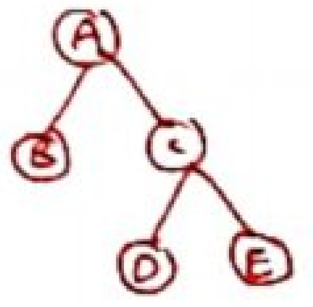
max hight of the tree

(leave ال Root ن path ايه)

(leave ال Root) ن deep

(Root ال leave) ن hight

EX)



$\Rightarrow 2^3 - 1 = 7$

A	B	C	-	-	D	E
1	2	3	4	5	6	7

[illegible] $\lfloor \frac{1}{2} \rfloor$ = 0 [illegible]

تطلع الاجابة تصح $\lfloor \frac{6}{2} \rfloor = 3$ هاد 3 الوالد ☑

$\lfloor \frac{4}{2} \rfloor = 2$ هاد (2) مش الوالد

Binary Tree Terminology *continued*

The number of children that a node x can have in a tree T is called its **degree**. The nodes in a binary tree can have at most a degree of 2. (عدد child لكل node يعرف degree) max ال في biny = 2 ات

The length of the longest path from root r to a node x in T defines the **depth** of x. Note that a root has depth 0. The **height** of a tree T is the largest depth of any node in T. The height of a node in a tree is the length of the longest simple path to a leaf node.

A **complete** binary tree contains only nodes that are leaf nodes or have a degree of 2, and all leaf nodes have the same depth. The number of nodes in a complete binary tree of height h is $\sum_{i=0}^{h} 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$.

Nodes that are leaf nodes are often called **external nodes**. Nodes that have at least one child are called **internal nodes**. The number of internal nodes in a complete binary tree of height h is $\sum_{i=0}^{h-1} 2^i = \frac{2^{h}-1}{2-1} = 2^h - 1$.

The Heap Data Structure *continued*

Note that we number the nodes in the heap from top to bottom, left to right. Then we can store the heap in an array, A, where the number associated with a node is its array index. The array A has two attributes:

1. *length*$[A]$ indicates the number of elements in the array; this stays constant throughout HEAPSORT.

2. *heap-size*$[A]$ indicates the number of elements in the heap. *heap-size*$[A] \leq$ *length*$[A]$. This value will decrease during the iterations in HEAPSORT.

We define three access functions:

1. PARENT(i) **return** $\lfloor i/2 \rfloor$

2. LEFT(i) **return** $2i$

3. RIGHT(i) **return** $2i+1$

The Heap Data Structure *continued*

Properties of a heap:

1. The root of a heap is the largest element.

2. Any path from leaf to root has values in ascending order.

3. The assignment of keys to a node in a heap structure is not unique, though the structure (i.e., shape) of the heap with a certain number of elements is unique.

4. All leaves are at depth $d-1$ or d, where d is the maximum depth.

5. All non-leaf vertices, except for possibly one, have two children. If a non-leaf vertex has only one child (which must be a left child), it is the right-most vertex at that depth with children.

Heap Operations

There are several heap operations that we will discuss:

- HEAPIFY: $\Theta(\lg n)$ ⇒ (heap of ...) (عند اتقف) ← لتضمن انه path acending order
- BUILD-HEAP: $\Theta(n)$
- HEAPSORT: $\Theta(n \lg n)$
- HEAP-EXTRACT-MAX: $\Theta(\lg n)$
- HEAP-INSERT: $\Theta(\lg n)$

اذا أعطاني 3 elem parent • (2) child • لعبد مقارنة ، بفرض انه ال largest هو ال Root (Parent) بقارن ال Parent مع (left child) يلي اكبر ... largest بعدين بقارن ال largest الكبير مع ال Right child اذا ال largest ... بس swap ... ال largest ... parent ... swap ... parent ال largest ... عشان انا كمان ال parent الجراش ب ليان انادي نفس ال Algo. مع ال child يلي بدلته ... لل parent (child) بلزم

وبنبادي ... heap property ... internal node

A heap with n elements has height $\Theta(\lg n)$; hence, many of the basic operations on a heap run in time proportional to $\Theta(\lg n)$. من ال Root ... لل leaves

#min heap properity ⇒ $Key_{parent} < Key_{child}$

نبادي من internal node ... leave من External node

↑ $\lfloor \frac{n}{2} \rfloor \dots 1$ (Bottom up ⇒ (لضمان ان كل ال tree صارت heap) (التحت ... الفوق))

The HEAPIFY Algorithm

[1, ... $\lfloor \frac{n}{2} \rfloor$]

* أي (External node) هو (heap) بنبادي ال (internal node) الكود ... heap

HEAPIFY takes two parameters, an array A and an index i into the array. It is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but the element at position i may be out of place. This routine moves the ith element into its proper place, so that the subtree rooted at i becomes a heap.

الترتيب تنازلي بنغير < ... بدل largest بنغير Smallest

```
HEAPIFY(A, i)              → Arry → parent index (بنطلع ال left Right)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l       (Smallest)
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
        HEAPIFY(A, largest)
```

* اذا كنت ال Root اكبر (element) ما نعمل swap ... اذا ... ال Alg.

* اذا واحد من Children اكبر من Parent ... swap ... element ... relation ship ... ال Alg. ... swap

... Parent ... Child اكبر

exit Condition ... Condition ... leave ... + length ... (Right, left indexes) ... اذا ... ال size ... left Child

Right →

* ... indese ... Value ... اذا (A(left) > A(P)) parent ... (left = largest) ... left child ... Parent ... اذا ال Right ... اذا A(Right) اكبر (A(left) > A(P)) ... largest ... Right (Right ... للقارنة)

اذا largest = i ... (طلع)

(Swap)

← سنبادي نفس ال Algo. ...

* ... (leave) ... out of index (left, Right)

من (child) ... من Parent

The HEAPIFY Algorithm

The way HEAPIFY works:

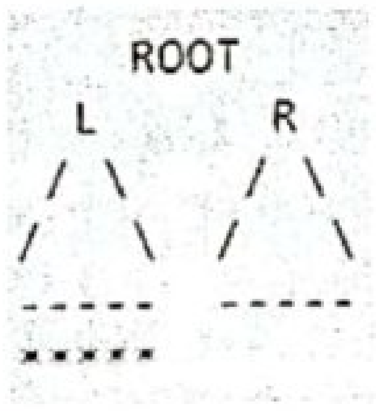
- Compare $A[i]$, $A[LEFT(i)]$, and $A[RIGHT(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

Worst-case Analysis of HEAPIFY

فين (subproblem الجديد)

(element ماينقسمش من (n) كاملة)

Assertion: A child of node i can have a tree of size at most $\frac{2n}{3}$.



$T(n) \leq T\left(\frac{n}{\frac{3}{2}}\right) + \Theta(1)$

$a = 1$, $b = \frac{3}{2}$

case 2

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

$\boxed{T(n) = \Theta(\lg n)}$

For a complete binary tree of height h, the number of nodes is $f(h) = 2^{h+1} - 1$. In above case we have nearly complete binary tree with the bottom half full. We can visualize this as collection of ROOT + LEFT complete tree (L) + RIGHT complete tree (R). If height of original tree is h, then the height of left is $h-1$ and right is $h-2$. So the total number of nodes in the tree (without the * nodes) can be expressed using the following equations:

$n = 1 + f(h-1) + f(h-2)$ ...(1)

We want to solve above for $f(h-1)$ expressed as in terms of n

$f(h-2) = 2^{(h-1)} - 1 = (2^h - 1 + 1)/2 - 1 = (f(h-1) - 1)/2$ ...(2)

Using above in (1) we have

$n = 1 + f(h-1) + (f(h-1) - 1)/2 = 1/2 + 3 * f(h-1)/2$

$\Rightarrow f(h-1) = 2 * (n - 1/2)/3$ $\Rightarrow$ Hence $O(2n/3)$

Worst-case Analysis of HEAPIFY *continued*

Complexity: Because the child of node i can have a tree of size at most $\frac{2n}{3}$,

$$T(n) \le T(\frac{2n}{3}) + \Theta(1)$$

By case 2 of the Master Theorem, $a = 1$, $b = \frac{3}{2}$, so $n^{\log_b a} = \Theta(1)$, and $f(n) = \Theta(1)$, so:

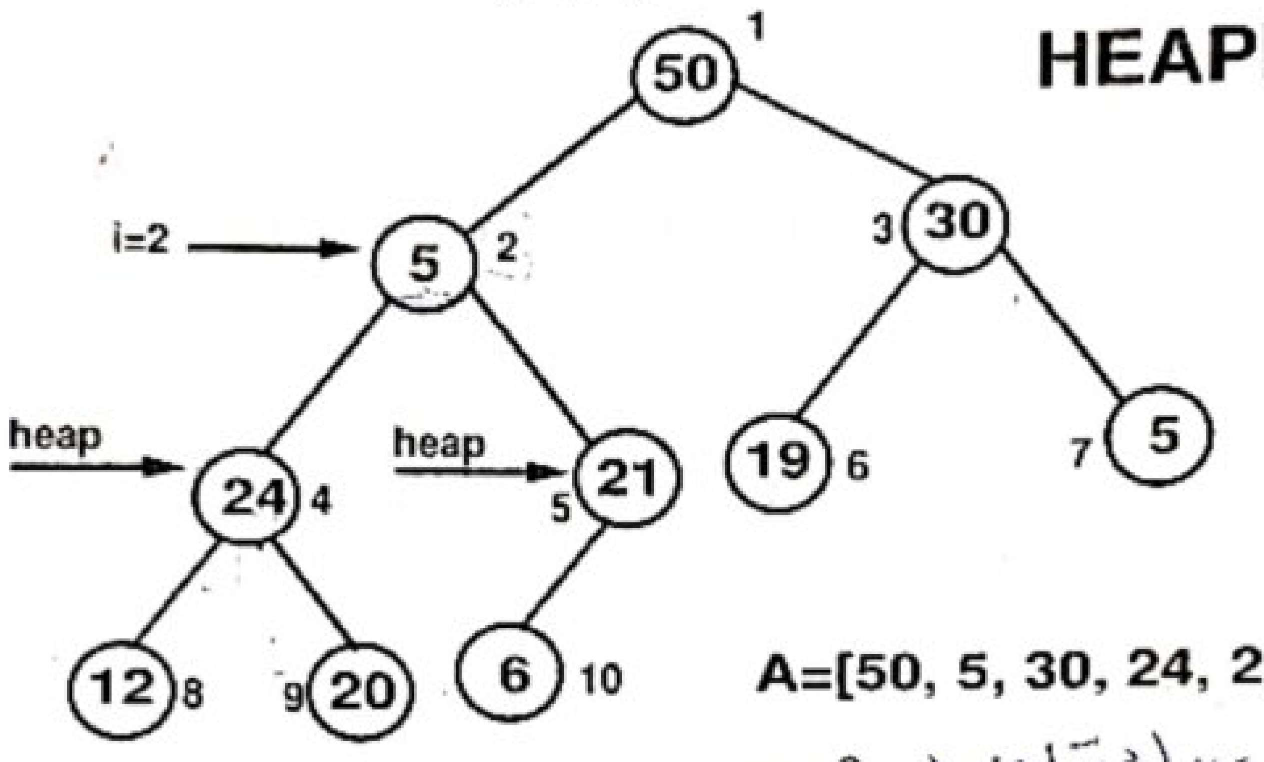
$$T(n) = O(\lg n) = O(h), \text{ where } h \text{ is the height of the heap.}$$

بنادي ال heap لل (2)

Example: Run HEAPIFY$(A, 2)$ on the following:

HEAPIFY(A,2)

(nearly Complet)



A=[50, 5, 30, 24, 21, 19, 5, 12, 20, 6] →

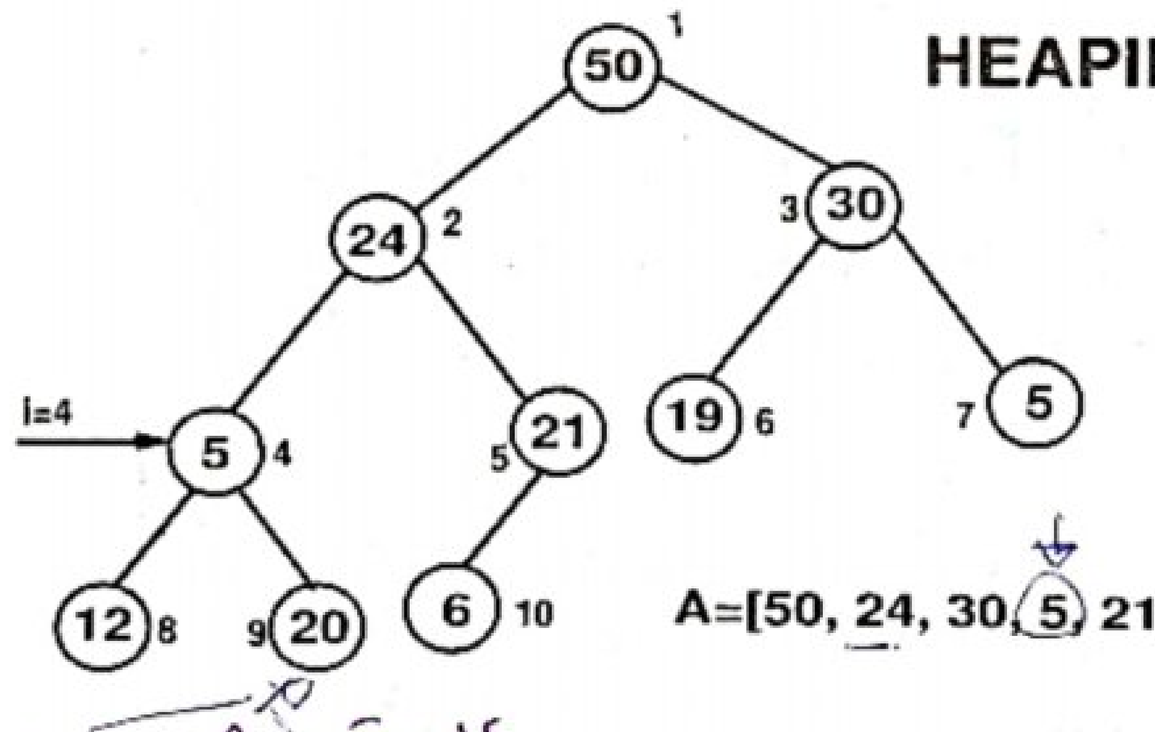
نكرره نكتب ال arry منفصل

كل ما تعمل ال swap بدك تنادي ال Heap IF y

The HEAPIFY Algorithm *continued*

بنادي ال heap لل 4

HEAPIFY(A, 4)

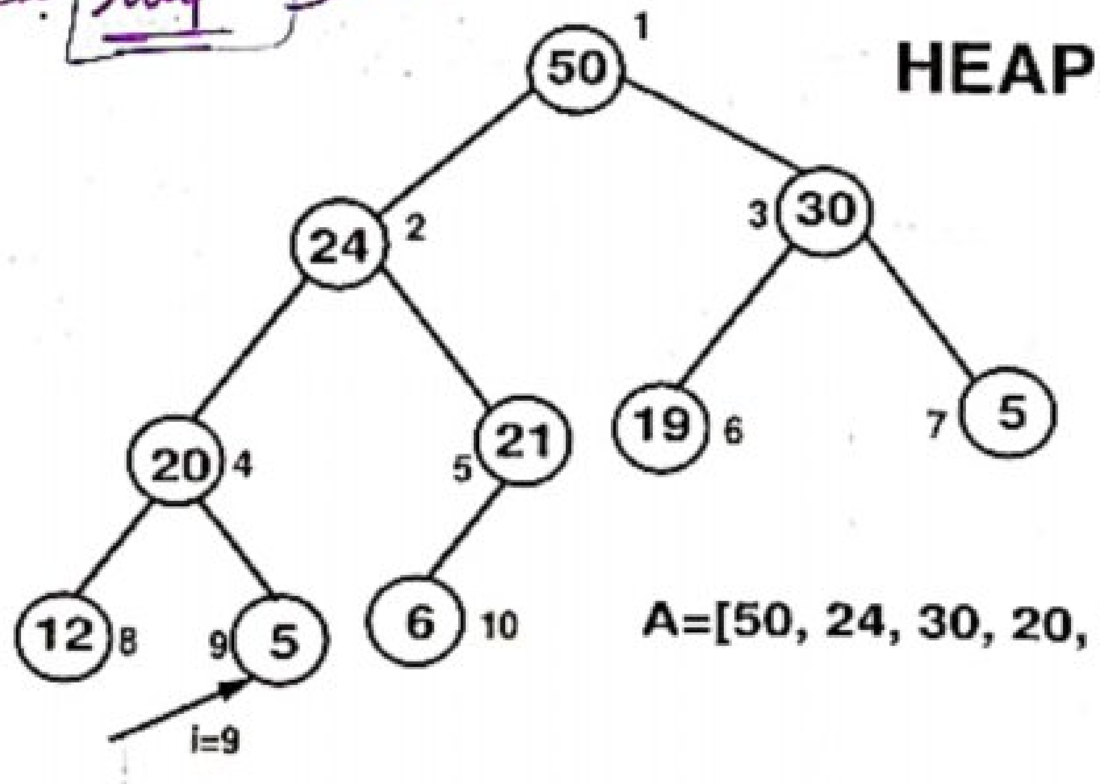


A=[50, 24, 30, 5, 21, 19, 5, 12, 20, 6] #

(hap IF y) بدك تنادي [Swap] كل ما تعمل

leave ☑

HEAPIFY(A,9)



A=[50, 24, 30, 20, 21, 19, 5, 12, 5, 6] #

(3call)

نكرر Call للطبقة

ال (arry)

Building a Heap

We can use HEAPIFY in a bottom-up way to convert $A[1..n]$, $n = length[A]$ into a heap. Since $A[(\lfloor \frac{n}{2} \rfloor + 1)..n]$ are leaves, they are already heaps; hence, BUILD-HEAP works bottom-up with the remaining nodes of A.

كل ال size بيجرينه ب heap

```
BUILD-HEAP(A)
1. heap-size[A] ← length[A]
2. for i ← ⌊length[A]/2⌋ downto 1
3.     do HEAPIFY(A, i)
```

(ينتقل) (bottom of Top)

(لنا اذا خطأ له بيني تحت انو له العقد)

* اذا بدي ابني heap بدي انادي على heap من (internal node) اللي بيكون (from bottom of Top)

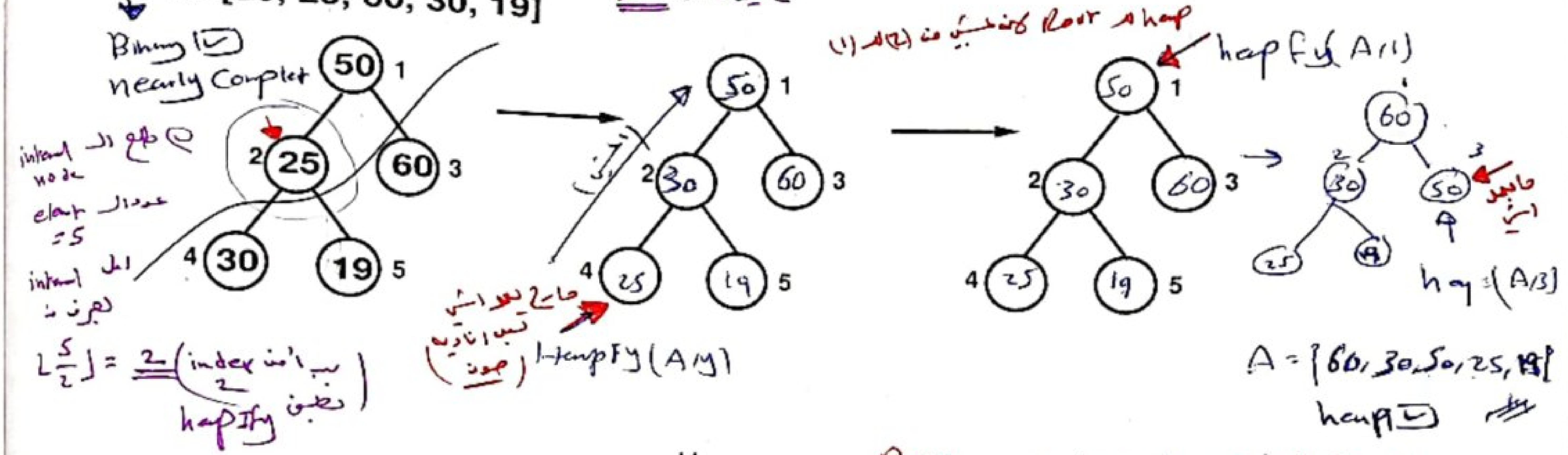
مثل ال (1) (ن)

كونو ال extreme left, Right (out of the index)

($T(n) = \lg n \Rightarrow$ ال Root (أكبر عدد من المقارنات ع ليوصل ال Root))

Example:

A=[50, 25, 60, 30, 19]



* اذا تدرك ابنيه في المزه بتبدا Build

انا ال Root صار ال max

بنلاقي heap

Building a Heap *continued*

Complexity:

A simple upper bound for BUILD-HEAP can be determined: each call to HEAPIFY is $O(\lg n)$ and there can be n such calls, giving an $O(n \lg n)$ worst-case running time. However, this is not asymptotically tight.

The time to HEAPIFY a node at height h is $O(h)$, so we can express the cost of BUILD-HEAP with the following summation (since the number of nodes at height h in an n node heap is at most $\lceil \frac{n}{2^{h+1}} \rceil$):

→ nearly

$\Rightarrow \lceil \frac{11}{2} \rceil = \lceil 5.5 \rceil = 6$ at most (leaves)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(\frac{n}{2} \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \lg n \rfloor} h \left(\frac{1}{2}\right)^h\right) =$$

$\to \frac{h}{2} \cdot \frac{1}{2}$

Because $\boxed{\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}}$, $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$. Hence,

$$T(n) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

Heapsort

(not stable) Alg.

[T(n) = nlgn] → (علشان كل مرة بنادي عن Root)

```
HEAPSORT(A)
1. BUILD-HEAP(A)
2. for i ← length[A] downto 2
3.     do exchange A[1] ↔ A[i]
4.         heap-size[A] = heap-size[A] − 1
5.         HEAPIFY(A, 1)
```

swap

(لما اوقف عندها) → لان الـ node هيكون لوحدها

عبارة عن heap max كاملا

للـ Root

(الـ element اللي بدلته انا)

بس البناء الـ heap أعمل Extract
لل max لطول الـ max وشيله
من heap مش محتاج Array
انا بس بطلع العنصر الـ max
من الـ heap)
علشان اجيب آخر leave حطه مكان
الـ Root بقلل الـ heap size دا وبطلع
heap array علشان اعمل heap مرة تانية
بنادي heapify للـ Root

⇒ عدد مرات انادي العملية (element − 1)

→ 2 آخر index

لأني مش هنادي على آخر (node وحدها)

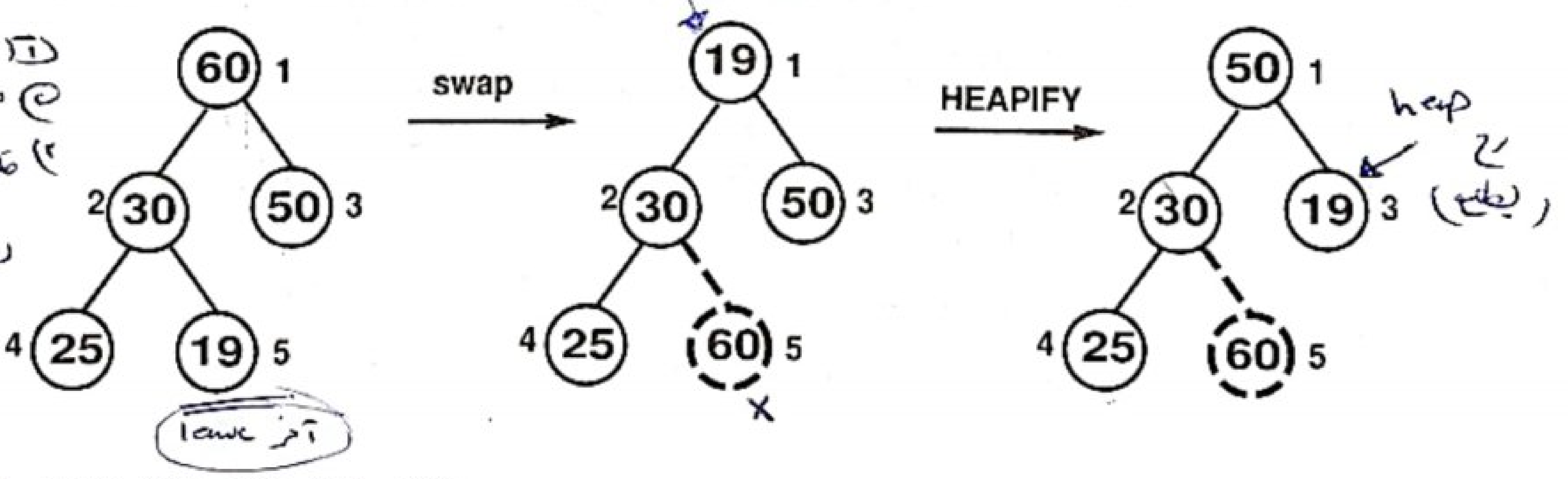
Complexity:

The $O(n)$ time to do BUILD-HEAP together with n calls to HEAPIFY gives HEAPSORT a worst-case running time of $O(n\lg n)$.

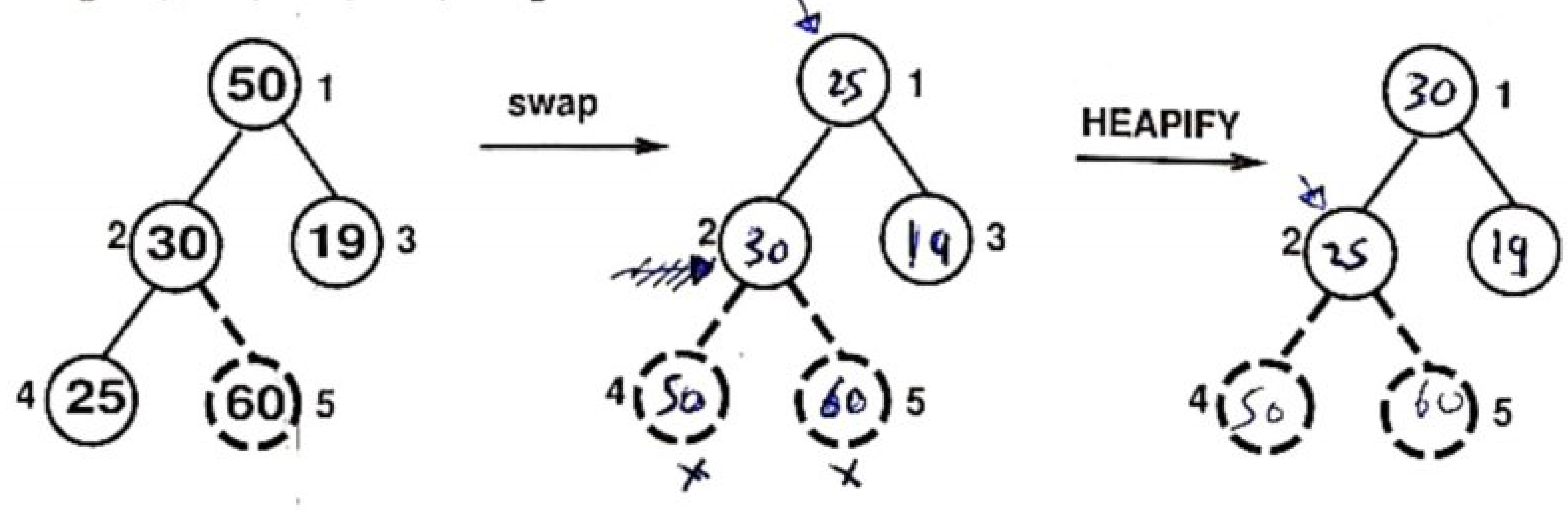
Heapsort Example

A=[60, 30, 50, 25, 19]

(1) بناء heap
(2) بلشيل آخر leave
(3) قلل (heap size) ليفضل
بطلت عنصر الـ heap الـ leave

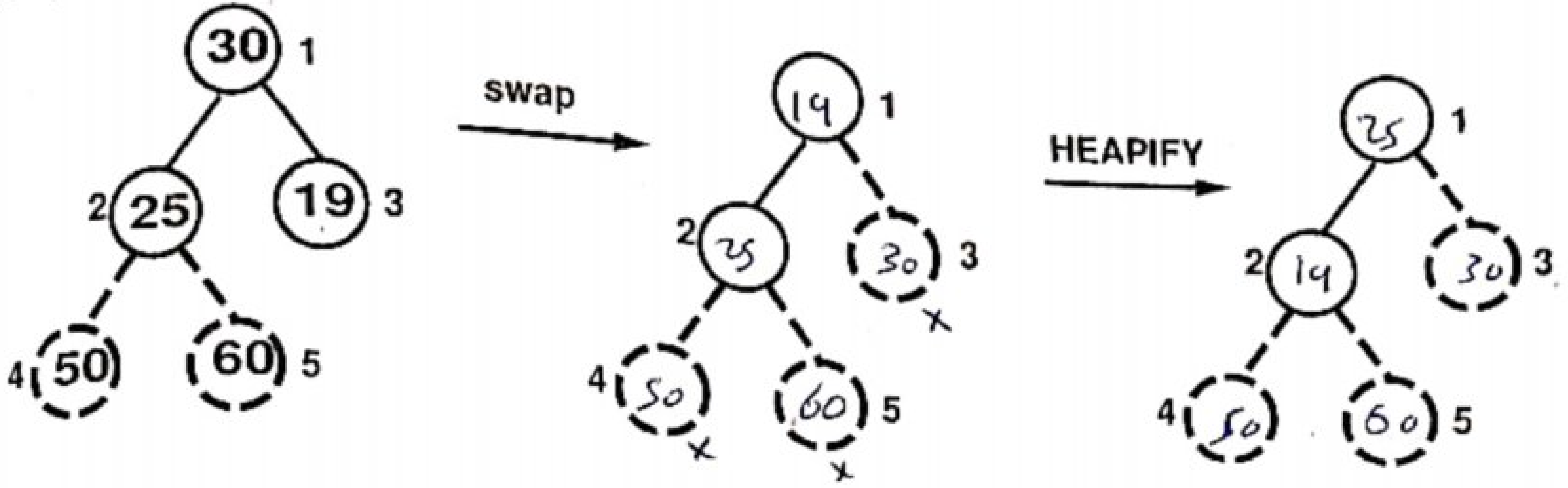


A=[50, 30, 19, 25, 60]

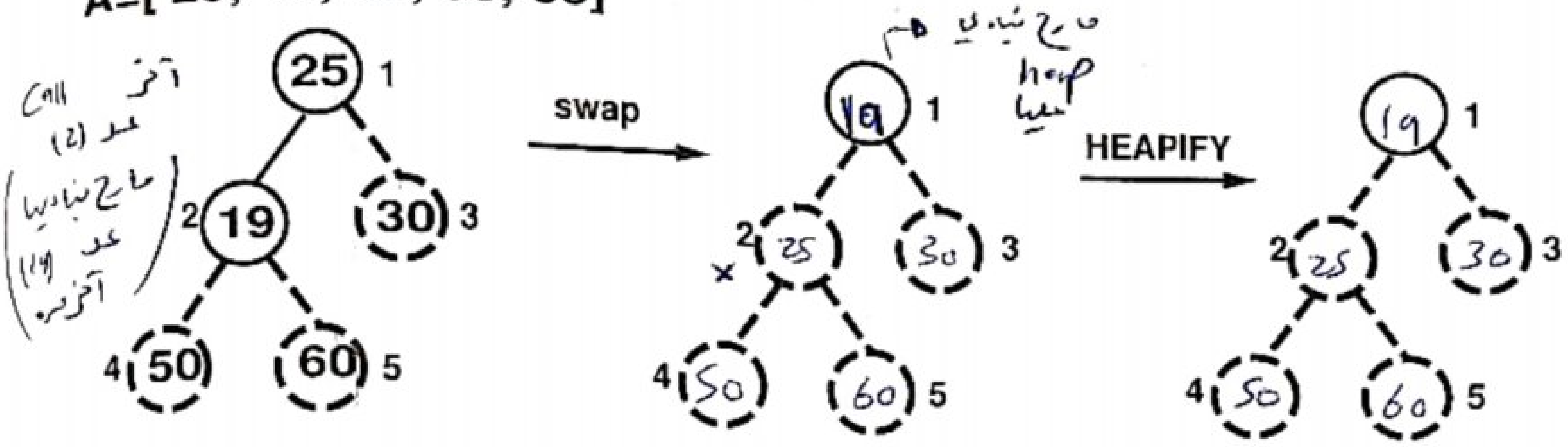


Heapsort Example *continued*

A=[30, 25, 19, 50, 60]



A=[25, 19, 30, 50, 60]



{19, 25, 30, 50, 60}

Priority Queues

A **priority queue** is a data structure for maintaining a set of elements, S, which supports the following operations:

- INSERT(S,x): insert element x into the set S. (اضيف)
- MAXIMUM(S): returns element with the largest key from the set S. (Top) Root
- EXTRACT-MAX(S): removes and returns the element with the largest key from the set S. (heap) طلعه برة
- INCREASE-KEY(S,x,k): increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

For example, a priority queue can be used in scheduling jobs with priorities on a computer system.

A heap is a good data structure for implementing a priority queue. We will examine the routines HEAP-EXTRACT-MAX, HEAP-INSERT and HEAP-INCREASE-KEY. The worst-case running time of these algorithms is related to the height of the heap, $O(\lg n)$.

The way HEAP-INSERT Algorithm work

Say that we want to insert element x into a heap with $(n-1)$ elements .

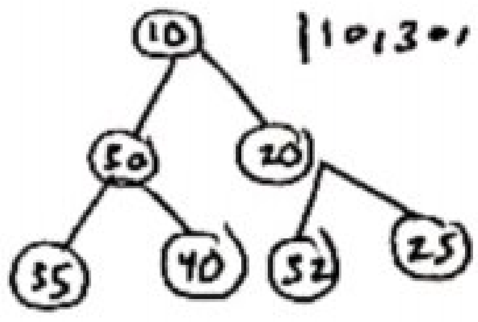
- Add x at the highest number level leaf.
- Increase heap size.
- Restore Heap property.
  - Repeat:
  - Compare x to its parent
  - If x is larger than its parent, then swap x with its parent until no swap is needed or x is at the root.

The way HEAP-EXTRACT-MAX Algorithm work

We want to delete the max element from the (n) elements heap .

- Delete the root (We need to restore Heap property).
- Decrease heap size.
- Put the right most element in the last level of the heap tree in the root position.
- Restore Heap property.

min heap |10,30,20,35,40,32,25|



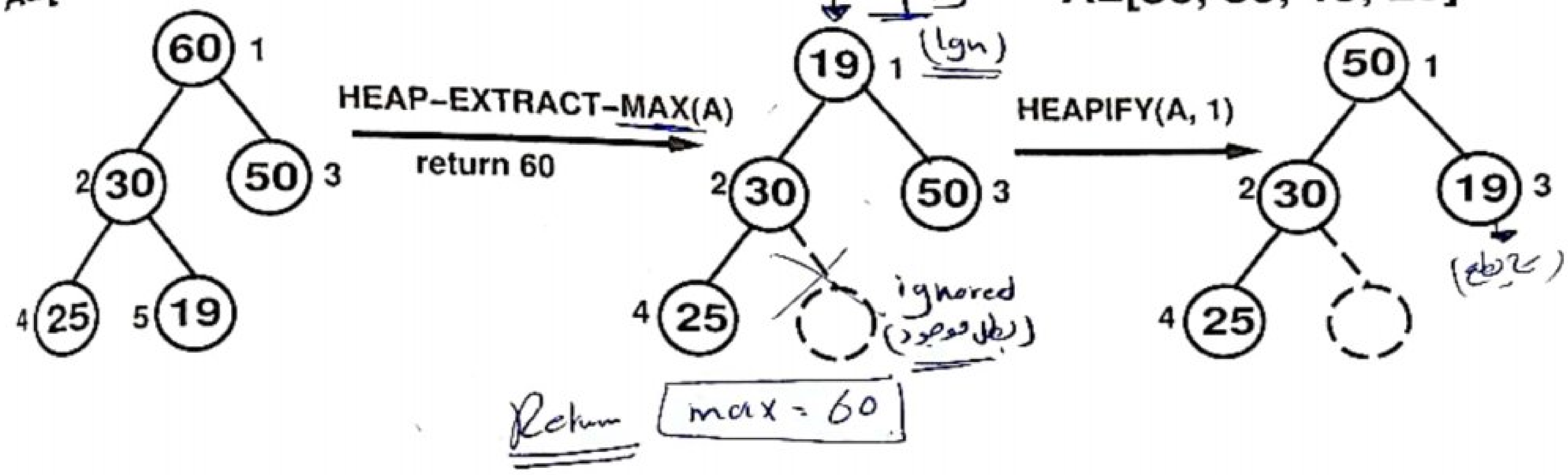
Priority Queues *continued*

```
      min
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] = heap-size[A] − 1
HEAPIFY(A, 1)
return max  min
```

(largest) → smallest ?
min نغير max بدل

A=[60, 30, 50, 25, 19]

A=[50, 30, 19, 25]



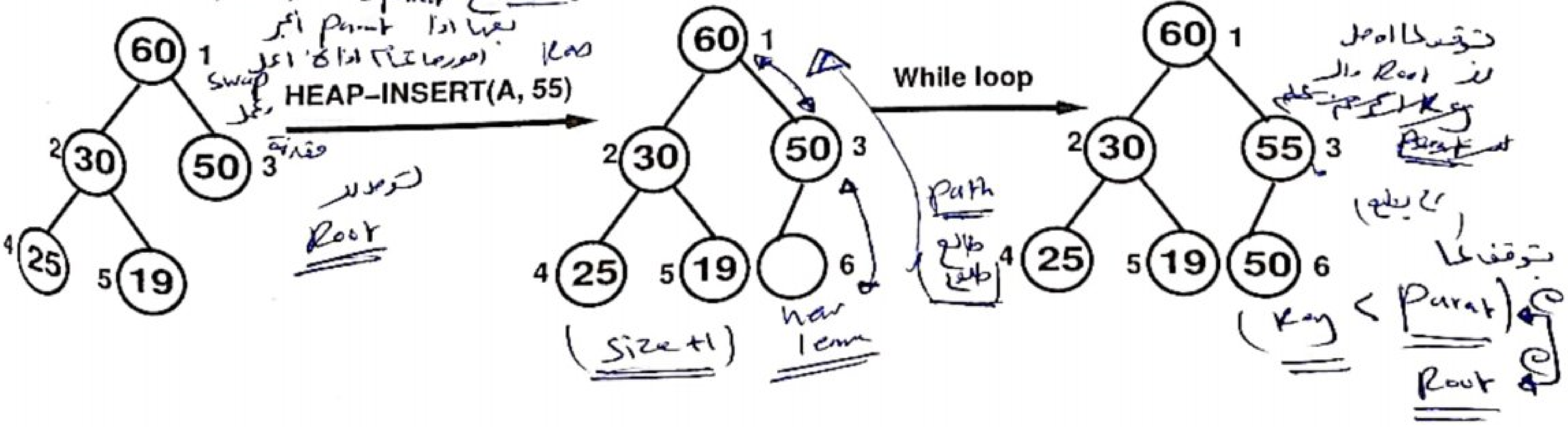
Priority Queues *continued*

```
          (Dec)
HEAP-INSERT(A, key)
heap-size[A] = heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[PARENT(i)] < key      (>)
        do A[i] ← A[PARENT(i)]
            i ← [PARENT(i)]
A[i] ← key
```

(1) لازم نزيد size الـ heap
(2) اذا رح نعمل insert مع Parent

A=[60, 30, 50, 25, 19]

A=[60, 30, 55, 25, 19, 50]



Decr نعدل اذا

node معينة

استبدال Current val

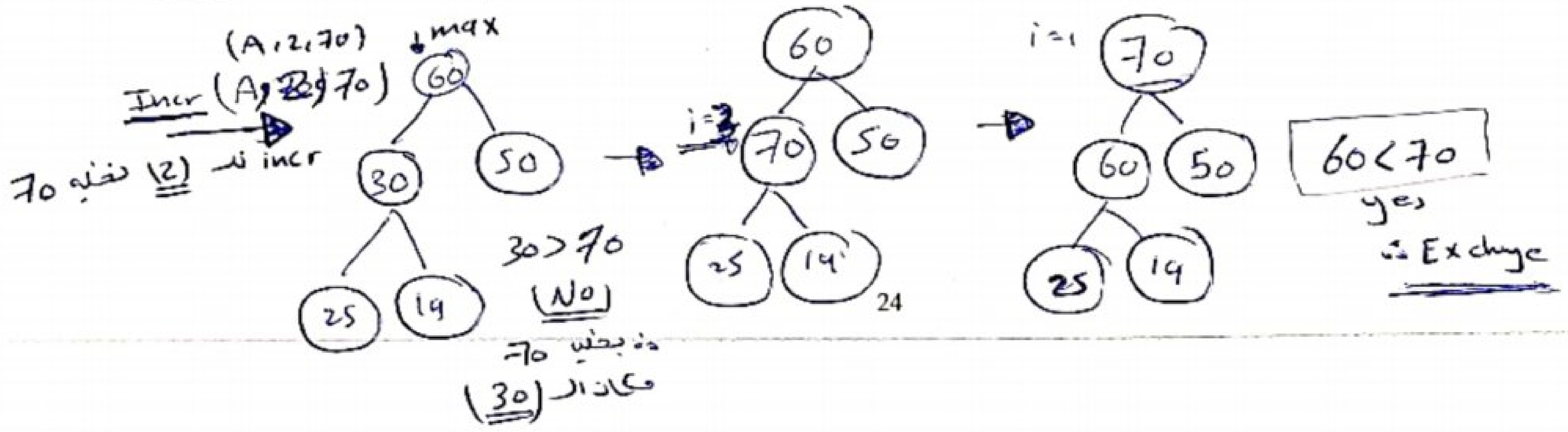
اشترط Condition

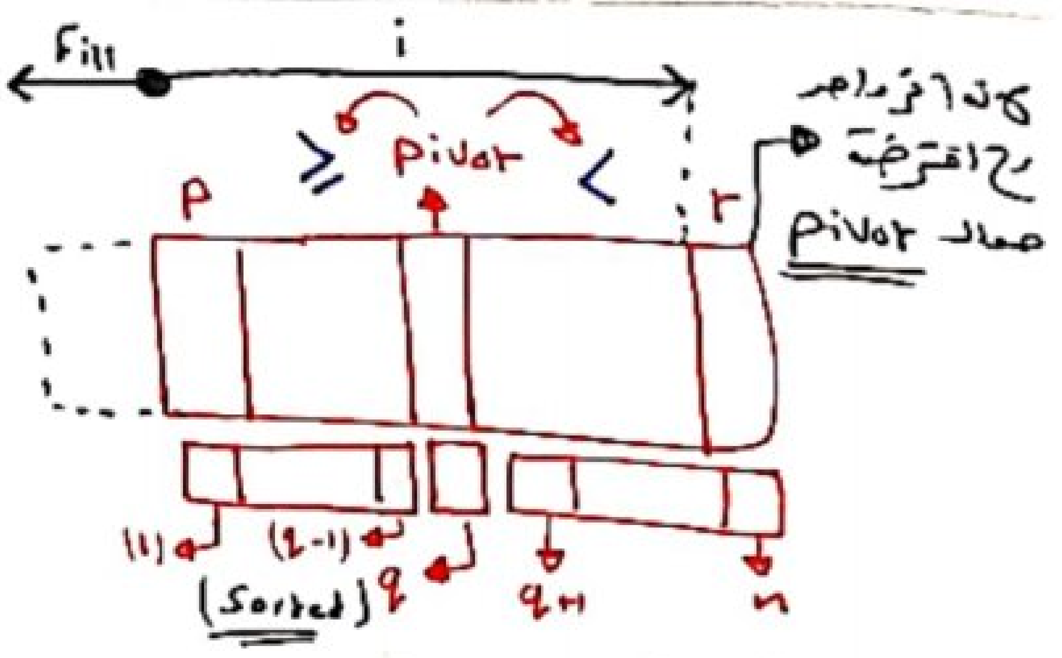
(new key > original val)

أي اذا لا x مضاعا مالقدر الجز increase

```
HEAP-INCREASE-KEY $(A, i, key)$
1. if $key < A[i]$
2.     then error / "new key is smaller than current key"
3. $A[i] \leftarrow key$ ( نغير تقابل مع (i) )
4. while $i > 1$ and $A[\text{PARENT}(i)] \overset{\boxed{1>}}{<} A[i]$
5.     exchange $A[i]$ with $A[\text{PARENT}(i)]$
6.     $i \leftarrow [\text{PARENT}(i)]$
```

Exercise: Modify the above routines to design a min-priority queue that supports the following operations INSERT, MINIMUM,EXTRACT-MIN, and DECREASE-KEY.





صار pivot رح افترض كون اخر واحد

صار يعمل تكرار ← (merge sort) الكن بحالة بعمل ال sorting بدون ما استخدم (axillary storage) in place

ب (merge sort) ضلني قسم لحد تعلّل (base case) لما اوصل لاساس رجع رتب (one element + one element) (merge)

اضطرينا هنا نعمل ال merge نتوفر (axillary storage) عشان نفصل ال array

برجع حسب المقارنات بدمج ترتيب ال (extra storage) هون بنضم ال element بعملية مقارنة بينهم

ال Quick Sort ما بدي استخدم (any axillary storage) بدي اعمل (partition) بحالة (original array) بحطهم ب ال (sorted element) بسبب لوجود (axillary storage)

ممكن يكون بأي مكان في array في البداية في المنتصف مش مهم ويكون بالمنتصف حتى اعداد يساويه بقل منه او اكبر منه (pivot)

Lecture 6: Quicksort

Dr. Khalil Yousef

(≥ كل اشي على شمال ال pivot اقل منه او يساويه
< كل اشي على يمينه ال pivot اكبر منه)

$$T(n) = T(q-1) + T(1) + T(n-q) + \square$$

$n-(q+1)+1 = n-q$

partition work at this division point

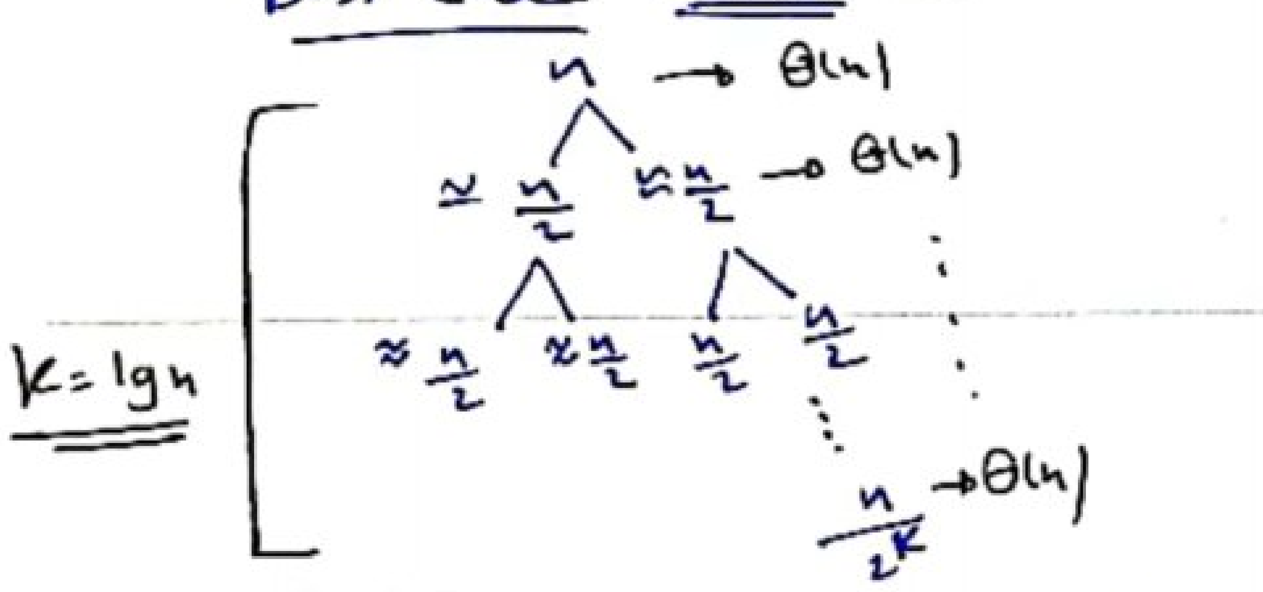
Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

* الكوك شورت يعتمد على مكان ال pivot ب Best او ال Worst case؟ على حسب شو بحط input

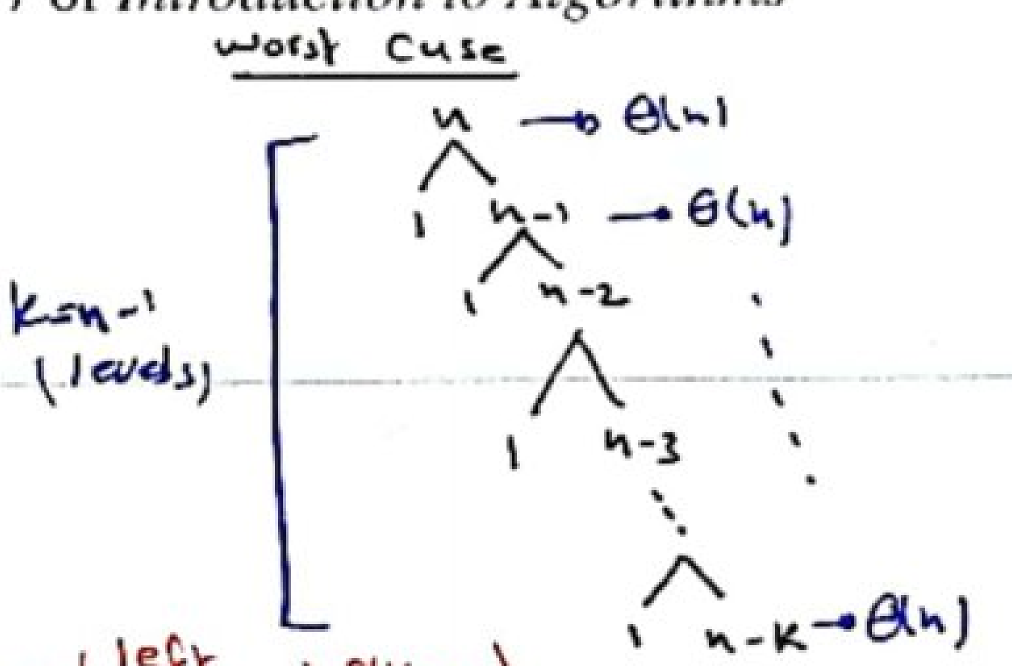
(insertion sort) عكس Avg. اقرب لل Best

Read Chapter 7 of *Introduction to Algorithms*

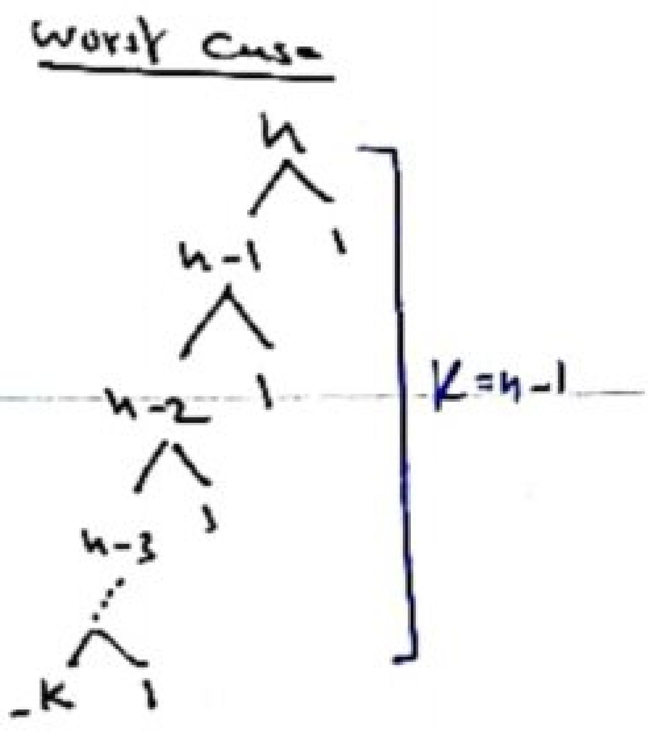
Best Case + Avg Case



Worst Case



Worst Case



Quicksort

(Stable)

$T(n) = 2T(\frac{n}{2}) + \theta(n)$ $\boxed{\theta(n\lg n)}$

لما يكون (left subarray) + ال pivot على اليمين $T(n) = T(n-1) + \theta(n)$ $\boxed{\theta(n^2)}$ ← $(n-1)(n) = n^2$

لما يكون (Right subarray) + pivot على الشمال $T(n) = T(n-1) + \theta(n)$ $\boxed{\theta(n^2)}$

Quicksort, like merge sort, is a divide-and-conquer algorithm for sorting a subarray $A[p..r]$.

Divide: Partition the subarray $A[l..r]$ into two subarrays: $A[l..q-1]$ and $A[(q+1)..r]$ so that all the elements in the first subarray are less than or equal to $A[q]$, and all of the elements in the second are greater than $A[q]$. The value q is computed during the partition process.

Conquer: Recursively sort the two subarrays.

Combine: No work required as the subarrays are sorted in-place.

The Quicksort Algorithm

```
array   اول elem
        آخر elem
index آخر الترتيب index اول ← QUICKSORT(A, P, r)
1. if P < r
                                        (pivot)
(pivot element) مكان ال ← 2.    then q ← PARTITION(A, P, r)
3.            QUICKSORT(A, P, q − 1)
4.            QUICKSORT(A, q + 1, r)
```

The initial call is QUICKSORT(A, 1, $length[A]$).

The Partition Algorithm

The Partition Algorithm (in English):

- Pick a **pivot** element (we'll use the element at the right end).
- Pass over elements from left to right, swapping any element $\leq$ the pivot to a growing region at the left end. Left end gets filled with the "small" elements.
- After the pass, the pivot gets swapped to the place just after the "small element" region.
- Return the final index of the pivot.

The Partition Algorithm *continued*

Time Complexity = n-1

```
PARTITION(A, P, r)
1. pivot ← A[r]
2. fill ← P − 1                      ( Fill = P-1 )
3. for j ← P to r − 1                ( for i=p to r-1 )
4.     [do if A[j] ≤ pivot]          Basic(n-1) operation
5.         then exchange A[++fill] ↔ A[j]     ( Fill=Fill+1 )
7. exchange A[++fill] ↔ A[r]   ▷ place pivot element     or pivot (pivot = A[r])
8. return fill                       (q) ابرجع ال
```

(pointer)

ال Fill بيتحرك opposite ~~opposite~~ direction)

Consept Fill : اذا ال Value (أي Value في array) الأكبر من Pivot خليها على Right Side بخليها على ال Right Side (incr. Fill من خلال ال) كل مالاقي Value أقل من pivot زيد ال Fill واعمل Exchange

(بنزيد ال Fill الا لو الشرط يتحقق عليه) ① ②

زيد ال Fill واعمل Exchange بين ال A الموضع Fill

The running time of PARTITION is $\Theta(n)$, where $n = r - l + 1$.

ال Fill في البداية رح احطه بساوي (p-1) ~~[illegible]~~

(يعني اذا بتبلش من zero index بحط ال Fill لـ zero

اذا ابتديت من one index بكون ال z=P وال 1=Fill)

بطلع او بتوقف لما يطلع واصل ال element لل الشمال ابرت Pivot (مابزيد ال Fill) بعمل skip د بجيب لبعدي اذا بلش ومن ثم الأمر بزيد ال Fill

الملخص (افترضنا آخر اصحو ال pivot ، بدي آخر كل ال element من (p) للـ (r-1)

بدي اتأكد كل يلي على شماله اكبر او يساوي ال pivot ، كل ما لاقيت element اصغر منه

اعمل swap ، اما اذا لقيت ال element أكبر منه ما بعمل swap . الطريقة بتضمن

انه ال pivot أصغر منه left subarray واكبر منه Right subarray

ال Fill بأكد مكان انت عملته swap.)

Partition Example: $A = <7, 8, 2, 6, 5, 1, 3, 4>$

Draw behavior of *fill*, *pivot*, and *j* during *PARTITION*$(A, 1, 8)$ on overhead.

Sol A = (7, 8, 2, 6, 5, 1, 3, ④) pivot, r=8 1<8 (نادي ال Partition)

Fill +1 Fill P=1 i swap

↳ 7<4 ⊗ , 8<4 ⊗ , 2<4 ☑

(A,1,8) ⇒ (2, 8, 7, 6, 5, 1, 3, ④)

↳ ① بنزيد ال Fill (Fill=1)
↳ ② i=3
↳ ③ Exchg i=3, Fill

Fill + ~~Fill~~ Fill swap i

↳ 6<4 ⊗ , 5<4 ⊗ , 1<4 ☑

↳ ① بنزيد ال Fill (Fill=2)
② i=6
③ Exchg i=6, Fill

فكرة ال Fill تخليه left اقل او يساوي pivot

⇒ (2, 1, 7, 6, 5, 8, 3, ④)

Fill + Fill swap i

↳ 3<4 ☑
↳ بنزيد ال Fill (Fill=3)
↳ i=7
↳ Exchg i=7 , Fill

ال i رح يطلع لانه خلص بس يطلع رح ازيد ال Fill واعمل swap مع ال pivot

⇒ (2, 1, 3, 6, 5, 8, 7, ④)

~~Fill~~ + Fill swap

index → 1 2 3 4 5 6 7 8
⇒ (2, 1, 3, ④, 5, 8, 7, 6)

return 4 (index) مش Value

Partition (A,1,3)

(2, 1, ③) pivot Fill i 2<3 ☑ , 1<3 ☑ تشبه حالة (worst case)

(2, 1, ③) Fill i 1<3 ☑

(2, 1, ③) 1 2 3 Fill Fill,i return = 3(index)

Partition (A,5,8)

(5, 8, 7, ⑥) Fill 5<6 ☑ 8<6 7<6 ☒

(5, ⑥, 7, 8)

The Partition Algorithm *continued*

The correctness of this algorithm can be shown using a loop invariant:

At the beginning of each iteration of the **for** loop, $A[l..r]$ contains the same elements it started with, possibly rearranged, and, for any array index k,

1. If $l \leq k \leq i$, then $A[k] \leq pivot$.

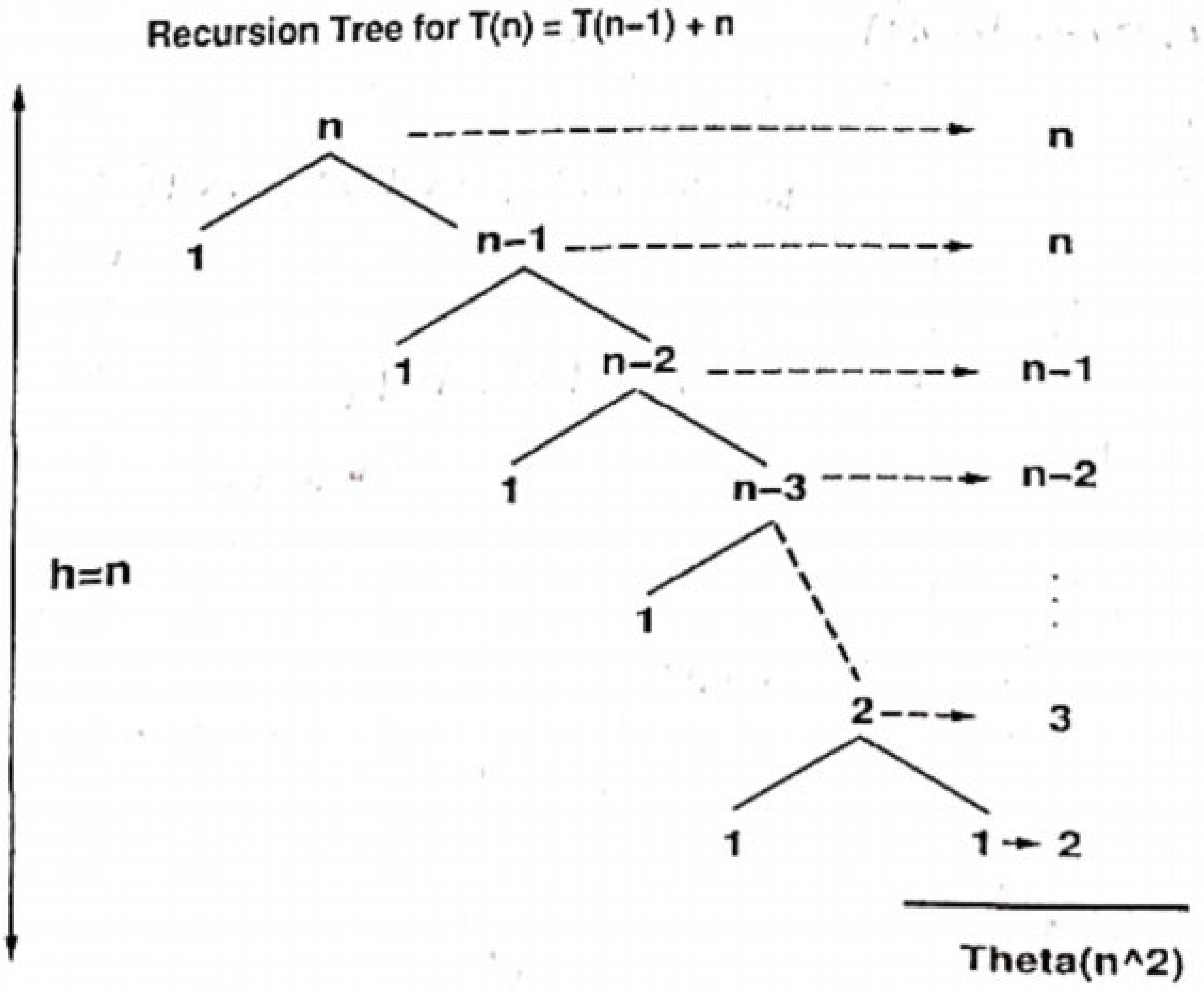
2. If $i < k < j$, then $A[k] > pivot$.

3. If $k = r$, then $A[k] = pivot$.

The running time of QUICKSORT depends on the sizes of the left and right partitions (which are affected by the choice of the pivot).

Worst Case Performance of Quicksort

Worst Case: The left or right partition has size of one element.



Worst Case Performance of Quicksort *continued*

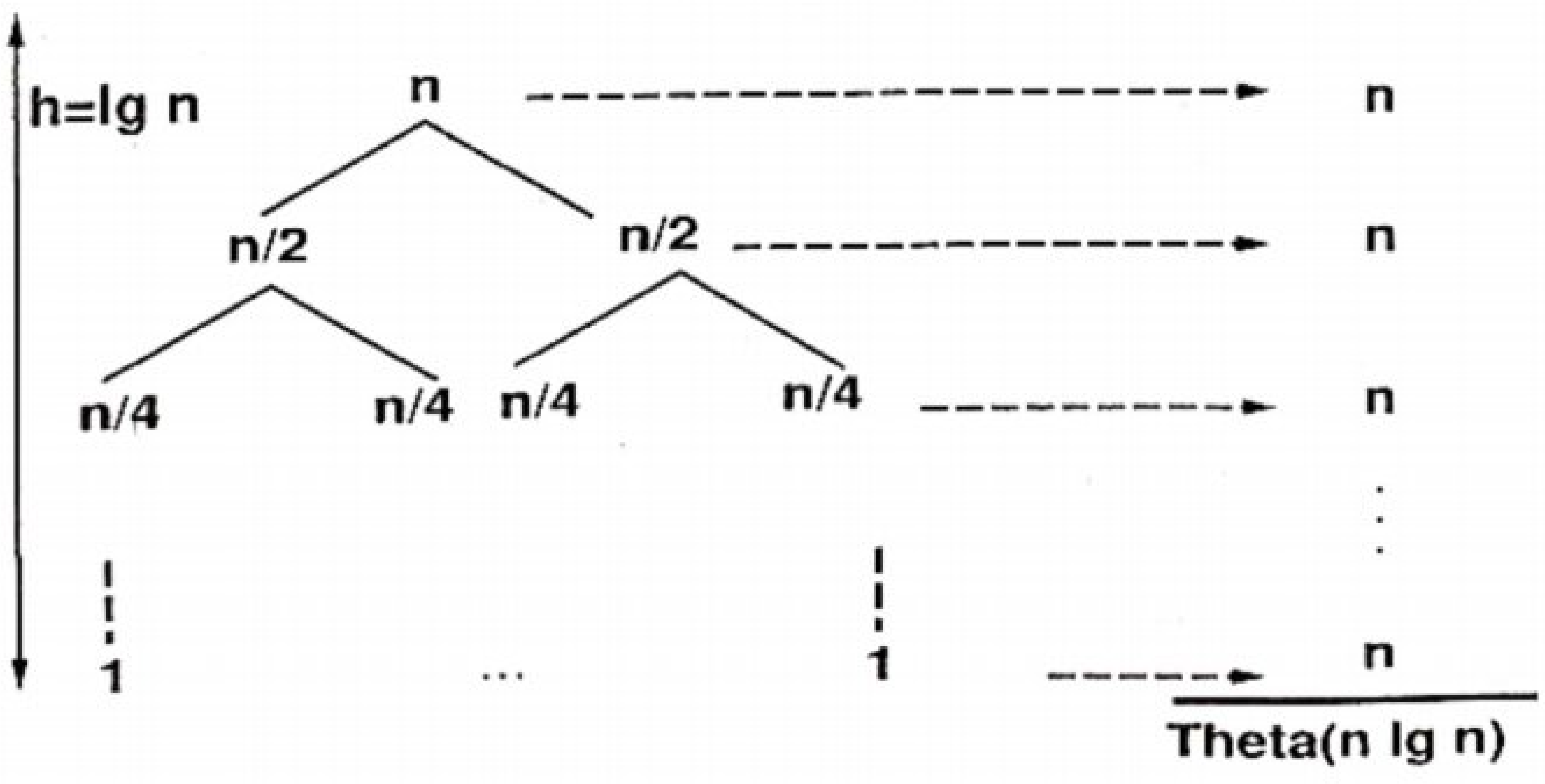
$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= T(n-2) + \Theta(n-1) + \Theta(n) \\
&= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&\vdots \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta(\sum_{k=1}^{n} k) \\
&= \Theta(n^2)
\end{aligned}
$$

When does the worst case occur?

Best Case Performance of Quicksort

The best case occurs when the sizes of the left and right partitions are equal sized.

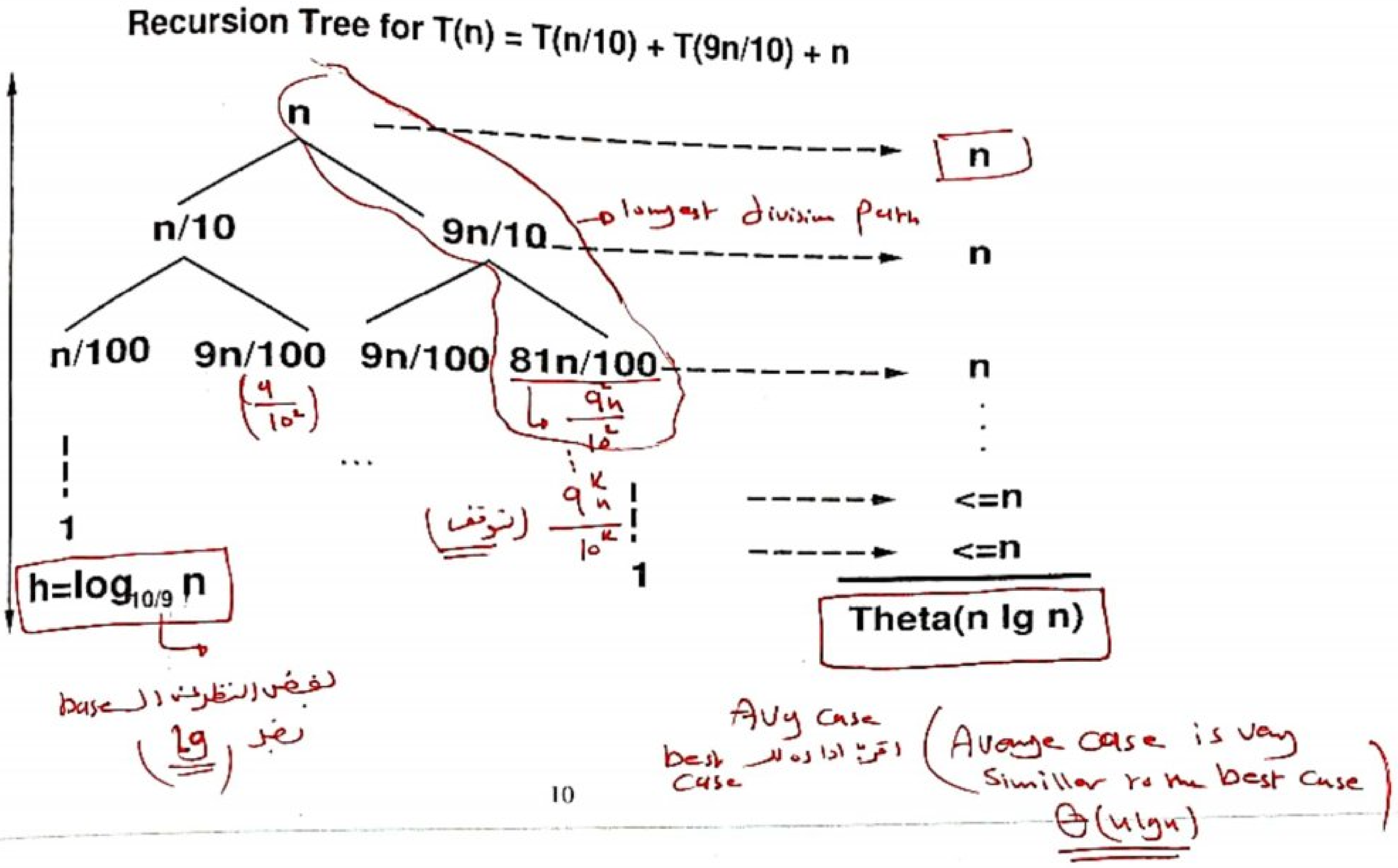
Recursion Tree for T(n) = 2T(n/2) + n



$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= \Theta(n \lg n)
\end{aligned}
$$

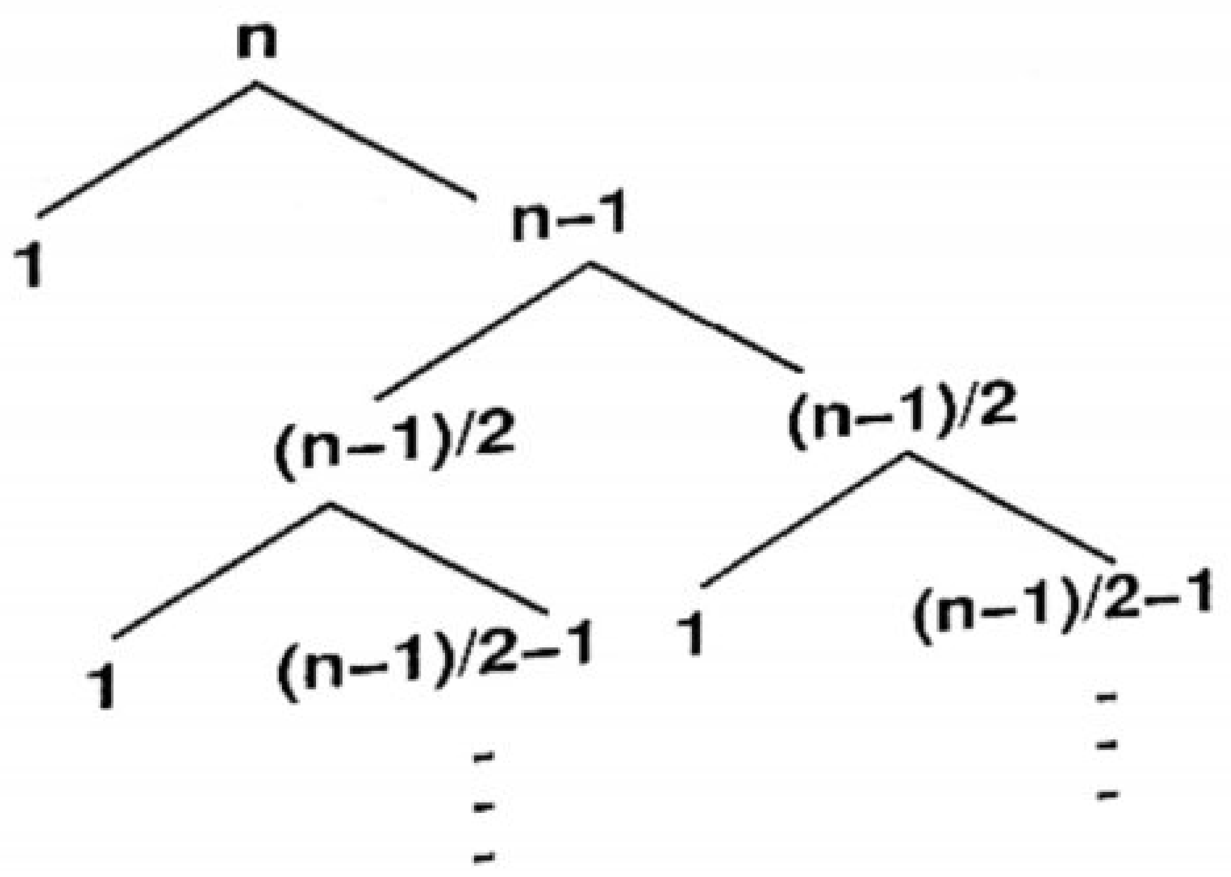
Balanced Partition Case

he average case running time of QUICKSORT is closer to the best case than the orst case. Even if there is an imbalance in the partitions of 9-to-1 (or 99-to-1 for at matter), we obtain a running time of $\Theta(n\lg n)$:



tuition for the Average Case

hen we run QUICKSORT, it is unlikely that the partition occurs in the same ay at every level; some will be balanced and some won't be. In the average case, ARTITION produces a random mix of "good" and "bad" splits. If the good d bad splits alternate over the levels, we end up with a nearly balanced partition hich is better than the 9-to-1 case:



verage Case will be $O(n\lg n)$

Randomized Quicksort

QUICKSORT's performance depends on the distribution of the input data. If we create a randomized version of QUICKSORT, the worst case behavior will rely on the random number generator hitting an unlucky and unlikely partition, again and again.

One possible implementation of RANDOMIZED-QUICKSORT is to randomize PARTITION by picking the pivot element at random from the range $l..r$, exchange $A[r]$ with that element, and run PARTITION as before.

(مش دايمًا بأخذ الـ Pivot آخر واحد بيكون عشوائي)

(Stable) ممكن يبطل تكون

ما بنزيل الـ worst case بس بنقلل

موجود الـ input اللي بتوديني للـ (worstcase)

```
RANDOMIZED-PARTITION(A, l, r)
  i ← RANDOM(l, r)
  exchange A[r] ↔ A[i]
  return PARTITION(A, l, r)
```

Randomized Quicksort *continued*

```
RANDOMIZED-QUICKSORT(A, l, r)
  if l < r
     then q ← RANDOMIZED-PARTITION(A, l, r)
          RANDOMIZED-QUICKSORT(A, l, q − 1)
          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Note: This does not improve the worst case running time of the algorithm, but it does prevent particular input cases from causing the worst-case behavior.

Thus the worst case running time is still $O(n^2)$.

proof of this was done through what is Knowns Desision trees ⇐ it was show no such Algo in the worst Case Could achive better than Θ(n lg n) ⇐ [Selection sort, Insertion Sort, merge sort, heap sort, Quicksort] Comparision based Sorting Algo ب ال

Lecture 7: Linear Sorts

بدل تستخدم n lg n افرض انه Performance الناتج بتحكي إذا بدك تحصل (for gather to sorting) تكنيك آخر ⇒ (linear sorting)

Course Learning Outcome :

Show how time and space complexities can be traded-off (e.g. distribution counting sort, hashing)

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

*linear Sorting Algorithms

1. Counting Sort
2. radix sort
3. Bucket sort

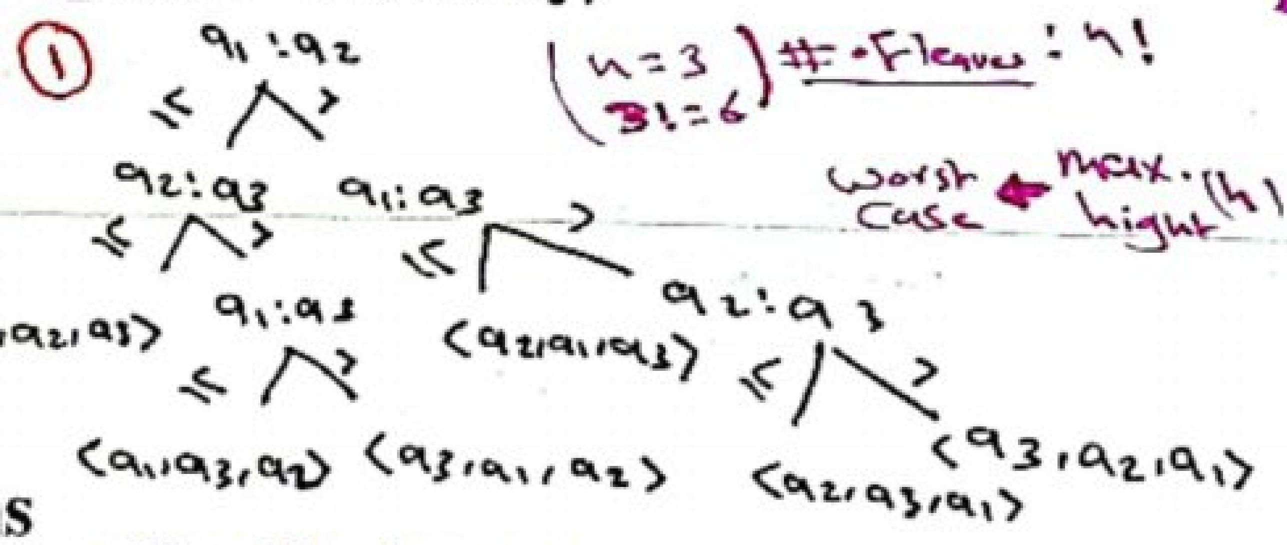
* Theorem: we can't acheive better than n lg n (i.e Ω(n lg n)), for any Comparison based Sorting Algorithm in the worst Case

Read Chapter 8 of *Introduction to Algorithms*

$\left[\begin{matrix} T(n) \geq n\lg n \\ T(n) = \Omega(n \lg n) \end{matrix} \right]$ worst Case

*let's Consider three elements in an array A = ⟨a1, a2, a3⟩

Decision Tree : (present or show all possible Comparison problem that has Comparison as its basic operation)



(n=3, 3!=6) # of leaves : n!

worst Case ⇐ max. hight (h)

leaves represent a solution

min hight ⇒ best Case

branch: عدد الـ branch أقل

(n-1) (best Case)

Comparison Sort Algorithms

* Time Complexity is measured by the hight of this tree from the root to the leave in solution

How fast can we sort? Much depends on the assumptions one makes about how the sorting is accomplished.

So far we have considered only algorithms that sort by comparing pairs of elements (i.e., INSERTION-SORT, MERGE-SORT, HEAPSORT, and QUICKSORT). They are called **comparison sorts**.

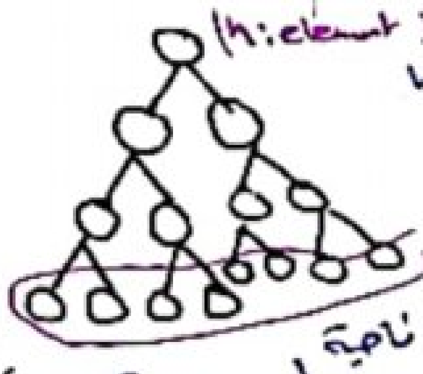
→ Best Case path

Any sort requires at least $n-1$ comparisons, or it won't have examined the input. However, all comparison sorts require $\Omega(n \lg n)$ comparisons to sort in the worst case. Can we achieve a sort that is faster than $\Omega(n \lg n)$? yes, linear Sorting

We assume that all elements are distinct while we are discussing the lower bound on comparison sorts. In this case, $a_i < a_j$, $a_i > a_j$, $a_i \leq a_j$, $a_i \geq a_j$ provide equivalent information relative to ordering; hence, we use $a_i \leq a_j$.

نقارن ①,② ②

بنينا tree لما نقارن (n) العناصر



(Depth of the tree (max hight = h)) h

now let's Consider a complet binary tree of hight h

# of leave = 2^h (# of external nodes) / # of internal nodes = $\sum_{i=0}^{h-1} 2^i$

$[2^h \geq n!]$ lg n , $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$

من ناحية أخرى $h \geq \lg n!$

أو للطرفين $(h \geq \Omega(n \lg n))$ ⇒ (longest path في decision tree بتكون أكبر من أو تساوي n lg n) ⇒ longest path (آخر leave)

in Decision tree ⇒ # of leave = n!

* Counting Sort: Assum the input array A: is a set of n integers in the Range linear 1:K (max)

A, <a1, a2, a3, ... an>

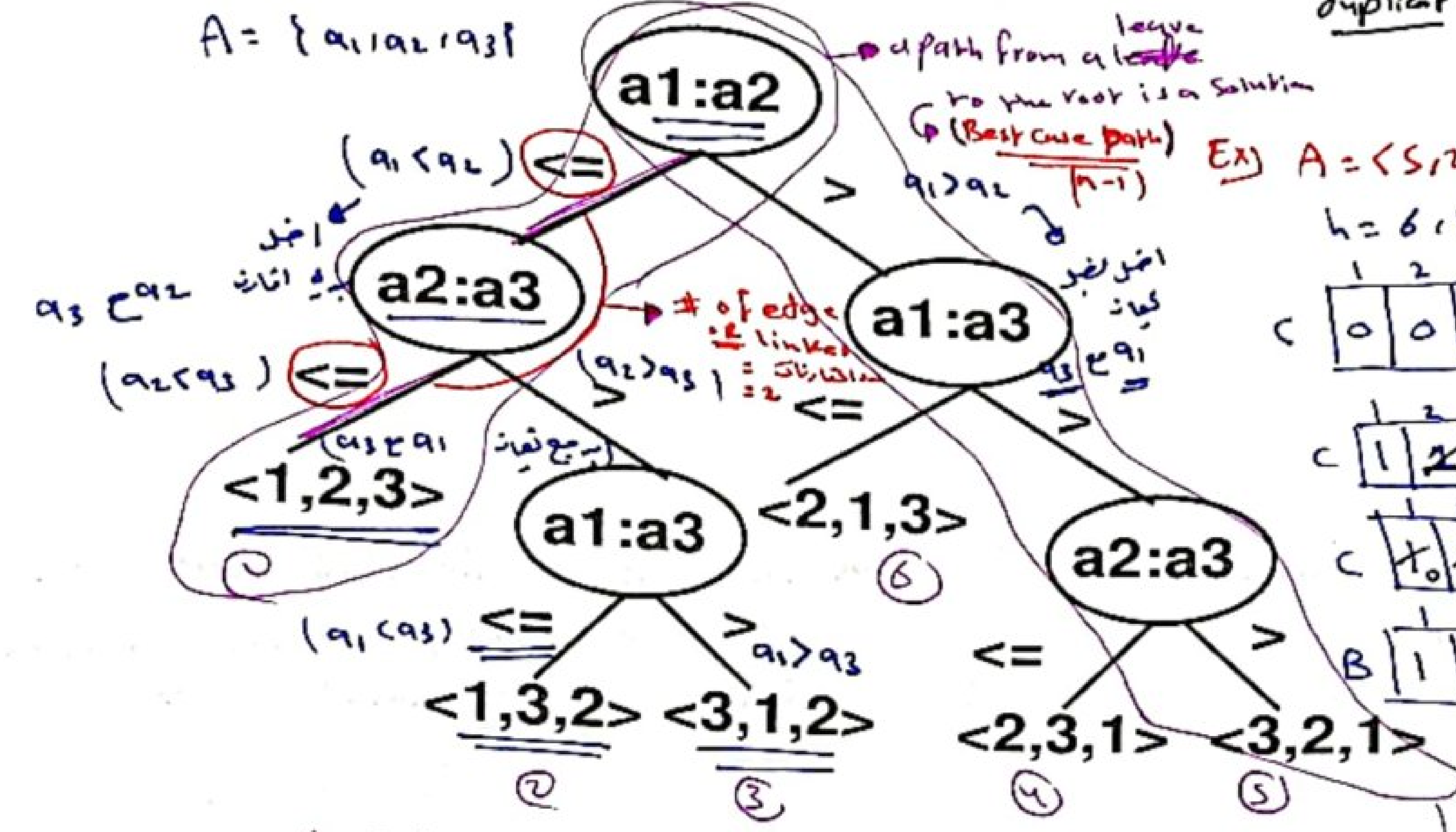
```
CountingSort (A, n, K)
{ for i=1 to K
    C[i] = 0
  for (J=1 to n)
    C[A[J]] = C[A[J]]+1
  for i=2 to K
    C[i] = C[i] + C[i-1]
  for J=n to 1
    B[C[A[J]]] = A[J]
    C(A[J]) = C[A[J]]-1
```

$\theta(n+K)$ K ماتكون قريبه من n عشان تضيف linear

Decision Trees

A comparison sort can be viewed in terms of a **decision tree**, which represents the comparisons performed by the algorithm operating on an input of a certain size.

بدل ما اطرح duplicat ال Count



Ex) A = <5, 2, 1, 3, 2, 4>

n = 6, K = 5

	1	2	3	4	5	6
C	0	0	0	0	0	
C	1	~~2~~ 2	1	1	1	
C	~~1~~ 0	~~3~~ 2	~~4~~ 3	~~5~~ 4	~~6~~ 5	
B	1	2	2	3	4	5

6 element منهم 5 أو أقل

5 element منهم 4 أو أقل

# of leaves represent # of all possible premutation or solution

n = 3

n! = 3×2×1 = 6 (possible solution)

(max)

Hight of the tree Captures the worst case Senario → (longest path)

worst case Path (n^2)

Decision Trees *continued*

# of leave in binary tree > # of leave in the Decision tree

$\boxed{2^h > n!}$

- There is a comparison tree for each input of size n.
- The leaves represent all possible permutations of the input array; hence, there are $n!$ leaves in the tree.

internal ←

أي node عبارة عن مقارنة بين (Possible element) / ال external الورق وصلك لل Solution

- Internal nodes represent comparisons between pairs of elements.
- The path from the root to a leaf shows the comparisons for arriving at the leaf's permutation (an execution trace).
- The length of the longest path in the tree indicates its worst-case running time.

length shortest path = Best Case

Theorem 9.1: Any decision tree that sorts n elements has height $\Omega(n \lg n)$, (i.e., at least one path is that length).

Complexity of Comparison Sorts

Proof: (We ignore data movement, bookkeeping operations, etc.)

The number of leaves in the decision tree is at least $n!$, or two permutations go to the same leaf.

The number of leaves in a binary tree is $\leq 2^h$, and:

$$\begin{aligned} 2^h &\geq n! \\ h &\geq \lg(n!) \end{aligned}$$

Recall **Stirling's approximation:** $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+\Theta(1/n)) > (\frac{n}{e})^n$, hence:

$$\begin{aligned} h &\geq \lg(\frac{n}{e})^n = n\lg n - n\lg e \\ \boxed{h} &\boxed{= \Omega(n\lg n)} \end{aligned}$$

better than nlgn in the worst case

longest path ← اذا طلع على

(linear Sorting) انتقل لل !? nlgn عشان نقدر نجيب احسن من

tool استخدامك → ① face detection and Recognition

Domain [illegible]

Decision Tree Questions

1. In a comparison sort decision tree, what do the leaves of the tree represent? How many leaves must there be in a decision tree sorting n elements? n!

possible Solution

2. In a comparison sort decision tree, what do the non-leaf nodes of the tree represent?
3. In a comparison sort decision tree, what do the edges of the tree represent?
4. What is the length of the longest possible path in any comparison sort decision tree for an input of size n such that comparisons are not duplicated? Why?
5. What is the length of the shortest possible path in any comparison sort decision tree? Why? (n-1) Best Case OR اذا اثرت (n-1) العنصر

ما مشيت للآخر (input)

Distribution Counting Sort or simply Counting Sort

→ Sorting process ~~of~~ is based on a Counting principle → So, there is no single Comparision being made by this algo. ↓ linear Sorting Algo.

(its linear Sorting Algo.)

Assumption: input elements are integers in the range 1 to k, where k can be different than n. Duplicates are allowed.

(value A = index) ① اليس

→ Input is give in the Array (n)
→ Axillary storage, will be defined: C/B

k must be known, which Represent the max integer value that the input could have.
if not given, then we know to find it linear time (Applying max Alg.)

Abstract: The basic idea is to count, for each element x, the number of elements less than that element so that we can determine the index of placement for x. If 10 elements are less than x, then x goes into the 11th position. (The algorithm is slightly more complex than this because it handles duplicates.)

We will show that COUNTING-SORT's complexity is $O(n+k)$. If $k = O(n)$, then the complexity is $O(n)$.

Size C = k ← C : will hold the Count of element that satify the following Condition that better be explained by an graph

B : output Sorted array , size (B) = size (A) = n

Data structures used:

Range index (1 - n=6)

Ex) let A =

1	2	3	4	5	6
4	2	1	3	2	3

n = 6
K = 4

1. **Input:** $A[1..n]$ where $A[j] \in \{1,2,\ldots,k\}$.
2. **Output:** $B[1..n]$, which is sorted.
3. **Auxiliary:** $C[1..k]$ is temporary storage such that $C[i]$ will contain the number of elements less than or equal to key i.

C (Initially = Zero)

1	2	3	4=K
0	0	0	0

→ Range = (1-K)

(For J=1 to K
C[J] = 0)

we need # of element in A whose value is equal to the Range 1,...,K and place this Count in C

For J=1 to n
C [A[J]] = C [A[J]] + 1

← استخدمنا الـ index للـ (C) كـ تعبيرا indeses من (1-4) وضلينا نزيد كنه الفئة الكاملة موجودة (Zero)

C

1	2	3	4
0+1=1	0+1=~~1~~ 1+1=	0+1= ~~1~~ 2	0+1= 1

2 = Value الـ two element عندي
1 = Value الـ one element عندي

two element (3) = 2
two element (4) = 1
Value

C

1	2	2	1

→ we have one element in A that has a value of 4

↓ construct a Commultive densiry fun. CDF of C

The COUNTING-SORT Algorithm

← تقرأ اول تسعات بجميع بليوراه (الزاد Cell تكون عبارة عن مجموع الـ Cell بل مكل)

2+1, 1+2+2, 1+2+2+1

1	3	5	6

C

ثابتة أول واحد = (وافانا كان أول رقم موجود) بيه نزله زي ما كان

بأضافة الـ Current Value بالمجموع . لما الـ result هنا نفس اللي قبله مجموع النبر

= مجموع الـ Value اللي قبلها كلها

```
COUNTING-SORT(A, B, k)
for i ← 1 to k
    do C[i] ← 0

for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1

▷ C[i] now contains the number of elements equal to i
for i ← 2 to k
    do C[i] ← C[i] + C[i − 1]
▷ C[i] now contains the number of elements less than or equal to i
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
```

for J=2 to K → current value
C[J] = C[J] + [J-1] ; اللي قبلها

1	2	3	4
1	3	5	6

→ we have 6 element in A whose value are 4 or les

← كده ستيول بالـ (Reverse ord) عشان الـ Algo. يكون (Stable)

بيه الـ B = size A

for J=n to 1

B[C[A[J]]] = A[J] موقعه الصحيح

C[A[J]] = C[A[J]] -1

→ to a Count For duplicate element

(بعدين احنا نمر A) اشتغلنا بالعكس

بأخد الـ A(Value) ونشوف قيمته الـ (C) الـ Count بتحليلها الـ Position الصحيح في matrix (B)

A(n)

4	2	1	3	2	3

اول اصلا (3) عشان نكتبها في (A)

الـ (3) موجودة في (5) كـ ننزلها كده في B والعدد mapping في (C) + 3,6 نطرح من (C)

B(n)

1	2	3	4	5	6
1	2	2	3	3	4

C(k)

1	2	3	4
~~1~~ 0	~~3~~ ~~2~~ 1	~~5~~ ~~4~~ 3	~~6~~ 5

Time Complexity = Θ(n+k)

Space Complexity = Θ(n+k)

A[3] ⇒ C[3] = 5 ⇒ B[5] = 3, c--

A[2] ⇒ C[2] = 3 ⇒ B[3] = 2, c--

A[3] ⇒ C[3] = 4 ⇒ B[4] = 3, c--

A[1] ⇒ C[1] = 1 ⇒ B[1] = 1, c--

A[2] ⇒ C[2] = 2 ⇒ B[2] = 2, c--

A[4] ⇒ C[4] = 6 ⇒ B[6] = 4, c--

(ما استخدمت اي عمليات المقارنات اعتماد على Assumption بتاع الـ Algo.)

لما نعمل dec اكتر من مرة للـ C cells هنا عندنا (duplicates) نفس الـ Value

Final C

0	1	3	5

← مهمة جدا بالامتحانات

COUNTING-SORT **Example:** $A = \langle 4,2,1,3,2,3 \rangle$

(stable)

n = 6
K = 4

Initially A is:

1	2	3	4	5	6
4	2	1	3	2	3

n = 6
K = 4

After executing lines 1 though 4, C is:

1	2	3	4
1	2	2	1

After executing lines 6 though 7, C is:

1	2	3	4
1	3	5	6

C

1	2	3	4
0	0	0	0

C

1	2	3	4
1	$\cancel{1}$ 2	$\cancel{1}$ 2	1

C

1	2	3	4
1	$\cancel{3}$ 2	$\cancel{5}$ $\cancel{4}$ 3	$\cancel{6}$ 5

6 element
[illegible]

1	2	3	4	5	6
1	2	2	3	3	4

C = 0, 1, 3, 5 ([illegible])

Now we begin iterating over the loop in lines 9 through 11.

Iteration 1: $j = 6$, so place $A[6]$ into the correct position in B given C: $B[C[A[6]]] = B[5] = A[6] = 3$ and $C[A[6]] = C[A[6]] - 1 = 5 - 1 = 4$, so:

B is:

				3	

C is:

1	3	4	6

Time complexity = $\Theta(n+\underline{k})$

↳ Can we get rid (بنتخلص) of K in terms of A?!

Sol Yes, but through Best case : $K \leq n \rightarrow K = O(n)$
Av case : $K \in \Theta(n)$] $T(n) = \Theta(n)$ linear
Worst case : $K \gg n \rightarrow T(n) = \Theta(K)$

* نحصروا في Av + Best (linear)
حكا المشاكل في Worst .

Consider when $A =$

1	2	3
3	7	1000000

← much larger K than n

n = 3 , K = 1000000

$T(n) = \Theta(k)$

Radix Sort ← [That means we must improve this Algo.

COUNTING-SORT **Example:** $A = \langle 4,2,1,3,2,3 \rangle$ *continued*

Iteration 2: $j = 5$, so place $A[5]$ into the correct position in B given C: $B[C[A[5]]] = B[3] = A[5] = 2$ and $C[A[5]] = C[A[5]] - 1 = 3 - 1 = 2$, so:

B is:

				3	

C is:

1	2	4	6

Iteration 3: $j = 4$, so place $A[4]$ into the correct position in B given C: $B[C[A[4]]] = B[4] = A[4] = 3$ and $C[A[4]] = C[A[4]] - 1 = 4 - 1 = 3$, so:

B is:

		2	3	3	

C is:

1	2	3	6

Iteration 4: $j = 3$, so place $A[3]$ into the correct position in B given C: $B[C[A[3]]] = B[1] = A[3] = 1$ and $C[A[3]] = C[A[3]] - 1 = 1 - 1 = 0$, so:

B is:

1		2	3	3	

C is:

0	2	3	6

COUNTING-SORT **Example:** $A = \langle 4,2,1,3,2,3 \rangle$ *continued*

Iteration 5: $j = 2$, so place $A[2]$ into the correct position in B given C: $B[C[A[2]]] = B[2] = A[2] = 2$ and $C[A[2]] = C[A[2]] - 1 = 2 - 1 = 1$, so:

B is: $\begin{array}{|c|c|c|c|c|c|}\hline 1 & 2 & 2 & 3 & 3 & \\ \hline\end{array}$ C is: $\begin{array}{|c|c|c|c|}\hline 0 & 1 & 3 & 6 \\ \hline\end{array}$

Iteration 6: $j = 1$, so place $A[1]$ into the correct position in B given C: $B[C[A[1]]] = B[6] = A[1] = 4$ and $C[A[1]] = C[A[1]] - 1 = 6 - 1 = 5$, so:

B is: $\begin{array}{|c|c|c|c|c|c|}\hline 1 & 2 & 2 & 3 & 3 & 4 \\ \hline\end{array}$ C is: $\begin{array}{|c|c|c|c|}\hline 0 & 1 & 3 & 5 \\ \hline\end{array}$

The COUNTING-SORT **Algorithm** *continued*

If the elements of A are distinct, then $C[A[j]]$ contains the correct final position for $A[j]$ in b; however, if there are duplicates, then $C[A[j]]$ is decremented each time we move $A[j]$ into B, so that subsequent duplicates of $A[j]$ can be correctly placed in B.

COUNTING-SORT is **stable.** A sort is stable if equal keys remain in the same relative order in the sorted sequence as in the original input sequence. For example,

$\begin{array}{|c|c|c|c|}\hline 4 & 2_1 & 3 & 2_2 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline 2_1 & 2_2 & 3 & 4 \\ \hline\end{array}$ ($K \gg n$) السيئة الوحيدة

لذا Radix Sort

Complexity: COUNTING-SORT runs in $\Theta(n+k)$ time. Hence, it is not bounded by $\Omega(n \lg n)$. Note that it is not a comparison sort, so there is no contradiction.

Counting sort : Assumption: Input are integer in the Range ~~[illegible]~~ 1, ..., K (must be Known)
↳ This Algo. because problematic when of size n when K>>n
T(n) = θ(K)

Radix Sort

If k is very large, it may dominate the function $O(n+k)$. k can be reduced by combining COUNTING-SORT with RADIX-SORT.

The basic idea is to sort n d-digit numbers from the least significant digit to the most significant digit (where digit 1 is the lowest-order digit and d is the highest-order digit). This requires d passes with a stable sort; hence, COUNTING-SORT is an excellent choice.

* . انا نفضل نستخدم Counting Sort لانه (linear)

```
RADIX-SORT(A, d)
1. for i ← 1 to d
2.     do use a stable sort to sort array A on digit i
```

① 1. for i ← 1 to d
② (Counting Sort on digit d)

merg Sort → θ(n lg n)
Insertion Sort → θ(n²)
Counting sort → θ(n+K)
total → θ(d(n+K)) → θ(d(n²)) → θ(d(n lg n))
نضرب بـ (d) عدد الـ(digit)

على الـdigit نشتغل عليها

D : digit
4 ← 1
0 0 0 5
1 0 0 0
0 0 0 2
من الـ least Significal digit لـ most

A = | 5 | 1000 | 2 | (n = 3)

C = | | | K = 1000

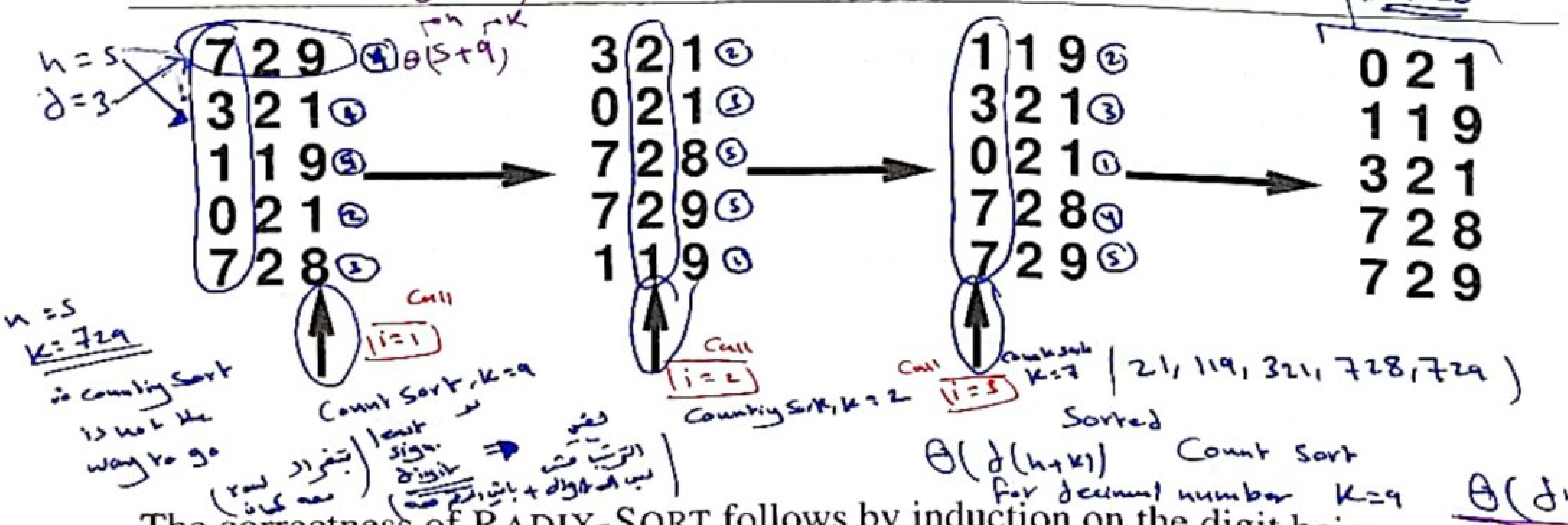
B = | 2 | 5 | 1000 |

اكل الـ تعبير الارقام رتعامل
digit نشتغل عليه
(ترتيب الـ digit بدل
ما ترتب الرقم كامل
انتقل digit - digit
مش مرتب الـ Counting Sort

decimal نقدر نستخدم في اي نظام حسب / Decimal : The max possible K=9

RADIX-SORT Example

θ(n+K) $\xrightarrow{\times 3}$ d(n+K) = 3(n+K) ⇒ linear Performance



n = 5
K = 729
∴ counting Sort is not the way to go

Count Sort, K=9

Counting Sort, K=2

Count sort K=7 (21, 119, 321, 728, 729) Sorted

(بتفرد least sign. digit ⇒ يعني الترتيب مش ...)
(لعد كما معه + digit الـ باشتغل)

θ(d(n+K)) Count Sort for decimal number K=9 θ(dn)

The correctness of RADIX-SORT follows by induction on the digit being sorted.

Why do we work from the lowest order digit to the highest? What would be required to work from high to low?

merg d n lg n
insert d n²

Correctness of Radix-Sort

loop invergent ماتتشوه

Base Case: If $d = 1$, there's only one digit, so sorting on that digit sorts the array.

Assume: The sort on the low-order $d-1$ digits correctly sorts the array if we ignore the higher order digits.

Inductive step: The sort on the dth digit will order the elements by their dth digit. Consider two elements, a and b, with dth digits a_d and b_d, respectively. There are three cases to consider:

1. If $a_d < b_d$, the sort will put a before b which is correct regardless of the lower order digits.
2. If $a_d > b_d$, the sort will put b before a which is correct regardless of the lower order digits.
3. If $a_d = b_d$, the sort will leave a and b in the same relative order because the sort is stable. Because the order was correct based on the $d-1$ lower-order digits, a and b are in the correct order.

Complexity of Radix-Sort

Counting-Sort is an excellent choice for the stable sorting algorithm used by Radix-Sort. In that case, $T(n) = \Theta(d(n+k))$. If $k = O(n)$, then the running time is $O(dn)$ for d passes of Counting-Sort.

Example 1: If we sort binary numbers in the range of 1 to n, $k = 2$ and $d = \lg n$, then $T(n) = O(n \lg n)$.

Example 2: Same scenario, except that we increase k and decrease d by grouping the bits into groups of r bits, so that $k = 2^r$ and $d = \lg n/r$. Then $T(n) = O(\frac{\lg n}{r}(n + 2^r))$.

Example 3: Taking $r = \lg n$ we would get $T(n) = O(n)$ and $d = 1$; we are just doing Counting-Sort.

Radix-Sort is also useful for sorting on multiple keys, where we treat each key as a "digit".

Bucket Sort

BUCKET-SORT sorts in linear time on the average by making a different kind of assumption than COUNTING-SORT.

مهم **Assumption:** input elements are evenly distributed over the interval $[0, 1)$.

A (uniformaly distributed) توزيع ① *** over the Range ②

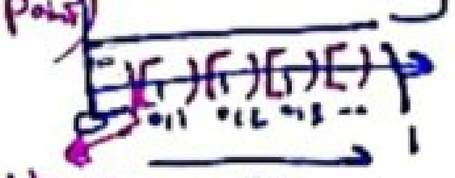
Data structures used: $A = \frac{A(\text{input array})}{\max(A)+1}$ (int → floating point) floating point

اكبر (اطلع ال max بقسم على كل الأرقام على (max+1))

floating point # لتصبح (floating point #) → (uniform function) Bucket sort

1. **Input:** $A[1..n]$ where $0 \le A[j] < 1$.

close open



(n: bucket) ٩٣ (0, n−1) $[\frac{i}{n}, \frac{i+1}{n})$

(n) Bucket

2. **Auxiliary:** $B[0..n-1]$ is a set of *buckets* corresponding to subintervals of $[0, 1)$ with which to classify and sort the elements of A. Each $B[i]$ is a pointer to a sorted list of $A[j]$'s that fall into bucket i. We place $A[j]$ into bucket $B[\lfloor nA[j] \rfloor]$, and bucket $B[i]$ holds $[\frac{i}{n}, \frac{i+1}{n})$.

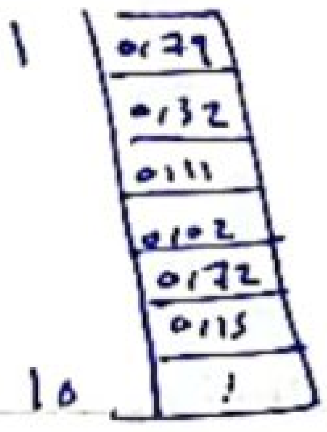
i=0 → i=n−1

كل bucket (i) by holds values to A

(Rang) → D

(توزيع ال #A Bucket

Given that the elements in A are evenly distributed over interval $[0, 1)$, the $B[i]$ lists should be short.



mapped

B [A4]

جزء → n

n=10 → 0.78 x 10 = ⌊7.8⌋ = 7

BUCKET-SORT Algorithm

```
BUCKET-SORT(A)
1. n ← length[A]

2. for i ← 1 to n
3.     do insert A[i] into list B[⌊nA[i]⌋]

4. for i ← 0 to n − 1
5.     do sort list B[i] with INSERTION-SORT

6. concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

going over the element in A (1, ..., n)

linear time

(linear time)

→ going over the Bucket (0, ..., n−1)

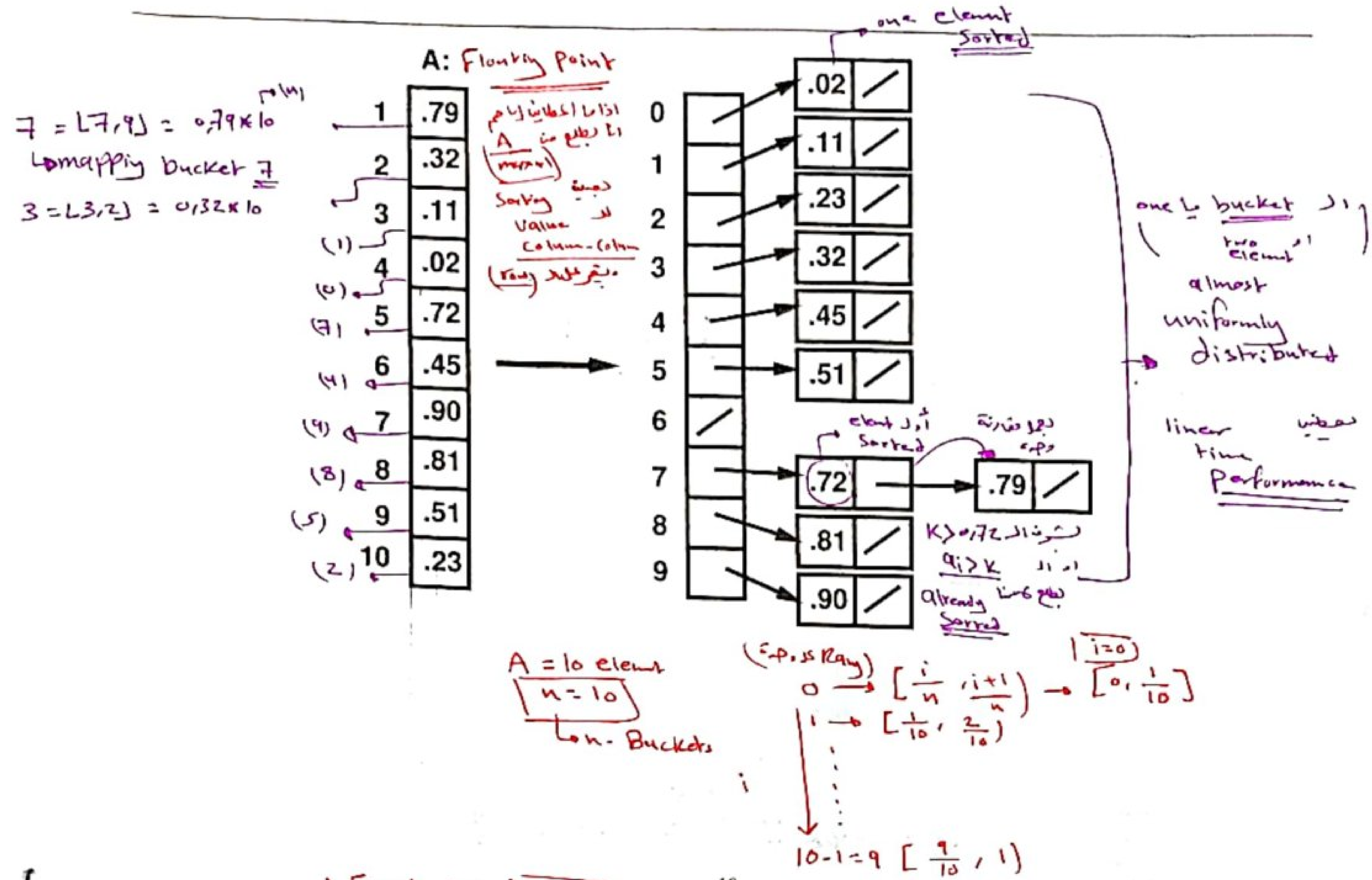
(linear time) (provided the condition) ⇒

Consider the uniformally distributed condition the insertion Sort work in Constant (كأنه بتنادي bucket واحد)

(Total linear time) Performance

linear خطي ⇐ Const x n ⇐ One element في bucket ال or two element

BUCKET-SORT Example



كيف اعمل mapping ?! ضرب كل K من عندك يا حزبه بـ (n) ⇐ n=10 \ , وبعدين ال ⌊Flor⌋

BUCKET-SORT Complexity

To analyze the running time, we observe that all steps except the INSERTION-SORT require $O(n)$ time. The question is how much time do the INSERTION-SORT calls require?

→ on AV + Best case

كل ال element تروح لل → one bucket

Bucket sort runs in expected time $\boxed{O(n)}$. The worst case is $\boxed{O(n^2)}$, however, as all numbers could conceivable land in the same bucket.

motivation ~~[illegible]~~ we need to have a support to performe Query and modification in Constant timing on AV

(data structure) عبارة عن

(dynamic data structure) المقصود هنا

Lecture 8: Search and Hash Tables

معاملات ال data ك operation (insertion OR Deletion OR Serching) وبيكون فيه هنا (other operation) نقسم ال operation لقسمين أساسيات

1 Modification → Ex) defined data structure by adding element OR insertion OR Deletion OR Inc/dec Key

2 ~~Quering~~ Query → Ex) Serching OR Return max/min

Course Learning Outcomes (CLOs):

- Use advanced searching techniques, such as hashing and 2-3 trees.
- Show how time and space complexities can be traded-off (e.g. distribution counting sort, hashing.

* لو الدنا عملية Serch لو ال element موجود في (array data structure) – قديش بدك وقت ?! لتبحث عن element

n : worst case + AV

1 : Best Case

(في Hash الهدف من الـ hash انه اعمل Serch بحيث يطلع Constant بدل linear في AV)

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

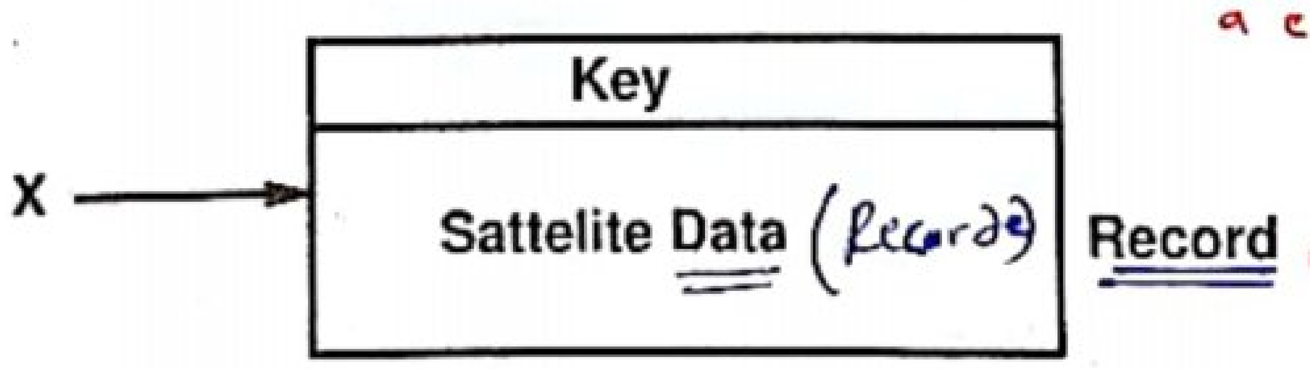
Review Chapter 10 and Read Chapter 11 of *Introduction to Algorithms*

Sets

set of element → static → we can't Add, remove element to the set.
→ dynamic → the size of the set can change due to insertion and Deletion.

A fundamental concept in computer science is the (dynamic **set**) which can grow and shrink over time. For example, a **dictionary** is a dynamic set that supports the insertion and deletion of elements, as well as a membership test.

Each element in a dynamic set is represented as an object whose fields can be examined or modified if we have a pointer to the object. Elements often have a **key field** as well as **satellite data**. For example:



Ex. of a dynamic set: Consider a set of user names to log in to a certain machin

each user name must contain 10 small alphabetic letters

Consider the size of the set to be 100

[illegible]

The best way to implement a set depends on the operations that are supported on the set. It is also important to consider the number of elements that will be stored in the set S, $|S|$, and compare it to the size of the universe U, $|U|$ (the set of elements that could conceivably be in S).

(بمعنى)

Operations on Sets

Operations on sets are either queries or modifying operations. The running time of these operations is reported in terms of the size of the set.

- SEARCH(S,k): a query that returns a pointer x (Record البيانات) to an element in S such that $key[x] = k$ or NIL لا if there is no such element in S. اذا مش موجود
- INSERT(S,x): a modifying operation such that $S \leftarrow S \cup \{x\}$.
- DELETE(S,x): a modifying operation such that $S \leftarrow S - \{x\}$.
- MINIMUM(S): a query on a totally ordered set S (اذا شرط مرتبين) such that a pointer to the element with the smallest key in S is returned.

Operations on Sets *continued*

- MAXIMUM(S): a query on a totally ordered set S such that a pointer to the element with the largest key in S is returned. (اذا مرتبة تصاعدي)
- SUCCESSOR(S,x): a query on a totally ordered set S (مرتبة) مهم such that a pointer to ترجع عدد لا ترجع المؤشر مهم لا K_x < element ود the smallest element in S greater than x is returned, unless x is the maximum, in which case NIL is returned. This can easily be extended to sets with duplicate keys.
- PREDECESSOR(S,x): a query on a totally ordered (Sorted) مهم set S such that a pointer to the largest element in S less than x is returned, unless x is the minimum, in which case NIL is returned. This can easily be extended to sets with duplicate keys.

Set Examples

- The login table for a computer system with 100 users where logins are of length 10 and consist of lower.case alphabetic characters; $|U| = 26^{10}, |S| = 10^2$. Support INSERT, DELETE, and SEARCH.

↳ Subset
مثال 100

universe | Set

$|U| \gg |S|$

$S \subseteq U$
(S) Subset من (U)

$\underline{26}$ (lower case letters) | (Size) 100

26^{10}

ال (100) حاجات من 26^{10}

*لو ما كبرت ال |U|: ح نصعب عليه احزر

↳ small + Cap ⇒ 26+26 = 52
↳ numerical ⇒ 10 +
↳ special char. ⇒ 10 +
mimal length is 14

= 52+10+10 = 72

$|U| = 72^{14}$

Implementation Issues for Dictionaries

array بحالة ؟

One way to implement a dictionary is as a linear list. In this case, SEARCH can take $\Theta(n)$ time in the worst case, and at least one of the INSERT or DELETE operations can take $\Theta(n)$ time in the worst case.

One way to implement a dictionary is as a heap. In this case, the INSERT and DELETE operations can take $\Theta(\lg n)$ time in the worst case. What about the running time of SEARCH?

We would like to use a datastructure that supports efficient dynamic set operations. In particular, we would like the INSERT, SEARCH, DELETE operations to take $O(1)$ time.

اي Key باخده مباشرة لتخزينه
بعبر ال index Table

The Direct-Address Table

To represent a dynamic set, we can use a direct-address table $T[0..m-1]$ where each slot corresponds to a key in the universe and:

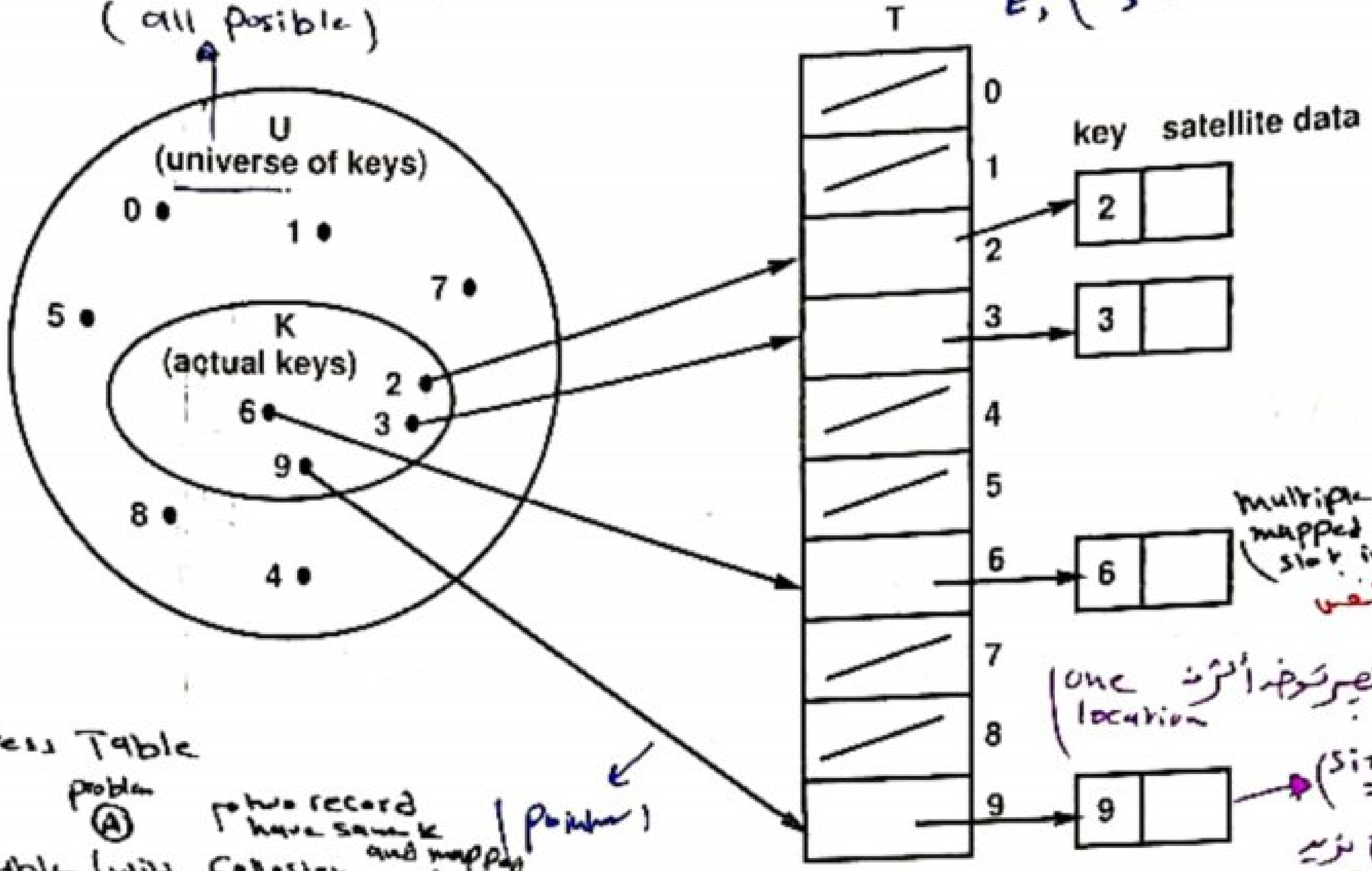
$$T[i] = \begin{cases} x & \text{if } x \in S \text{ and } key[x] = i, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

k: index بعبر للـ dah (Table)

يعني 2 بـ (slot 2) يروح عليه و 3 بـ (slot 3) الخ

(0,.....,m-1)
Size of the Table = m
Size of the Key = n
$\alpha = \frac{n}{m}$ = (load factor)
n>m , α>1 (over flow)

(all posible)



المشاكل اللي ممكن تحصل
1 duplicat (multiple Key mapped same slot in table)
(أفرض Records يكون لهم نفس ال Key) ⟸ اكثر (تحتاج لو أفترض one location)

المفروض كل واحد ال (size =1)
لكن اذا اصبح duplicat ح يزيد في (direct Table)

2 ال size K يكون اكبر من حجم ال Table
الحل اضافة element ح يحصل overflow (m<n)
(Key اكبر من size Table)

Direct Address Table
حل مشاكل ال direct
Hash Table (with Collosion Avoidance stratiges)
1 chaining (سلسلة)
2 open Addresing

problem A two record have same k and mapped same slot in direct address Table (Pointer)

problem B # of K is larger than the Direct Table size

(subset) بحجم ال Table * بالبداية تكون (empty)

6

(الحل insertion لازم يكون Constant)

(duplicat) تجي اليه ح يجرني

The Direct-Address Table *continued*

All of the following operations on a direct-address table are $\boxed{O(1)}$:

صلاحيات 1+2 بروح لـ Table
سطل اسمه direct address Table
بصير اسمه (Hash Table)
Can support duplicat

```
1. DIRECT-ADDRESS-SEARCH(T, k)
       return T[k]   → constant

2. DIRECT-ADDRESS-INSERT(T, x)
       T[key[x]] ← x

3. DIRECT-ADDRESS-DELETE(T, x)
       T[key[x]] ← NIL   → للـ Pointer
```

اذا صار duplicat في two method تخزين لتفادي Collosion
→ two Key mapped Same slot

(استبدال ال Pointer بـ Table)

Direct Addressing is a simple technique that works well if the universe of keys, $U = \{0, 1, 2, \ldots, m-1\}$, is relatively small and no two elements have the same key.

Hash Tables

from 0 to m-1

Problem: usually the range of keys is much larger than the desired table size m (e.g., ASCII strings).

(over flow + duplicat) ⇒ collision (Problem)

لو تجمعوا الـ عدد (Key) > عدد الـ size (Table) الالانه أكبر الـ Table (لعبا بنفس الوقت ما بقدر ، انه وصول لهم ليكونوا duplicat)

(mapped sorting) A **hash table** is a generalization of an ordinary array which supports the direct addressing of an element in the table in $O(1)$ average case time, although the worst case time is $\Theta(n)$.

→ (كل الـ element موجودين في نفس الـ index)

صح ≅ تقريبا ≅ (usually) → doubt

The idea is to store keys in a large hash table of size m (say $n \leq m \leq 2n$, where n is the number of keys to store) such that the data is scattered across the table uniformly. (→ Bucket Sort)

نحاول ما نوقع في مشكلة الـ Collision

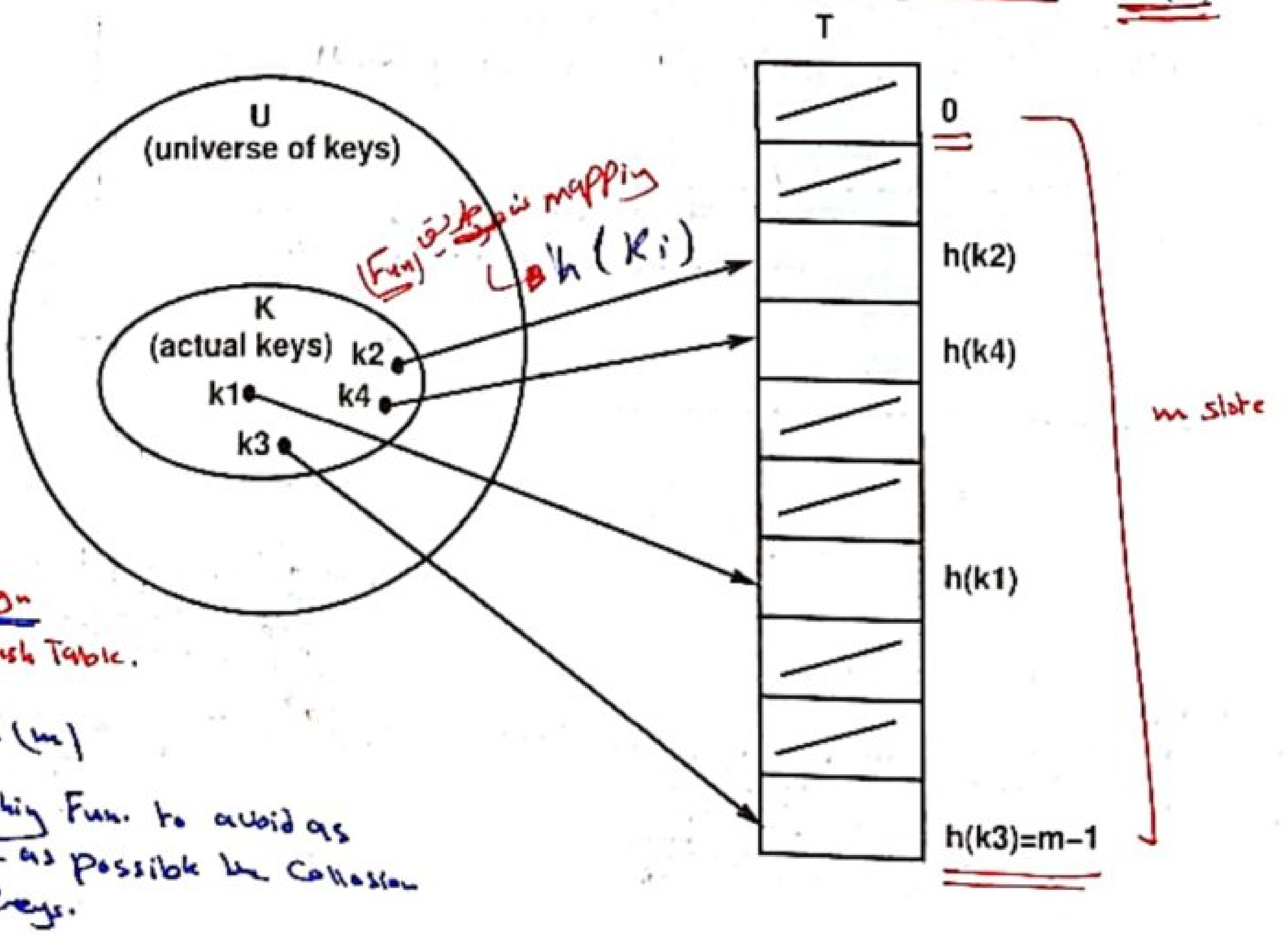
The INSERT, SEARCH, DELETE operations on a hash table take $O(1)$ average case time, as we shall see.

Example: A symbol table in a compiler is often represented using a hash table, where each element key consists of the string of characters to store in the table.

Hash Tables *continued*

Computable in constant time O(1) → hash Fun. (بحطها بـ الـ slote الي الها) / mapping {0, ..., m-1} → one of the hash Table Slots

The element with key k is stored in slot $h(k)$ given that h is a function which maps a key $k \in U$ to an index of hash table $T[0..m-1]$. Note that $h: U \to \{0, 1, \ldots m-1\}$ given that m is the size of the hash table. h should be computable in $O(1)$ time.



two important design parameter to the hash Table.

1. Hash Table size (m)
2. choise of the hashing Fun. to avoid as reduce as much as possible the Collision of multiple keys.

Hash Tables *continued*

(مثلا الـ Password) مثال

If K represents the set of keys to store and $|K| << |U|$, then, by using a hash table, we can reduce the storage space for T to $\Theta(|K|)$ and still access the elements in $O(1)$ average case time.

حجم الـ Table
بحكي للـ K مش الـ (U)

Hashing schemes perform very well in practice when the maximum number of keys is known in advance and can be implemented in fewer than 200 lines of C or C++ code!

Problem: A **collision** occurs if $\boxed{h(k_1) = h(k_2).}$

two key same hash value
نفس الـ hash
طلعت متشابهات

It is best to avoid collisions altogether by obtaining a suitable hash function that minimizes the number of collisions. However, if $|K| > m$, there will be keys that hash to the same index; hence, collision avoidance is not possible. ما في مفر من (Collision)

Fortunately, there are effective techniques for resolving the conflicts created by collisions.

اذا الـ n يزيد $(n < m \leq 2n)$

اذا كثير الـ (Lots of Deletion) بنصح باستخدام الـ (Chaining)

اذا الـ Deletion عدده قليل بنصح باستخدام الـ (open addressing)

Collision Resolution Methods

There are two methods for resolving collisions: linked: (Pointer)

لـ بيرجع طريقة ادخل الـ element + الثاني طريقة احل الكو ماحيم
ما في Collision بتتعاملي لـ (duplicat) → our hope (linked list يكون حجم الـ بكل slot قليل) → عشان نضمن يكون on AV we have Constant Performance

1. **Chaining:** Keep a linked list of the elements that hash to the same index. This method is simple to implement and degrades gracefully when the number of keys n exceeds the table size m. It uses dynamic memory allocation.

(K>m) بيصير اسوء الـ Performance

worst Case
ح تحصل الـ case يكونوا الـ mapped same slot بالتالي ح يكون عندي one linked list

Bucket Sort

$(n < m \leq 2n)$: m (Prime #) ما بقدر الـ كيف نحدد؟ نختار الـ Table size اداة لكي نضمن حجم الـ Keys

$|m > n|$ long Table (element اثر العدد Collision غشان لو صار من بعضها (1 او) نقصه)

ادل Double تبعه

2. **Open Addressing:** Store all keys in the table itself, and if a collision occurs, then use some method to obtain another location to place the item in the table. This is easy to implement and no dynamic memory allocation is used; however, n must be less than or equal to m, performance degrades as $n \to m$, and deletion operations are difficult to support, as we shall see.

(اذا ما في Pointer)

$(n \leq m)$ load factor
يقل Performance — (m) تقارب n(Keys)

لازم حجم الـ Table الكبير يساوي حجم Keys

Serch اي عملية عليها on AV برضه

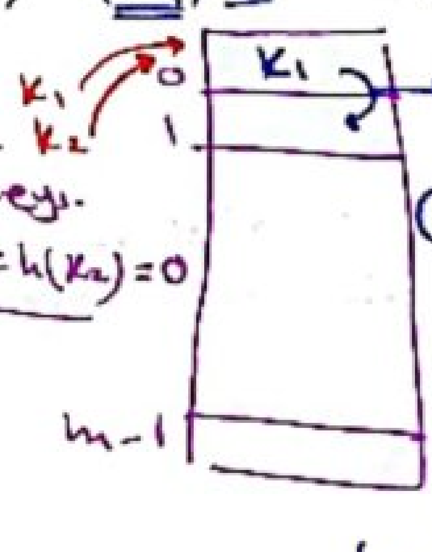
(Delet marker) (الحل) مشكلة اذا حذفت K_1 ما نطابه K_2 خارج بيدلي فيه كان حذفت K_1 (NIL)

(linear timing)

(أكبر الـ Table عشان الحل المشكلة)

[1] insert $K_1 \to 0$
[2] insert $K_2 \to$ ح على نفس المكان بس المكان بتخزن one slot → use a probing method to find and empty location to stor it must be systematic and
شاركنا (Collision) ← شارك الـ
(hashing الـ نفسه) الـ ←
$\boxed{\frac{n}{m} \leq 1}$

Assume we have two keys. K_1, K_2
$h(K_1) = h(K_2) = 0$
K_1, K_2



اذا فيه عد next location

Possible Probing Sequence
① one possible method to go linearly (what is the max. # of probing sequences to store a key? m) (m: i=0 → m-1) (sequence)
② quadric
③ double hashing

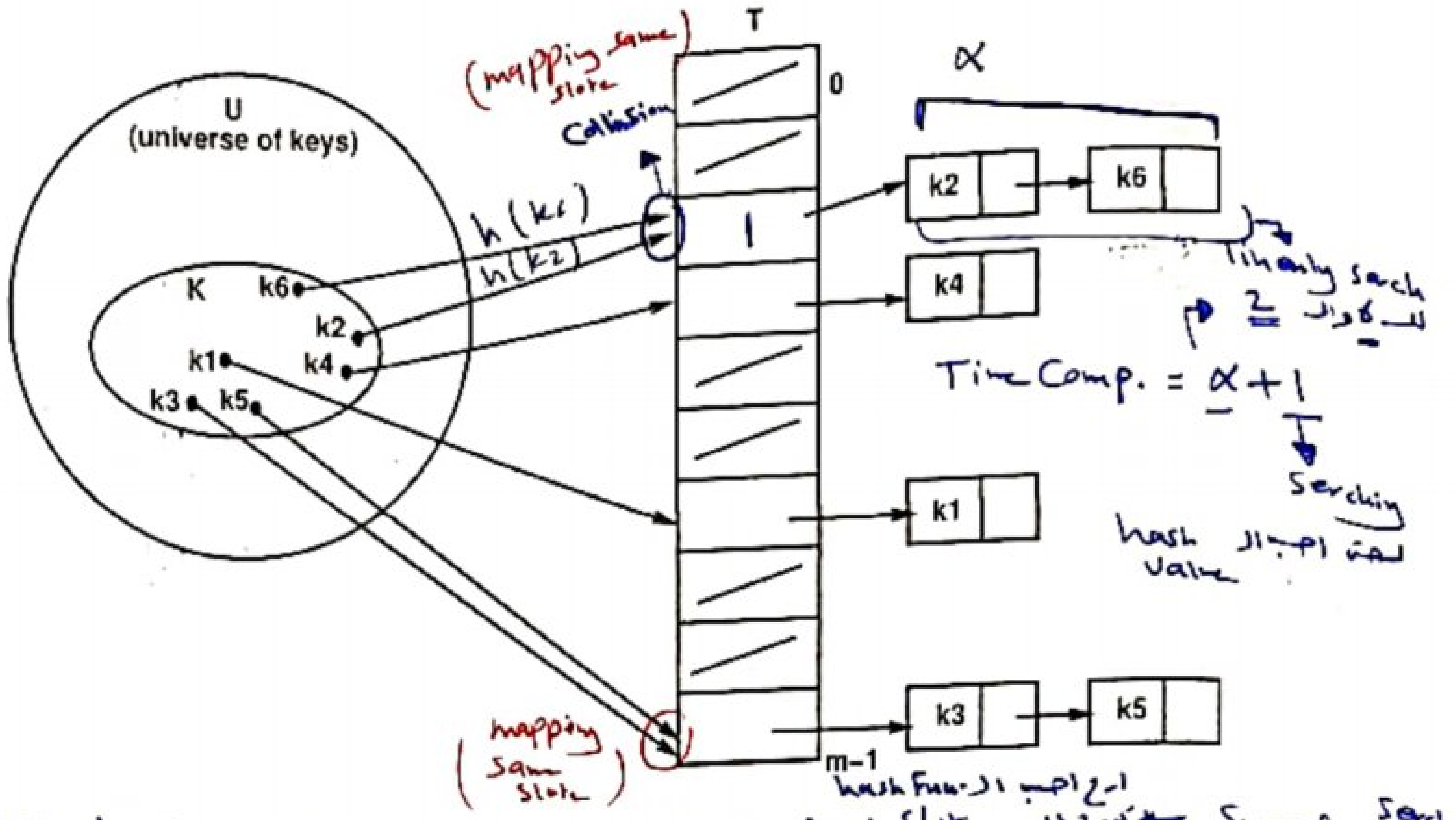
$h(K, i) = h(K) \bmod m$
لـ اذا كانت اختفاء (فارغ)

linked list ؟ ما هي شكل الـ ؟ chaining ؟ شو الـ
لازم نستخدم method to obtain another location

α = load factor = $\frac{n}{m}$ / m at least larger or equal to n
α in open addressing most be (1 or less)
عدد الـ element لـ

Collision Resolution by Chaining

The simplest collision resolution technique is called **chaining**. The basic idea is that all items that hash to the same index are stored in a linked list pointed at by that table entry.



اروح احسب الـ hash Func

A = Θ(1+α) / linked list الـ احتمال Value الـ موجودة فيه Slot لـ تروح ← Sucssful Serch Stat

(linked liste فيه element تروح عـ Slot المفروض يكون موجودة في [illegible] ~~[illegible]~~ linkedliste وأتكون آخر ← in Sucssful

AV = Θ (α+1)

Fun:

Collision Resolution by Chaining *continued*

The operations on the chained hash table are defined below:

1. CHAINED-HASH-SEARCH(T,k) (اذا موجود بترجعه) (بـ Chaing اذا كان linked list بدك تعمل Serching داخل الـ linked linearly list)

   search for an element with key k in $T[h(k)]$ → key = k / Best (حجم الـ linked list بيكون قليل)

   The worst case running time is proportional to the length of the list. (α)

   n = linked liste لـ كودال Case راح على طول الـ

2. CHAINED-HASH-INSERT(T,x)

   insert x at the head of the list $T[h(key[x])]$

   The worst case running time is $O(1)$.

3. CHAINED-HASH-DELETE(T,x) → ادور عليه بالخانية بالأول واحطه Nil

   delete x from list $T[h(key[x])]$ (الـ pointer اكثر من الحذف اسهل بلا حساب الـ space)

   The worst case running time is $O(1)$ if the list is doubly linked. (double pointer) If singly linked, it must search for x's predecessor to perform the operation.

   → ادور عليه + الغي (1 + α)

لـ الغي الـ pointer بشيلة

An Example of Hashing with Chaining

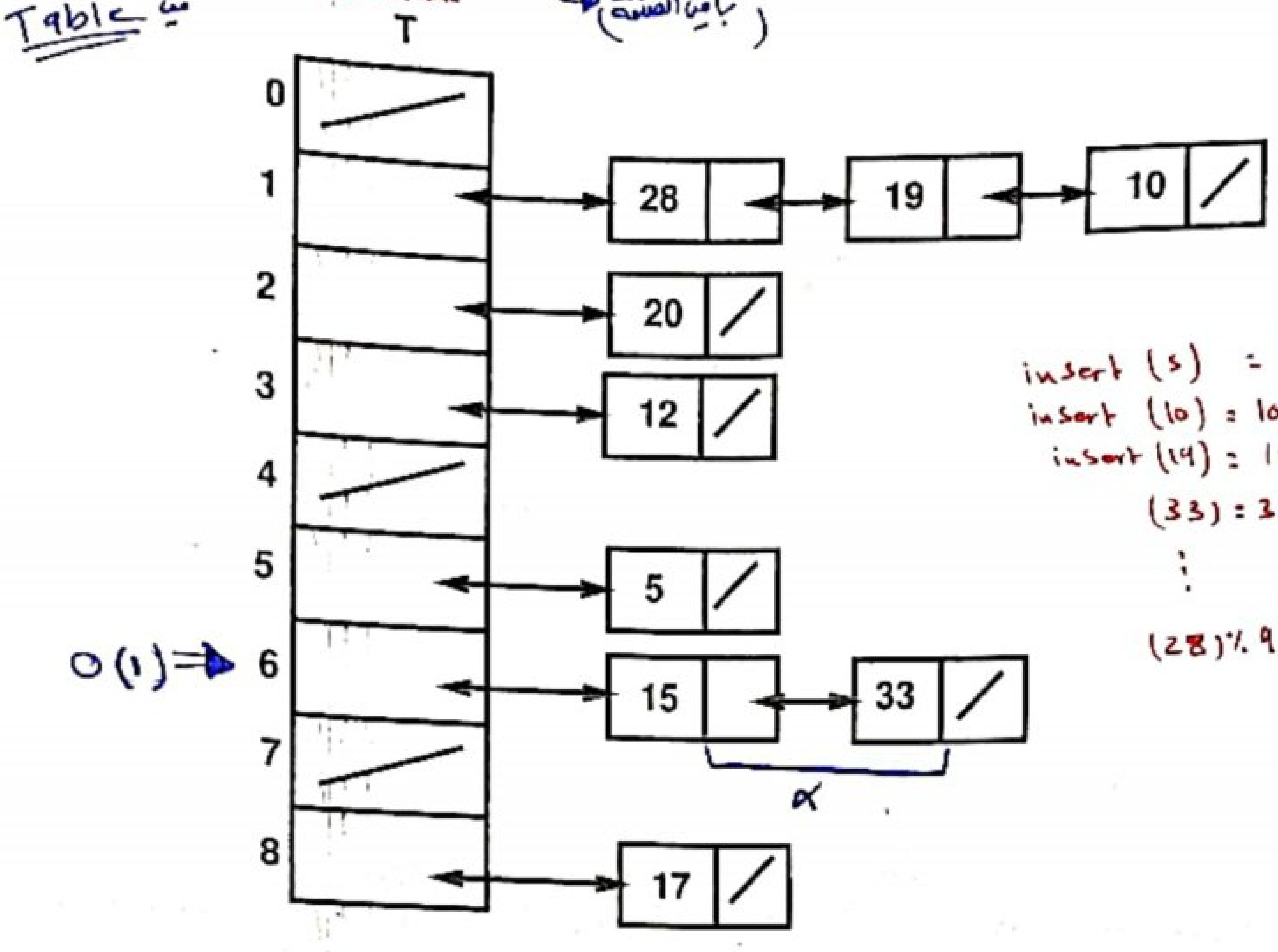
Example: $m = 9, h(k) = k \bmod 9, K = \{5, 10, 19, 33, 20, 15, 12, 17, 28\}$

(0-8)

slot ل عدد ال
Table في

Hash Fun.
باقي القسمة ( )

inserting لطبق
insert كود ال



h(K) = Kmod 9

insert (5) = 5%9 = 5
insert (10) : 10%9 = 1
insert (19) : 19%9 = 1
(33) : 33%9 = 6
⋮
(28)%9 = 1

Analysis of Hashing with Chaining

The worst case running time to search for an element in hashing with chaining is $\Theta(n)$, when all keys hash to the same slot.

simple uniform hashing $\boxed{\alpha = \frac{n}{m}}$, Av size of each linked list = α (load factor)

hash table من ال $\Theta(1 + \alpha)$ length list

linked list ال length تزيد performance

The average case depends on how well the n keys are distributed among the m slots.

Given a hash table of size m and n elements to store in the table, we define **load factor** as $\alpha = \frac{n}{m}$. Our analysis will be in terms of α.

نتعامل مع (Chaining) فقط

(any key is equally likely to hash any of the m slots of the hash Table)

For analysis, we will make the assumption of **simple uniform hashing**, which means that any given element is equally likely to hash into any of the hash table slots, independently of the other elements.

prob. [1 key mapped to a one of the n slots] = $\frac{1}{m}$

Compare with (uniform hashing assumtion)
→ open addressing نتعامل مع

Analysis of Hashing with Chaining *continued*

Theorem: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time $\Theta(1+\alpha)$, on average, under the assumption of simple uniform hashing.

Proof:

- The assumption of simple uniform hashing implies that any key k is equally likely to hash to any of the m slots. $(\frac{1}{m})$
- The average time to search unsuccessfully for key k is the average time to search one of the m lists.
- Given that the average length of a list is $\boxed{\alpha = \frac{n}{m}}$, the expected number of elements to examine in an unsuccessful search is α.
- Hence, the total time required is $\boxed{\Theta(1+\alpha)}$ (including the time to compute $h(k)$).

chaining

Analysis of Hashing with Chaining *continued*

Theorem: In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on average, under the assumption of simple uniform hashing.

Proof: See the textbook.

Hash Table Size

The following are guidelines on the selection of hash table size, m: [illegible] usually

- Select m as a prime number or power of two. In theory the prime is slightly better (more choice when picking a hash function and universal hashing can be used more easily), but in practice we tend to choose $m = 2^k$ (eliminates the need to do modulo operations).

- If using chaining, use $n \leq \boxed{m} \leq 2n_{max}$, where n_{max} is the maximum number of keys to be stored in the table at one time. Nothing hinges on a particular choice. [illegible] (n)

- For open addressing with double hashing, use $\frac{3}{2}n_{max} \leq \boxed{m} \leq 2n_{max}$. Larger values result in better performance. أكبر مرة ونصف (n)

two design any [illegible] hashTable → $n \leq m \leq 2n$; $\frac{3}{2}n \leq m \leq 2n$

Hash Functions

Often keys are similar or clustered, but we do not want this regularity of key distribution to affect uniformity of the hash function. $h(x)$ and $h(x+\varepsilon)$ should differ, where ε represents some small change (e.g., integer, real, or string).

A good hash function should come close to satisfying the assumption of simple uniform hashing. More formally, assume that each key is drawn independently from U according to the probability distribution P, where $P(k)$ is the probability that key k is drawn, then:

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad j = 0, 1, \ldots, m-1$$

Unfortunately, P is generally unknown (though sometimes it is, for example, random reals in the range $[0..1)$).

Usually heuristic techniques are used to create hash functions that perform fairly well. Sometimes stronger assumptions are required than simple uniform hashing.

Most hash functions assume that the universe of keys is the set of natural numbers (or that they can be mapped to natural numbers).

Hash Functions *continued*

① **Division method:** $h(k) = k \bmod m$

فقط
ع النظري

Example: If $m = 52$ and $k = 235$, then $h(k) = 27$.

Rules of thumb:

- Avoid powers of 2 as the value of m; otherwise, not all bits of the key will be used by the function. If $m = 2^p$, then only the lowest order p bits will be used!
- Avoid powers of 10 when decimal numbers are used as keys.
- If $m = 2^p - 1$ and k is a character string interpreted in radix 2^p, then keys that are permutations of the same digits hash to the same slot! For example, 51 in radix 2^4 ($5 \times 16^1 + 1 \times 16^0 \bmod 15 = 6$) collides with 15 in radix 2^4 ($1 \times 16^1 + 5 \times 16^0 \bmod 15 = 6$) when $m = 15$. (See 11.3-3.)
- Good values of m are primes not too close to a power of 2.

  Example: If $n = 2000$, and we are willing to examine 4 elements on average during search, then $h(k) = k \bmod 491$ is a good choice (512 is the power of 2).

Hash Functions *continued*

Multiplication method:

Refer to the book for further details on this method.

Open Addressing

(lots of Pointer) ماعنده → (zero pointer)

Open Addressing: All elements are stored directly in the hash table (i.e., there are no pointers); hence, either there is an element or NIL in a table entry.

(Serch) Detection قدر اللي فيه بوجود الـ (delete marker) أو Nil Pointer بعد delet ←

The hash table is now a static entity that can fill up, so $\boxed{\alpha \leq 1.}$ This method trades pointers for table size. (برنامج كبير Page Table الاستخدام Pointer) $\frac{3}{2}n$ $\leq m \leq 2n$ لانه مفهوم انه كبرنامج Table لما

To perform a table insertion, we now **probe** the hash table for an empty slot in some systematic way. (→ linear, quadric) Instead of using a fixed order; however, the sequence of positions probed depends on the key to be inserted.

→ probing sequence ID (اول ID تبدأ منه الـ ...)

$h(k,i) = h(k+i) \bmod m$

Total (m Sequance)

(m-Sequance) linear

(m-probing Sequance) Quadric

m^2 possible probing seq. double hashing (اساس) Lo it uses two hashing Fun.

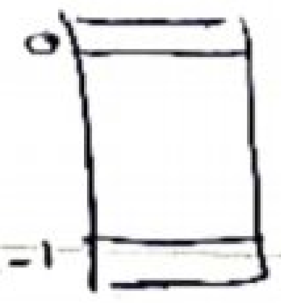
The hash function is redefined as:

$$h : U \times \{\underbrace{0,1,\ldots,m-1}_{\text{(Trials)}}\}^{(i)} \rightarrow \{0,1,\ldots,m-1\}^{(\text{Slot واحد})}$$

For every key k, the **probe sequence** $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ is considered. As the table fills up, every position in the table is a possible option for the insertion of a key. (اذا مشت عليهم كلهم هتلاقيه) Slot ID (i=0 ... m-1)

means ⟸ empty slot

(Table Full (k) / (element عدد الـ / slote عدد الـ)) $1 \leq \alpha$ ∴ will examing all the probing sequence empty slote ملاقاش



problem ① (primary clusetring)

$h'(k) = k \bmod m$ ← متاخد ليه

$i = [0, \ldots m-1]$ ⇐ $h(k,i) = (h'(k) + i) \bmod m$ linear probing

$i = [0, \ldots, m-1]$ ⇐ $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ (quadric probing)

Trials index

Proble ② (Secundy clustering)

Open Addressing *continued*

اكثر انتشار ③ Double Hashing (two differ hash fun.)

$h_1(k) = k \bmod m$] → Design Parameter ← (انا هنا اختار ما يبقى)

$h_2(k)$

$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$, $i = [0, \ldots m-1]$

We assume that we are storing only keys in the table; there is no satellite data; hence, a table slot either contains a key k or NIL.

(not empty) To empty

$\overline{n=3}$

$m_0 \; m_1 \; m_2$

$m_0 <$ $m_1 - m_2$ / $m_2 - m_1$

$m_1 <$ $m_0 - m_2$ / $m_2 - m_0$

$m_2 <$ $m_0 - m_1$ / $m_1 - m_0$

$3! = 6$

```
HASH-INSERT(T, k)
  i ← 0                          ← دائما كل insertاله new Key = 0
  repeat j ← h(k, i)             ← hash value للوصول ← loop (اخر عشر كنت ما لقيتش empty = Nil  or  i = m)
      if T[j] = NIL              ← Nil أو Delet marker  (empty: OR T[j] = Delet marker)
          then T[j] ← k          insert الـ بعد Search لـ ما عليها مش فرق بكثير
               return j
      else i ← i + 1             ← اذا مش empty  (next probe / اروح لـ)
  until i = m                    → Table is full  α ≥ 1
  error "hash table overflow"
```

لم جربت كل possible seq.

(linear probing مقطع الـ مخصص ...)

($1 \geq \alpha$) هتشتغل لما مش كده ←

# of Possible Permutation of the m slot $= m!$ possible probing seq.

linear ① } مش هيطلع (m! من m)

quadric ② }

Double ① ← m^2

linear only acheam (m) out of the m!

$\text{prob}[\;] = \frac{1}{(m-1)!}$ $\boxed{\frac{m}{m!}}$

Open Addressing *continued*

HASH-SEARCH uses the hash function to search for key k, and it will terminate if it finds k in the slot, returning the index where the item is found. It also terminates with NIL if it finds NIL in a probed slot or if it searches the entire table without success.

insertion تل
برجعلها اذا ما لقى
Nil بدور عن K

(Deletion) مشكلة في *
Serch موجود؟ مش

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k, i)
        if T[j] = k
            then return j (index)
    i ← i + 1
  until T[j] = NIL or i = m
return NIL
```

بحسب ال hash value

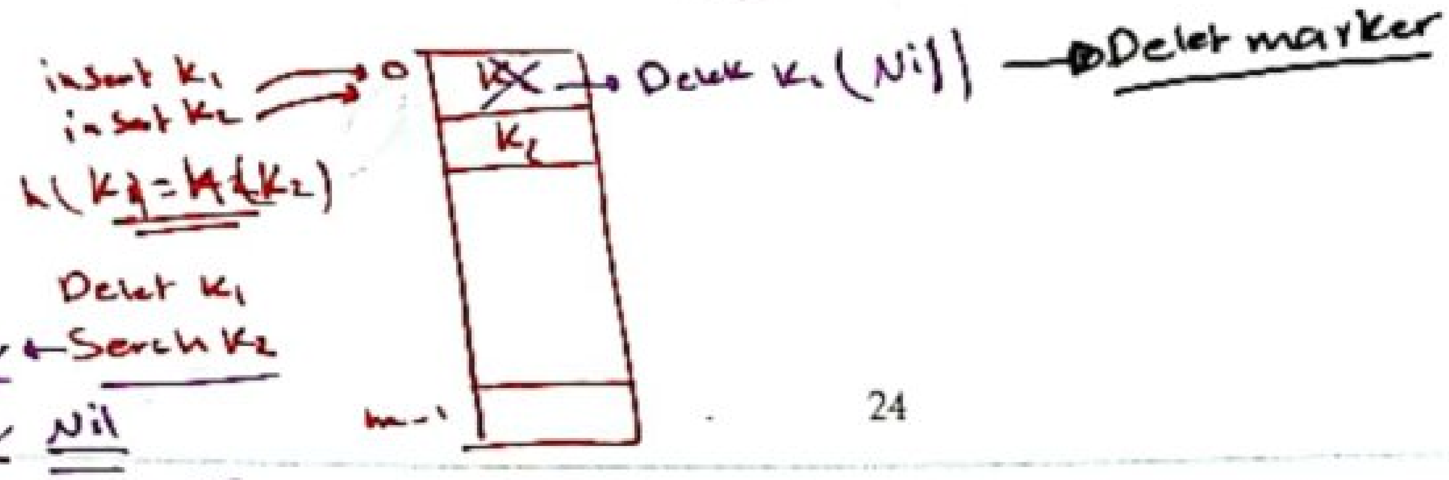
لقيت ال K بـ Table slot j

(Probe seq.) اذا لا اذا زيد

حل ال Deletion صعب (Deltion marker)

(بوقف)

مش موجود



h(K1=h(K2)

Delet K1
Serch K2 → رح تودينا ل (0)، الاول
رح اتوقف وهاي مشكلة Nil
عن K2 موجودة مش Nil
اكلا لبس انقر Delet بحط Nil حط
Delet marker

Deleting in an Open-address Table

We cannot simply delete an element in a table by inserting a NIL; it could break the probe sequence for other elements in the table.

① حل

It is possible to get around this by using pointers, which goes against the spirit of open addressing.

② حل

We could also use a special purpose delete marker instead of NIL when an element is removed.

Then HASH-SEARCH will automatically search further when looking for other elements when encountering the delete marker. HASH-INSERT can be modified to treat the delete marker as NIL. However, using this approach, the search time is no longer dependent only on the load factor α.

→ Because deletion in open-address hash tables is difficult (صعب احتماله), usually open-address hashing is not done when delete operations are required.

اذا عندك lost of deletion بفضل تستخدم ال (Chaining) بزعل (open address) للتبحصم

Probe Sequences

In the analysis of open addressing, we make the assumption of **uniform hashing,** which requires that each key considered is equally likely to have any of the $m!$ permutations of $\{0,1,\ldots,m-1\}$ as its probe sequence.

مقارنات simple مع

Porb. [ ] = $\frac{1}{m!}$

Three techniques are commonly used to compute probe sequences for open addressing:

$\frac{1}{m!} \neq \frac{m}{m!}$ (not equal)

1. linear probing   m out of m!
2. quadratic probing   m out of m!
3. double hashing m^2 out of m!   (m^2) افتراضية

These techniques guarantee that $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ is a permutation of $\langle 0,1,\ldots,m-1\rangle$ for each key k, but none fulfills the assumption of uniform hashing since none can generate more than m^2 sequences.

Linear Probing

Given $h' : U \to \{0,1,\ldots,m-1\}$, **linear probing** uses the hash function: $h(k,i) = (h'(k)+i) \bmod m$ for $i = 0,1,\ldots,m-1$.

كون i=0 من البداية   تاني عادة   تالت عادة

Given key k, the first slot probed is $T[h'(k)]$, then $T[h'(k)+1]$, $T[h'(k)+2]$, etc. Hence, the first probe determines the remaining probe sequence.

Example: $h'(k_1) = h'(k_2) = j$, $h'(k_3) = j+1$, $h'(k_4) = j$, and the keys are entered in the order: k_1, k_2, k_3, k_4, giving the following table:

$h(k_1) = J / h(k_2) = J / h(k_3) = J+1 / h(k_4) = J$

hash Table (open addressing)

1	2	3	4	5	6	7	8
K_1	K_2	K_3	K_4				
J	J+1	J+2	J+3.	J+4	J+5	J+6	J+7

m = 8

insert 4 Keys

$K_1 \to$ i=0
K_2 i=1 — one Colosion
K_3 i=1 — one Colosion
K_4 i=3 (problem)

Compute (0=K) الـ ... الدائرة

$h(k,i) = [h'(k)+i] \bmod m$

$K_1 = h(k_1)+0 \to K_1 \to 0$ empty

$K_2 = h(k_2)+0 \to K_2 \to 0$ not empty

(i=1) رح نجرب بـ

$K_3 = h(k_3)+0 = J+1$

Collision i=1 → i++ (i=2)

Colosion i++

This method is easy to implement, but suffers from **primary clustering**, that is, two keys that hash to different locations compete with each other for successive rehashes. Hence, long runs of occupied slots build up, increasing search times.

$K_4 = h(K_4)+0 = J$

Colosion (i=1)
i++
Collosion (i=2)
i++
Collosion (i=3)
i++
empty ∴ insert

Colloston 3=i

K_2, K_3

$h(k_2) = J$ $h(k_3) = J+1$

ودايماً بدو نفس المكان اضطرت اقارنه ببعض انتقل للي بعديه رح يعمل شاغل

Quadratic probing

Solved the Primary clustering problem but faced another Problem secondary clustering.

Quadratic hashing uses a hash function of the form:

شالت مشكلة linear اللي هي الـ (Primary)

$p0 = c_1$, $p(1) = c_2$

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where h' is an auxiliary hash function, $c_1, c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$. Note that m, c_1, and c_2 must be selected carefully.

The initial probe position is $T[h'(k)]$, with subsequent positions depending on a quadratic function of the probe number i.

Example: probe sequence given $c_1 = c_2 = 1$: → $i + i^2$

	J	J+1	J+2	J+3	J+4	J+5	J+6	J+7	J+8	J+9	J+10	J+11	J+12	J+13
	p_1	.	p_2				p_3						p_4	

Quadratic probing is better than linear probing because it spreads subsequent probes out from the initial probe position. However, when two keys have the same initial probe position, their probe sequences are the same, a phenomenon known as

مشكلة

secondary clustering.

Solving this problem Double (Hashing)

resolved

{ P1, P2, P3, P4 }

4 keys to be inserted in an open addressing hash Table in which Colosion are ~~Sorted~~ by Probing $c_1 = c_2 = 1$ / Assum $h'(k) = J$ for all keys.

let $h'(p_1) = h'(p_2) = h'[p_3] = h'(p_4) = J$

كم نتكرر (i)

$h(k,i) = h(k) + i + i^2$

Insert Collision عدد + لنفس الشي (i = # of collision)

$p_1 \Rightarrow h(p_1) + 0 + 0 = J$ empty insert $\boxed{i=0}$

$p_2 \Rightarrow h(p_2) + 0 + 0 = J$ (Collosion), i++

$h(p_2) + 1 + 1 \Rightarrow J+2$ ☑ empty insert $\boxed{i=1}$

$p_3 \Rightarrow h(p_3) + 0 + 0 \Rightarrow J$ Collasion, i++

$h(p_3) + 1 + 1 \Rightarrow J+2$ Collosion, i++

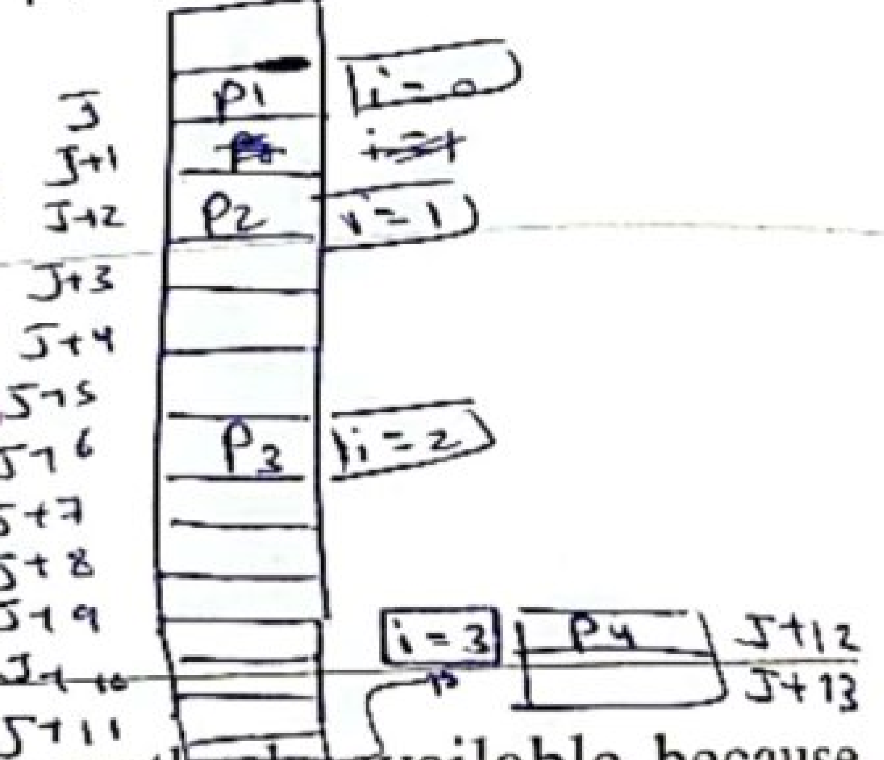
2 collosion $\boxed{\text{insert } i=2}$

$h(p_3) + 2 + 4 = J+6$ ☑

$\boxed{i++}$

$h(p_4) + 3 + 9 = J+12$ ☑ $\boxed{\text{insert } i=3}$

3 collosion



28

Double Hashing

Double hashing is one of the best open addressing methods available because the permutations produced have many of the characteristics of randomly chosen permutations. It uses a hash function of the form:

Solved Secondy

Secondy & Primary حلت

m^2

$$\boxed{h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m}$$

→ أول مرة (i=0) نستخدم أول Fun

where h_1 and h_2 are auxiliary hash functions.

إذا صار Collision (i=1) نستخدم hash Fun الثانية , i++

The initial position probed is $T[h_1(k)]$, with successive positions offset by the amount $(i h_2(k))$ modulo m.

Now, keys with the same initial probe position can have different probe sequences.

Note that $h_2(k)$ must be relatively prime to m for the entire hash table to be accessible for insertion and search. If $d = \text{GCD}(h_2(k), m) > 1$ for some key k, then search for key k would only access $\frac{1}{d}$th of the table. (See Chapter 33.)

Double Hashing *continued*

A convenient way to ensure that $h_2(k)$ is relatively prime of m is to select m as a power of 2 and design h_2 to produce an odd positive integer. Or, select a prime m and h_2 to produce a positive integer less than m.

$(m' < m)$

Example: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$, where m' is slightly less than m.

Double hashing is an improvement over linear and quadratic probing in that $\Theta(m^2)$ sequences are used rather than $\Theta(m)$, since every $(h_1(k), h_2(k))$ pair yields a distinct probe sequence, and the initial probe position, $h_1(k)$, and offset, $h_2(k)$, vary independently.

Double Hashing *continued*

دائماً نبدأ $\boxed{i=0}$

Example: Use double hashing to store the keys 10, 18, and 34 in a hash table of size $m = 8$, using:

K_1 K_2 K_3

$$h_1(k) = k \bmod 8, h_2(k) = 1 + (k \bmod 6)$$

0	1	2	3	4	5	6	7
		16	18				34
		$i=0$	$i=1$				$i=1$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 5$

Insert $k = 10$, $\boxed{i=0}$

$10\%8 = 2 + 0(\overset{5}{1 + 10\%6}) = 2$ ✓

↳ slot insertion

Insert $K_2 = 18$, $i = 0$

$18\%8 = 2 + 0(\overset{1}{1 + 18\%6}) = 2$ Collision

$i{+}{+}$, $\boxed{i=1} = 2 + 1(1+0) = 3$ ✓

↳ slot insertion

Insert $K_3 = 34$, $i = 0$

$34\%8 = 2 + 0(\overset{5}{1 + 34\%6}) = 2$ Collasion

$i{+}{+}$ $\boxed{i=1}$ → $2 + 1(1+4) = 7$ ✓

Slot insertion

Analysis of Open-Address Hashing

element حجم ال < Table حجم

Theorem: Given an open-address hash table with load factor $\boxed{\alpha = \frac{n}{m} < 1}$ the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming uniform hashing. (Theoretically assumption)

للتبسيط

max

كم مرة ح تتغير لحظة (n) لحد ما نعمل Insertion, Serching, Deletion

اذا اختير probly seq. بحقق ال (uniform hasing)

ازا اجو امتحان بجيبها

Proof:

- Refer to the book for the proof.

let n=20, m=40 Find ① α ?!
② expected # of probe.

Sol

① $\alpha = \frac{n}{m} = \frac{10}{20} = 0.5$

② expected $= \frac{1}{1-\alpha} = \frac{1}{1-0.5} = 2$

Analysis of Open-Address Hashing *continued*

Corollary 12.6. Inserting an element into an open-address hash table with load factor α requires at most $\frac{1}{1-\alpha}$ probes on average, assuming uniform hashing.

under uniform (ص)

Proof:

Insertion of a key requires an unsuccessful search followed by the placement of the key in the first empty slot found. Thus the expected number of probes is at most $\frac{1}{1-\alpha}$.

let n=20, m=40, Find α, Considring in serting an element in open addressing hash Table with load vector it Computed above, what is the expected # of probe in Av to performa. insertion ?!

① $\alpha = 0.5$

② $\frac{1}{1-\alpha} = 2$

Analysis of Open-Address Hashing *continued*

Theorem: Given an open-address hash table with load factor ① $\alpha < 1$, the expected number of probes in a successful search is at most $\boxed{\frac{1}{\alpha} \ln \frac{1}{1-\alpha}}$, assuming uniform hashing and ② assuming that each key in the table is equally likely to be searched for. ②

→ behavor (Constant timing in AVg (Serching + Delecting + insertion)

Proof: Refer to the book for the proof.

let n = 10, m = 20 Successful Serch

Sol | α = 0,5)

$$\frac{1}{0,5} \ln \frac{1}{1-0,5} = \underline{\underline{2 \ln 2}} = \underline{\underline{1,39}}$$

→ اتوش(1) (0,693) × 2

(V, E) → Vertices, → Edge

Lecture 10: Graphs, BFS, DFS, and Topological Sort

Graph : usually denoted by the letter G(V, E) → Two tuple
and it is composed from Two sets :- → e.g → V = {a, b, c, d, e}, |V| = 5 (size) → could be given or written using alphabet letters or numerical number

[1] Set of vertices on nodes: V → usually captured by circles
[2] set of edges or links : E

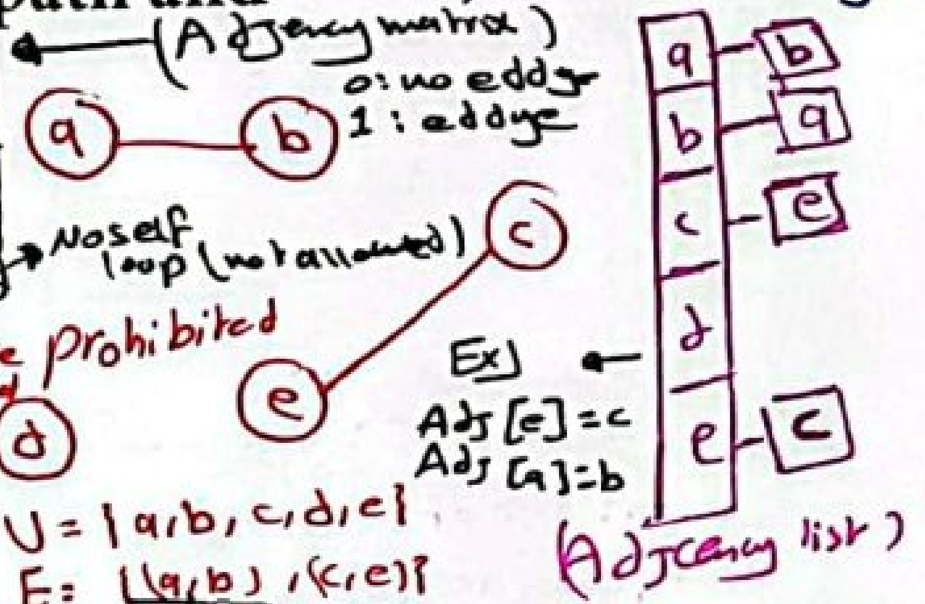
Note : The edges could be classified as : [1] Direct edge → visually or graphically represented by arrows between the vertices → to be three the virtices of the edge,
[2] undirected edges → visually or graphically represented by lines. (ال direction غير موجود) of the edge.

Course Learning Outcome

Use fundamental graph algorithms, like traversal, shortest path and spanning tree in the solution of real-life problems

	a	b	c	d	e
a	0	1	0	0	0
b	1	0	0	0	0
c	0	0	0	0	1
d	0	0	0	0	0
e	0	0	1	0	0

← (Adjacency matrix) 0: no edge 1: edge
→ No self loop (not allowed)



Ex) Adj [e] = c
Adj [a] = b
(Adjacency list)

self loop are prohibited in an undirected graph.
(undirected) V = {a, b, c, d, e}
E: {(a, b), (c, e)}

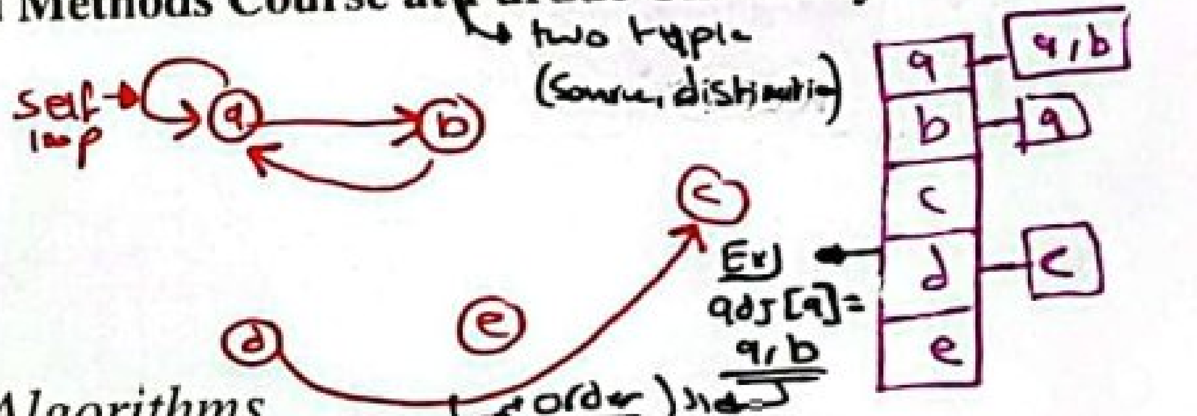
e.g : Consider Two nodes or vertices from V
i.e. v, u ∈ V
then an example of: (source) (البداية) (Sorce) (distination) (destination)
(A) Directed edge: (v, u), or (u, v)
(self loop are allowed) Two tuple
(B) unidricted: (v, u) = (u, v)
(self loop not allowed)

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University → two tuple (Source, distination)

(3) Set of weights : W, when $w_i \in \mathbb{R}$
Captured by inserting the value of wi on top ~~but then~~ of every edge (directed on undirected)
but then our graph would be given as :
G(V, E, W) → Three tuples
(weighted graph) (edge weight ال كـ)

Read Chapter 22 of *Introduction to Algorithms*



Ex) adj [a] = a, b
(order ترتيب)

(directed) E = {(a, a), (a, b), (b, a), (d, c)}
→ self loops are allowed

	a	b	c	d	e
a	1	1	0	0	0
b	1	0	0	0	0
c	0	0	0	0	0
d	0	0	1	0	0
e	0	0	0	0	0

* classification of the graphs
1- Based on the edges (e.g ① Directed graph ② undirected graph)
2- Based on the existance of the weight (e.g ① weighted graph ② unweighted graph)
3- Based on the density of the edges (e.g ① Dense graph : (V^2 يقارب edges أعداد ال) ② Sparse graph : $E << |V^2|$ in a direct graph
4- Based on the Connectivity of the graph (e.g fully connected graph)

Graphs

Ex) what is the max # of edges is an a directed graph?) V^2 (Dense graph)
(V-1) = عدد ال edges to connected vertice
(عدد ال vertices) V = edges تقريباً عدد ال ∴ (self loop) لأنه مافي (V^2) متحيل الوصلال (un.direct. graph) في ال

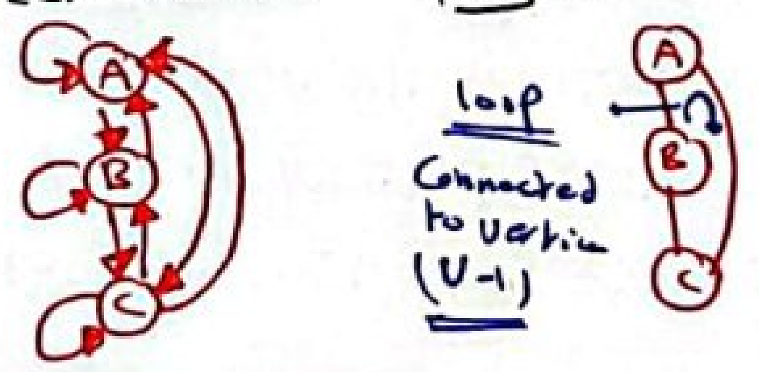
Graph representations of data structures are very useful in a variety of disciplines.

Examples of graph applications:

Ex) on fully connected graphs
[1] Directed Case [2] undirect case

- Network Analysis
- Project Planning (PERT charts)
- Shortest Route Planning
- Compilers: Data-Dependency Graphs, Control-Flow Graphs
- VLSI
- Natural Language Processing

V = {A, B, C}
|V| = 3
|E| = $|V|^2$ = 9



V = {A, B, C}
|E| = |V| = 3
E = {(A, B), (A, C), (B, C)}

E = {(A, A), (B, B), (C, C), (A, B), (A, C), (B, A), (B, C), (C, A), (C, B)}

* Question
How to represented a graph in a memory of a Computer?!
(Direct edge) إذا بيشير ← [1] Adjacency list
Storage : $\Theta(|V| + 2|E|) = \Theta(U + E)$ (→ 2 higher constant) → Applies (Dire., undri)
إذا كانت بدنا نعرف وين الاتصال ← [2] Adjacency matrix "No pointed"
size = $|U|^2$, storage = $\Theta(|U|^2)$

trade ~~trade~~ off ← [!? Best ال Serching (صغيرة) matrix (كبيرة) list : Dase / sparse] → صح
Storage Complexity

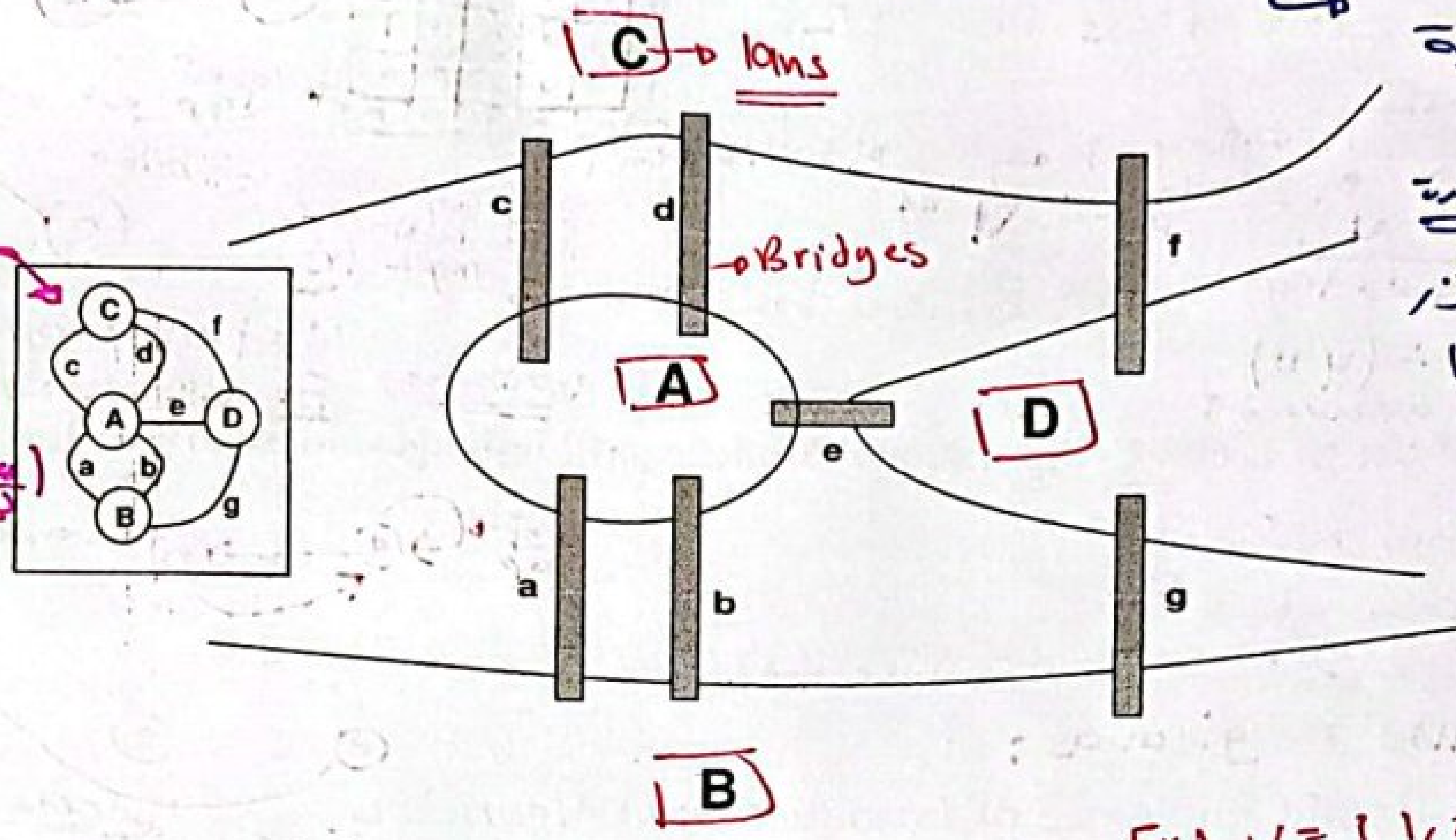
We will discuss techniques for representing graphs and perform some basic operations on them (e.g., breadth-first search, depth-first search, topological sort).

* Notations
[1] degree of a node → in degree (عدد ال arrows يلي داخلة من node) / → out degree] → in directed graph
[2] Adjacency principle (out طالعة) (عدد ال arrows يلي طالعة من node معينة)
the nodes u, v are to be called adjacent iff there exist a link of edge between them

The First Use of Graphs

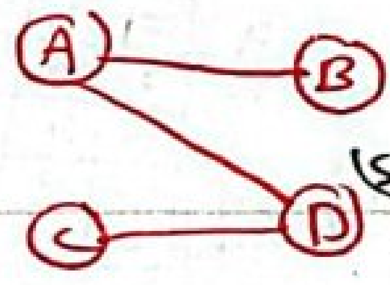
Graphs were first used by Euler in 1736, when he worked on the **Königsberg Bridge Problem**. There are four land areas (or vertices): A, B, C, D and seven bridges (or edges): a, b, c, d, e, f, g. The problem is to determine whether there is a way to start out from one land area and walk across each of the bridges exactly once and return to the original land area (i.e., Is there a Eulerian walk?).

اذا بلشنا من جزيرة معينة
المفروض انك تستخدم
كل Bridge فقط مرة واحدة
بشرط انك تزور كل الجزر
وترجع للجزيرة يلي كنت فيها



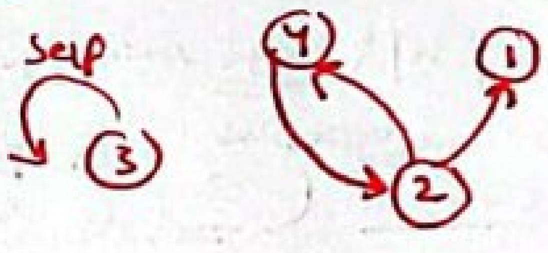
Graph: set of vertices (nodes) and edges (links)
بينهم

Ex) let V = {A,B,C,D} , undirected graph.



E = {(A,B), (C,D), (D,A)}
(size) |E| = 3
|V| = 4

Ex) V = {1,2,3,4} , Directed graph.



E = {(2,4), (4,2), (3,3), (2,1)}
|V| = 4
|E| = 4

Definition of a Graph

A graph G consists of a finite, non-empty vertex (or node) set V and an edge set E (possibly empty), i.e., $G = (V, E)$. Note that $E \subseteq V \times V$; hence $|E| = O(V^2)$.

If $u, v \in V$ and there is an edge between those vertices, we can represent this by storing an ordered pair (u, v) in E. Normally, (u, v) appears in E only once (unless you are working with a **multigraph**, which allows multiple edges between vertices).

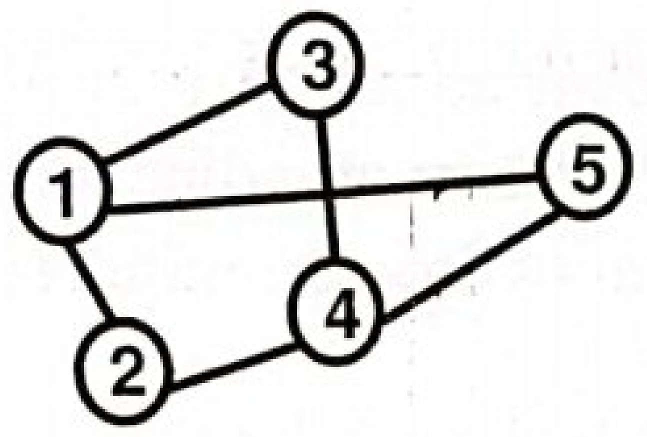
استخدم arrows

If an edge between two vertices is **directed** from one vertex to the other, $u \rightarrow v$, it is represented as a tuple $(u, v) \in E$. Note that (v, v) represents a self-looping directed edge.

If an edge between two vertices is **undirected**, then for $(i, j) \in E$, $(i, j) = (j, i)$. There are no self-loops allowed, i.e., $(v, v) \notin E$.

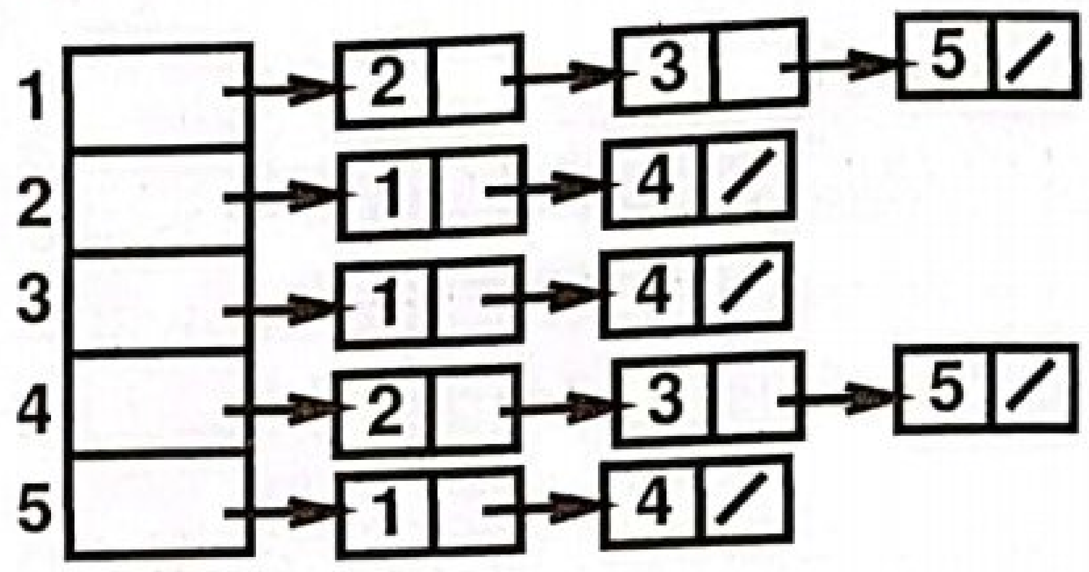
Example of an Undirected Graph

Undirected Graph, G



G = (V, E)
V = {1, 2, 3, 4, 5}
E = {(1,2), (1,3), (1,5),
(2,4), (3,4), (4,5)}

* Adjacency List for G



* Adjacency Matrix for G

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 0 & 0 & 1 & 0 \end{array}$$

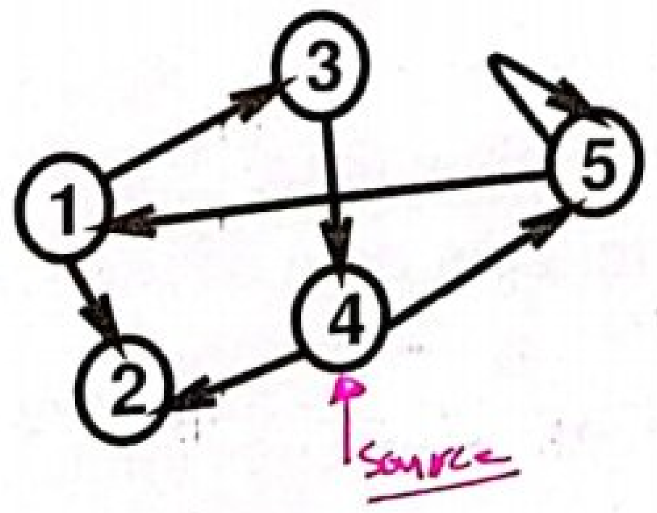
no self loop
undirected

* اذا في wighr
بنخزنه بدل الـ 1, 0
اذا ما في weight (عادي)

Example of a Directed Graph (or Digraph)

حل السؤال عن (BFS)

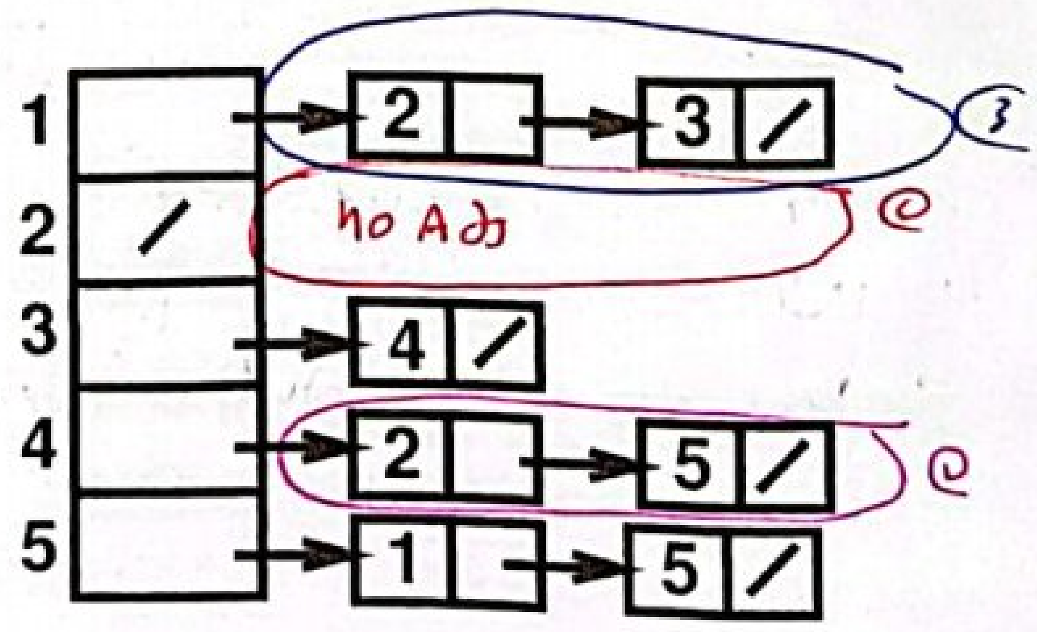
Directed Graph, G



G = (V, E)
V = {1, 2, 3, 4, 5}
E = {(1,2), (1,3), (3,4), (4,2),
(4,5), (5,1), (5,5)}

node	1	2	3	4	5
c[ ]	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B	~~G~~ B	~~W~~ G
d[ ]	~~∞~~ 2	~~∞~~ 1	~~∞~~ 3	0	~~∞~~ 1
π[ ]	~~Nil~~ 5	~~Nil~~ 4	~~Nil~~ 1	Nil	~~Nil~~ 4

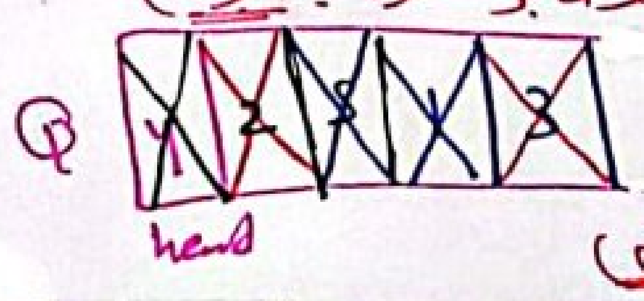
Adjacency List for G



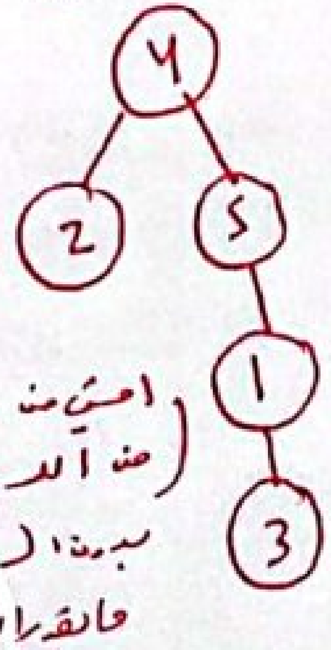
Adjacency Matrix for G

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 & 1 \\ 5 & 1 & 0 & 0 & 0 & 1 \end{array}$$

(امشي من [illegible] هنا الـ 3 الاعلى للاكبر مش عشوائي [illegible])



BFS Tree



بدونا الـ Parent
فابقى ارسم
BFS Tree
(صح)

1. Graph Representation: Adjacency Lists

For graph $G = (V,E)$, an adjacency list representation consists of an array of length $|V|$, Adj, such that for each vertex $v \in V$, $Adj[v]$ keeps a list of those vertices adjacent to v.

→ (يعني من direct edge)

Given that the **degree** of a vertex v is the number of incident edges to v in an undirected graph $G = (V,E)$, we know that the number of items in an adjacency list representing G is $\sum_{v \in V} degree(v) = 2|E|$. Hence, the amount of storage required to represent G is $\Theta(V+E) = \Theta(max(V,E))$.

A directed graph's vertices have both **in-degree** and **out-degree**. The number of items stored in an adjacency list for a directed graph $G = (V,E)$ is $\sum_{v \in V} out\text{-}degree(v) = |E|$, resulting in storage requirements of $\Theta(V+E) = \Theta(max(V,E))$.

Adjacency lists are good representations for sparse graphs $\boxed{|E| << |V|^2}$; however, there is no quick way to determine whether a given edge (u,v) is present in the graph without searching the list associated with $Adj[u]$.

2. Graph Representation: Adjacency Matrix

For graph $G = (V,E)$, an adjacency matrix consists of a $|V| \times |V|$ matrix A, such that:

→ weight ممكن تكون اذا Real number

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$

Using this method, we can determine very quickly whether there is an edge between two vertices; however, it uses $\Theta(V^2)$ memory, independent of the size of E. E={(1,2)}

→ mean's (graph transpose ⇒ من directed كان تصبح undirected بكرر نفسه ⇒ الـ edges بنعكس (يعني بكون الـ source بيصير dest بالعكس) Ex) ① → ②

Note that for an undirected graph G represented with an adjacency matrix A that $A = A^T$, i.e., $A[i,j] = A[j,i]$, which can be used to cut the memory requirements in half. We can further reduce space by using a bit matrix representation. ① ← ② 2nd ع نعكس الـ direction (graph transpose) نفس الـ V لـ الـ (set of edges transpose)

This is a good representation for dense graphs, i.e., $|E| \approx |V|^2$ or for small graphs. Usually this method requires too much storage for large graphs, especially since many graphs are sparse (e.g., planar graphs). E={(2,1)}

Weighted Graphs

In a **weighted graph** G, weights can be associated with the edges or the vertices of the graph.

If each edge has an associated **weight** given by a weight function $w : E \to \mathbf{R}$, adjacency-list and adjacency-matrix representations must be modified to represent this additional information.

The adjacency list for G can be easily adapted to represent this weight information. For each edge $(u,v) \in E$, store the weight $w(u,v)$ with vertex v in u's adjacency list.

Adjacency matrices can also be easily adapted to represent a weighted graph, G. For each edge $(u,v) \in E$, store the weight $w(u,v)$ in $A[u,v]$ (NIL if no edge). Note that no bit-matrix representation is possible in this case.

[1] distance function d[ ]: ① Initialy all vertices in the graph except for the Sorce node will be initinlized to a very lang value (∞) → means all vertices are assumed initialy unreachable from the Source
② distance all the Source is zero (0 = من نفسي)

[2] Parent function π[ ]: maintanse the parent relationship of node. help in constracting the BFS tree.

[3] Color function: 3 colors will be used
(W) white :
(G) Gray :
(B) Black :
→ see the slide

Function
→ d[ ]: ~~destanse~~ distance function
→ π[ ]: parent function (أبو Source ال)
→ c[ ]: ~~Coller~~ Color function
→ adj[ ]: adjencency fun.

③ Defined set of functions to mange the information about traversal of the node in the graph from the source node

② use FIFO Queue to mange the transal of the nodes from the Source vertex

① Assumption: The graph will be represented using AdJency list

Breadth-First Search (BFS)

Input ال Graph

Given a directed or undirected graph $G = (V,E)$ and a distinguished source vertex s, the breadth-first search algorithm systematically explores (يستكشف) the edges of G to discover whether every vertex is reachable from s. It does this by visiting all vertices at distance $\boxed{k}$ (# of edge) before vertices at distance $\boxed{k+1}$. The algorithm:

- computes the distance (fewest number of edges) from the source vertex to every other vertex,
- creates a breadth-first tree (يرسم الـ node ع طريق ال (Parent Fun)) with root s to all reachable vertices such that the path from s to v represents the shortest path between those two nodes.

Ancestors and descendants in the breadth-first tree are defined with respect to the shortest path from the source to each vertex.

Breadth-First Search Algorithm: BFS

BFS:

- assumes that $G = (V,E)$ is represented using an adjacency list.
- uses a first-in-first-out (FIFO) queue, Q, to manage the traversal of vertices. operation (Head, Enqueue (insert عنصر 1), Dequeue (remove عنصر 1)
- keeps track of the color of a vertex $u \in V$ using $color[u]$. If u is WHITE, it has not been discovered; if GRAY, it has been put on Q; if BLACK, its successors have been added to Q. initially all Vertices is white ما عدا ال Surce شيلناه (Q)
- keeps track of the distance between the source vertex s and $u \in V$ in $d[u]$.
- uses $\pi[u]$ to store the predecessor of u in order to construct a BFS-Tree. If there is no predecessor, the value of $\pi[u]$ is NIL. Parent مالو زي ال Sorce

```
BFS(G,s)
for each vertex $u \in V[G] - \{s\}$   except the sorce
      do $color[u] \leftarrow$ WHITE
         $d[u] \leftarrow \infty$   (unreachable)
         $\pi[u] \leftarrow$ NIL
```

Breadth-First Search Algorithm: BFS *continued*

```
$color[s] \leftarrow$ GRAY   (حطيته داخل ال (Q)
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \{s\}$   in Q (S) → داخل ال (Q)
while $Q \neq \emptyset$ ( Q not empty)
      do $u \leftarrow head[Q]$   (أول node في Q)
         for each $v \in Adj[u]$
             do if $color[v] =$ WHITE
                   then $color[v] \leftarrow$ GRAY   ((داخل ال (Q))
                        $d[v] \leftarrow d[u] + 1$
                        $\pi[v] \leftarrow u$
                        ENQUEUE(Q,v)
         DEQUEUE(Q)   Remove ( شيل ال node اللي كنت شغالة عليها وسميتها (u) )
         $color[u] \leftarrow$ BLACK
```

داخل ال for

داخل ال if

ها color

تبع (u) + طلعته Q

(Shortest distance)

ال distance من a لل a = 0
ال distance من b لل a = 1
ال distance من c لل a = 1
ال distance من d لل a = 2
ال distance من e لل a = 1

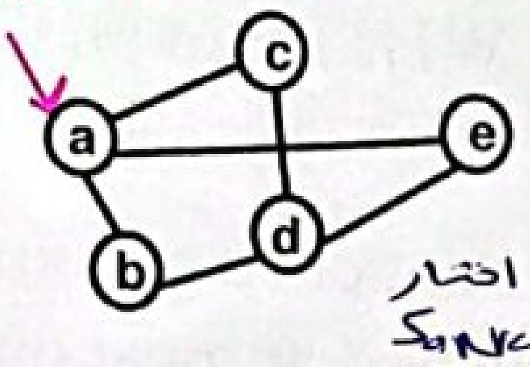
فوائد ال Table

Example: BFS(G,a)

V = {a,b,c,d,e}

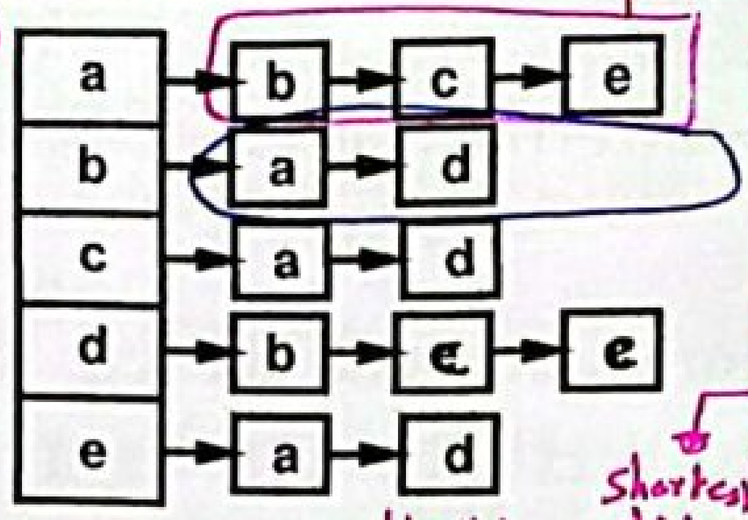
E = {(a,b),(a,c),(a,e), (c,d) ، (c,a) ، (b,a) (b,d) ، (d,e) ، (d,c) (d,b)}

source node



①

اختار Source (a)



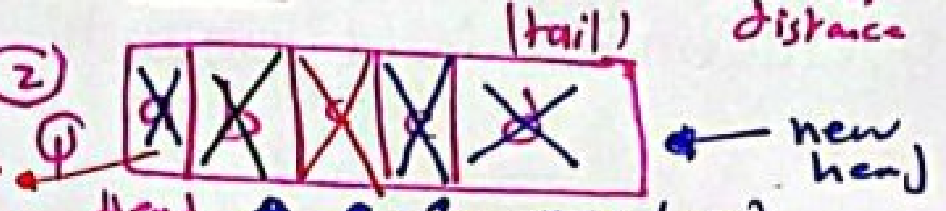
Node	a	b	c	d	e
c[]	~~G~~ B	~~W~~ B ~~G~~	~~W~~ B G	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B
π[]	NIL	~~NIL~~ ~~∞~~ a	~~NIL~~ a	~~NIL~~ b	~~∞~~ ~~NIL~~ a
d[]	0	0+1= 1	~~∞~~ 0+1= 1	~~∞~~ 1+1= 2	~~∞~~ 0+1= 1

Shortest distance

d(parent a)

d(parent b)

②

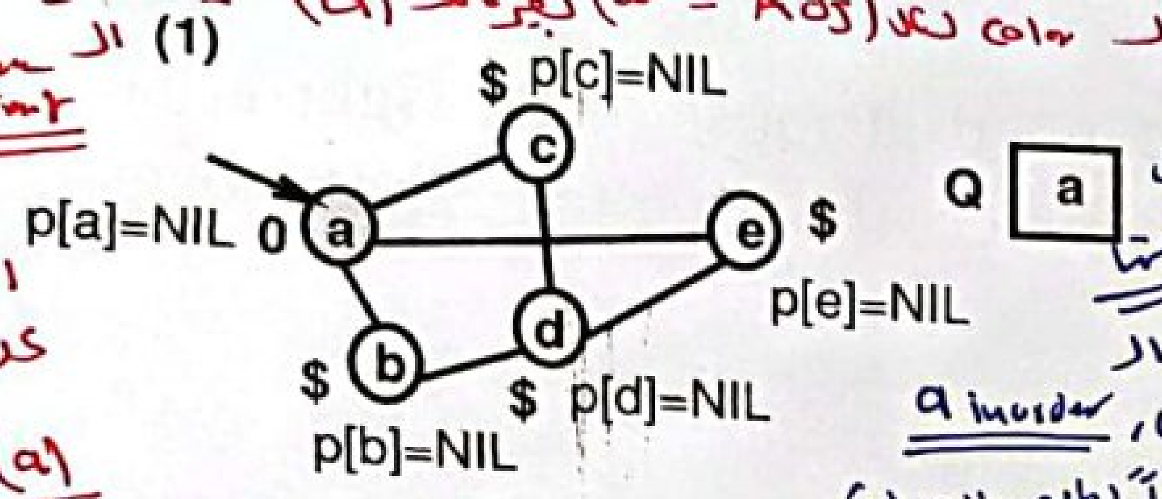


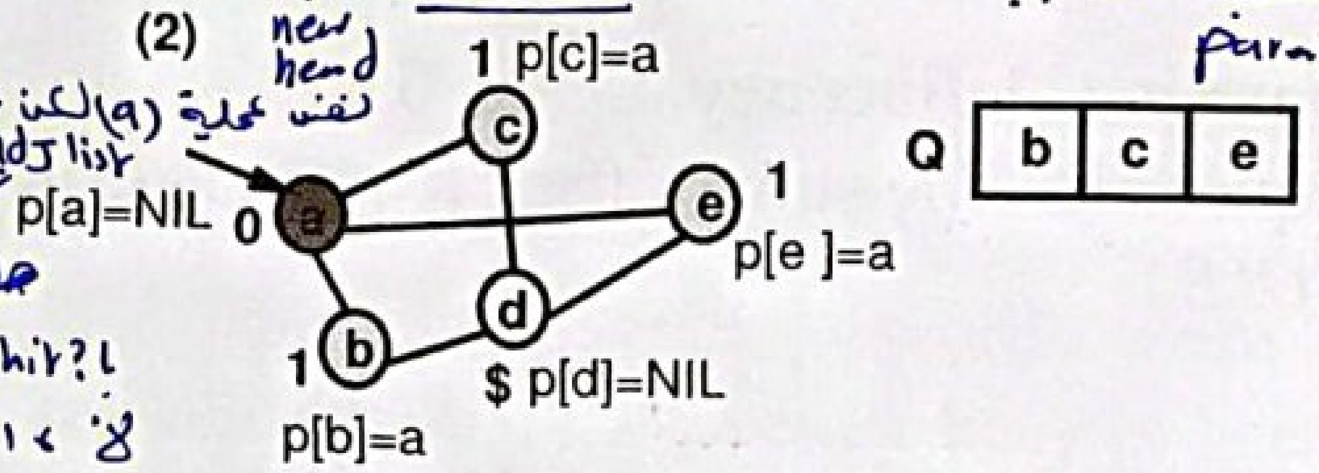
صار ال Aly. خزن ال (a) head بعد ما Ad لل (a) وخزن ال order

اذا ال color لكل (Adj = w) بيغير ال (G) بعين صفوفيا Q وبغير

ال distance وال parent

بس حلقة ال (list) كلها بعد deq(a) بعين الغير (a) وبتغير (B)





نفس عملية (a) لكن عل adj list تاعتها

حلال

a in order, white?!

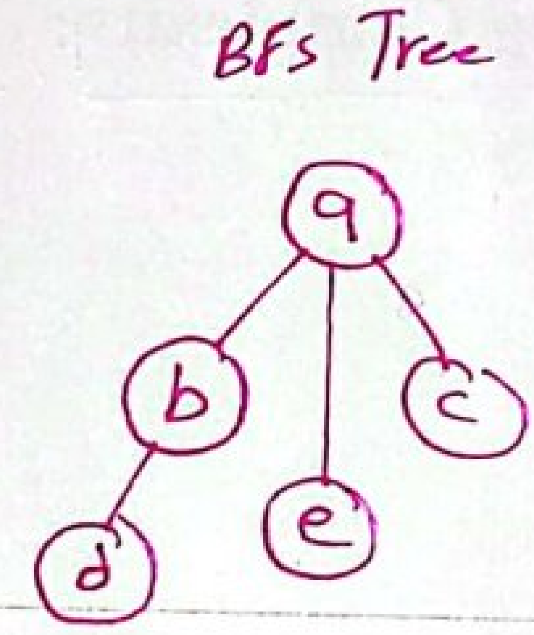
لأ ، اذا تطلع سابقا

اذا ال while ننفذ ال statement الموجودة

والسؤال بسأل ؟!

parent, distance بـ Finish

BFS Tree



بروح لل node (b) وبسأل مين ال Source ؟! الـ = a اذا كانت مرسوم ال Branch ال distance = 1 : one edge

مين ال parent الـ = a

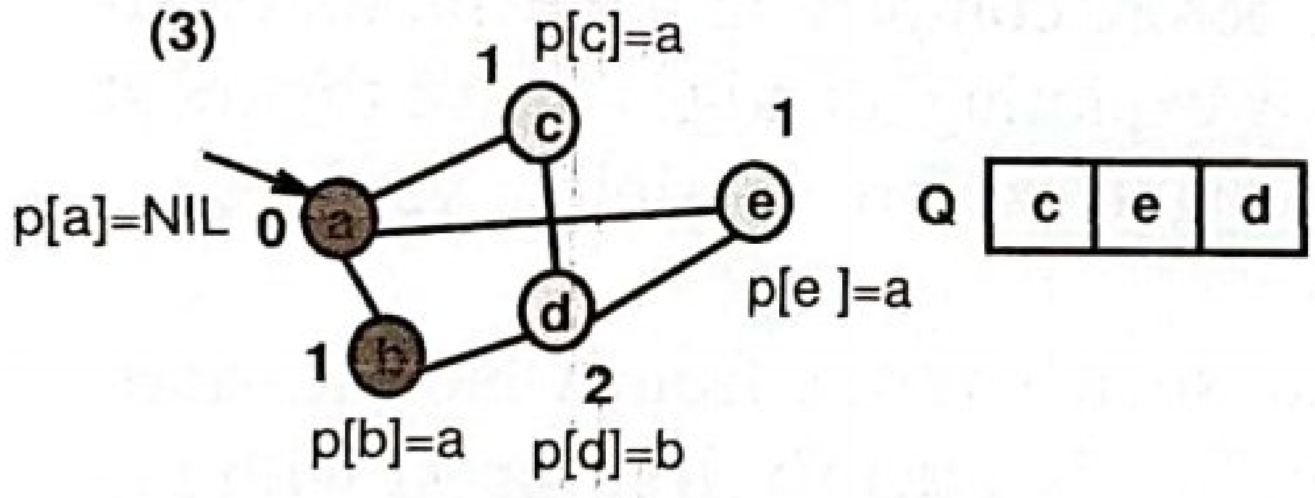
بروح لل (c) مين ال parent ؟! a : بعدها ال Branch ال distance = 1 one edge : (one link) لسبب

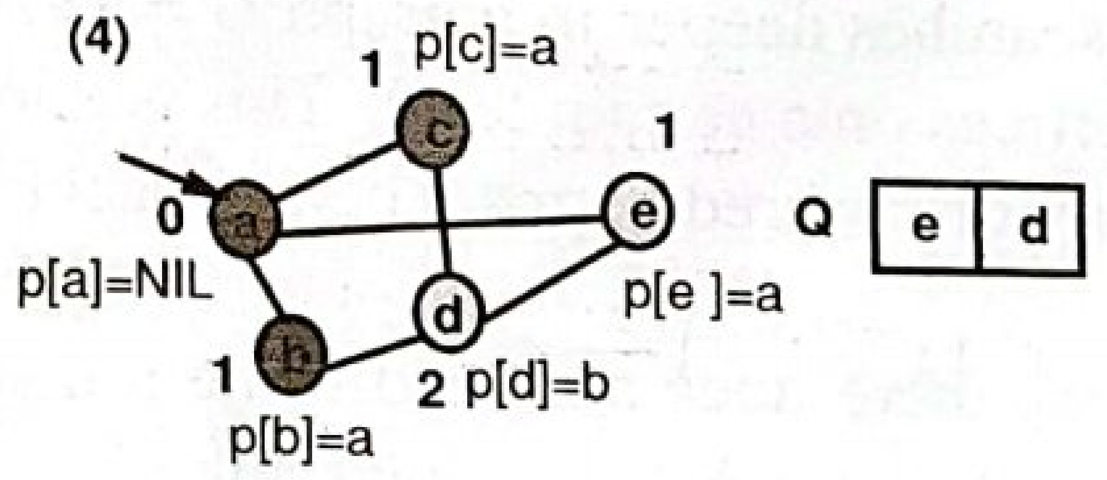
بروح لل (d) مين ال parent ؟! b : Branch ثاني

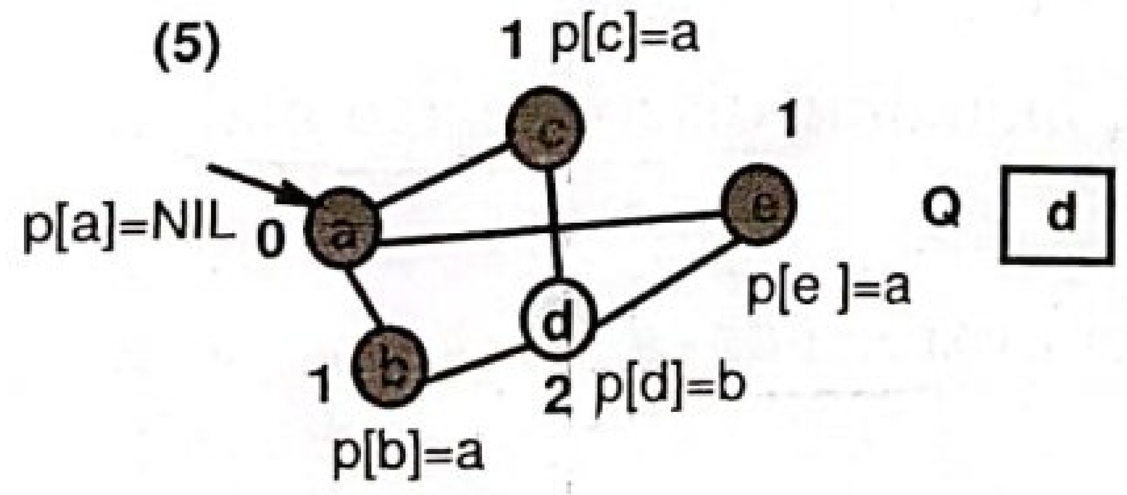
ع ال distance = 2 (two edges)

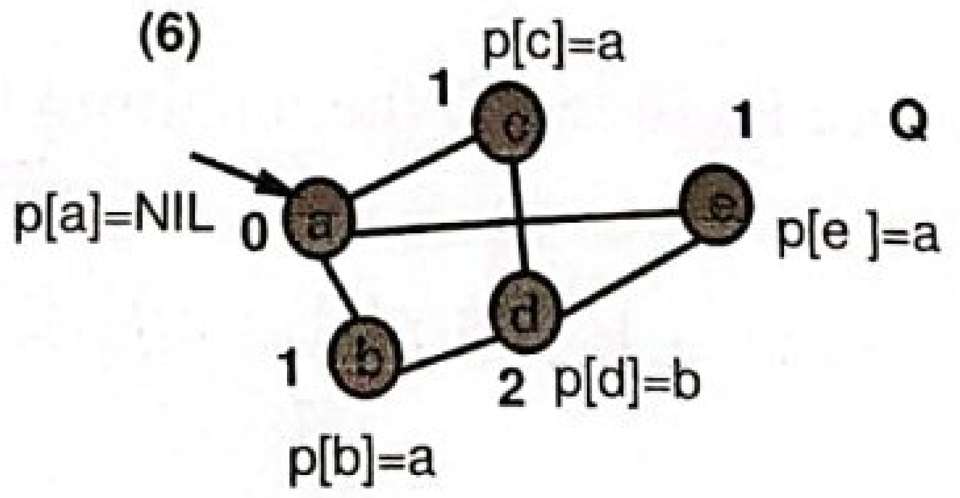
ال (e) من (a) + one edge (dis=1)

Example: BFS(G,a) *continued*









Complexity of BFS

- **Initialize:** $|V|-1$ vertices are initialized to WHITE in $\Theta(V)$.
- **Queue Operations:** Because vertices when enqueued become GRAY and they are never reset to WHITE, vertices are enqueued at most once. Each enqueue and dequeue operation uses $O(1)$ time; hence, all queue operations take $O(V)$ time.
- **Scanning Adjacency Lists:** The lists are scanned at most once, right after dequeuing. The sum of their lengths is $\Theta(E)$. Hence, the time to scan the lists is $O(E)$.
- **Resulting Complexity:** $O(V+E)$

(العمق)

The Depth-First Search Algorithm: DFS

(حاجة تستخدم ① مز BFS)

لانه لا ستراتيجية بتطلق بتحكيانه
اذا شفت (node) ما رح اترك
ال (node) يلي ب my Bredth (سطحي)
اذا شفت node ما رح اترك ال node الاقرب مني
بس اسرق الابعد ثم الابعد يعني (كما واحد لاعمق node) max. depth لعند بدي ارجع بنفس الطريقة)

DFS searches deeper in a graph $G=(V,E)$ before completing the exploration of the vertices on a certain level (القراب من). This is done by exploring all edges out of the most recently discovered vertex v before backtracking to explore v's sibling vertices.

(unique Source) ما بدي — عن node تجربة ⇒ Tree كانت

In DFS, there does not need to be a distinct source vertex from which to start. The resulting predecessor subgraph $G_\pi=(V_\pi,E_\pi)$ is a **depth-first forest** with the following properties:

- All vertices are included in the resulting forest (غابة), including disconnected nodes.
- $\boxed{E_\pi} = \{(\overset{\text{Parent}}{\pi[v]},v) : v \in V \text{ and } \pi[v] \neq nil\}$ is the set of **tree edges**.

كل node e (parent تبعا)

(مش disconnect node)

كل ال edges يلي ال parent إلهم لكل Vertex مش Nill

The Depth-First Search Algorithm: DFS *continued*

DFS:

- assumes that $G = (V, E)$ is represented using an adjacency list.
- uses backtracking to manage the traversal of vertices.
- keeps track of the color of a vertex $u \in V$ using $color[u]$. If u is WHITE, it has not been discovered; if GRAY, it has been discovered; if BLACK, its adjacency list has been completely explored. (مرّيت عليه مش انه خلصته Φ منه) (Finsh)
- uses $\pi[u]$ (Parent) to store the predecessor of u in order to construct a DFS-Forest. If there is no predecessor, the value of $\pi[u]$ is NIL.
- keeps track of two timestamps for each vertex $v \in V$; $d[v]$ records when v is first discovered and grayed; $f[v]$ (Finish) records when v's adjacency list has been completely examined, at which point v is made BLACK. Note that $d[v] < f[v]$ and both timestamps are integers in the range $[1, 2|V|]$. Each time stamp will be set only one time for each vertex v.

discovery + Finish (مرة واحدة فقط مافيش update)

DFS(G)

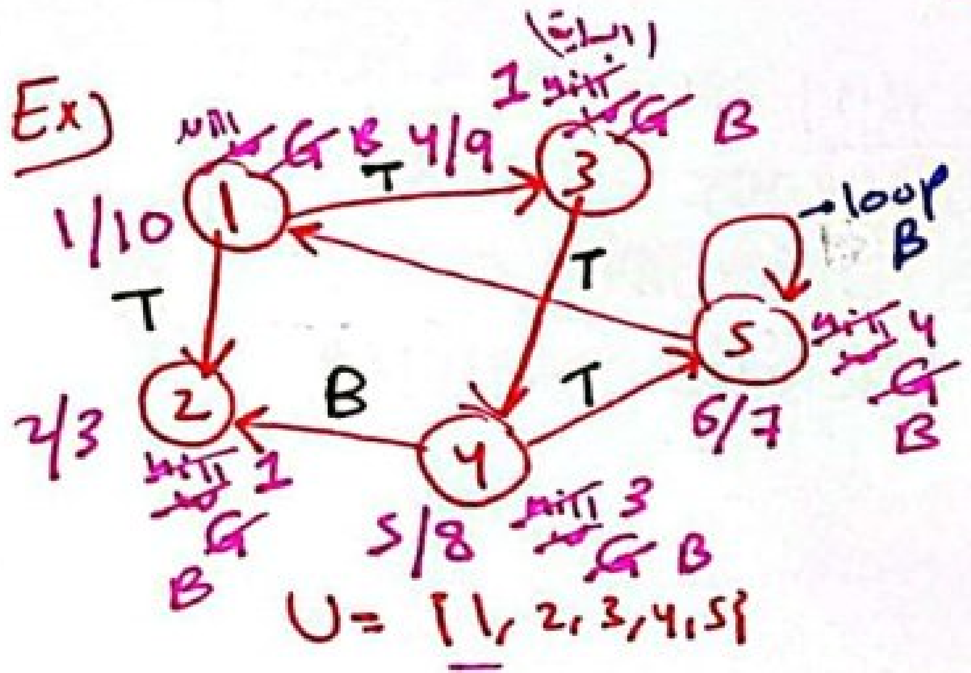
```
DFS(G)                                  مهم (الـ order مش مهم)
for each vertex $u \in V[G]$
    do $color[u] \leftarrow$ WHITE
       $\pi[u] \leftarrow$ NIL   (Parent)
$time \leftarrow 0$      ▷ global timestamp
for each vertex $u \in V[G]$
    do if $color[u] =$ WHITE
          then DFS-VISIT($u$)
```

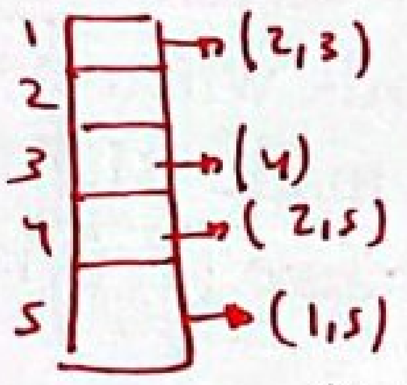
بيشتغل على كل node بشرط انها تكون While

DFS-VISIT(u)

```
DFS-VISIT(u)
1. color[u] ← GRAY
2. d[u] ← time ← time + 1
3. for each vertex v ∈ Adj[u]
4.       do if color[v] = WHITE
5.             then π[v] ← u            parent حيدد ال
6.                  DFS-VISIT(v)         مبدئيا ازوره
7. color[u] ← BLACK
8. f[u] ← time ← time + 1
```

Ex)




1 → (2,3)
2
3 → (4)
4 → (2,5)
5 → (1,5)

node	1	2	3	4	5
c[ ]	B	B	B	B	B
π[ ]	nill	1	1	3	4
d[ ]	1	2	4	5	6
F[ ]	10	3	9	8	7

(اهم اشي)

Final Table

(Tree) التطبيق عد

Classification

Tree edge: T (أول مرة بزوره وكانت node white)

Back edge : B (اول مرة بزوره وكانت node مش white (loop))

Complexity of DFS

- **Initialize:** $|V|$ vertices are initialized to WHITE in $\Theta(V)$.
- DFS-VISIT: is called once for each vertex $v \in V$ because vertices are never reset to WHITE. During the execution of DFS-VISIT(v), the loop at lines 3-6 is executed $|Adj[v]|$ times. Because $\sum_{v \in V} |Adj[v]| = \Theta(E)$, the total cost to execute the loop over all vertices is $\Theta(E)$.
- **Resulting Complexity:** $O(V+E)$

Example: DFS(G)

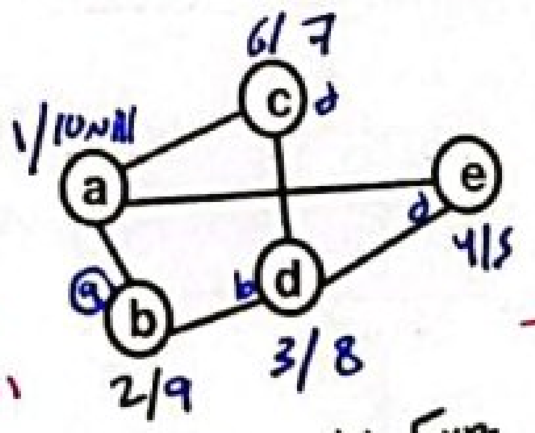
ال order حسب صار

V={a,b,c,d,e}

بـ أ'ب (a) هل white ?! yes

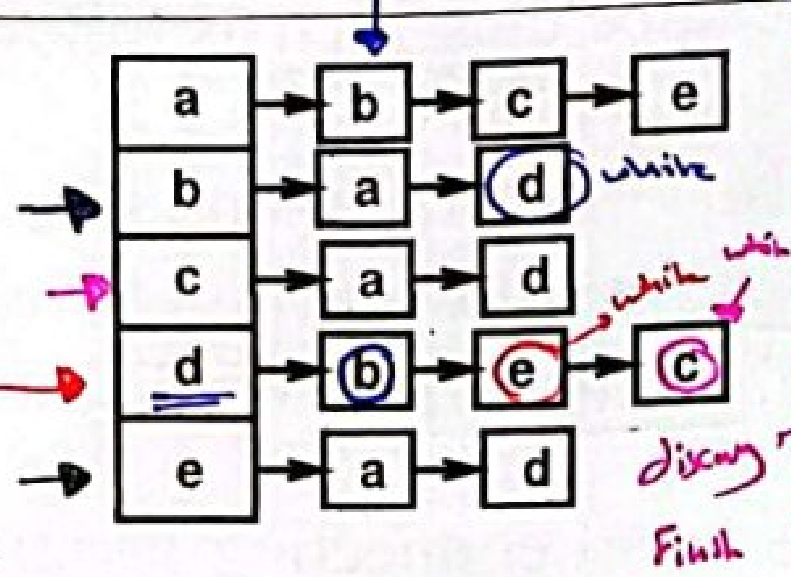
اذا نعمله Visit بغير ال color (G)

بزيد ال time (ما بزيد)



white ?! yes ➔ 6

برجع من (b)



node	a	b	c	d	e
c[ ]	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B	~~W~~ ~~G~~ B
π[ ]	Nill	~~Nill~~ a	~~Nill~~ d	~~Nill~~ b	~~Nill~~ d
d[ ]	1	2	6	3	4
F[ ]	10	9	7	8	5

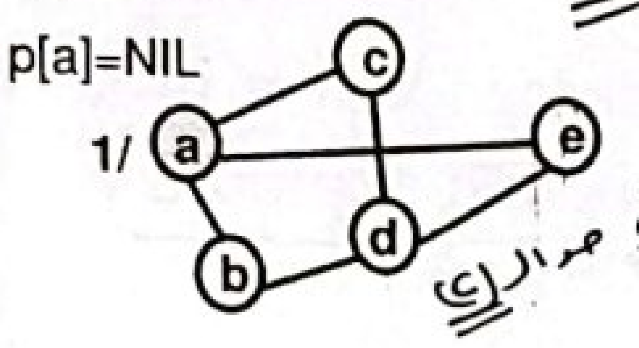
Exp ماشي ∴ بتغير (B) بعمله Finish

برجع برجع هنا وصلنا احيث يا e ?! مش (d) برجع بتأكد مد (b) وهل هنا d جدال (b)

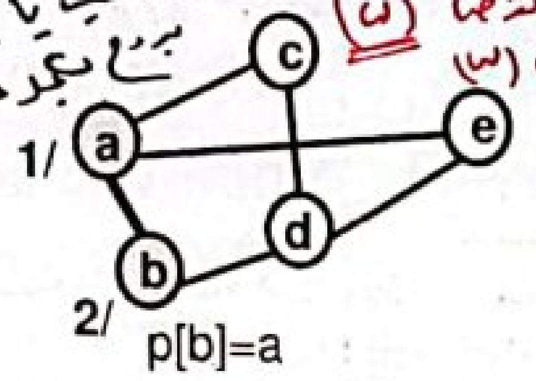
بجينا ما حلصنا مايد ال ① Call برجع عن b

متأكد اذا المنحنا (b) كله (b) ماشي (b) ∴ خلصنا

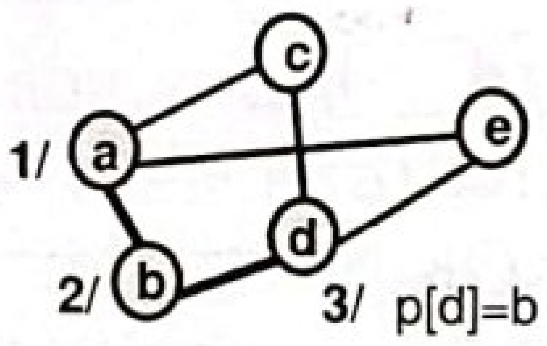
(1)



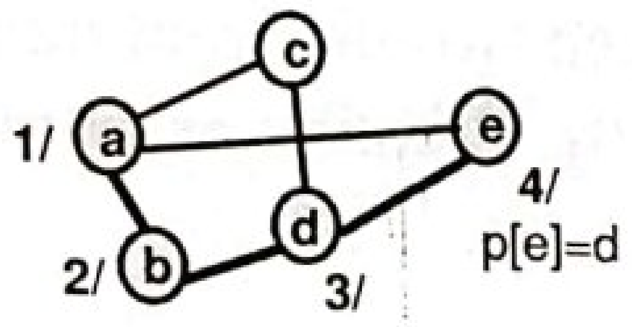
(2)



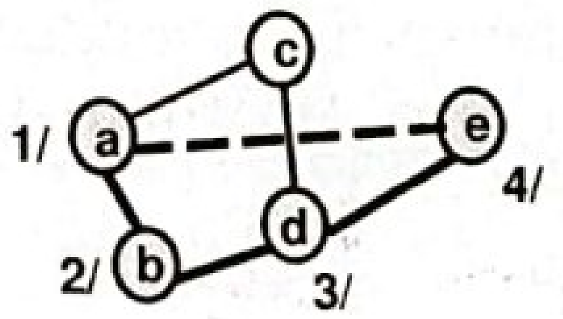
(3)



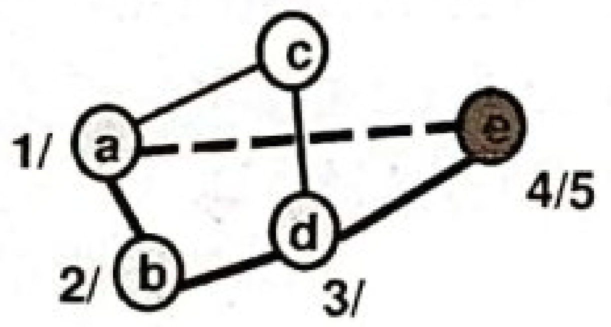
(4)



(5)

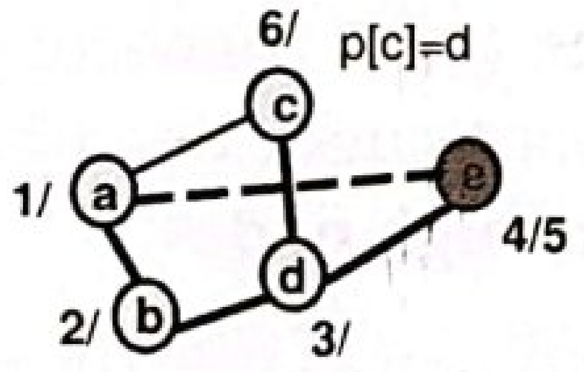


(6)

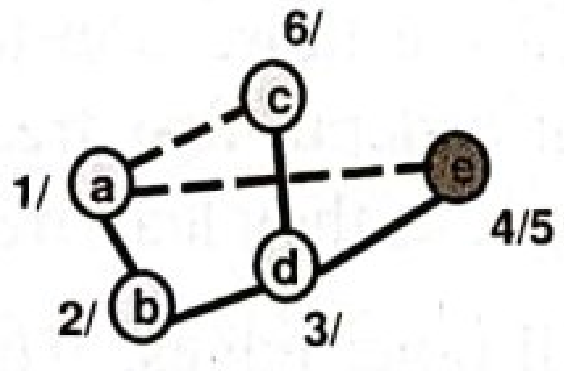


Example: DFS(G) *continued*

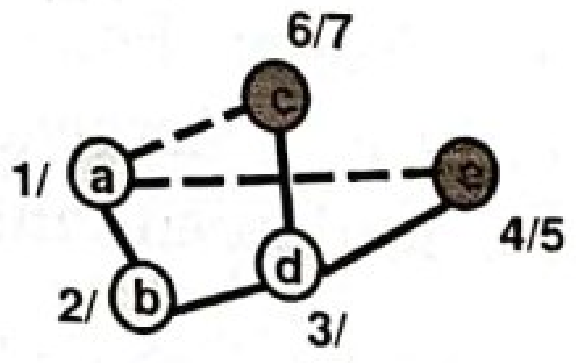
(7)



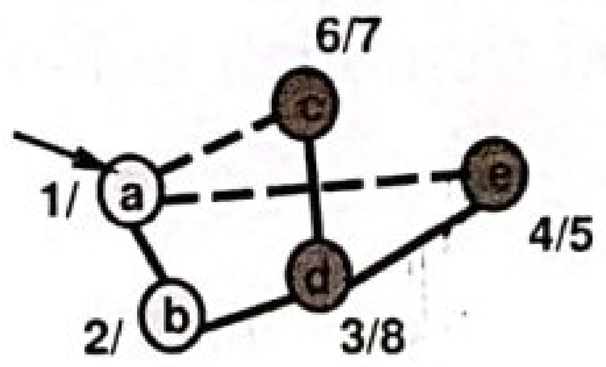
(8)



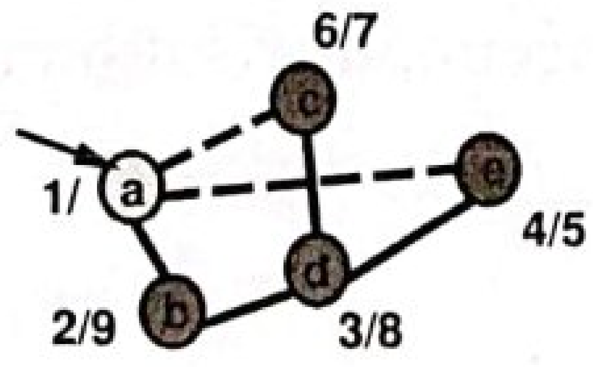
(9)



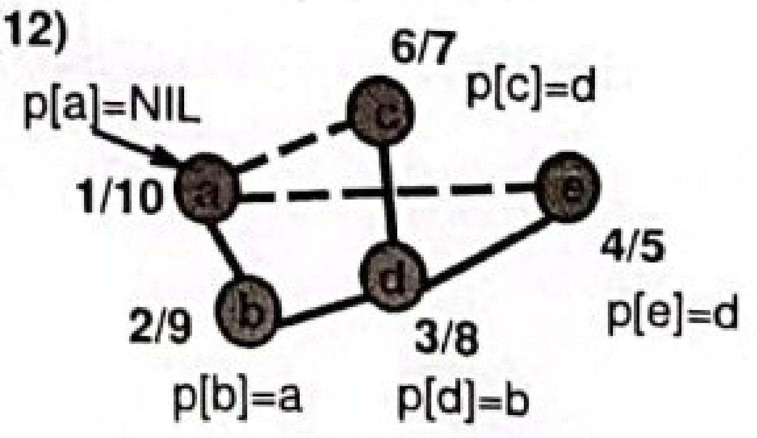
(10)



(11)



(12)



DFS(G) Edge Classification

The edges in directed G can be classified as follows by DFS(G):

→ Source (Gray)
→ Dist. (White)

1. **Tree Edge:** an edge (u,v) in a DFS-Forest G_π resulting from the discovery of a vertex v from a GRAY vertex u $(d[u] < d[v] < f[v] < f[u])$. When the edge (u,v) is first explored v is WHITE. (اذا زرت الـnode كانت white)

(loop) لو edge تعود lup

2. **Back Edge:** an edge (u,v) connecting vertex u to an ancestor v (or u itself) $(d[v] < d[u] < f[u] < f[v])$. When the edge (u,v) is first explored v is GRAY.

↳ Source

Lo (dif الـ Dist. و Source الـ dif الـ لتنفيذ) +

Dist.

3. **Forward Edge:** a non-tree edge (u,v) connecting u to a descendent v in a DFS-Tree $(d[u] < d[v] < f[v] < f[u])$. When the edge (u,v) is first explored v is BLACK.

شبه الـ Tree edge (كأنه من أكثر من path لا يوجد)

↳ نفس الـ Tree (الحل الفرق)

edge باقيا ما تبقى من cross تكون (ما وصلنا لها إلا من الطريق)

4. **Cross Edge:** all remaining edges; either within a single tree where there is no ancestor relationship between the vertices or edges between two different trees in the forest $(d[v] < f[v] < d[u] < f[u])$. When the edge (u,v) is first explored v is BLACK.

d(dist.) f(dist) d(Source) f(Source)

(Dist. node) لعين يكون زائره الـ Dist. node (Black)

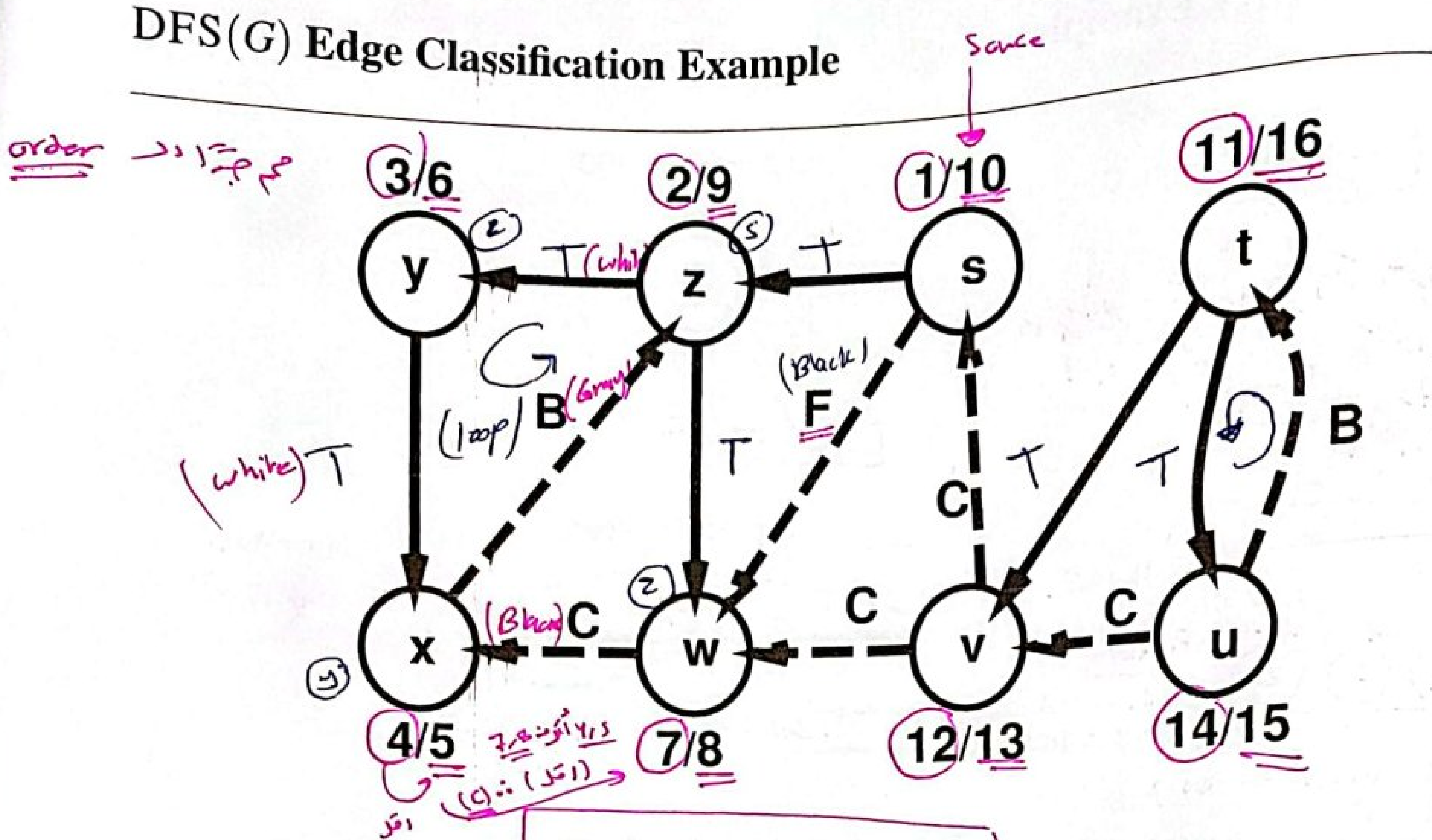
DFS(G) Edge Classification

More about the Forward and Cross Edges.

1. **Forward Edge:** Forward edges are those non-tree edges (u,v) connecting a vertex u to a descendant v in a depth-first tree. Forward edges describe **ancestor-to-descendant relations**, as they lead from **low to high nodes**.

2. **Cross Edge:** Cross edges are all other edges. They can go between vertices in the same depth-first tree as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees. Cross edges link nodes with **no ancestor-descendant relation** and point from **high to low nodes**.

DFS(G) Edge Classification Example



V={s,z,y,x,w,t,v,u}

Q: Design Question

Design an Algorth. to discover if a given directed graph Contains a loop(s) or Not?!

→ Run DFS on the directed graph.

→ classify the edges in the graph.

→ check if there are any Back edges.

→ if yes, Return yes the graph Contains a loop

otherwise, Return No.

Time Complexity ⇒ $\theta(U+E)$

Topological Sort

graph [illegible] الترتيب بعد (الشرط) Cycle (Back edges) ما عنا يعني

A **topological sort** of a directed acyclic graph (or DAG), $G=(V,E)$ is a linear ordering of all of its vertices such that if $(u,v)\in E$, then u appears before v in the ordering.

A DAG expresses a partial order on its vertices; a topological sort generates a linear ordering of the vertices such that the partial order relations are preserved.

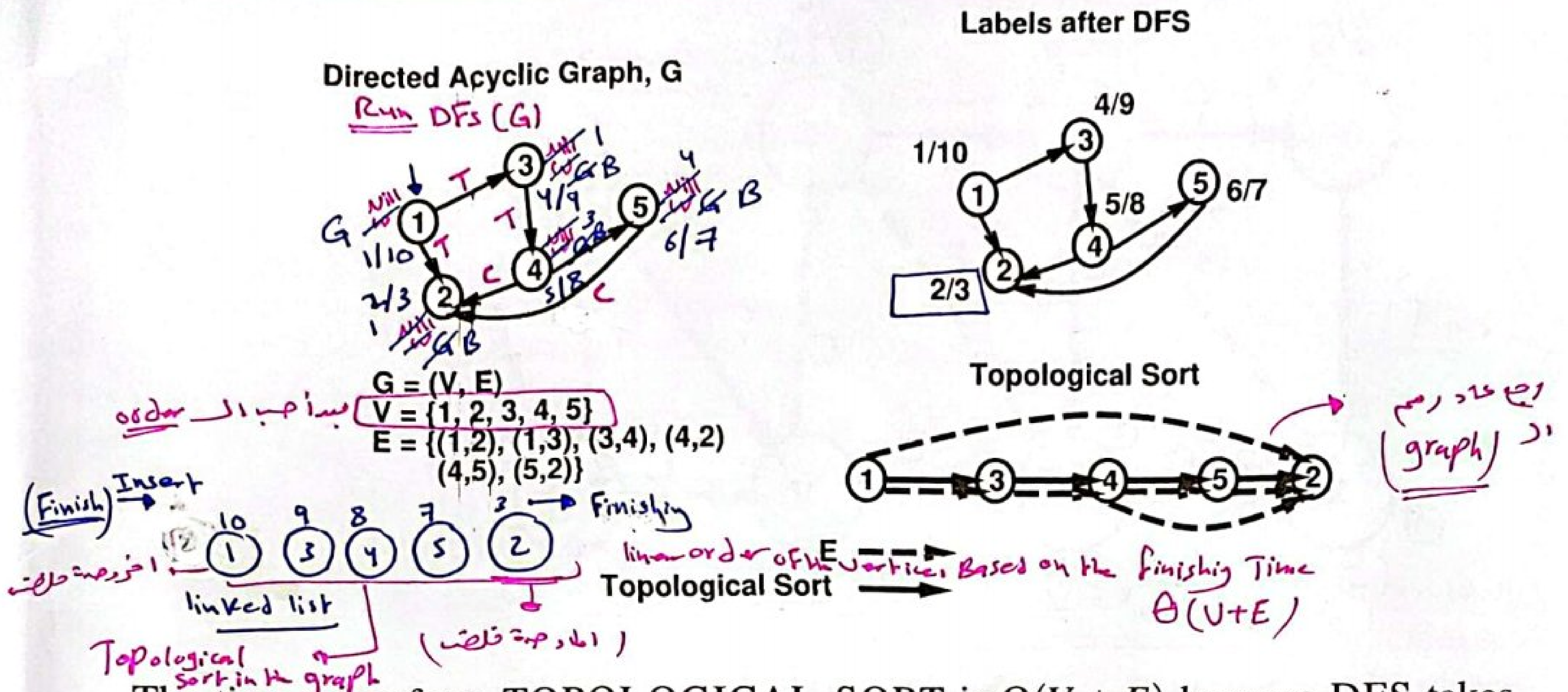
Basic Idea: The finish times produced by DFS form a total order needed to produce a topological sort of a DAG.

الترتيب حسب Finishing Time

```
TOPOLOGICAL-SORT(G)
1. call DFS(G) to compute finish times f[v] for each v ∈ V.
2. as each vertex is finished, insert it into the front of a linked list.
3. return the linked list of vertices.
```

(السبب) → Time Complexity = U+E

TOPOLOGICAL-SORT Example



Directed Acyclic Graph, G

Run DFs (G)

Labels after DFS

G = (V, E)

V = {1, 2, 3, 4, 5}

E = {(1,2), (1,3), (3,4), (4,2) (4,5), (5,2)}

order ابدا حسب ال

Topological Sort

رح اعيد رسم ال (graph)

(Finish) Insert

10 9 8 7 3 → Finishing

اخر وحدة طلعت

linked list

Topological sort in the graph

(اول وحدة طلعت)

linear order of the vertices, Based on the finishing Time

$\Theta(V+E)$

The time to perform TOPOLOGICAL-SORT is $\Theta(V+E)$ because DFS takes $\Theta(V+E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of a linked list.

Fibonacci Series

Top down

$F_n = F_{(n-1)} + F_{(n-2)}$
$F_{n-1} = F(n-2) + F(n-3)$
$F_{n-2} = F(n-3) + F(n-3)$
⋮
$F_3 = F_2 + F_1 = 3$
$F_2 = F_1 + F_0 = 2$
$F_1 = 1$
$F_0 = 1$ } Base case

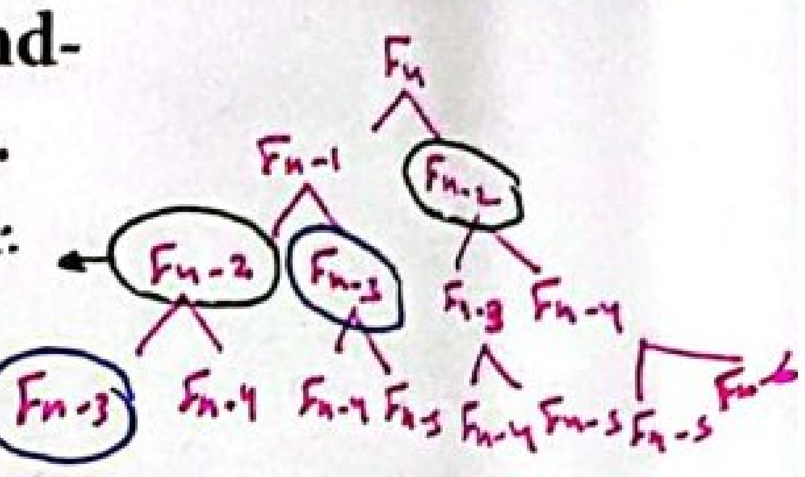
Bottom
Bottom up
up

down
F_0 1 → Base case
F_1 1 +
F_2 2 + → $F_1 + F_0$
F_3 3 + → $F_2 + F_1$
F_4 5 → $F_3 + F_4$
⋮
$F_n = F_{(n-1)} + F_{(n-2)}$
TOP

Lecture 11: Dynamic Programming Matrix-chain Multiplication Problem

Course Learning Outcome

- Use algorithm design methods, such as exhaustive search, divide-and-conquer and dynamic programming, to develop efficient algorithms.



Black cycle: تكرار ال Sub problem

مكرر نفس ال (Sub problem)

Dr. Khalil Yousef

not efficiant when subproblem recure over and over ⟸ (Soultion: use dynamic prog.)

not be attention (Divide and Conquer) كونه في تكرار في حالة ال (Subproblem)

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

(Base-case لـ نتوصل ال التذكر من الحل كثير ← (memoization Table) (more efficiant) ← نستخدم
(Soultion نتأكد اذا في ال إلهم ونتذكرهم
لنضيف اله ال Subproblem يلي وصلنا (Solution) إلهم)

(Dynamic programming) ⟸ (Bottom up) ال more better تكراراته ليشغل من

متى استخدم ال dynamic programming ?!

Read Chapter 15 of *Introduction to Algorithms*

هو الحالات يلي نستخدم فيها ال dynamic programming ?!

(optimization problems [1] (نكون فيها اكثر من حل و نختار افضل واحد

(مكررة subproblem عدد) subproblem recurse again and again [1]

# of distmic or unige subprobla $O(n^2)$ [2]
(memo. Table) بحدد حجم ال

Key word discribe : (مثل Shortest, smallest, longest, largest, min, max)
(Compute or Find multiple Solution and than Select)
(Compair with a given threshold and divide) → Decision problem [2] (are more harder than the optimization problem)
Yes NO

* dynamic programming Techniqes, Conside the matrix chain multiplication optimization problem.

Dynamic Programming

Dynamic programming is a metatechnique (not an algorithm) like the divide-and-conquer method. It is used to create algorithms for problems that can be solved by combining solutions to smaller subproblems.

However, it is a method that is most effective when a subproblems recur again and again in other subproblems.

In such a case, the divide-and-conquer techniques would redo the subproblems each time, resulting in much unnecessary work.

Dynamic programming is often used for solving **optimization problems**, in which a set of choices must be made to arrive at an optimal solution to a problem (minimizing or maximizing some value associated with the problem). Note that there may be more than one optimal solution to a problem.

- Optimization problems: e.g find the shortest path
- Decision problems (YES/NO): e.g is there is a path of k (threshold or less) ...

Dynamic Programming *continued*

Examples:

- Minimizing scalar multiplications in a chain of matrix multiplications
- Scheduling problems
- Longest common subsequence in two sequences

Example Dynamic programming:-
Matrix chain multiplication
↓
minimization : optimization problem

(Dimintion)
A_1 A_3
P : < 1, 5, 2, 3, 4 >
P_0 P_1 P_2 P_3 P_4
A_2 A_4
↓
(usually a product vector is given to represent the matrices)

Σ dimintion
4 matrices
$A_1 \cdot A_2 \cdot A_3 \cdot A_4$
1x5 5x2 2x3 3x4
$P_0 x P_1$ $P_1 x P_2$ $P_2 x P_3$ $P_3 x P_4$

Consider Four matrices
Find the best paratesization that will result in the minimum # of Scalar multiplication

matrices [illegible] (" ")

[illegible]
$A_1 (A_2 \cdot (A_3 \cdot A_4))$
$(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$
$((A_1 \cdot A_2) \cdot A_3) \cdot (A_4)$
$(A_1 \cdot (A_2 \cdot A_3) \cdot A_4)$

Base Case → Cost of the multiply A_1 and A_2
$P_0 x P_1$ $P_1 x P_2$
rows عدد
Column عدد
Same لازم يكونوا نفس بعض عشان يكون عندنا
Valid multiplication
↓
only multiply two matrices
Consecutive (ورا بعض)

$A_1 \cdot A_2 = P_0 * P_1 * P_2$
Scalar or Pairwise Scalar multiplication

m-table
multiplication cost ①
Choice of division in memo. Table ②
Table
S-table
Bottom up
(Store)

K=1 (choice)
A_1 ① $\cdot A_2$
A_2 ② $\cdot A_3$ → K=2
A_3 ③ $\cdot A_4$ → K=3
Base case
2
2 matrices

K=1 ←
$A_1 \cdot A_2 \cdot A_3$
A_1 ① $(A_2 \cdot A_3)$
$(A_1 \cdot A_2)$ ② $\cdot A_3$ → K=2
Table
3 matrices
Choice the Best { K=1, K=2 } and store in the memo. table

Dynamic Programming Steps

K=2
K=3
$A_2 \cdot A_3 \cdot A_4$
A_2 ② $(A_3 \cdot A_4)$
$(A_2 \cdot A_3) \cdot A_4$
Choice the Best + store the table

The development of a dynamic programming algorithm is characterized by four steps:

$A_1 \cdot (A_2 \cdot A_3 \cdot A_4)$ K=1
$(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$ K=2
$(A_1 \cdot A_2 \cdot A_3) \cdot A_4$ K=3 → (Choice the Best)
4 matrices

Bottom up
$F_0 = 1$
$F_1 = 1$
Fibonacci series
→ Store in memoization Table
$F_2 = F_1 + F_0$ → Restor from The table = 1+1 = 2] Store

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Compute an optimal solution from the computed information.

(خطوة مهمة)

Step of Recursive :
[1] Perform Renaming of the indices of the matrices to allow expression all possible subproblems
→ Consider $[A_i \ldots A_K A_{K+1} \ldots A_J]$
K: division
$1 \leq i \leq n$ $i \leq K < J$ $1 \leq J \leq n$
matrices عدد

[illegible]

(Extra step usually)

$(A_i \ldots A_K)$: $(A_{K+1} \ldots A_J)$
m(i,K) Choice of division of K m(K+1,J)
where m() is a Function would Compute and store the min. # of Scalar multiplication For multiply Sequence

Solution to those subproblem must be Recoverd from the memo. table ~~exer~~ except for the base cases.

The first three steps form the basis of dynamic programming, and the last is needed only if we want to report an optimal solution, not just return its value.

possible of Sub problem (optimal solution) [illegible]
→ Recursive equation

Goal : Compute m(1,n)
A_i A_K A_J
P_{i-1} P_K P_J
$= P_{i-1} * P_K * P_J$

(the ~~most~~ most important step in any dynamic programming problem) ∴ $\left[m(i,J) = \min \left(m(i,K) + m(K+1,J) + P_{i-1} * P_K * P_J \right) \right]$
$i \leq K < J$ (Check all Possible K: choices)

Example: Matrix-chain Multiplication

We will first discuss the dynamic programming method in terms of a chain of n matrices to be multiplied. For example, the following chain

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4$$

can be computed in the following ways:

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))) \quad or$$
$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)) \quad or$$
$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4)) \quad or$$
$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4) \quad or$$
$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

Two important properties to consider Dynamic Programming method :

1. overlapping subproblem
   distinct subproblem = $O(n^2)$
2. optimal substructure
   the optimal solution to the general problem containing optimal solution to the subproblem.

This parenthesization can have a dramatic impact on the cost of evaluating the product.

Example: Matrix-chain Multiplication

The main idea of this dynamic programming example is to answer the following question: Which parenthesization will give the best (the cheapest cost) i.e. How many basic operations at minimum will it take under the best parenthesization to multiply the sequence $A_1, A_2, ..., A_n$.

m- table + m- Recursive equation

Thus we have two questions to answer: (1) what does it cost to get the best parenthesization and (2) How to get it.

- Natural way: try all possible parenthesization
- So, for each parenthesize (division point), make a recursive call on the left side of the division point and another recursive call on the right side. Then compute the combined cost remembering (momization) if this division point is the best so far (the cheapest).
- A recursion call is terminated if the problem is small enough and give the trivial answer (optimal solution).
- Return the best cost found

بكل مرة

Example: Matrix-chain Multiplication

Note that: The idea in dynamic programming is different than quick sort (i.e. divide and conquer)

- In quick sort, we only look at the location of one pivot.
- In dynamic programming, we look at all of the pivot's possible locations to answer the question of what is the best pivot placement in the quick sort algorithm.

Note that, as we will see in this example
Dynamic programming = optimal division point + overlapping subproblems.

The above in red defines the Elements of Dynamic Programming

Elements of Dynamic Programming

- **Optimal Substructure:** The optimal solution is built from optimal solutions to subproblems. In the case of the matrix-chain algorithm, if $A_1 \cdot A_2 \cdot \ldots \cdot A_k$ is a prefix subchain of an optimal parenthesization of $A_1 \cdot A_2 \cdot \ldots \cdot A_n$, then $A_1 \cdot A_2 \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot A_{k+2} \cdot \ldots \cdot A_n$ must be optimally parenthesized. (This allows using divide and conquer approach i.e. when independent smaller optimization problem of the same form contribute to the solution of the original problem)

- **Overlapping Subproblems:** The space of subproblems must be small (in the sense that the recursive algorithm solves many of the subproblems over and over again) for this method to be useful. The number of distinct problems should be polynomial. In the case of the matrix-chain algorithm, the number of subproblems as we will see, is $\Theta(n^2)$.

Example: Matrix-chain Multiplication

The general approach of Dynamic programming (Top down approach) can be summarized as follows:

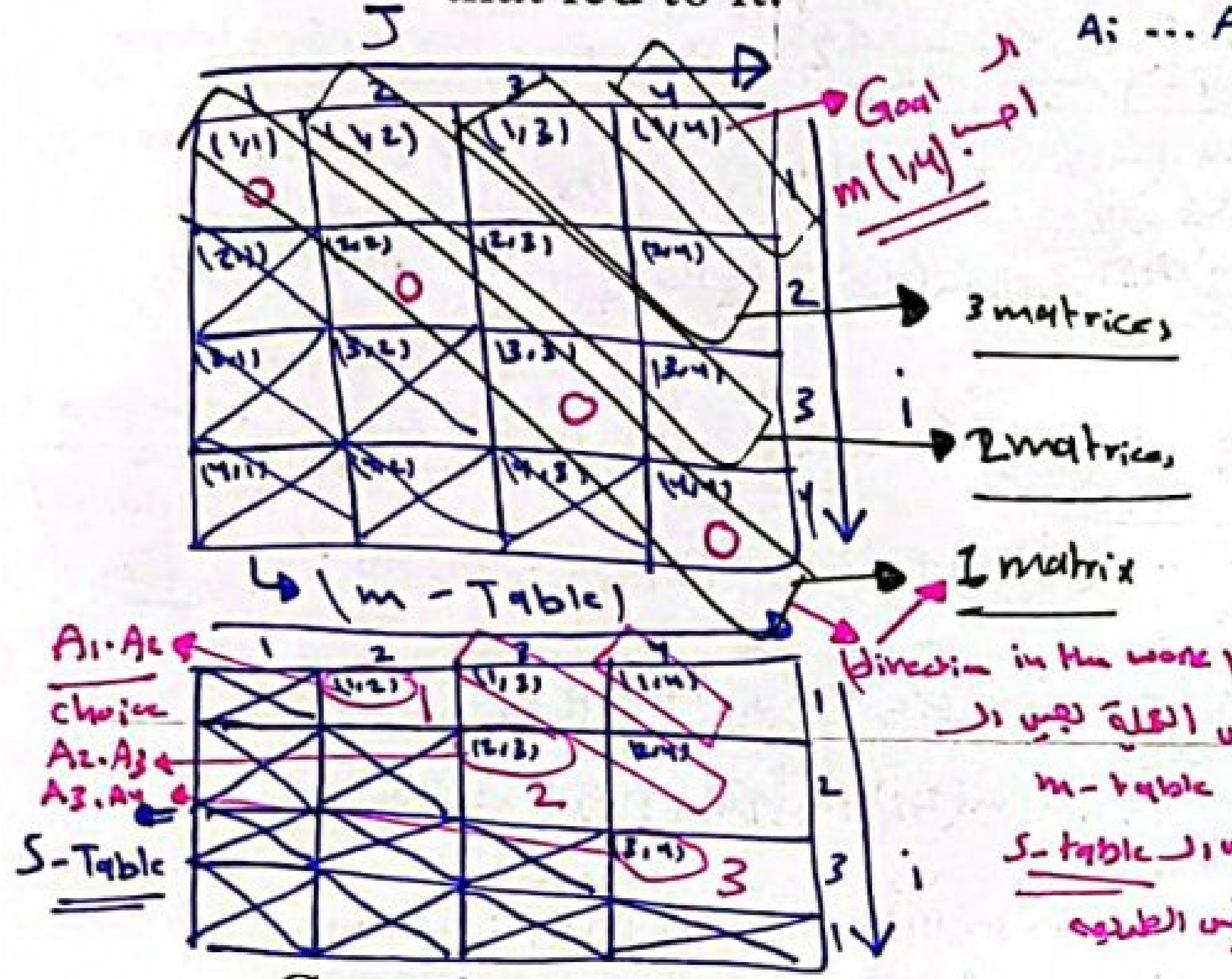
(تذكر)

- **Memoization**: Check if the subproblem was already solved (trivial solution)
  - For each parenthesize (choice or division point)
    - ∗ Divide and Conquer: make a recursive call on the left side of the division point and another recursive call on the right side.
    - ∗ Compute the combined cost checking if this division point (choice) was the best so far (the cheapest).
  - Return and remember the best cost (solution value) found and the choice that led to it.

$A_1 A_2 A_3 A_4$

$A_i \ldots A_k \ldots A_j$: $\boxed{\begin{matrix} 1 \le i \le n-1 \\ i \le k < j \\ i < j \le n \end{matrix}}$

منعرفش ال i < J ⇒ لسه مكتبناش بداخل ال cell ☒



$m(i,j) = \underset{i \le k < j}{\min} [m(i,k) + m(k+1,j) + P_{i-1} * P_k * P_j]$

S-table دي (K) اللي تطلع بتخزنها في

نفس العملية بتاعت ال m-table

بجيب ال S-table بنفس الطريقة $S(i,j)$

Counting Scalar Multiplications in Matrix Multiplication

```
MATRIX-MULTIPLY(A, B)
1. if columns[A] ≠ rows[B]
2.     then error "incompatible dimensions"
3.     else  for i ← 1 to rows[A]
4.               do for j ← 1 to columns[B]
5.                      do  C[i, j] ← 0
6.                          for k ← 1 to columns[A]
7.                              do C[i, j] = C[i, j] + A[i, k] · B[k, j]
8.           return C
```

$\boxed{n \times m} \cdot \boxed{m \times l} = \boxed{n \times l}$

The number of scalar multiplications is: $n \times m \times l$

The Matrix-Chain Multiplication Problem

The matrix-chain multiplication problem: given a chain $\langle A_1, A_2, \ldots, A_n\rangle$ of n matrices, such that for $i = 1, 2, \ldots, n$, matrix A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ in a way to minimize the number of scalar multiplications.

Example: $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ such that the dimensions of A_1 is 10×20, A_2 is 20×50, A_3 is 50×1, and A_4 is 1×100.

تكملة ذلك

Cost 2200, but how it was obtained

$((A_1)(A_2)(A_3))A_4$ $\therefore \Rightarrow ((A_1)((A_2)\cdot(A_3)))(A_4)$

① step

② step

③ step

$A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)) \rightarrow 125{,}000$ scalar multiplications

① الـ S-table (3)

② نجيب Choice 1 الـ Parentheses 1,3

③ نجيب 2,3 2

(طريقة التوصل للنتائج)

$(A_1 \cdot (A_2 \cdot A_3)) \cdot A_4 \rightarrow 2{,}200$ scalar multiplications

Ex) $P = [10, 20, 50, 1, 100]$ ($P_0\ P_1\ P_2\ P_3\ P_4$; $A_1\ A_2\ A_3\ A_4$)

	1	2	3	4
1	0	10000	1200	2200
2		0	1000	3000
3			0	5000
4				0

m-Table

Done ②

Done ① ☑

Done ①

$A_1A_2 \rightarrow m(1,2) = \min[m(1,\cancel{K}) + m(\cancel{K+1},2) + P_0 P_{\cancel{K}} P_2 = 10\times20\times50 = 10000$

$1 \le K < 2 \Rightarrow \boxed{K=1}$

$A_2, A_3 \rightarrow m(2,3) = \min(m(\cancel{2},2) + m(\cancel{3},3) + P_1 * P_2 * P_3 = 20 * 50 * 1 = 1000$

$2 \le K < 3 \Rightarrow \boxed{K=2}$

$A_3, A_4 \rightarrow m(3,4) = \min(m(3,3) + m(4,4) + P_2 P_3 P_4 = 50 * 1 * 100 = 5000$

$3 \le K < 4 \Rightarrow \boxed{K=3}$

$m(1,3)$ $A_1A_2A_3$ Possible choice $K=1,2$

$\rightarrow m(1,3) = \min(m(\cancel{1,1}) + m(2,3) + P_0 P_1 P_3 = \boxed{1200}$ (1000+200) $\boxed{K=1}$ choice

$\rightarrow m(1,3) = \min(m(1,2) + m(3,3) + P_0 P_2 P_3 = 10500$ $K=2$

(ما بنحرزنا جدول table نقارن مع choice الثاني)

Compare and select the min

$m(2,4)$ $A_2A_3A_4$

$\rightarrow m(2,4) = \min(m(\cancel{2,2}) + m(3,4) + P_1 P_2 P_4 = 105000$ $K=2$

$\rightarrow m(2,4) = \min(m(2,3) + m(\cancel{4,4}) + P_1 P_3 P_4 = \boxed{3000}$ $\boxed{K=3}$

Done ② ✓

	2	3	4	
	1	1	3	1
		2	3	2
			3	3

S-Table

عند الـ (3) أعمل Parentheses

$A_1A_2A_3A_4$

$\rightarrow m(1,4) = \min(m(\cancel{1,1}) + m(2,4) + P_0 P_1 P_4 = 23000$ $K=1$

$\rightarrow m(1,4) = \min(m(1,2) + m(3,4) + P_0 P_2 P_4 = 65000$ $K=2$

$\rightarrow m(1,4) = \min(m(1,3) + m(\cancel{4,4}) + P_0 P_3 P_4 = \boxed{2200}$ $\boxed{K=3}$

How to Choose the Best Parenthesization

Exhaustive checking will not provide an efficient algorithm because this would result in an exponential running time algorithm. To see this let's devise a recurrence:

Let $P(n)$ be the number of alternative parenthesizations for an n matrix chain. We can split the sequence between the kth and $(k+1)$st matrix for any $k = 1, 2, \ldots, (n-1)$, and then parenthesize the remaining subsequences, giving the recurrence:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \ge 2. \end{cases}$$

Note that this recurrence has a closed form solution of $P(n) = C(n-1)$, where $C(n)$ is the nth Catalan number, and $C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(\frac{4^n}{n^{3/2}})$.

1. Characterizing the Structure of an Optimal Parenthesization

To characterize the structure of an optimal solution for the matrix-chain multiplication problem, we must split the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ between A_k and A_{k+1} for some integer $1 \leq k < n$.

$A_{i..j}$ denotes the matrix resulting from multiplying $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

Using this notation, the cost of the optimal parenthesization is the cost of computing $A_{1..k}$ and $A_{(k+1)..n}$ along with the cost of multiplying the two matrices together.

Note that if $A_1 \cdot A_2 \cdot \ldots \cdot A_k$ is a prefix subchain of an optimal parenthesization of $A_1 \cdot A_2 \cdot \ldots \cdot A_n$, then $A_1 \cdot A_2 \cdot \ldots \cdot A_k$ must be optimally parenthesized. If not, then we could provide a lower cost alternative. An optimal solution to the matrix-chain multiplication problem must contain optimal solutions to subproblems.

2. Defining the Value of an Optimal Solution Recursively

Let $m[i,j]$ denote the minimum number of scalar multiplications to compute $A_{i..j}$; hence, $m[1,n]$ is the minimum number to compute $A_{1..n}$. Note that $A_{i..i} = A_i$ requires no scalar multiplications; hence, $m[i,i] = 0$ for $i = 1, 2, \ldots, n$.

Assuming that the optimal parenthesization splits the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ between k and $k+1$ where $i \leq k < j$, then $m[i,j]$ is the minimum cost of computing the subproducts $A_{i..k}$ and $A_{(k+1)..j}$ plus the cost of multiplying them together, $p_{i-1}p_kp_j$:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

Note that there are $j-1-i+1 = j-i$ possible values of k to check for a given i and j. Below is the recursive definition for the minimum cost of parenthesizing the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j & \text{if } i < j. \end{cases}$$

3. Compute the Optimal Solution Bottom-Up

MATRIX-CHAIN-ORDER is a bottom-up algorithm for computing the optimal solution, which takes a single argument $p = \langle p_0, p_1, \dots p_n \rangle$, where the dimension of A_i is $p_{i-1} \times p_i$, for $i = 1, 2, \dots, n$.

It uses an auxiliary table $m[1..n, 1..n]$ to store the $m[i, j]$ costs and $s[1..n, 1..n]$ to record the index of k that achieves the optimal cost in computing $m[i, j]$.

```
MATRIX-CHAIN-ORDER($p$)
1. $n \leftarrow length[p] - 1$
2. for $i \leftarrow 1$ to $n$
3.     do $m[i,i] \leftarrow 0$   *
```

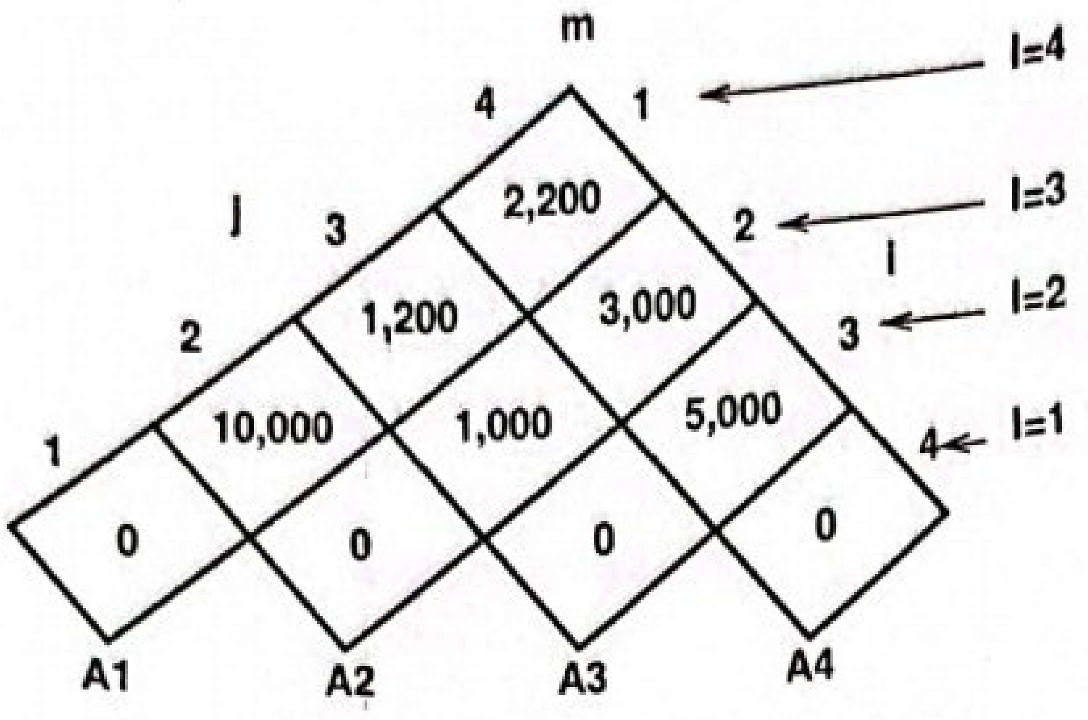
3. Compute the Optimal Solution Bottom-Up *continued*

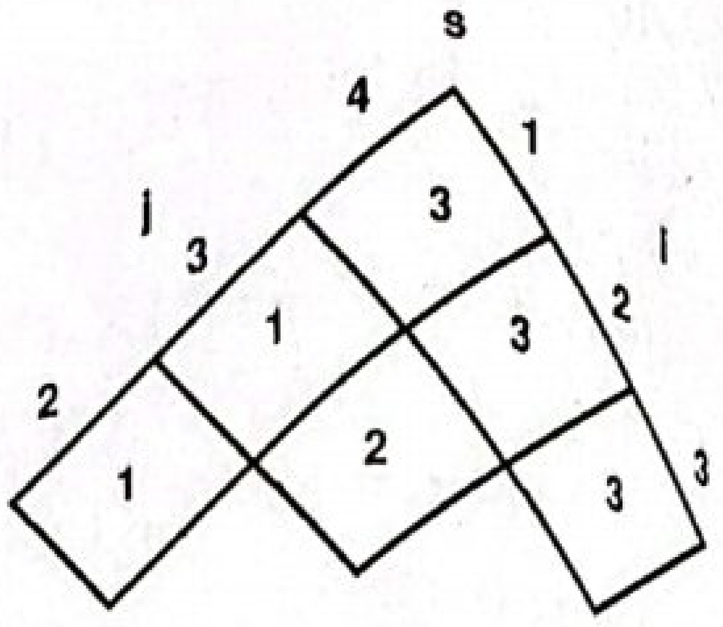
```
          index = L
4. for $l \leftarrow 2$ to $n$
5.     do for $i \leftarrow 1$ to $n - l + 1$
6.            do $j \leftarrow i + l - 1$
7.           ** $\boxed{m[i,j] \leftarrow \infty}$
8.            * for $k \leftarrow i$ to $j - 1$
9.                * do $\boxed{q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j}$
10.                     if $q < m[i,j]$
11.                        then $m[i,j] \leftarrow q$
12.                             $s[i,j] \leftarrow k$
13. return $m$ and $s$
```

At each step, the $m[i, j]$ computed in lines 9-12 depends only on the table entries $m[i, k]$ and $m[k+1, j]$ already computed.

MATRIX-CHAIN-ORDER **Example**

Consider MATRIX-CHAIN-ORDER(p), where $p = \langle 10, 20, 50, 1, 100 \rangle$.





MATRIX-CHAIN-ORDER **Example** *continued*

The computations are provided below:

$$m[1,2] = min_{1 \le k < 2}\{m[1,1] + m[2,2] + p_0 p_1 p_2\} = 10,000$$
$$m[2,3] = min_{2 \le k < 3}\{m[2,2] + m[3,3] + p_1 p_2 p_3\} = 1,000$$
$$m[3,4] = min_{3 \le k < 4}\{m[3,3] + m[4,4] + p_2 p_3 p_4\} = 5,000$$

$$\begin{aligned} m[1,3] = min_{1 \le k < 3}\{&m[1,1] + m[2,3] + p_0 p_1 p_3 = 1,200, \\ &m[1,2] + m[3,3] + p_0 p_2 p_3 = 10,500\} = 1,200 \end{aligned}$$

$$\begin{aligned} m[2,4] = min_{2 \le k < 4}\{&m[2,2] + m[3,4] + p_1 p_2 p_4 = 105,000, \\ &m[2,3] + m[4,4] + p_1 p_3 p_4 = 3,000\} = 3,000 \end{aligned}$$

$$\begin{aligned} m[1,4] = min_{1 \le k < 4}\{&m[1,1] + m[2,4] + p_0 p_1 p_4 = 23,000, \\ &m[1,2] + m[3,4] + p_0 p_2 p_4 = 65,000, \\ &m[1,3] + m[4,4] + p_0 p_3 p_4 = 2,200\} = 2,200 \end{aligned}$$

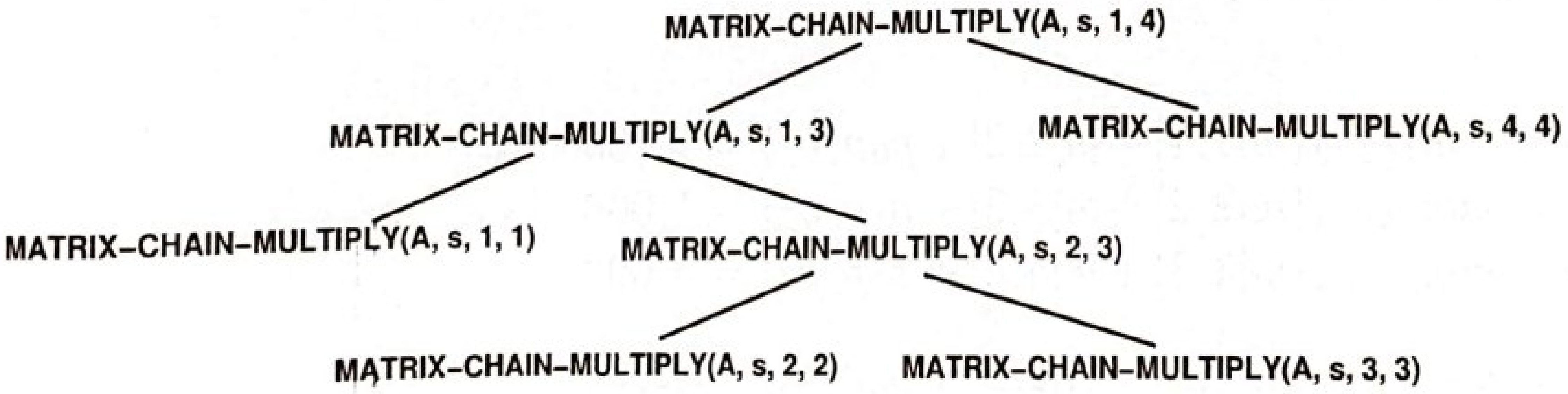
4. Constructing the Optimal Solution

The following algorithm uses the table $s[1..n, 1..n]$ to determine the best way to multiply the matrices.

```
MATRIX-CHAIN-MULTIPLY$(A, s, i, j)$
if $j > i$
    then $X \leftarrow$ MATRIX-CHAIN-MULTIPLY$(A, s, i, s[i, j])$
         $Y \leftarrow$ MATRIX-CHAIN-MULTIPLY$(A, s, s[i, j] + 1, j)$
         return MATRIX-MULTIPLY$(X, Y)$
    else return $A_i$
```

For our example, this computes the matrix multiplication as $((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$.

Constructing the Optimal Solution for the Example


MATRIX–CHAIN–MULTIPLY(A, s, 1, 4)
MATRIX–CHAIN–MULTIPLY(A, s, 1, 3)
MATRIX–CHAIN–MULTIPLY(A, s, 4, 4)
MATRIX–CHAIN–MULTIPLY(A, s, 1, 1)
MATRIX–CHAIN–MULTIPLY(A, s, 2, 3)
MATRIX–CHAIN–MULTIPLY(A, s, 2, 2)
MATRIX–CHAIN–MULTIPLY(A, s, 3, 3)

Complexity of MATRIX-CHAIN-ORDER

The complexity of the algorithm for $2 \leq l < n$ is computed as follows:

$$
\begin{aligned}
T(n) &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{j-1} 1 && (1)\\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (j-i) && (2)\\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (i+l-1-i) && (3)\\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1) && (4)\\
&= \sum_{l=2}^{n} (l-1)(n-l+1) && (5)\\
&= \sum_{l=1}^{n-1} (n-l)\cdot l = n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2 && (6)\\
&= \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6} && (7)\\
&= \frac{n^3-n}{6} = \boxed{O(n^3)} && (8)
\end{aligned}
$$

(3)

$\underline{\underline{n = \text{loop}}}$ لكل For loop

let p(n): number of all possible n-matrices parantesization of the sequence Ai ··· An

$$\boxed{p(n) = \sum_{K=1}^{n-1} p(K) * p(n-K)} \quad n \geq 2 \qquad \underbrace{(A_1 \cdots A_K)}_{K} \underbrace{(A_{K+1} \cdots A_n)}_{n-K}$$

$= \Omega\left(\frac{4^n}{n^{3/2}}\right)$ $\underline{\underline{\text{exponential}}}$

Lecture 12: Greedy Algorithms

Fractional Knapsack Problem

→ Greedy Algorithms have with Dynamic program.
- → Similarities → optimal substructure + Applied to optimization problem.
- → differancies → ① there is (greedy) no assumption of having precomputed Solution to the subproblem. → one of the advantages of the greedy over the dynamic program. (الإيجابية ما في storage
- ② ther is no gurantee to acheive or reach to the optimal Solution of the underlying problem. → disaduntge → dynamic programming → ther is a gurantee to acheive or reach to the optimal solution

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

→ uniq properity (Dynamic programming): Greedy choice → make locally optimal choices in the hope to reach an optimal Solution.

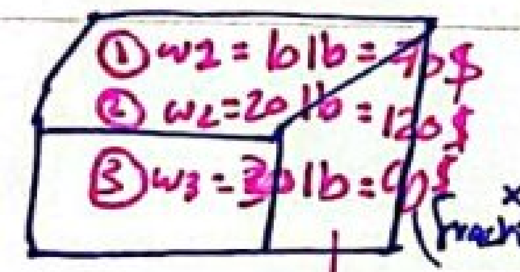
Ex) Computing the shortest path from a single Source in ~~the~~ a graph.

Read Chapter 16 of *Introduction to Algorithms*

diffant example
- → Knapsack problem → Integer / → Fraction
- → minimum-spanning-tree algorithms] work in the graph problem
- → Single Source Shortest path

فكرة الـ Knapsack Solution

شنطة + الها مساحة محددة ممنوع تتجاوزها (اقيمها item الكمية الي وزنا)



Example

item (n=3), W = 50lb = 23 Kg

Solution 1 (binary → 0 → ما باخذه / 1 → باخذه) ⇒ Dynamic programming

x (fraction) لـ ما بقدر اجزئها

احتمالات
- → w1 + w2
- → w2 + w3
- → w1 + w2 + $\frac{2}{3}$w3 (Fractional Knapsack) → greedy

حسب الـ (greedy choice) بختار الـ items

حسب نسبة الـ (value / weight) بعبي الـ band الواحد قبل الثاني لحد ما تتعبى الـ KnapSack فل

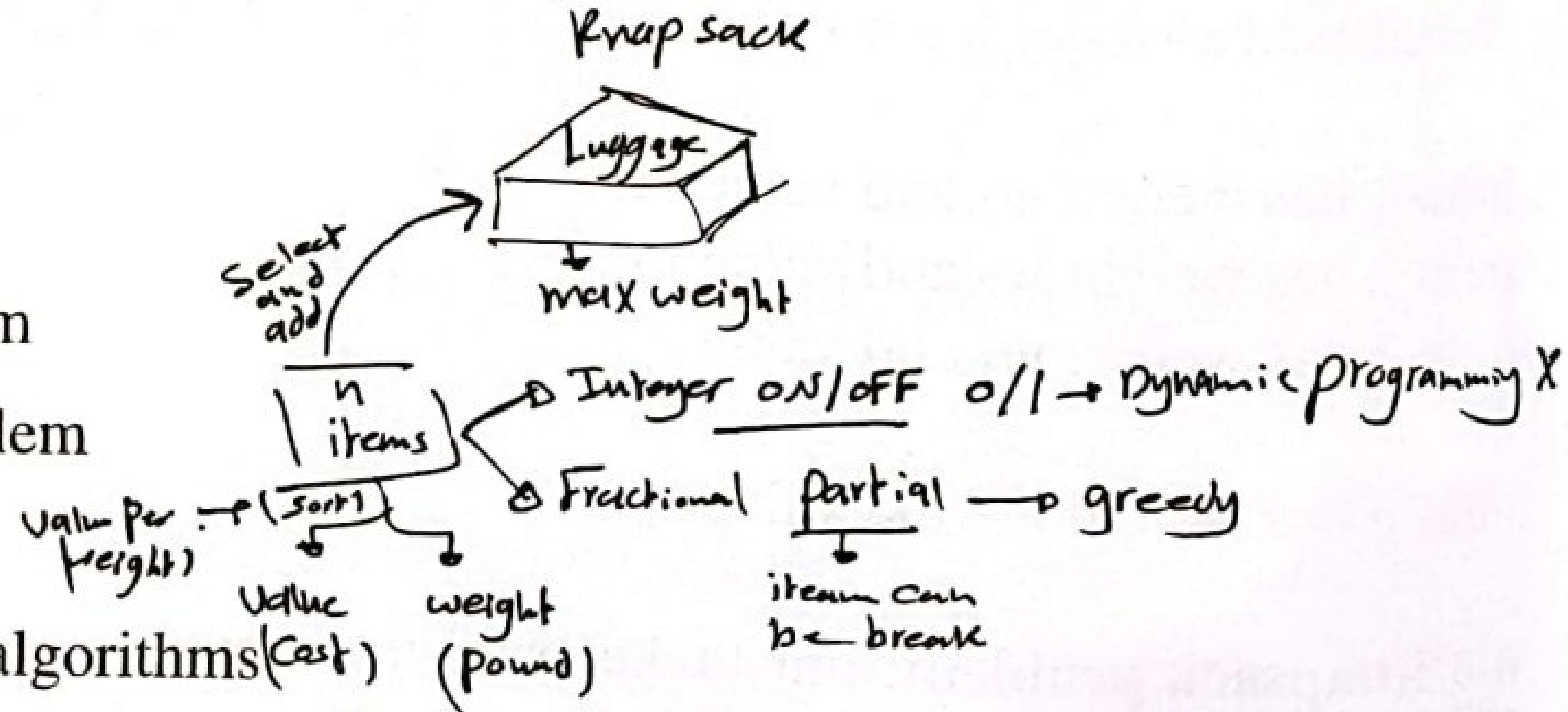
Greedy Algorithms

A greedy algorithm makes the choice that looks best at the moment, i.e., it makes a locally optimal choice in hopes of achieving a globally optimal solution.

Dis It is a powerful and widely applicable method, but it may not always lead to the optimal solution.

Examples:

- Activity-selection problem
- Fractional knapsack problem
- Huffman coding
- Minimum-spanning-tree algorithms
- Single-source shortest path



- → Integer ON/OFF 0/1 → Dynamic programming X
- → Fractional partial → greedy (item can be break)

Value (cost) weight (pound)

① sort the items based on the ratio $\frac{\text{Value}}{\text{weight}}$

② continuous selected the item and added based on largest ratio until the Knapsack is Full

Complexity : O(nlgn)

When is a Greedy Algorithm Applicable?

A greedy algorithm obtains an optimal solution by making a sequence of choices that seem best at the moment they are chosen. This heuristic strategy; however, does not always produce an optimal solution for all problems.

There are two properties that are exhibited by most of the problems that lend themselves to a greedy strategy:

مشكلة في عملية ال storge ماالقدر التصميم م ①
ال dynamic

- **Greedy-choice property:** It is possible to make locally-optimal choices in order to arrive at a globally optimal solution. Note that greedy choices are made without relying on having solutions to subproblems first. This is an important difference from the dynamic programming method

②

- **Optimal substructure:** The optimal solution contains subproblems which are optimal solutions to subproblems. This property is important for the dynamic programming method as well.

Greedy Strategy versus Dynamic Programming

To illustrate how to choose between dynamic programming and a greedy approach, we will consider two variants of a classical optimization problem, **the knapsack problem**. The basic idea is to maximize the value of goods in a knapsack, which can only hold W pounds. Note that each item to consider for packing has both a weight and a value:

item 1 has weight w_1 and value v_1.
item 2 has weight w_2 and value v_2.
item 3 has weight w_3 and value v_3.
⋮
item n has weight w_n and value v_n.

0-1 knapsack problem: must take or leave the entire item.

Fractional knapsack problem: can take fractional items.

Greedy Strategy versus Dynamic Programming *continued*

Both knapsack problems exhibit the optimal-substructure property:

- **0-1 knapsack problem:** consider the most valuable load weighing W pounds. If we remove j, the remaining load must be the most valuable load weighing $W - w_j$ using the $n - 1$ items excluding j.

- **fractional knapsack problem:** consider removing a weight w of one item j from an optimal load weighing W pounds. The remaining load must be the most valuable load weighing $W - w$ given the $n - 1$ items and $w_j - w$ pounds of j.

Even though the fractional knapsack problem can be solved with a greedy approach, the 0-1 knapsack problem cannot be.

Greedy Strategy for the Fractional Knapsack Problem

- First, for each item i, compute the value per pound $\frac{v_i}{w_i}$ and sort the items by this value in $O(n \lg n)$ time.

- Obeying a greedy strategy, take as much of the item with the **greatest value per pound**.

- If there is still capacity, continue with the next most valuable item, taking as much as possible.

- Continue with the next most valuable item until the knapsack is full.

- This greedy algorithm runs in $O(n \lg n)$ time.

↳ Sorting

Greedy Strategy for the 0-1 Knapsack Problem

Consider the packings for a 0-1 knapsack problem with the following items and a knapsack with capacity, $W = 50$ pounds.

item 1: $w_1 = 10, v_1 = \$70, \frac{v_1}{w_1} = \$7$ (ratio)

item 2: $w_2 = 20, v_2 = \$120, \frac{v_2}{w_2} = \$6$ (ratio)

item 3: $w_3 = 30, v_3 = \$135, \frac{v_3}{w_3} = \$4.5$ (ratio)

1. **The greedy packing:** pick item 1 and then item 2, leaving a capacity too small for item 3. The weight is 30 pounds, the value is $190. → this is optimal ?! No (ممكن يطلع افضل)
2. **A better non-greedy packing:** pick items 1 and 3, leaving a capacity too small for item 2. The weight is 40 pounds, the value is $205. (better) → this is optimal ?! → binary : yes, Fraction : NO نقدر احصل افضل
3. **An optimal non-greedy packing:** pick items 2 and 3, leaving a capacity too small for item 1. The weight is 50 pounds, the value is $255.

This problem can be solved using dynamic programming because of the overlapping subproblems and optimal-substructure property.

weight eddges

Lecture 13: Minimal Spanning Trees

Course Learning Outcome

- Use fundamental graph algorithms, like traversal, shortest path and spanning tree in the solution of real-life problems

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

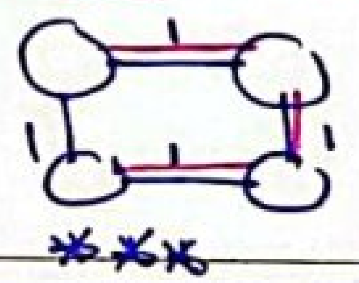
Read Sections 21.1-21.3 and Chapter 23 of *Introduction to Algorithms*

MST : Q : Is it min.spaning tree unique?! and if it is not, when this can happen

optimal solution "MST" is unige iff the graph Contains distinct weight (weight دupplict اذا ما في) | Cost=3)

|V| vertice

|V-1| MST = eddge



***

MST in this example is not unige, in fact every spaning tree here is a MST

Spanning Trees

A **spanning tree** $S=(V,T)$, for a connected undirected graph $G=(V,E)$ is an undirected tree (i.e., connected and acyclic) that connects all vertices V with $T \subseteq E$.

vertices → subset of eddges

(loop) cycle يعني ما فيه

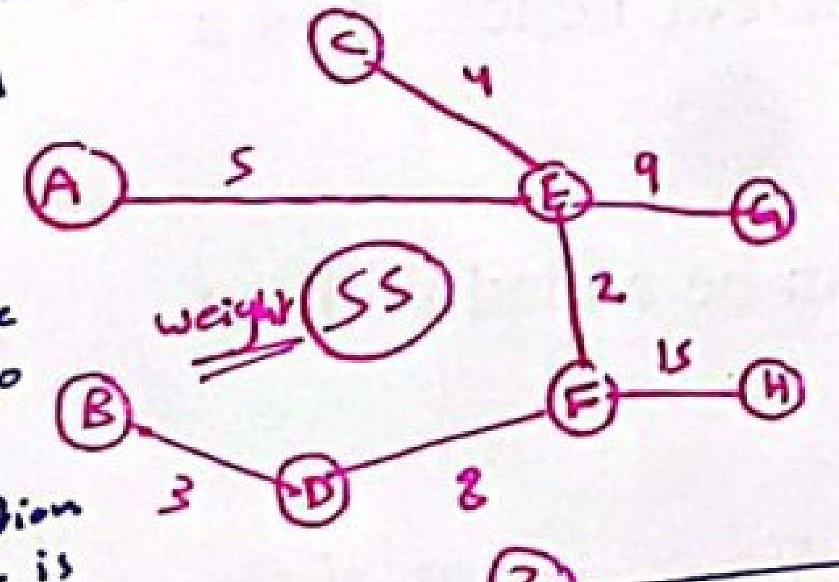
كامل ال E او جزء منها)

A **minimum spanning tree** (or MST) for a connected undirected graph $G=(V,E)$, given a weight function $w: E \to R$, is the spanning tree $S=(V,T)$ for which $w(T)=\sum_{(u,v)\in T} w(u,v)$ is minimized. For example:

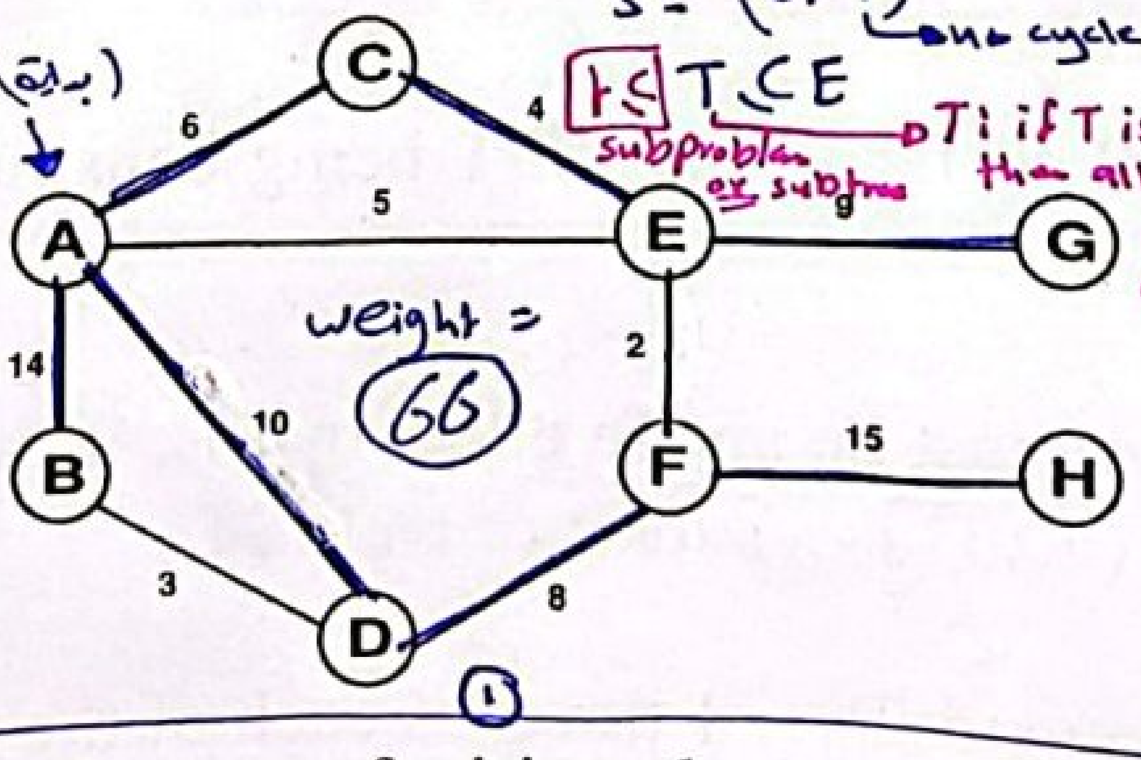
MST min. Spanning tree's (not unige)

اطلع على ال Solution

Effective Solution to this optimization problem is found through



(بداية)

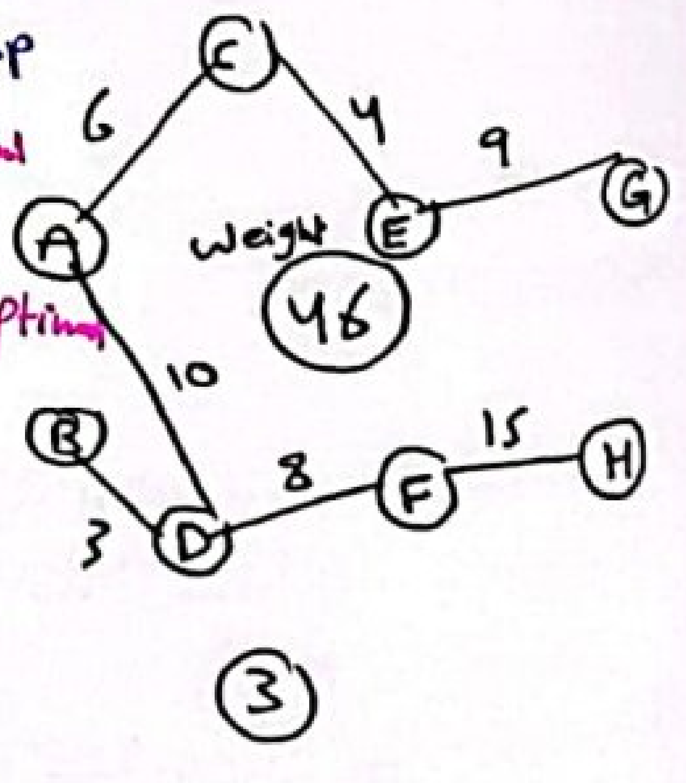


S = (V, T) all vertices; one cycle or loop

t ⊆ T ⊆ E

subproblem or subtree

T: if T is optimal then all t's are also optimal



2 **Applications:** Minimize the cost of wiring electronic circuits. → greedy Algo. (تتبعل)

Greedy Alg. → must ① Greedy choice → (no cycle) Safe and light weight edge (least cost) means ⇒ Generic min spaning tree

② optimal substructure

MST which initialy empty ← A = { } keep adding stop when A is indeed a spaning tree)

Optimal Substructure Property of MST

An MST has the optimal substructure property in that an optimal MST contains optimal subtrees.

- To demonstrate this, consider an MST for the connected undirected graph $G = (V,E)$, T, containing an edge (u,v).
- If we remove the edge, T is partitioned into two subtrees T_1 and T_2.
- T_1 must be an MST of $G_1 = (V_1, E_1)$, the subgraph of G induced by the vertices of T_1 (i.e., V_1 are the vertices of T_1 and $E_1 = \{(x,y) \in E : x,y \in V_1\}$).
- Similarly, T_2 must be an MST of G_2.
- Because $w(T) = w(u,v) + w(T_1) + w(T_2)$, there cannot be a more optimal tree than T_1 or T_2; otherwise, T would be suboptimal.

GENERIC-MST

```
GENERIC-MST($G, w$)
1. $A \leftarrow \emptyset$   (empty) initial
2. while $A$ does not form a spanning tree ⇒ $A = \{\emptyset\}$ empty  تبدأ ب
3.         do find an edge $(u,v)$ that is safe for $A$
4.             $A \leftarrow A \cup \{(u,v)\}$        ↳ no cycle + least weight (cost)
5. return $A$
```

A is the set of edges added so far to the MST being constructed; hence, it is a subgraph of an MST.

Each edge (u,v) added to A must be a **safe edge**, that is, it can be added without violating the fact that $A \cup \{(u,v)\}$ is a subset of an MST.

The loop in lines 2-4 is executed $|V| - 1$ times as each of the $|V| - 1$ edges of the MST is determined. Initially, there are $|V|$ trees, with each iteration reducing the number by 1. When there is a single MST, the algorithm terminates.

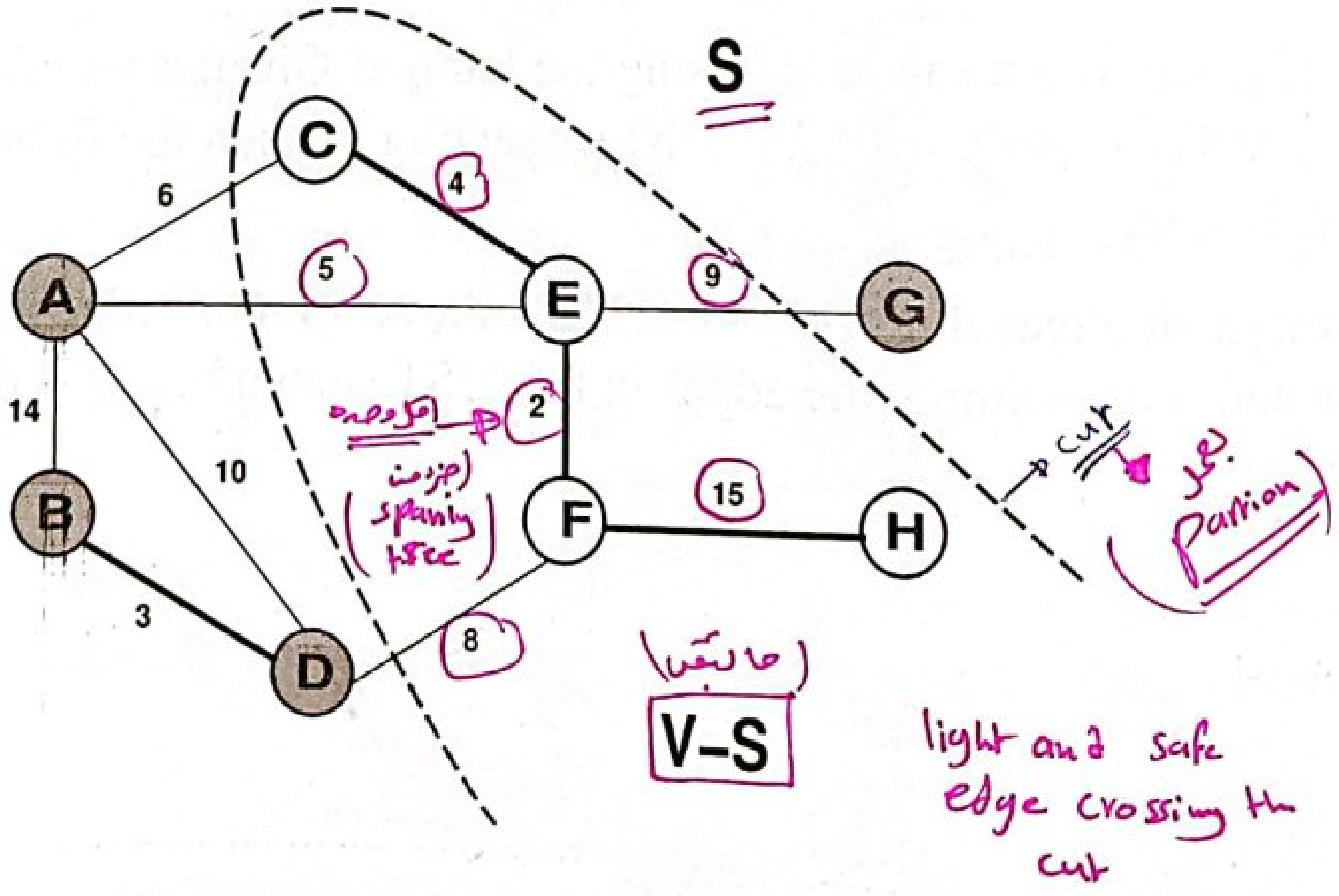
Terminology for Safe Edge Selection

We need some definitions to help us to understand GENERIC-MST:

فكرة ال Cut في السؤال التالي

- A **cut** $(S, V-S)$ of an undirected graph $G=(V,E)$ is a partition of V.
- An edge $(u,v) \in E$ **crosses** the cut $(S, V-S)$ if one endpoint is in S and the other is in $V-S$.
- A cut **respects** the set A if no edge in A crosses the cut.
- A **light edge** is the the minimum-weight edge crossing the cut.
- A **safe edge** is an edge that can be added to A, a subset of the MST, so that is is still a subset of the MST.

Example Illustrating Terminology



light and safe edge crossing the cut

4, 5, 2, 9, 15, 8

A={(B,D), (C,E), (E,F), (F,H)}

بختار اقل وحدة ال (2) ∴ لازم تكون جزء من spanning tree بحيث بغير عدد cut لا تتجاوز ال (2) وترك الباقي

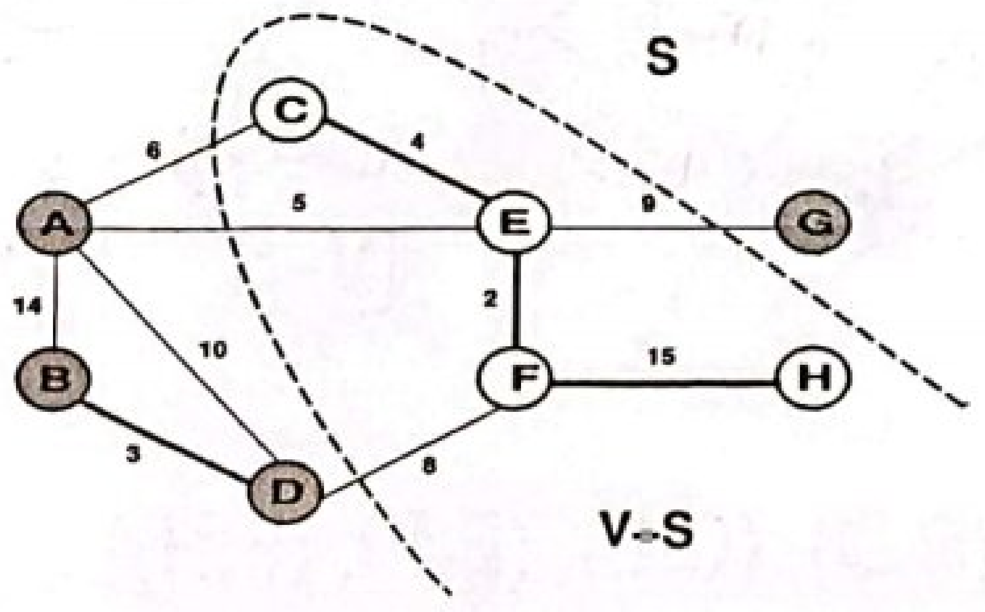
نضيفه ل (A) S = {E,F}

MST: Important Notes:

Distinct weights in the a graph guarantee that the minimum spanning tree of the graph is unique. Without this condition, there may be several different minimum spanning trees. For example, if all the edges have weight 1, then every spanning tree is a minimum spanning tree with weight $V-1$.

MST: Important Notes:

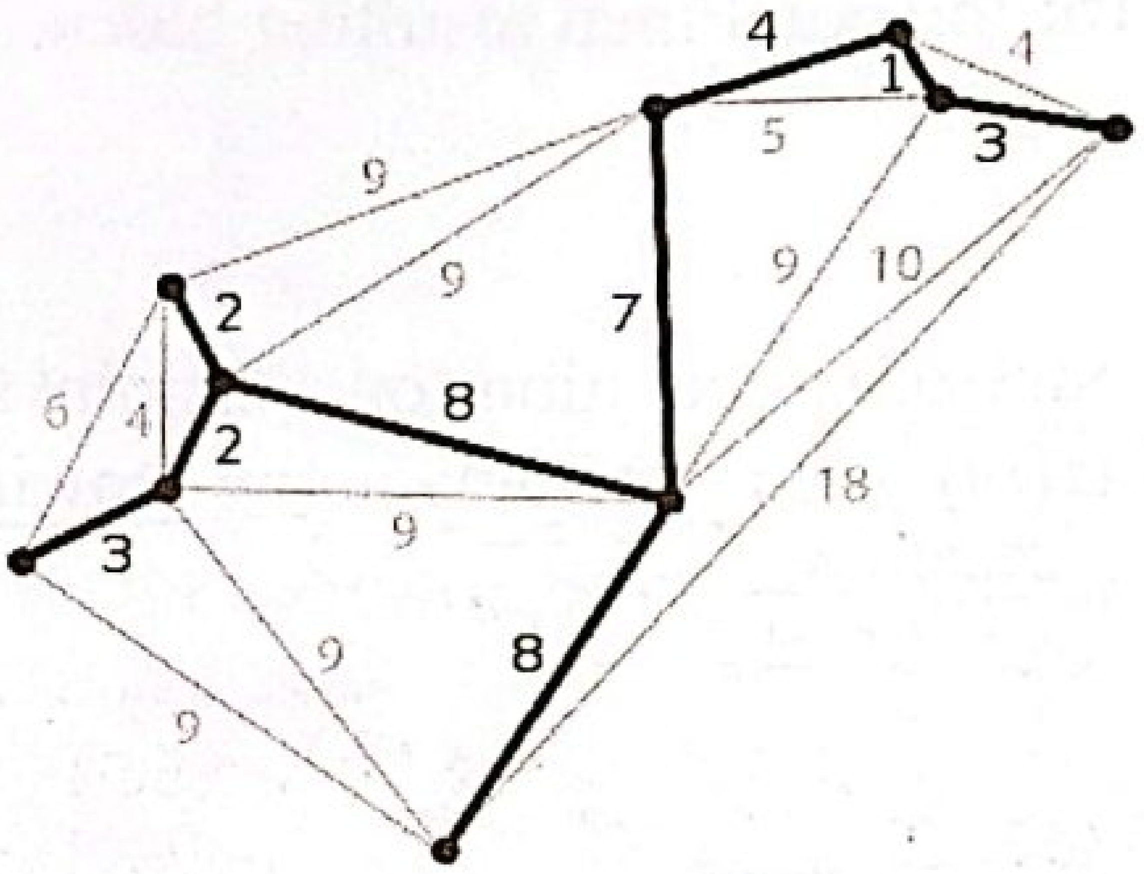
- For Greedy algorithms, there are efficient means to identify safe choices of the edges; Prim and Kruskal Algorithms.
- For MST, we find a safe edge using the lemma: Given a set of edges A that is part of MST A and a cut $(S, V-S)$ respecting A, then the light edge crossing the cut is safe.
- Given a set of edges A that is part of MST, the set of these edges is not necessary connected. For example the edges in the MST shown below is disconnected.



A={(B,D), (C,E), (E,F), (F,H)}

MST: Important Notes:

- The MST actual algorithm doesn't developed yet; because how to find a cut and how to find a lightest edge. The following algorithms describe the safe edge but use the same Generic MST.
  - Kruskal's MST Algorithm
  - Prim's MST Algorithm



→ Requires using special disjoint set operation

make-set
FIND-SET
Union

are associated with low order Time complexity

Kruskal's MST Algorithm

→ those operation will be used in a way to sure that we add to A a light and safe edge

Repeatidly:

Add (Update partition (tree) by UNION (u,v) i.e. $A = A \cup \{(u,v)\}$)

the lightest (lowest weight edge)

legal edge to A (doesn't cause a cycle to A; Question: how to find this edge; This edge should join two different blocks of the partitions (trees) satisfying the condition that FIND-SET(u) $\neq$ FIND-SET(v)).

ماتعملي loop

Note that UNION (u,v), FIND-SET(u), FIND-SET(v), and others are operations on a disjoint data structure. We will now describe some of these operations.

Kruskal's MST Algorithm

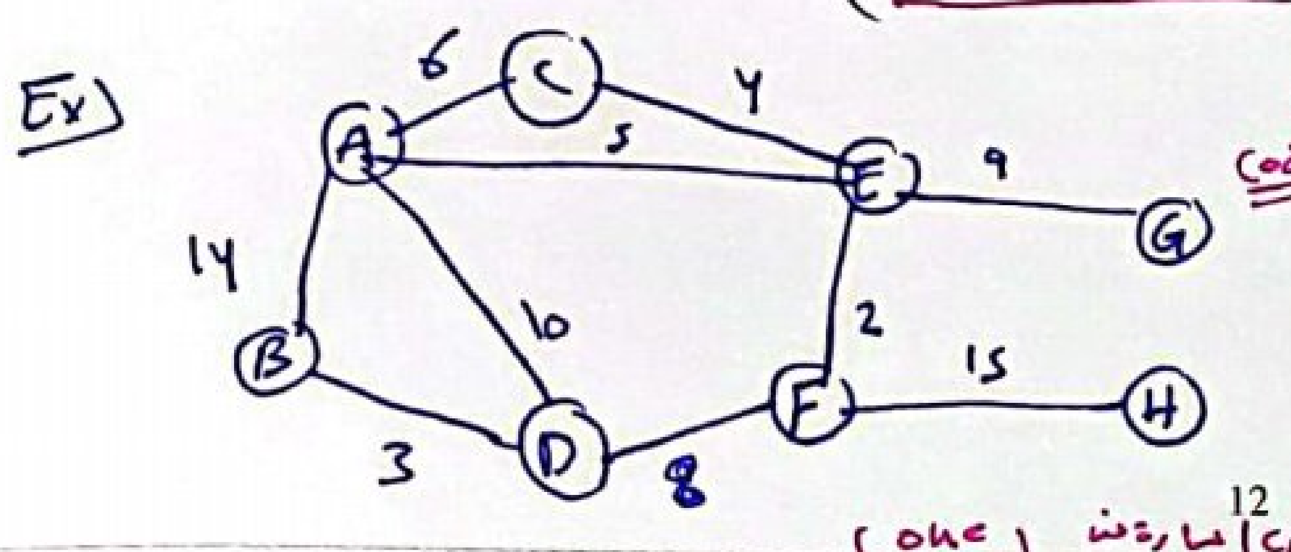
Disjoint Set Data structure operations:

- **MAKE-SET**(x): adds x to the partition, extending the domain. (تعمل الـ createset)
- **FIND-SET**(x): returns the representative element (name of the set) of the partition block containing x.
- **UNION**(x, y): (بيجمع) combine the partition blocks for x and y and choose a new representative element for the combined resulting block.

Running time cost: A sequence of m operations on a disjoint set data structure runs in $O(m\alpha(m))$ time, where $\boxed{\alpha(m)} \leq 4$ (extremely a low growing function)

(Constant time)

Ex)



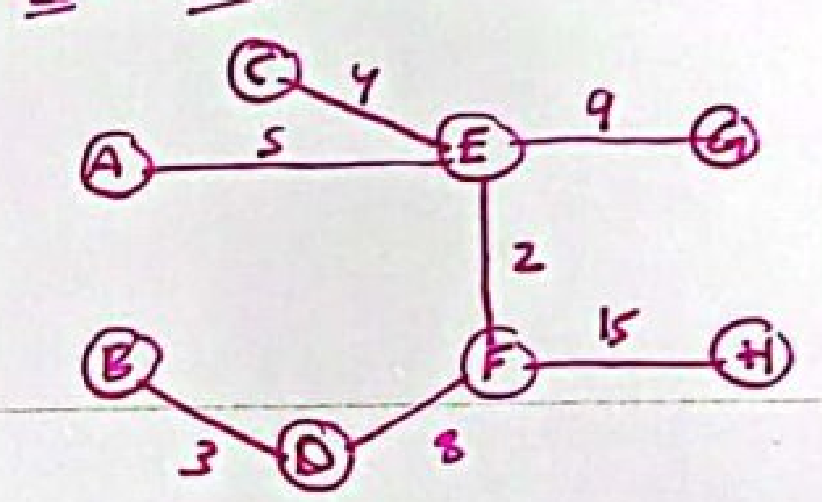
Step 4 مثال الـ code

Sorting edges weight ①

✓ (E,F) 2
✓ (B,D) 3
✓ (C,E) 4
✓ (A,E) 5
cycle تعمل ⇐ × (A,C) 6
✓ (D,F) 8
✓ (E,G) 9
(cycle loop) × (A,D) 10
× (A,B) 14
✓ (F,H) 15

(one set) بارت من (C,E,A) 12

② بنعمل edge ونضيف بحيث ما نعمل cycle + A



A = {(E,F), (B,D), (C,E), (A,E), (D,F), (E,G), (F,H)}

(Cost MST) 46 = 2+3+4+5+8+9+15 = weight

Kruskal's MST Algorithm (Ex ↑)

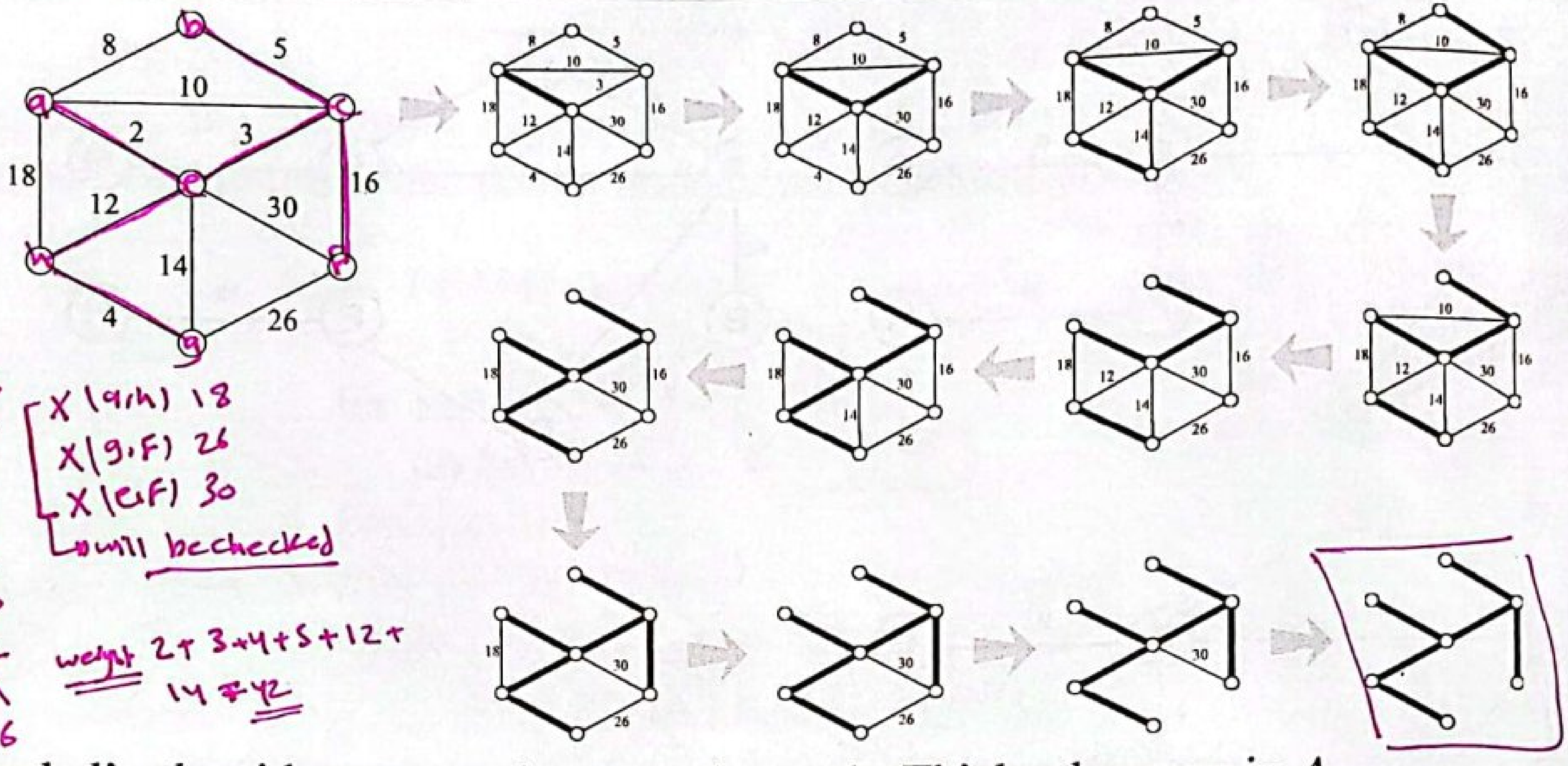
This algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees, the edge (u, v) of least weight. Let T_1 and T_2 denote two trees connected by edge (u, v). Since (u, v) must be a light edge connecting T_1 to another tree, Corollary 24.2 implies that (u, v) is a safe edge.

```
MST-KRUSKAL(G, w)
1. A ← ∅
2. for each vertex v ∈ V[G]                        كل node في graph
3.     do MAKE-SET(v)                              لعمل isolated
4. sort the edges of E by non-decreasing weight w  (من الأقل للأكبر)  (ح ارتب الـ edge في graph حسب الـ weight  O(E lg E))
5. for each edge (u, v) ∈ E, in order of w         (source, Dis)  نتابع weight من sorted
6.     do if FIND-SET(u) ≠ FIND-SET(v)             (not belong same path (not have loop))
7.         then A ← A ∪ {(u, v)}
8.              UNION(u, v)                        union من هنا اعمل
9. return A
```

أهم خطوة من خطوات الـ Algo.

Time complexity ⇒ $O(|E| \lg |E|)$

MST-KRUSKAL Example



1 Sorting
✓ (g,e) 2
✓ (e,c) 3
✓ (g,h) 4
✓ (b,c) 5
X (a,b) 8
X (a,c) 10
✓ (h,e) 12
X (e,g) 14
✓ (c,F) 16

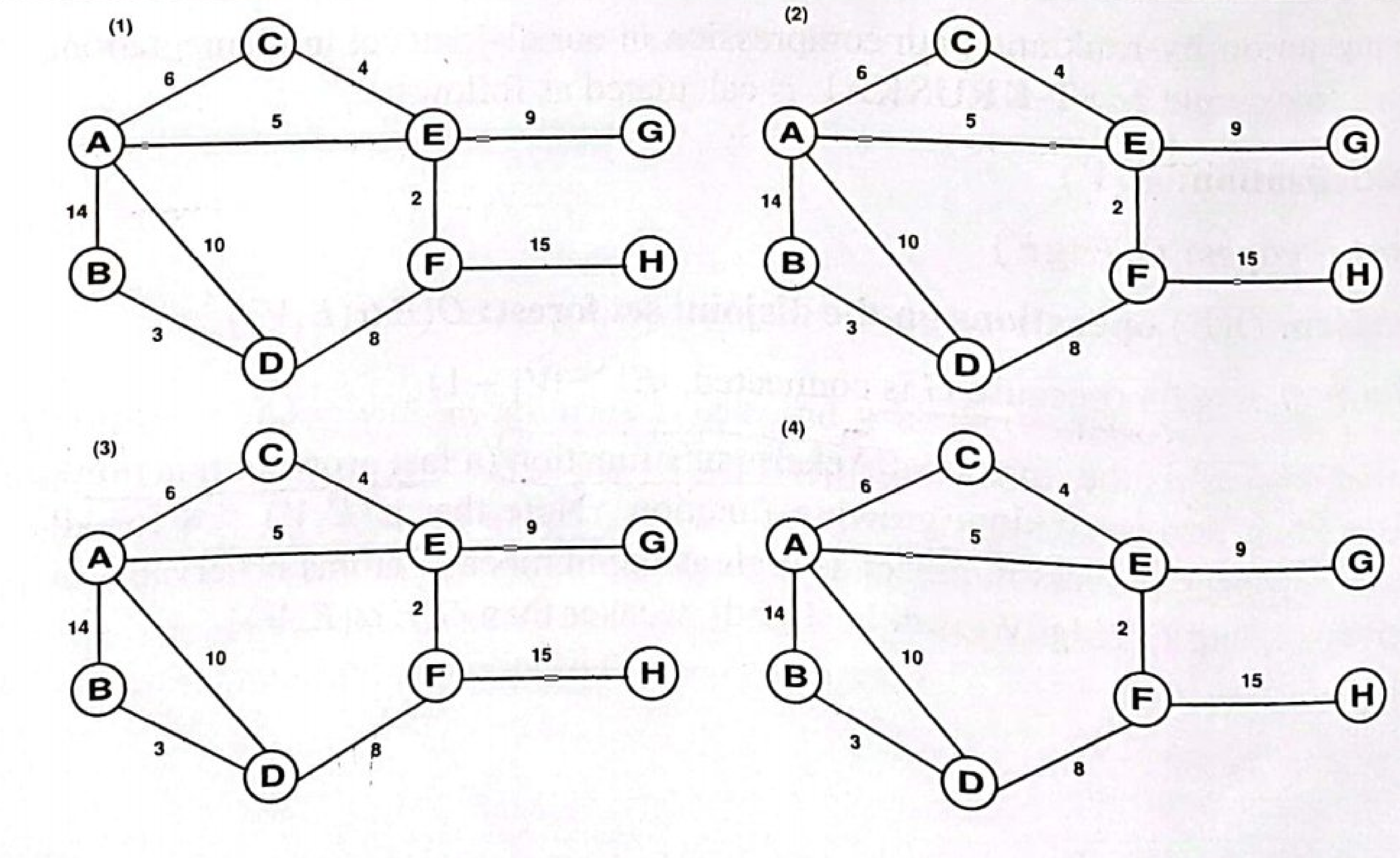
X (a,h) 18
X (g,F) 26
X (e,F) 30
will bechecked

weight 2+3+4+5+12+14 = 42

Kruskal's algorithm run on the example graph. Thick edges are in *A*.

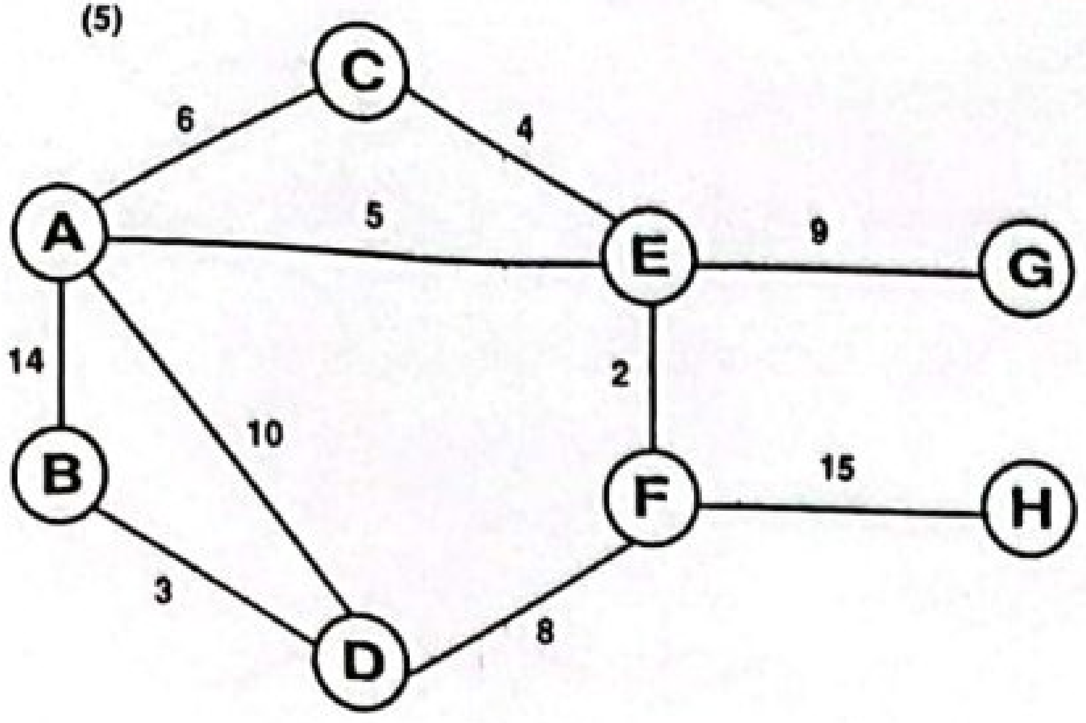
http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/13-mst.pdf

MST-KRUSKAL Example

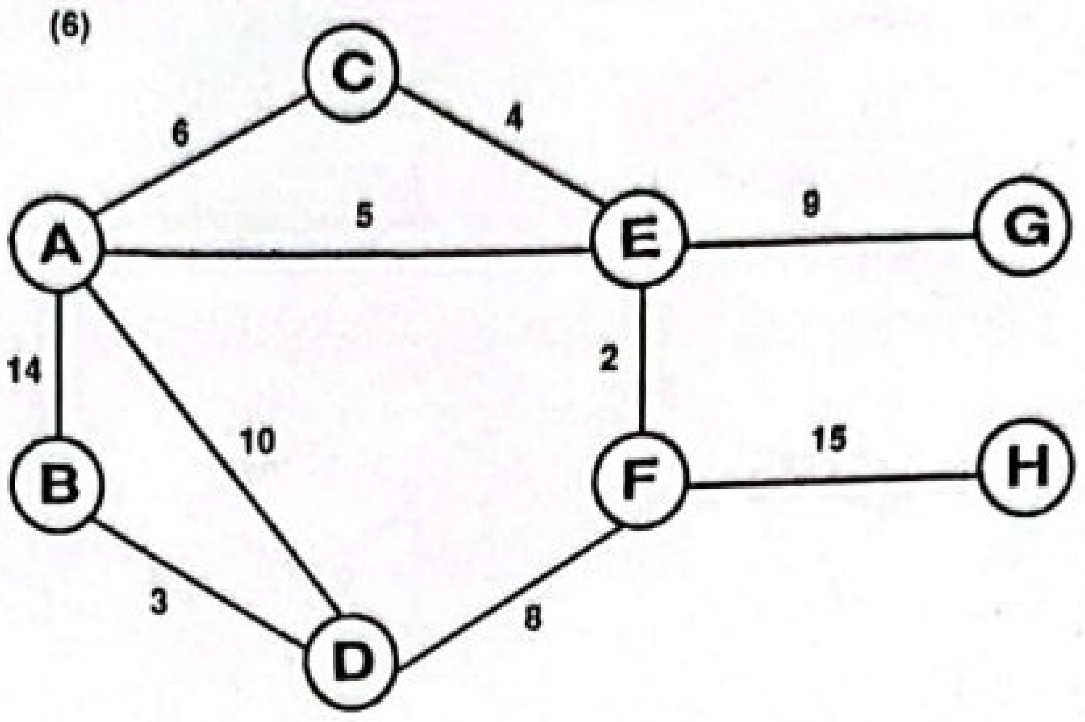


MST-KRUSKAL **Example** *continued*

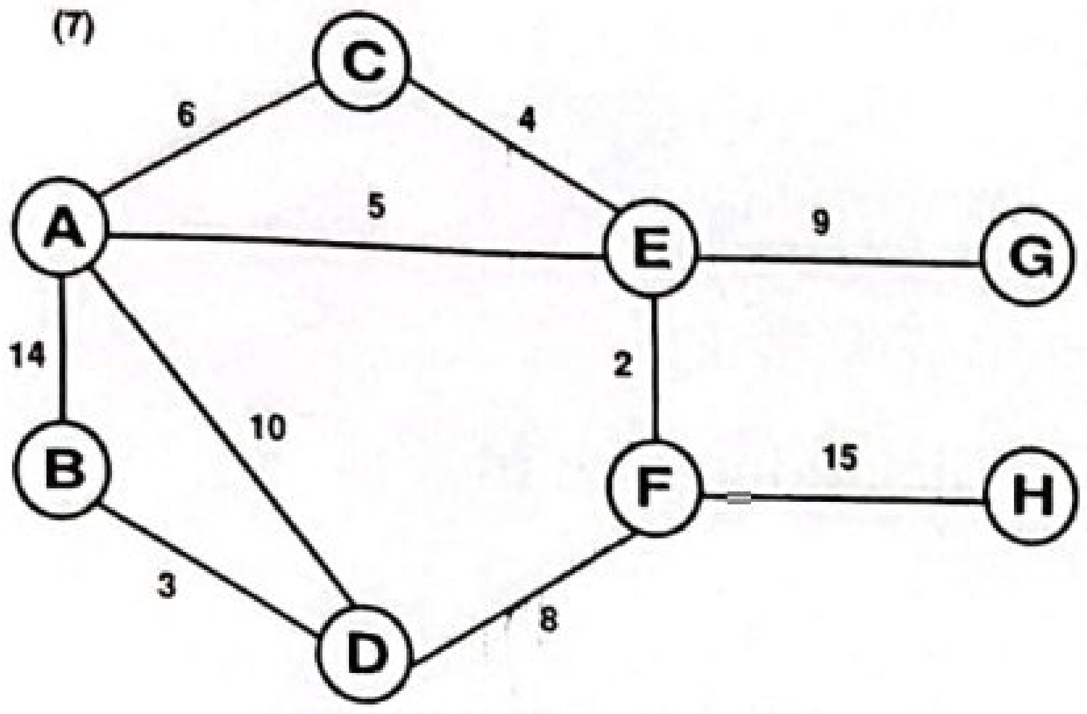
(5)



(6)



(7)



MST-KRUSKAL **Running Time**

If we use union-by-rank and path compression in our disjoint set implementation, the time to execute MST-KRUSKAL is calculated as follows:

- **Initialization:** $\Theta(V)$
- **Sort E edges:** $O(E \lg E)$
- **Perform $O(E)$ operations on the disjoint set forest:** $O(E\,\alpha(E,V))$
- **Total:** $O(E \lg E)$ (because G is connected, $|E| \geq |V| - 1$)

معكوس (α)

Note that $\boxed{\alpha(m,n)}$ is the inverse of $\boxed{\text{Ackerman's function}}$ (a fast growing function), and as such it is a very slow growing function. Note that $\alpha(E,V) \leq 4$ for all practical purposes (covers numbers as high as the number of atoms observable in the universe), and $O(E \lg^* V)$ is only slightly weaker than $O(E\,\alpha(E,V))$.

① → Very Similer to BFS (will use a Queue (Priorty Queue = min heap) + will use what we call Keys of the vertices to help us pick the light weight and safe edge

Prim's MST Algorithm

② → uses Key [v] Function for all the vertices to Keep the min. weight of any edge connecting to v from vertices in the tree → (MST)

The algorithm uses a queue, Q, to select an edge to add to A. For each vertex v, $key[v]$ keeps the minimum weight of any edge connecting to v from vertices in the tree, and $\pi[v]$ points to the parent node in the tree. Initially, $key[v] = \infty$.

③ → uses π [v] parent function to point to the Parent node in the Tree MST

④ → Input to the Algorithm
* undirected graph
* unique Source node

لحين لازم يكون فيه Source node → لا يكون الـ adj. (اضيفهم كل Queue) تراد BFS

لكن في BFS كل الـ node تعتبر unit distance فالـ parent بتعلم أماني (MST)

لا فالأمر مش كذلك فين Key بتعبر عن الـ weight للـ edges (الأشخصم) الـ weight لـ edge من Key vertices

واكمل extract min

```
MST-PRIM(G, w, r)
Q ← V[G]   (كل الـ vertice داخل الـ Q)   (min heap)
for each vertex u ∈ Q
    do key[u] ← ∞   (للكل ما عدا (E) Source node)
key[r] ← 0
π[r] ← NIL   (Source)   ← لسه مرحلة من BFS
while Q ≠ ∅   (بين دخلت Source in الـ min)   ← V lgV        DEQ في prims   حاليا
    do u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]   → (cut) الـ   ← E
            do if v ∈ Q and w(u,v) < key[v]   (Key → update)   ← heap Dec. Key: O(1) Fibonacci, O(lgV) binary
                then π[v] ← u   (Parent → update)
                     key[v] ← w(u,v)   → Dec Key
```

Time Complexity (Finally heap)
Total = $\theta(V\lg V + E\lg V)$
$= \theta((V+E)(\lg V))$

Time Complexity Fibonacci heap
$= V\lg V + E$

→ Source

~~2~~ الشخص Table						
node	a	b	--			
Key[ ]	0	∞	∞	∞	∞	∞
π [ ]	NiL	Nil	Nil	Nil	Nil	Nil

Key[ ] ← بدل distance في BFS

Recall the operation on the min heap Q

size → $|Q| = |V|$

Consider a Binary heap → Extract-Min( ) = O(lgV); → Decrease-Key( ) = O(lgV)

use other Kind of heap ⇒ Fibonacci heap → Extract Min( ) = O(lgV); → Decrease Key( ) = O(1) ← as a fact

MST-PRIM Discussion

During the course of the algorithm:

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$$

At the end:

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}$$

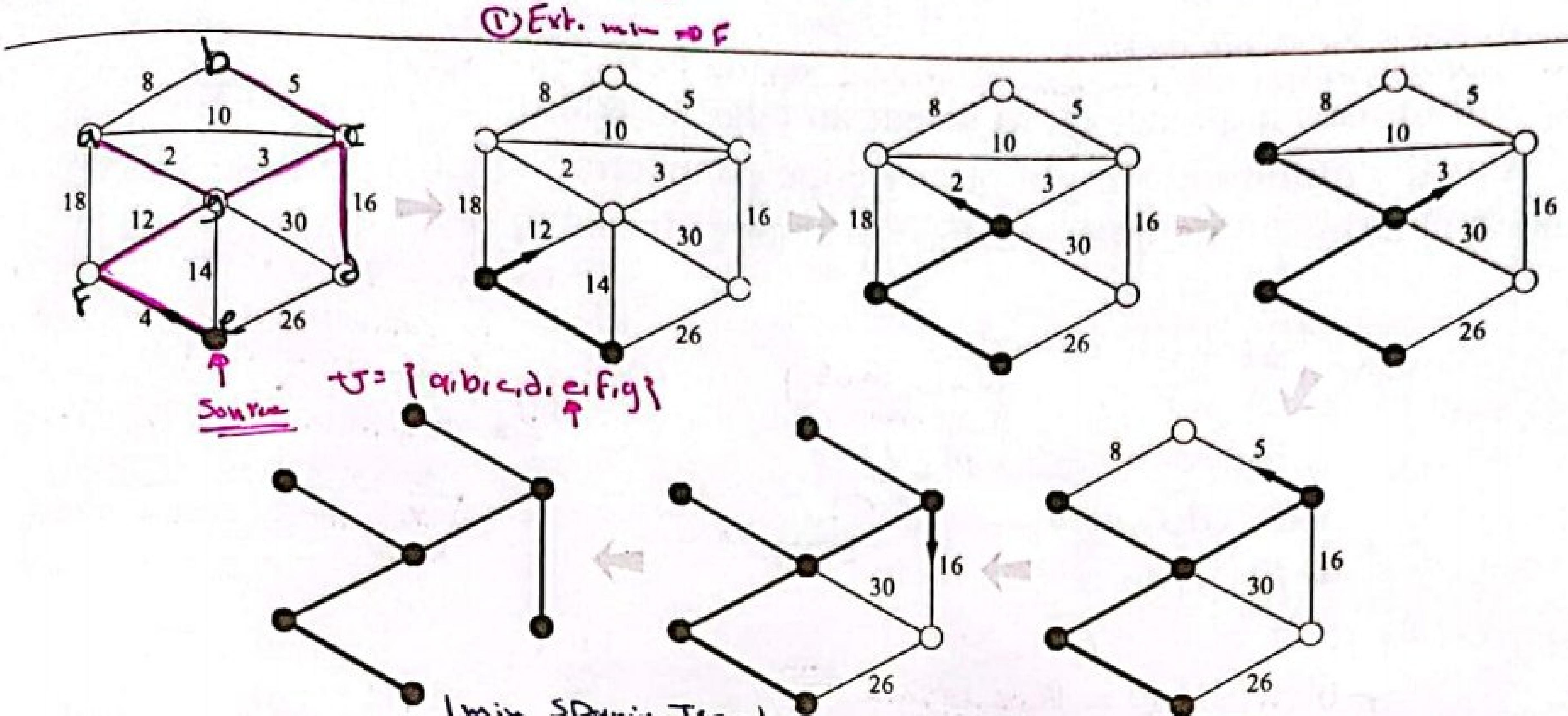
MST-PRIM starts with an arbitrary root r and it loops until it constructs a tree spanning the vertices of V. At each step, a light edge connecting a vertex in A to a vertex in $V - A$ is added to the tree. By corollary 24.2, the algorithm will add only edges that are safe for A, so the resulting tree will be an MST. The strategy is clearly a greedy one, since at each step, the tree is augmented with an edge that contributes a minimum amount to the tree's weight.

node	a	b	c	d	e	f	g
Key[ ]	~~∞~~ ~~18~~ 2	~~∞~~ ~~8~~ 5	3	~~∞~~ ~~26~~ 16	0	4	~~∞~~ ~~14~~ 12
π[ ]	~~nil~~ ~~f~~ g	~~a~~ c	g	~~e~~ c	Nil	e	~~e~~ F

Cost ⇒ 2+5+3+16+0+4+12 = 42

MST-PRIM Example

① Ext. min ⇒ F



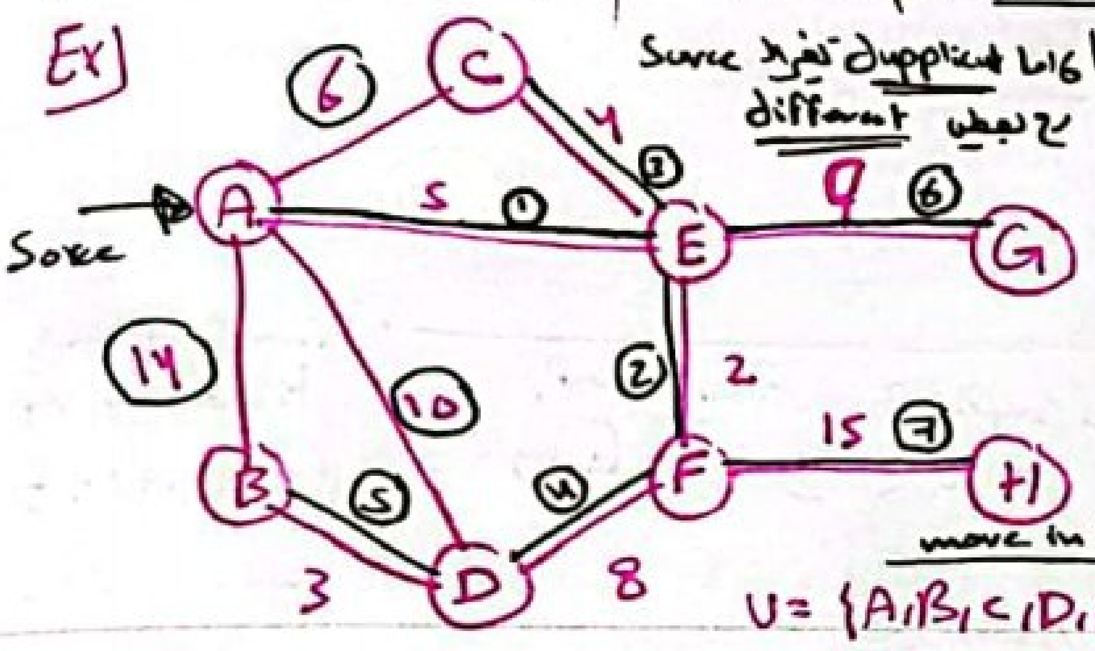
(min Spaning Tree)

PRIM's algorithm run on the example graph. Thick edges are in A.

http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/13-mst.pdf

ال (٢) Sorce مو مهم اذا المشترك

Sorce تفرق اذا duppliced weight (unique weight) different يج نعطي



move in (α) order → 20

U = {A,B,C,D,E,F,G,H}

node	A	B	C	D	E	F	G	H
Key[ ]	0	~~14~~ 3	~~6~~ 4	~~10~~ 8	5	2	9	15
π[ ]	Nil	~~A~~ D	~~A~~ E	~~A~~ F	A	E	E	F

priority Queue (min heap)

node	~~A~~	~~B~~	~~C~~	~~D~~	~~E~~	~~F~~	~~G~~	~~H~~
Key	~~∞~~	~~∞~~ 14	~~∞~~ 6 ④	~~∞~~ 10	~~∞~~ 5	~~∞~~ 2	~~∞~~ 9	~~∞~~ 15

① لعمل ال Table بجدول node

① Extract ال min ونبدع لو (A) ال Adj داخل Q

(α) order مع ال العطي

ال (B) داخل ال Q صح ال weight السابق لما خلصنا نبدع من ال edge (14) اكبر من (∞) ؟ نعم update ⇐ next adj (2) C ⇐ (6 من ∞) داخل ال Q update to

نفس الشي لل (D)

⇒ ال Q مو not empty نبعد Extract ال (E) ا نبدأ بعد شيله ال edge بين A,E

(G) ال ②

(Q) داخل ال E لل adj نشوف مين

(G,F,C)

C داخل ال Q بعد decr ونبدل ال parent (α) يعني ؟ 6 > 4 yes

تجربة نرجع ال Q ونشوف مين اقل دائم

Extract بعد كذا (F) ال F

③ ال (F) ال adj ⇐ D,H

ال E ⇐ 8 < 10

update

ال D صح ؟ (✓)

ال adj مافي

ال Q not empty

Extract ال (C) كانه اقل ماعكنا

ال edge بين (E,F)

ال A و E ال Adj و ال C ماحتاج اعمل اي اشي

التنين مش داخل ال Q

نعين ال Q لل Adj ال E

edge بين C,E + علقت

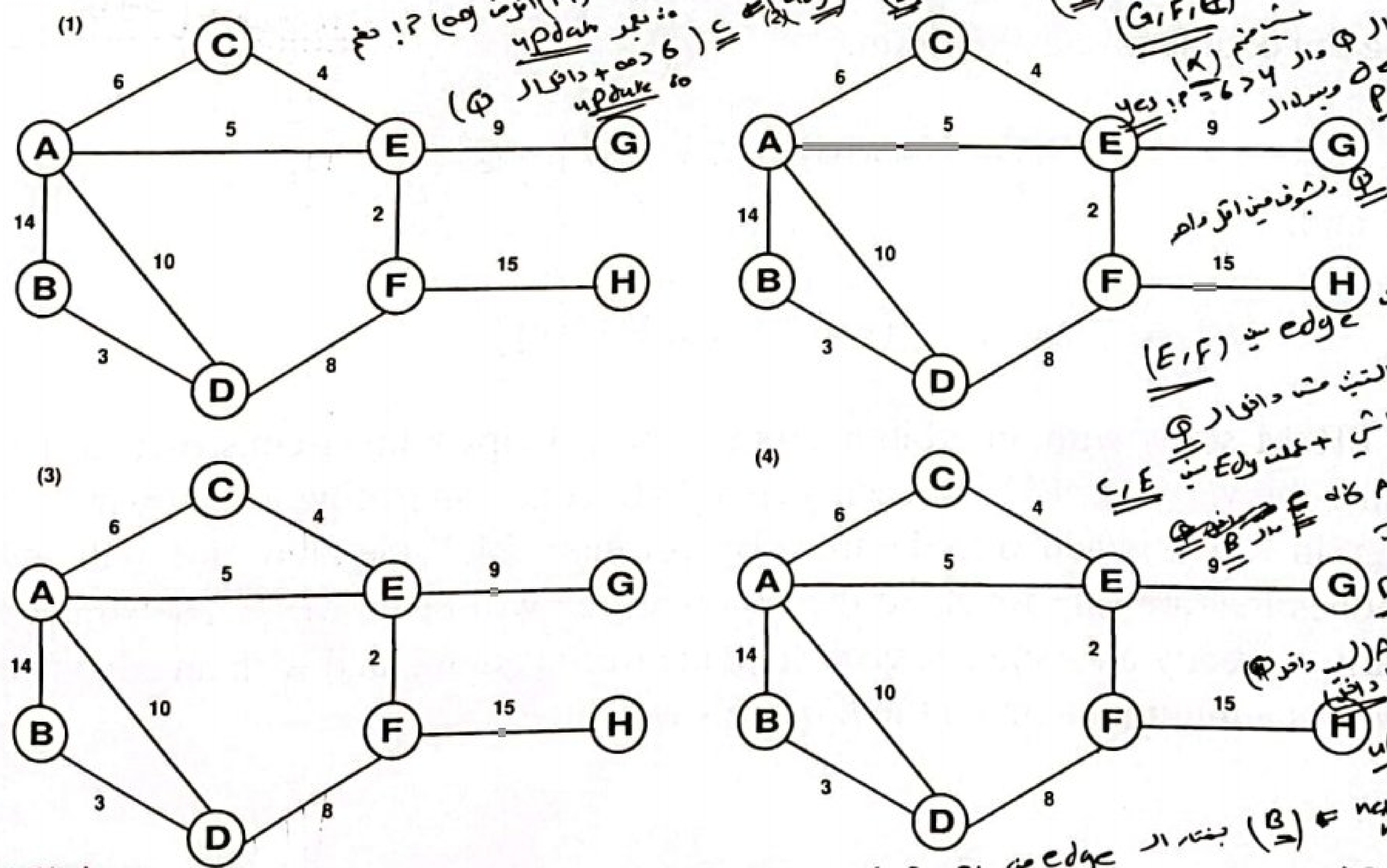
ال (F) parent نحرك

ال edge بين D-F

ال D ال Adj هي A (ليس داخل Q) و B داخل

14 > 3 ∴ update

MST-PRIM Example



④ Extract ال next min ⇐ (B) بنختار ال edge من (B - D) ومنين ال adj ال B صوار A مش داخل Q مابعمل اشي

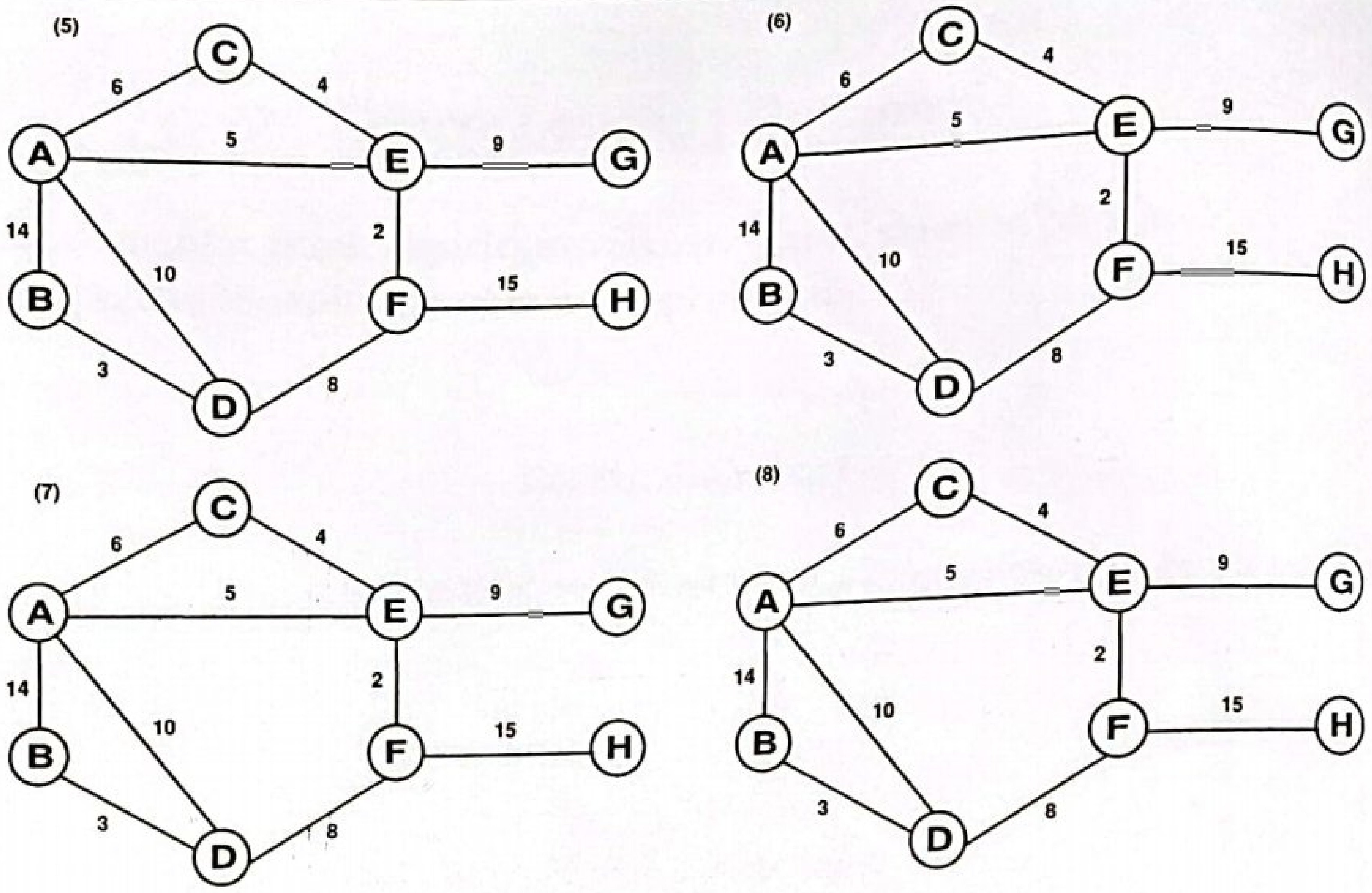
ال next min (G) ⇐ ال Adj E ماحاج اعمل شي + بعمل edge (E - G)

ال next min H ⇐ ال Adj F = (مش داخل Q) + بعمل edge بين (F - H) ومابعمل اشي بطلع

ال weight بطلع استخدمت لتعقب لحساب ال Cost (MST)

6+3+4+8+5+2+9+15 = 46

MST-PRIM **Example** *continued*



MST-PRIM **Complexity**

The running time of MST-PRIM depends on the implementation of the priority queue selected.

١ مم **Heap:** DECREASE-KEY runs in $O(\lg n)$ time.

- Initialization: $O(V)$
- EXTRACT-MIN Operations: $O(V \lg V)$
- DECREASE-KEY Operations: $O(E \lg V)$
- **Total:** $O((V+E)\lg V)$

(مم) **Fibonacci Heap (see Chapter 21):** DECREASE-KEY runs in $O(1)$ time.

- Initialization: $O(V)$
- EXTRACT-MIN Operations: $O(V \lg V)$
- DECREASE-KEY Operations: $O(E)$
- **Total:** $O(E + V \lg V)$

Lecture 14: Single-Source Shortest Path Algorithms

Course Learning Outcome

- Use fundamental graph algorithms, like traversal, shortest path and spanning tree in the solution of real-life problems

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

Read Chapter 25 of *Introduction to Algorithms*

The Shortest-Paths Problem

1

A **shortest-paths problem**, given a weighted, directed graph $G = (V, E)$ with 2 weight function $w : E \to \mathbf{R}$, is to find the shortest path between two arbitrary vertices, u and v.

Examples:

Determine the shortest route from Aqaba to Irbid.

Determine the shortest distance between two intersections from a street map.

This problem is a generalization of breadth-first search to handle weighted graphs. In BFS, the weight of each edge is 1.

Edge weights can be interpreted, instead of as distances, as time, cost, penalties, etc.

Shortest-Path Terminology

The **weight** of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of the member edges:

$\downarrow$ ↳ To source ↳ dist.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

لحساب ال weight
لل path's بجمعه
(لطلع ال path بياناته الجمع weight)
[illegible]

The **shortest-path weight** from u to v is defined as:

$$\delta(u,v) = \begin{cases} \boxed{\min}\{w(p) : \overset{\text{source}}{u} \overset{p}{\rightsquigarrow} \overset{\text{dist.}}{v}\} & \text{if there is a path from } u \text{ to } v \\ \infty \text{ (open cct) (unreachable)} & \text{otherwise. (اذا ما في path)} \end{cases}$$

↳ shortest (path)
d(u,v) (لاختصار الرمز d)
اذا مش shortest

A **shortest path** from u to v is any path p with $\boxed{w(p) = \delta(u,v).}$

The Single-Source Shortest-Paths Problem and Variations

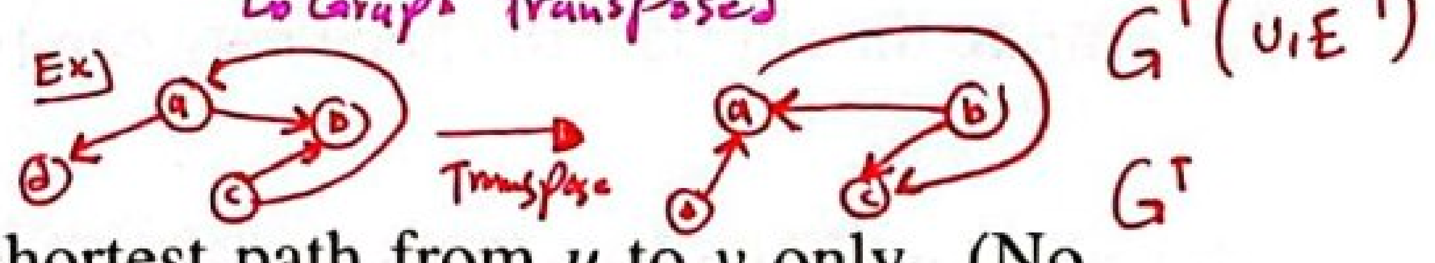
This lecture focuses on the **single-source shortest-paths problem**: given a graph $G = (V,E)$, find the shortest path from a given **source vertex** $s \in V$ to every other vertex, $v \in V$. There are many variations that can be solved with the algorithm for this problem:

s → d
→ (روت) بعكس ال ← s ← d

Single-destination shortest-paths problem: Find the shortest path to a given **destination** vertex t from any vertex v. (Compute G^T and use the single-source shortest path algorithm with t as source.)

↳ To Graph Transposed

Ex)



Single-pair shortest-path problem: Find a shortest path from u to v only. (No algorithms for this problem are known that run asymptotically faster than the best single-source algorithm in the worst case.)

من كل node اطلع ال shortest → distance لكل ال other vertices graph ⇒ بنمشي مع next vertice ⇒ [illegible]

All-pairs shortest-paths problems: Find the shortest path from u to v for every pair of vertices u and v. (Can use the single-source algorithm for each vertex, but it is best handled as an all-pairs problem.)

A Single-Source Shortest-Paths Example

(Directed weighted)

(uniqe sonrce)

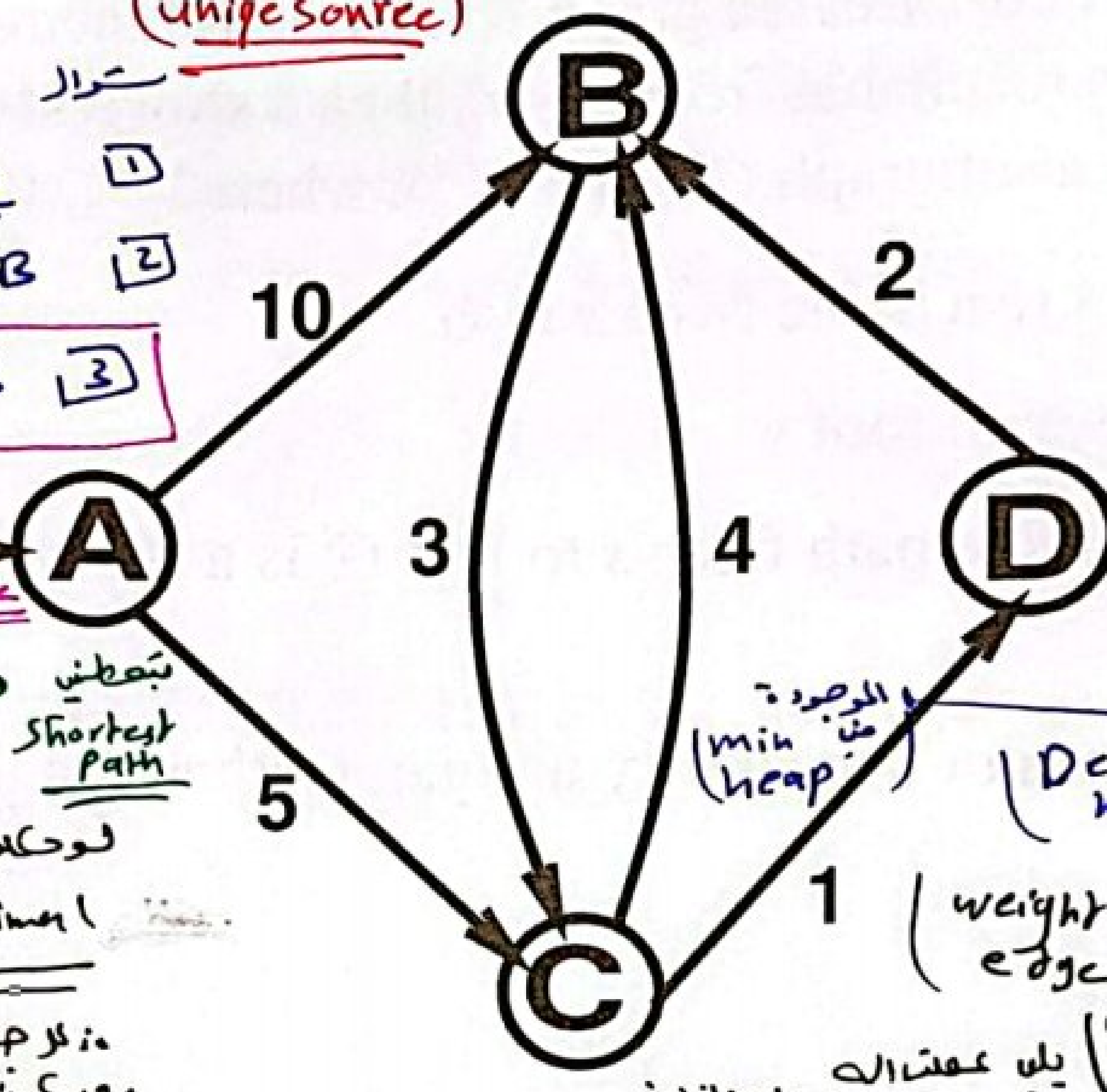
سؤال Path من A لـ B ؟!

① A → B (Direct) = 10 (weight)

② A → C → B = 5 + 4 = 9

③ A → C → D → B = 5 + 1 + 2 = 8 *

بدي اختار الـ min الـ path



Source

(predecessor Tree) → بتعطيني Shortest Path

optimal substructer

A → optimal
C → optimal
D → optimal
B

لو حكينا انه optimal

اذا كل جزء منه رح يكون optimal

(min heap) من الموجودة

مبدأ الـ Greedy choice : ← Relaxation (Decrease Key)

رح اجمع الـ (weight edge + distance Source)

وانا كل ما اشوف اقل من الـ (Current weight) يلي عطيتها initilization

في البداية بتكون (∞) اذا اقل بعمل (Decrease Key)

ملاحظة: عادي الـ problem بتنحل بـ Greedy Algo.

① optimization problem
② optimal substructure
③ Greedy choice

① Dijkstra's Algo. ⇒ it has very similar implementation to Both Prims MST and BFs Alg.

لكن بدل Key في Table نستخدم (distance)

② Bellman Ford Algo.

A Representation for the Shortest Paths

We typically want to compute, not only the weight of each shortest path, but also the vertices in each shortest path.

As in the BFS algorithm, the shortest-paths algorithms will, for a given graph $G = (V, E)$, maintain for each vertex $v \in V$, a **predecessor,** $\pi[v]$ (has a value of some $u \in V$ or NIL), so that PRINT-PATH(G, s, v) can be used to print the shortest path from s to v.

The shortest-paths algorithm will use a predecessor subgraph, $G_\pi = (V_\pi, E_\pi)$, induced by the π values, such that:

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

We shall prove that the π values produced by the shortest-paths algorithms when they terminate have the property that G_π is a **shortest-paths tree**, a tree rooted at s containing a shortest path from s to each vertex $v \in V$ reachable from s.

Shortest-Paths Tree

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$ with no negative-weight cycles reachable from $s \in V$, then a **shortest-paths tree** rooted at s is defined formally as a subgraph $G' = (V', E')$, where $V' \subseteq V, E' \subseteq E$ such that:

1. V' is the set of vertices reachable from s in G.
2. G' forms a rooted tree with root s.
3. $\forall v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Because shortest paths are not necessarily unique, neither is a shortest-paths tree always unique.

Optimal Substructure of a Shortest Path

The shortest-paths algorithms exploit the property that subpaths of a shortest path are shortest paths.

Lemma 25.1. (Subpaths of shortest paths are shortest paths)
The proof is provided in the book

Shortest-Paths Algorithms: Initialization

The single-source shortest-paths algorithms operate on a weighted, directed graph $G = (V, E)$ and use two arrays, π and d, to calculate the shortest paths from s to each $v \in V$. When the algorithms terminate, for $v \in V$, $\pi[v]$ is the predecessor of v in the shortest path from s to v and $d[v]$ is the weight of that path.

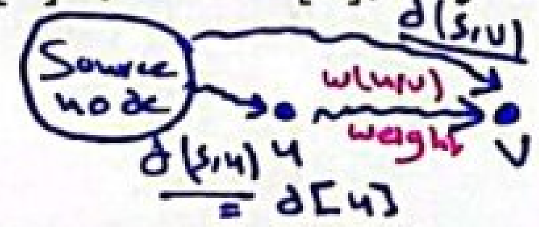
The following routine is used by the single-source shortest-paths algorithms to initialize the arrays:

```
INITIALIZE-SINGLE-SOURCE($G, s$)
for each vertex $v \in V[G]$
    do $d[v] \leftarrow \infty$
       $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$
```

Shortest-Paths Algorithms: Relaxation

The shortest-paths algorithms use a technique called **relaxation**. The basic idea is that $d[v]$ is an upper bound on the weight of a shortest path from source s to v, and as such is called a **shortest-path estimate**, which is reduced from ∞ until it finally reaches the shortest-path weight value $\delta(s, u)$.

RELAX(u, v, w) updates $d[v]$ (and $\pi[v]$) by examining the impact of the weight of edge (u, v).



upper Bound on the distance from s to v, so it is not a Tight Bound δ(v) ← (this is what we disired Tight bound)

```
RELAX($u, v, w$)
if $d[v] > d[u] + w(u, v)$
    then $d[v] \leftarrow d[u] + w(u, v)$   (Relaxation, updat dist.)
         $\pi[v] \leftarrow u$   (updat parent)
```

من ال Source directly من s لل v

اذا أقل يعني d[v] > d[u]+w[u,v]

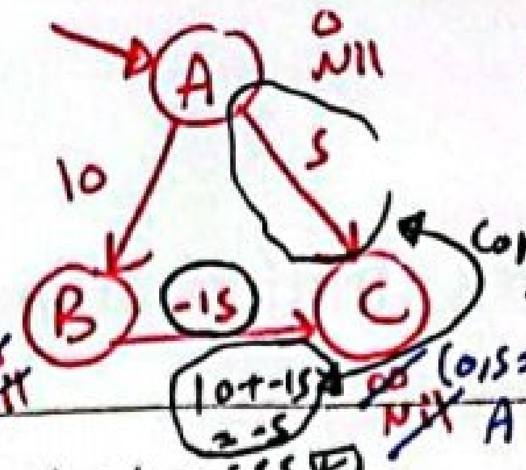
ساعتها بعد update distance ال + update parent ال

يتزيد بناقص اسن

Relaxation is the only means by which shortest-path estimates and predecessors change once they are initialized in all of the single-source shortest-path algorithms.

Ex) DijKstra Algo

وجود (negative weight)



U = {A, B, C}

S = {A, C, B}

لعين من A لل B (shortest path)

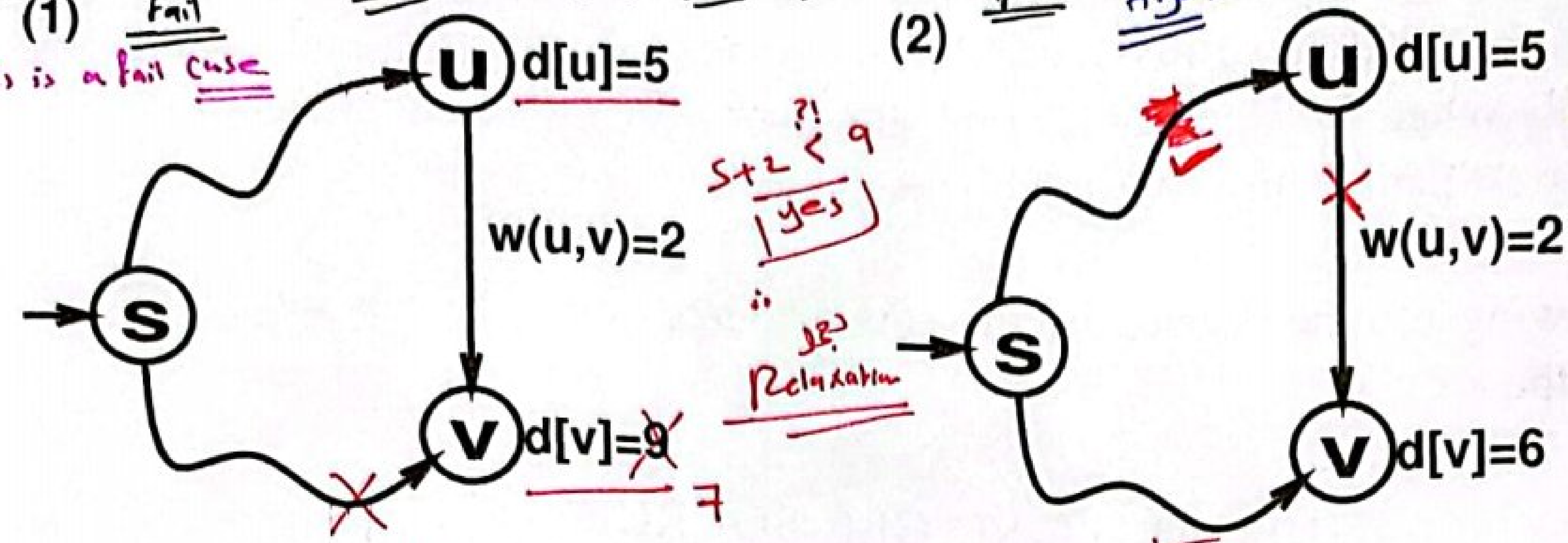
Contradiction اي دامن C وجود لعين Path (A→C) is the min
source

Shortest-Paths Algorithms: Relaxation

(Contradiction) Alg. Fail بحليل ال path الداخل path تاني افوض path معناه لعينة ← -5<5 ☑ update

↳ this is a fail case

Algo. يعني اى update كانت هيا مكملة ع حلو ح بيكون في مشكلة Extract الموا من و خفتها لا اى لعين لعينة (Shortest Path)



5+2 < 6 ?!

No

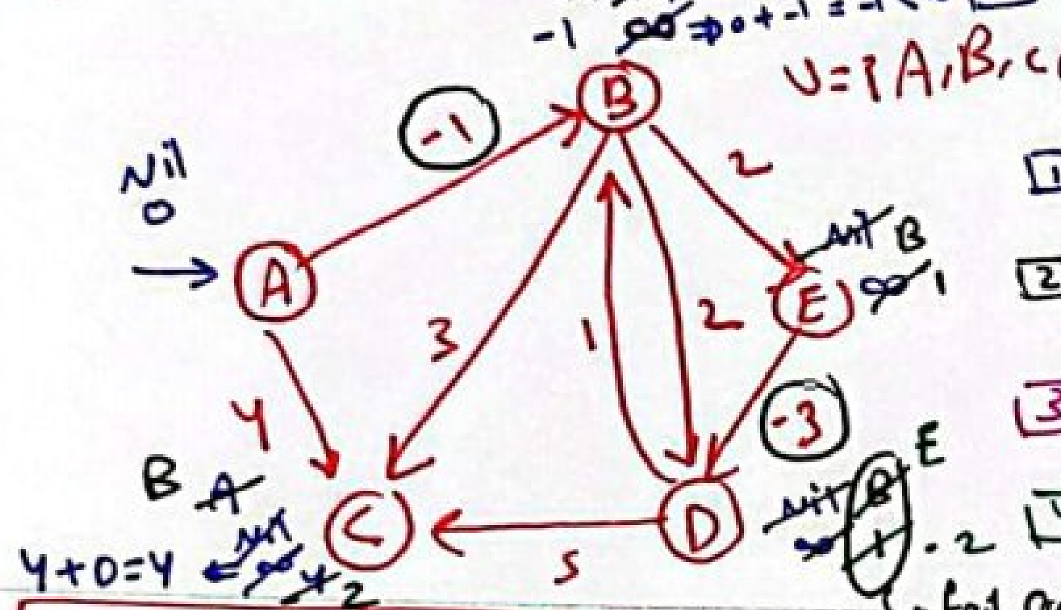
⇓

No Relaxation

بيحصل update ال path + بعد ال parent

ال weight (u,v) يعتمد بين ال pair من ده لا افضل اشي و من ده لا افضل اشي

Ex) DijKstra Algo.



U = {A, B, C, D, E}

1. Ext. min A ⇒ min B → Adj: C ⇒ -1+3 = 2 < 4 ☑; D ⇒ -1+2 = 1 ☑; E ⇒ -1+2 = 1 ☑
2. Ext. B ⇒ min D (α order) → B ⇒ 1+-1 = 0 < -1 ☒; C ⇒ 1+5 < 2 ☒
3. Extract D ⇒ min E → Adj D ⇒ 1+-3 = -2 < 1 ☑
4. Extract E ⇒ min C ⇒ Adj مالة ما نبحر اي اشي

↳ fail case

وجود ال negative weight edges في هذا المثال ⇓ (a problem this example) (Fail Case)

node	A	B	C	D	E
d[ ]	0	~~-1~~	~~2~~	~~1~~	~~1~~
π[ ]	Nil	~~A~~	~~B~~	~~B~~	~~B~~

(Fail Case) ← Final X Table

S = {A, B, D, E} (Fail Case)

Properties of Relaxation

(خالصفات الـ node اي لعين لعينة الـ shortest path)

Lemma 25.4. Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, and let $(u, v) \in E$. Then, immediately after relaxing (u, v) using RELAX(u, v, w), $d[v] \leq d[u] + w(u, v)$.

Proof:

If $d[v] > d[u] + w(u, v)$ just prior to the call RELAX(u, v, w), then $d[v] = d[u] + w(u, v)$ afterward. On the other hand, if $d[v] \leq d[u] + w(u, v)$ just prior to the call, then nothing changes.

□

Lemma 25.5. Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, let s be the source vertex, and let the graph be initialized using INITIALIZE-SINGLE-SOURCE(G, s), then $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on E. Once $d[v]$ achieves its lower bound $\delta(s, v)$, it never changes.

Dijkstra's Algorithm

***

- Dijkstra's Algorithm uses a greedy strategy together with the assumption of no negative edge weights to determine the shortest paths from a source vertex s in a weighted, directed graph $G = (V, E)$, represented using adjacency lists, to all $v \in V$.

من S → all v في graph

- The algorithm creates a set S of vertices whose shortest-path weights from s have been determined.

ال vertices لقيت لها ال Shortest path من ال Source إلهم

ال shortest path اذا مش موجودين ب (S) بكون بدي احسب ال shortest path من (S) إلهم

- The algorithm repeatedly selects the minimum vertex from a priority queue Q, containing vertices in $V - S$ keyed by their d values.

- It is like BFS except that it uses a priority queue, keyed on d. It is also similar to MST-PRIM.

V → S ⇒ initially empty except for the Source node

V → V-S ⇒ initially inserted in a priority queue (min heap)

DIJKSTRA(G, w, s)

```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)        distance = ∞
                                     parent = Nil
S ← ∅  ( initially empty )
Q ← V[G]  ( كلها vertices في queue )
while Q ≠ ∅  ( not empty )
   do u ← EXTRACT-MIN(Q)             أول node ← (Source) أجزال بتطلع لها قيمة (S)
      S ← S ∪ {u}                    حطه في (S)
      for each vertex v ∈ Adj[u]
         RELAX(u,v,w)
```

d[S] = 0 + π[V] = Nil + d[V] = ∞

|V|

V lg V

V lg V +

(binary) lg V

(Fibonacci) (1)

= V lg V + E lg V

OR

V lg V + E

E

V

V

= V² (normal array)

كل Adj بعمله Relax

بستمر في العملية لحد ما تصير Q (empty)

Time Complexity = { V lg V + E lg V ; V lg V + E }

DIJKSTRA Example

Ex

node	A	B	C	D
δ[ ]	~~∞~~ 0	~~∞~~ ~~10~~ ~~9~~ 8	~~∞~~ 5	~~∞~~ 6
π[ ]	Nil	~~Nil~~ ~~A~~ ~~C~~ D	~~Nil~~ A	~~Nil~~ C

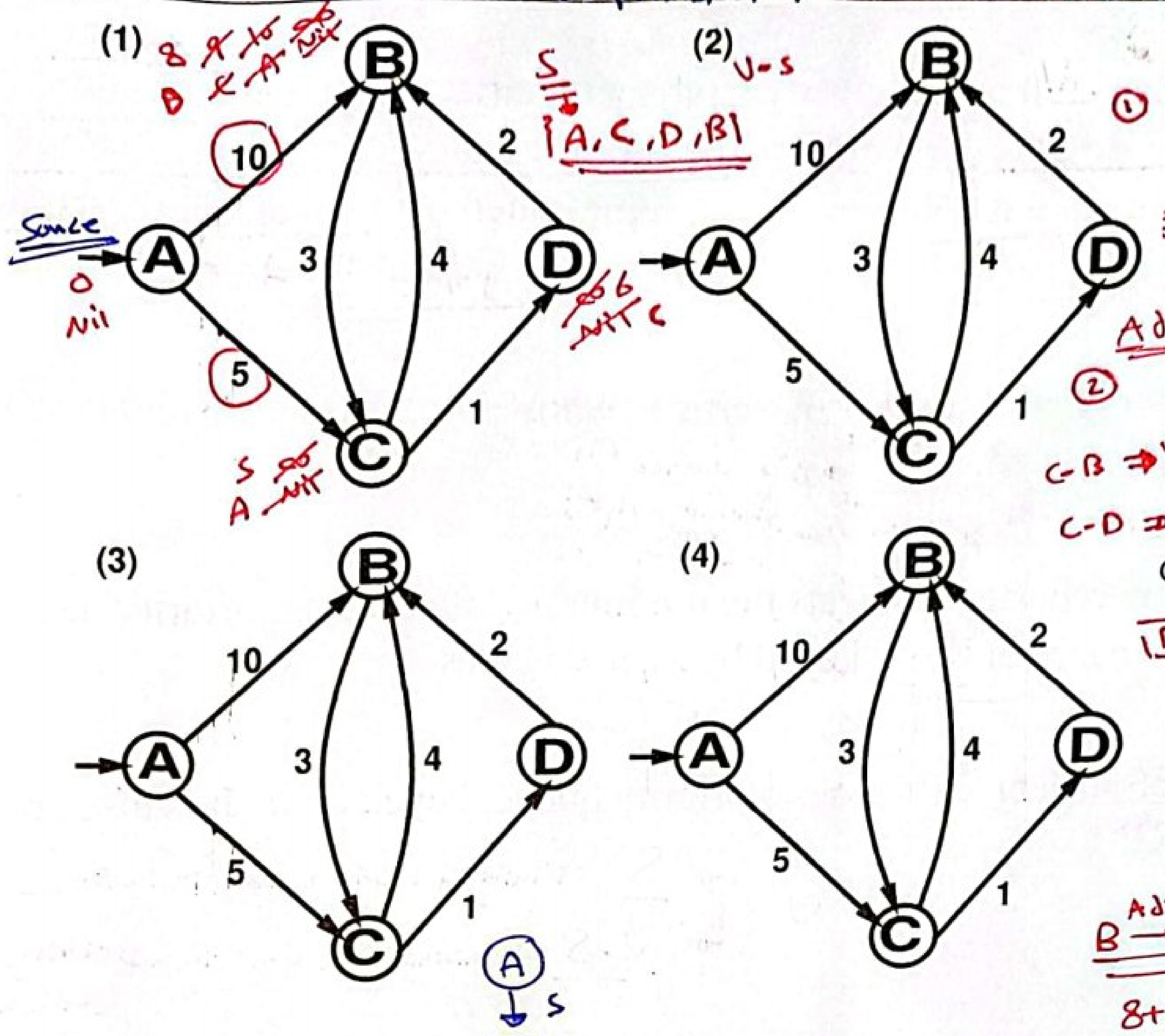
V = {A, B, C, D}

① initialization

② Source (δ = 0)

To vertices (all are initilly inserted into a priority Queue (min heap))

Extract min (A) ⇒ Adj A → B, C (اعمل Relax)



① A - B ⇒ 0+10 = 10 < ∞ , yes ∴ Relax + update parent

A - C ⇒ 0+5 = 5 < ∞ , yes ∴ Relax

⇒ Extract min (بس اخلص ال Adj)

Q not empty

C → بطل موجود مني ال Queue + ال Adj

② C → B, D — order is important

C-B ⇒ 4+5 < 10 , yes ∴ Relax

C-D ⇒ 1+5 < ∞ , yes ∴ Relax

Q not empty ∴ Extract min

D → Extract, بطل موجود مني Q

D → B (Adj)

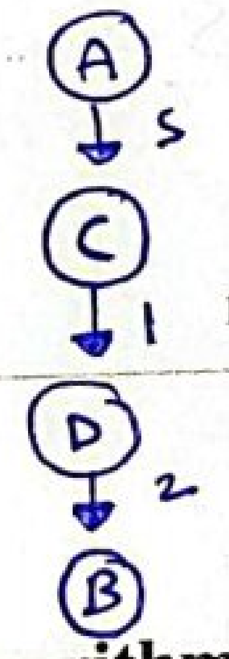
6+2 < 9 , yes ∴ Relax

Q not empty ∴ Extract min

B → C (Adj)

8+3 < 5 , No (مابعمل اي اشي)

Q is empty ∴ Finish Algo



(برسم ال Tree معتمدين لل Parent)

order S ⇒ {A, C, D, B}

V - S ⇒ ∅

Running Time of Dijkstra's Algorithm

EXTRACT-MIN: $|V|$ times.

DECREASE-KEY: $|E|$ times.

Hence, the worst case running time of DIJKSTRA can be characterized by the equation $|V| T_{\text{EXTRACT-MIN}} + |E| T_{\text{DECREASE-KEY}}$.

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
Array	$O(V)$	$O(1)$	$O(V^2)$
Binary Heap	$O(\lg V)$	$O(\lg V)$	$O((E+V)\lg V)$
Fibonacci Heap	$O(\lg V)$	$O(1)$	$O(V \lg V + E)$

(افضل شي)

The Bellman-Ford Algorithm

** BELLMAN-FORD(G, w, s) is a single-source shortest-paths algorithm that supports negative edge weights. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbf{R}$, (it returns a solution if and only if there is no negative-weight cycle reachable from s (a boolean value of FALSE indicates that no solution exists because there is a negative-weight cycle).

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)            ⇒ V      initialization نفس الـ
for i ← 1 to |V[G]| − 1                   ⇒ V      iteration    V-1 relaxation
    do for each edge (u, v) ∈ E[G]                  VE
        do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]                        If we were able to perform one more relaxation
    do if d[v] > d[u] + w(u, v)
        then return FALSE                          → نزل negative weight cycle → must return False
return TRUE
```

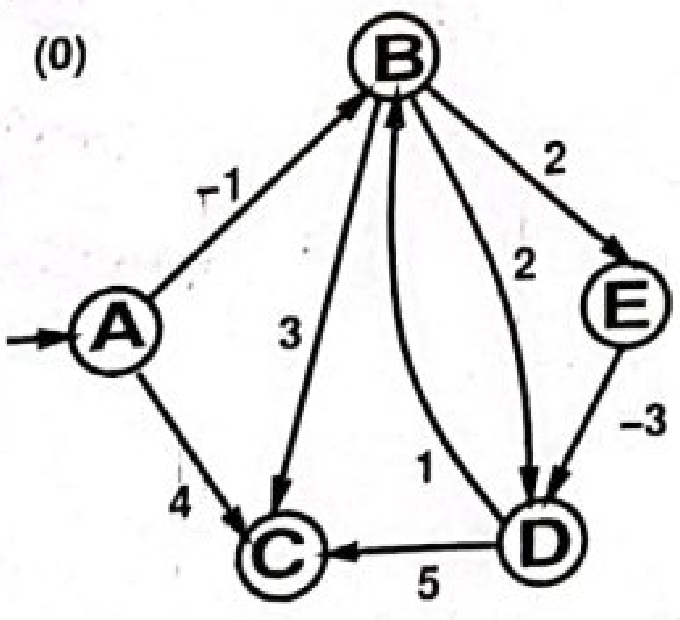
بزعيل ح يرجع ~~False~~ (True)

Time Complexity = O(VE)

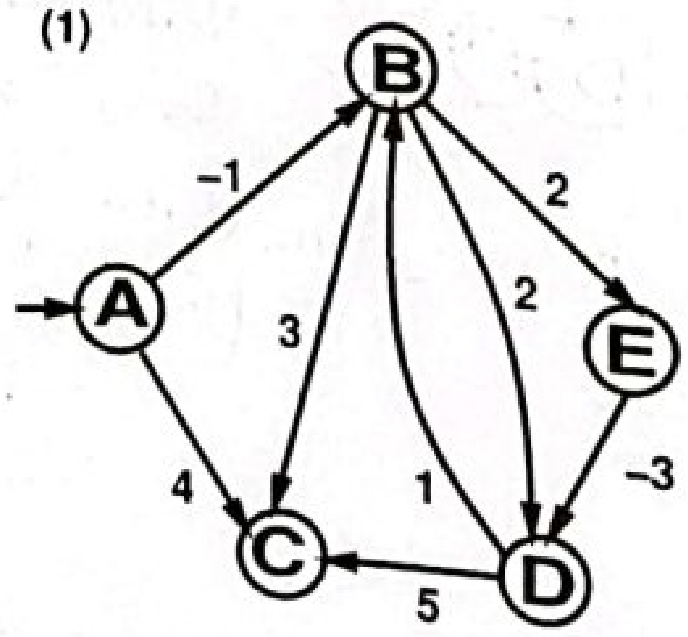
BELLMAN-FORD Example

Consider the edges in the following order: (A,B), (A,C), (B,C), (B,D), (D,B), (D,C), (E,D), (B,E).

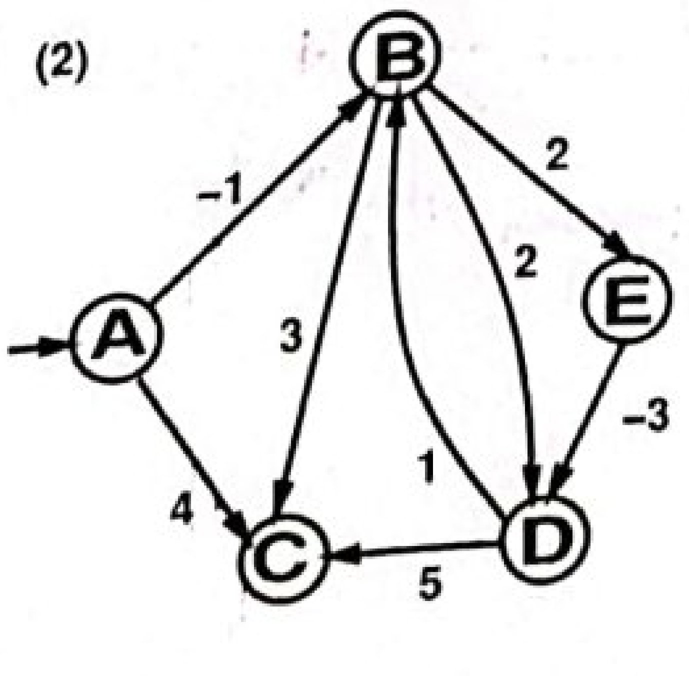
(0)



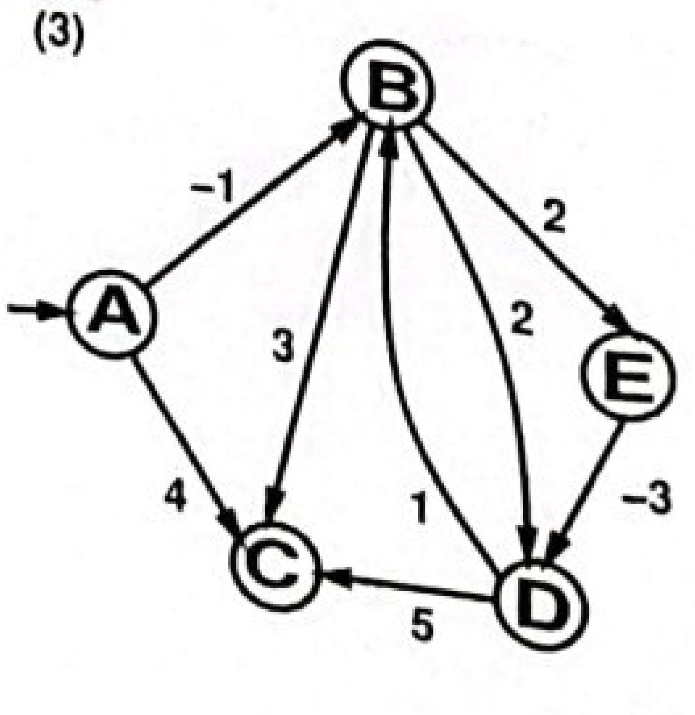
(1)



(2)



(3)



The Running Time of BELLMAN-FORD

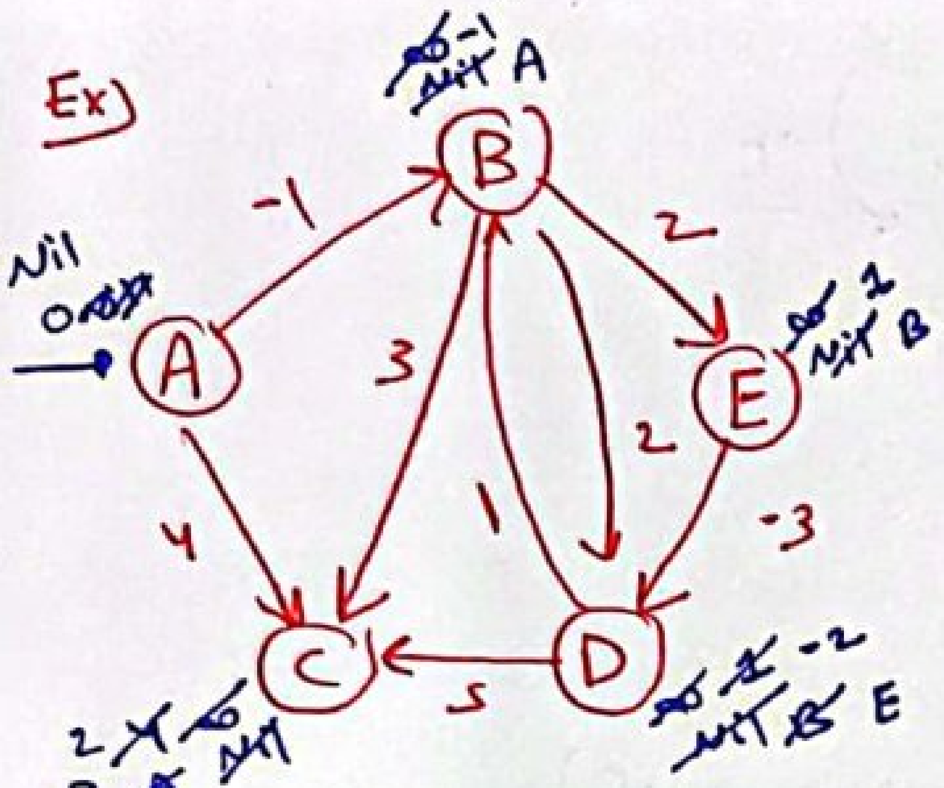
INITIALIZE-SINGLE-SOURCE takes $\Theta(V)$ time.

In lines 2-4, there are $|V|-1$ passes over E edges, which takes $O(VE)$ time.

Lines 5-7 takes $O(E)$ time.

Hence, the running time of BELLMAN-FORD is $\boxed{O(VE)}$.

Ex)



$U = \{A, B, C, D, E\}$

$|U| = 5$ (size) ∴ For loop = 4 ← $U-1 = 5-1 = 4$

order	i=1	i=2	i=3	i=4] relax
(A,B)	✓	×	will be the same	will be the same
(A,C)	✓	×		
(B,C)	✓	×		
(B,D)	✓	×		
(B,E)	✓	×		
(D,B)	× \ no update	×		
(D,C)	×	×		
(E,D)	✓	×		

Done First iteration ☑

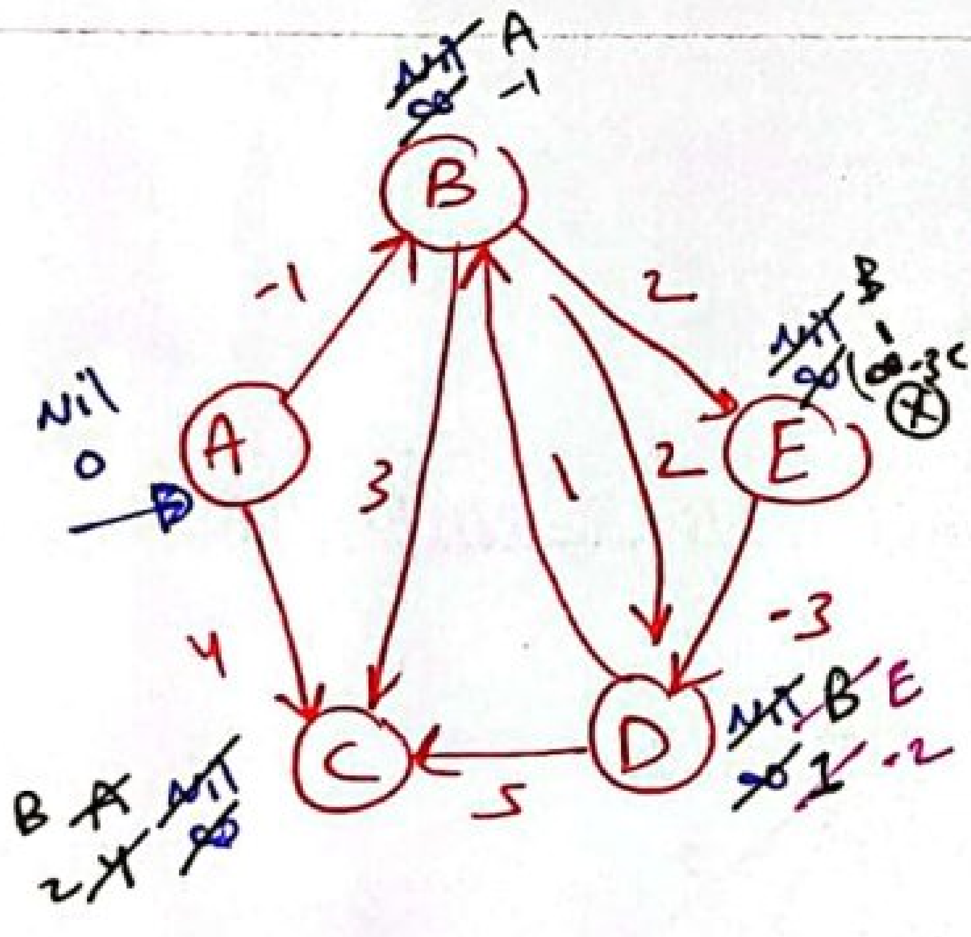
✓ : update (relax)
× : no update (no relax)

مافي update
∴ لو جربنا ال i=3, i=4
نفس الجواب رح يطلع

⇒ ∴ Alg. return True ⇒ ∴ Final Table

node	A	B	C	D	E
d[ ]	0	-1	2	-2	1
π[ ]	Nil	A	B	E	B

Result is True ⇐



order	i=1	i=2	i=3
(A,B)	✓	×	will be the same
(A,C)	✓	×	
(B,C)	✓	×	
(B,D)	✓	×	
(D,B)	×	×	
(D,C)	×	×	
(E,D)	⊗	✓	
(B,E)	✓	✓	

بغض النظر عن ال order
ما رح يتأثر ال Final ANS
(بس رح يأثر على ال iteration
ال Relaxation في كل iteration)

Final ANS

Note * ال graph اذا فيه
negative weight edges ⇐ لازم نستخدم (Bellman Ford) ← عشان نكتشف ال (negative weight cycle)

اذا مافي إشي ← Bellman Ford Alg. / Dijkstra Alg.