

Algorithms

CPE 110408300

Dr. Khalil Ahmad Yousef

Office: E3039

Email: Khalil@hu.edu.jo

Computer Engineering Department

The Hashemite university

Introduction

Reading Assignment:

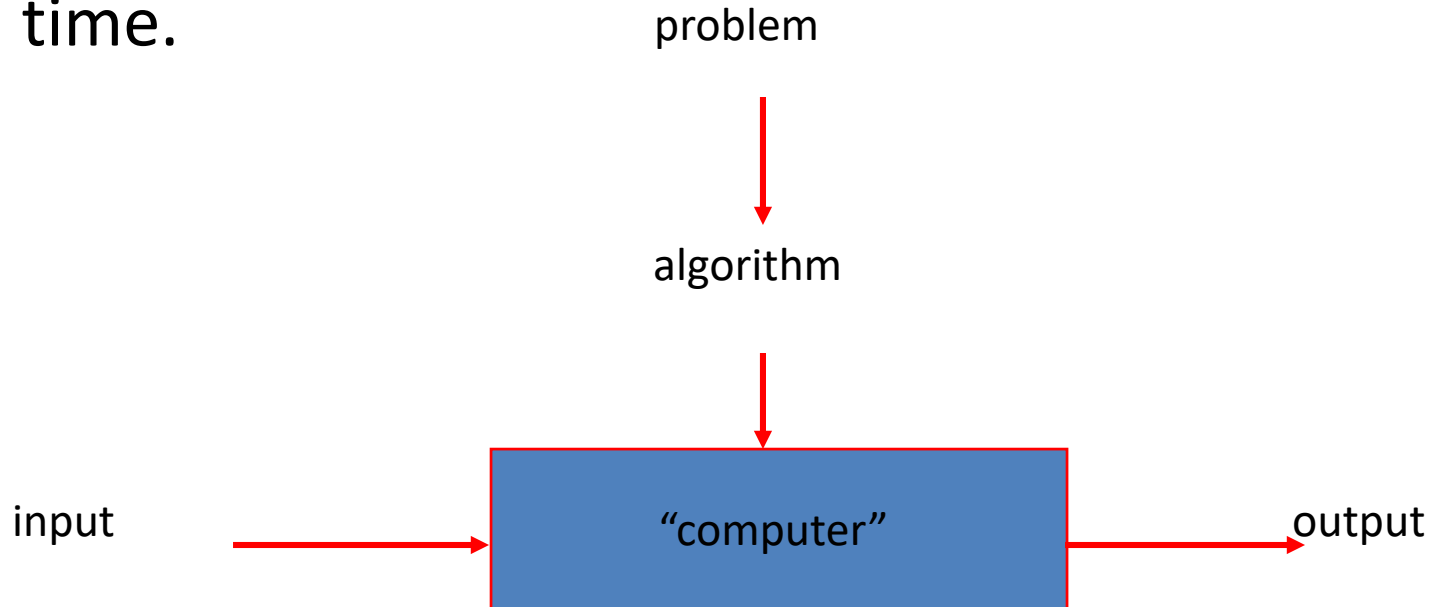
Read Chapters 1 and 2 of *the book*

Course Learning Outcomes

Show how time and space complexities can be traded-off.

What is an algorithm?

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



What is an algorithm?

- Other definitions
 - **A precisely defined sequence of (computational) steps** that transform a given input into a desired output.
 - A method for solving a problem(s)
 - Correctness of the method
 - Performance of the method
- Not all problems can be defined by an algorithm
 - Example: **Halting problem**
 - *Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever*
- A problem is said to be solvable
 - If a computer program can be written to produce the correct output for any input if we let it run long enough and allow it the required storage space

What is an algorithm?

- The way to write an algorithm

```
ALGORITHM NAME {  
    INPUT(s)  
    OUTPUT(s)  
    STEPS  
}
```

Title of the Course

- Design and Analysis of **Algorithms**
- Two main issues related to **algorithms**
 - How to **analyze** algorithm efficiency
 - How to **design** algorithms

Analysis of algorithms

- How good is the algorithm?
 - **Time** efficiency or complexity
 - **Space** efficiency or complexity
- Does there exist a better algorithm?
 - lower bounds
 - optimality

Analyzing Algorithms: WHY?

- To asymptotically compare the performance of multiple algorithms used to solve the same problem
 - Question: How to compare which one is better
 - What do we mean by the word “better”
 - Time Complexity
 - Space Complexity
 - Etc.
- **Focus:** Comparison based on Time Complexity
 - Assumption to compute the time complexity:
 - Running time (**Asymptotic Performance Analysis**)
 - Counting the basic operations as a function of the size of the input of the problem under study

Asymptotic Performance Analysis

- How does the algorithm behave as the problem size gets very large?
- Why?
 - Consider certain problem, which assumed to have several solutions expressed in terms of time complexity functions (n is size of the input):

—◆— $f(n) = n$

—■— $f(n) = \log(n)$

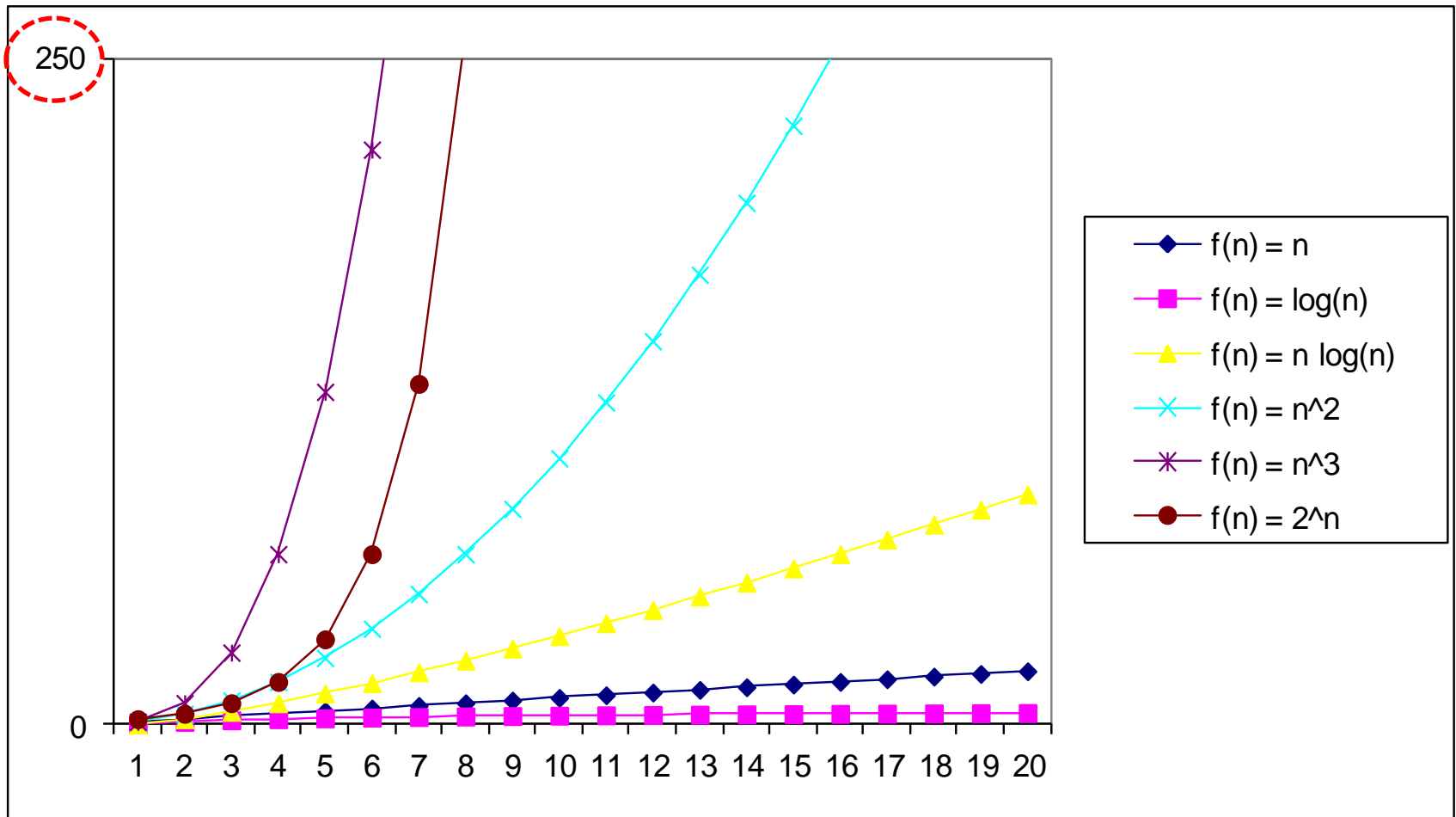
—▲— $f(n) = n \log(n)$

—×— $f(n) = n^2$

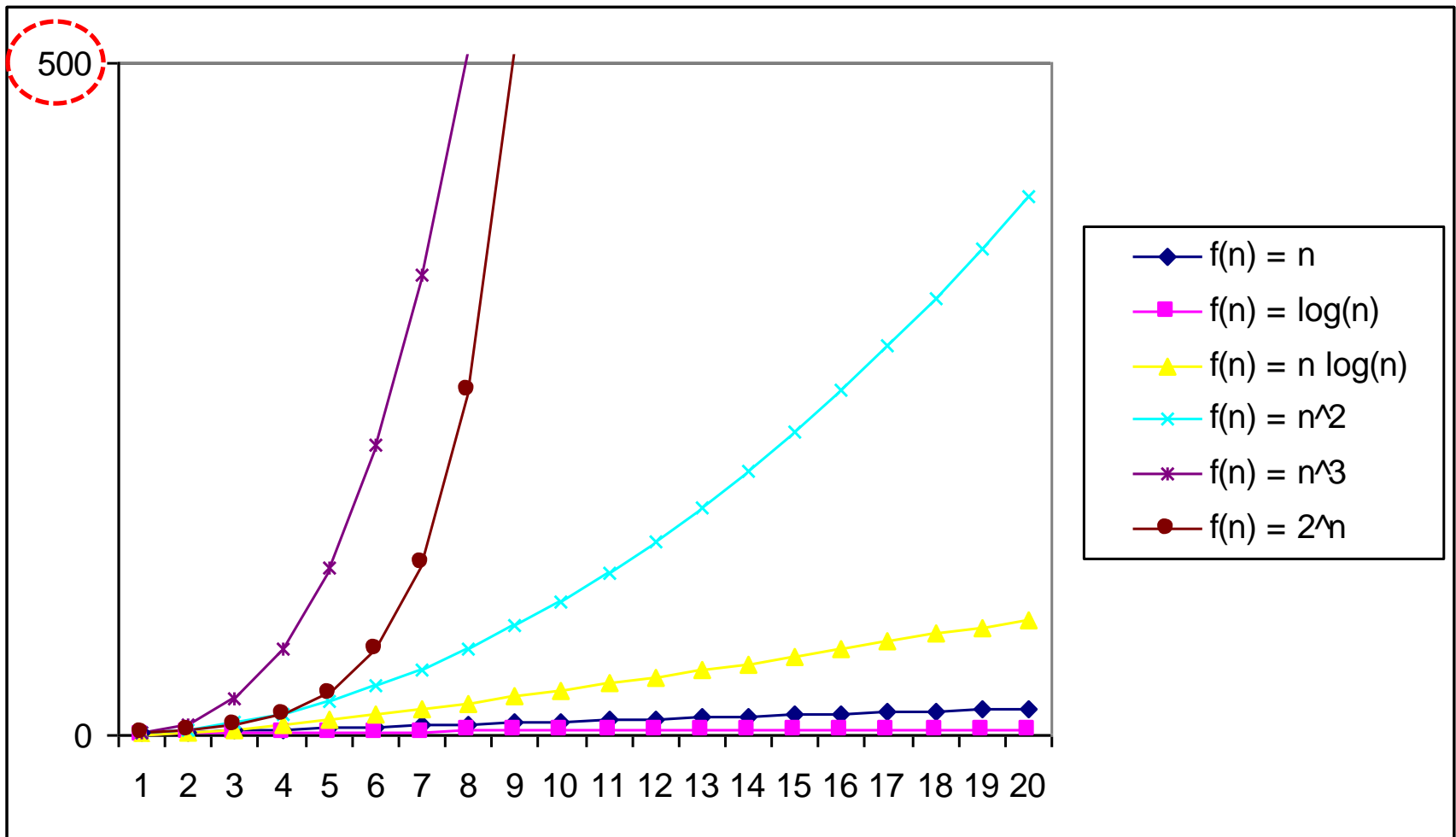
—✱— $f(n) = n^3$

—●— $f(n) = 2^n$

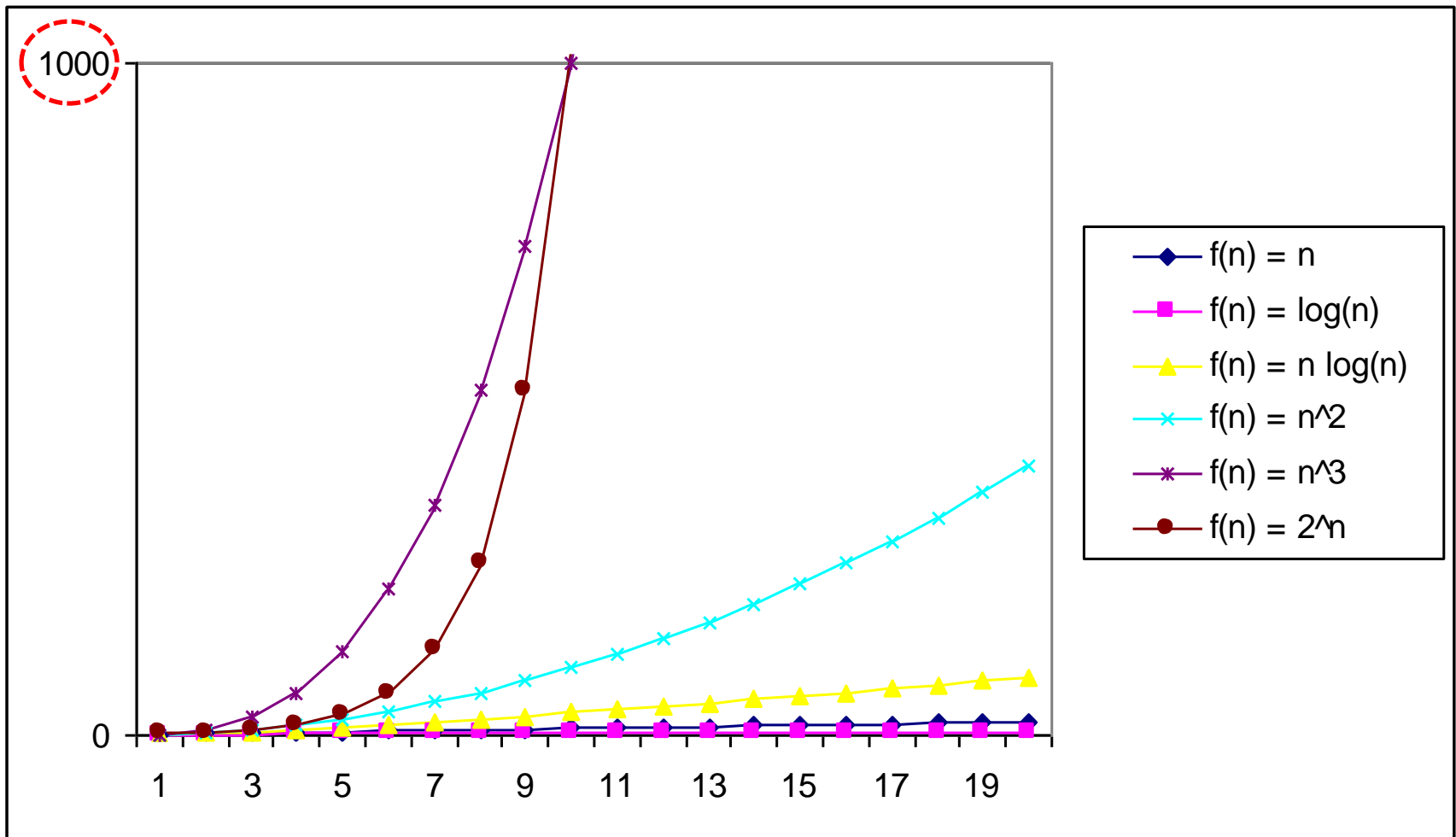
Practical Complexity



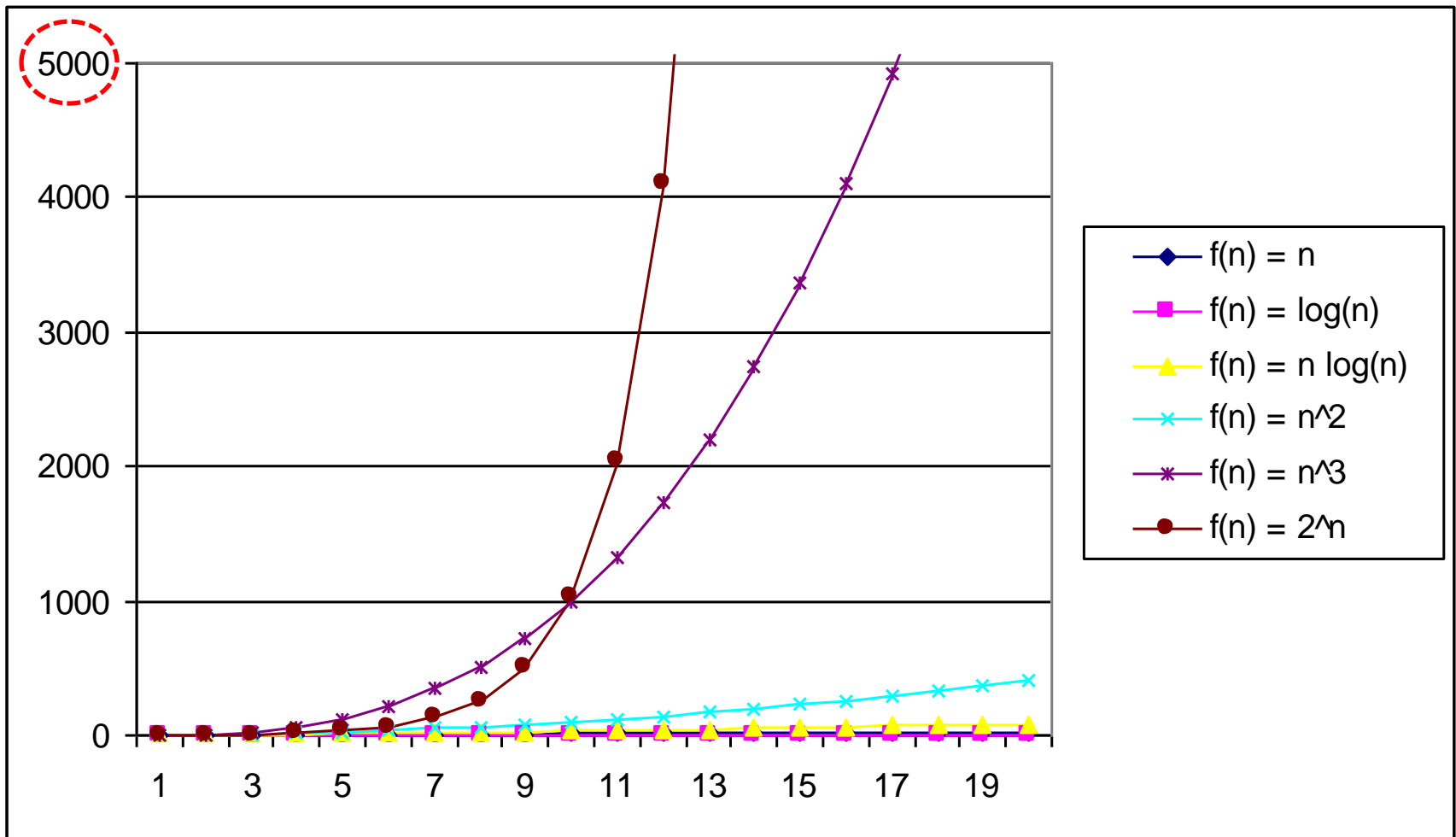
Practical Complexity



Practical Complexity



Practical Complexity

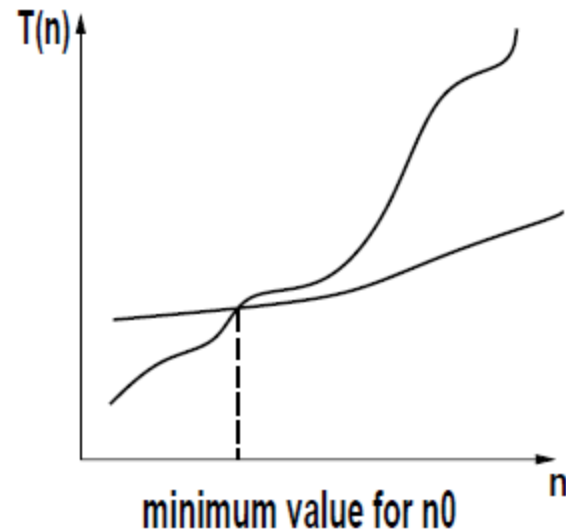


Summary: Analyzing Algorithms

- In this course, how we shall characterize or analyze an algorithm?
 - Use the resources in terms of:
 - Time Complexity
 - Space Complexity
- Goal:
 - Find the running time as a function of the input size in terms of the numbers of basic operations
 - Or finding **Order of Growth**
 - Or Performing **Asymptotic Analysis (Performance)**

Asymptotic Analysis

- How does the running time of the algorithm grow if the input gets very large?
 - Running time
 - Memory/storage requirements



- For large datasets the worst-case running time of INSERTION-SORT increases proportionally to n^2 . We use the notation $O(n^2)$ to express this.
 - Note that we ignore low-order terms and leading constants.
 - We will introduce O notation formally very soon.

Analyzing Algorithms

1. Correctness

2. Simplicity (Number of statements, Readability, the easiness of implementation)

3. Time Complexity

- Speed of the algorithm regardless of the hardware
- Includes compile + running time
- The amount of work to be done

4. Space Complexity

- Storage space
- The amount of space needed by the program to execute till completion
 - Fixed space: Static data (array)
 - Variable space: space allocated during running time

5. Optimality

Important **problem** types

- **Sorting**
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Fundamental data structures

- list
 - array
 - linked list
 - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary

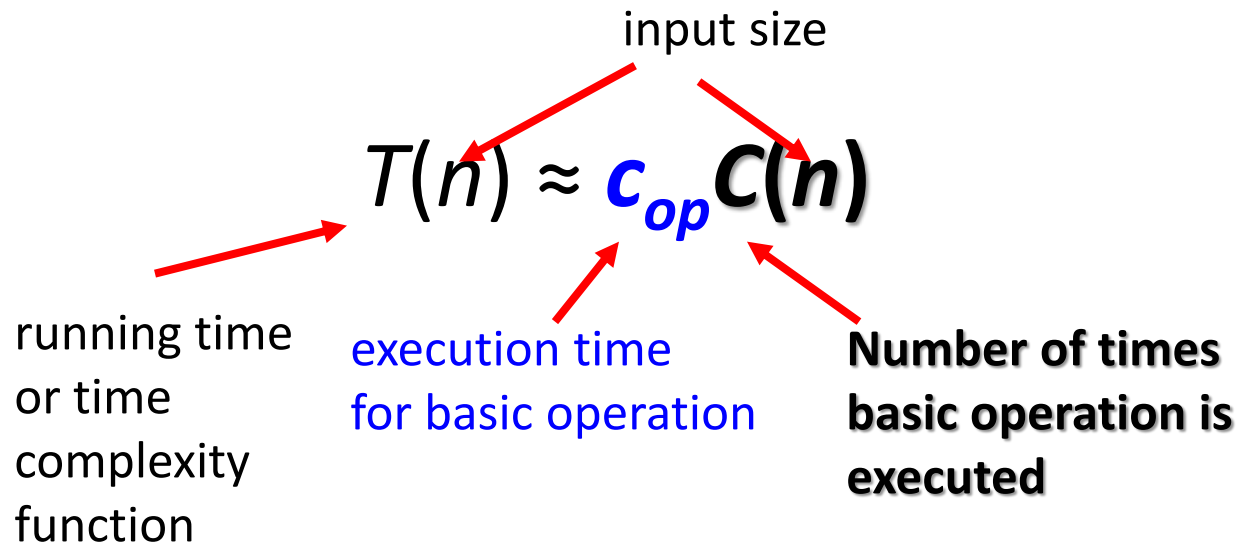
Analyzing Algorithms: Focus

- Time Complexity and Space Complexity
 - Approaches:
 - **Theoretical** analysis
 - **Empirical** analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the **basic operation** as a function of input size

- **Basic operation**: the operation that contributes most towards the running time of the algorithm



Input size and basic operation examples

<u>Problem</u>	<u>Input size measure</u>	<u>Basic operation</u>
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Input size and basic operation examples

<u><i>Problem</i></u>	<u><i>Input size measure</i></u>	<u><i>Basic operation</i></u>
Sorting a list of n items	Number of list's items, i.e. n	Comparison of two elements

Time Complexity: Running Time

- Number of primitive steps (Basic Operations) that are executed
 - Except for time of executing a function call most statements roughly require the same amount of time
 - $y = m * x + b$
 - $c = 5 / 9 * (t - 32)$
 - $z = f(x) + g(y)$
- We can be more exact if need be

Time Complexity

- Worst case:
 - Maximum number of basic operations performed by the algorithm on an input of a specific size.
- Best case:
 - Minimum number of basic operations performed by the algorithm on an input of a specific size.
- Note: For any input of size n , there are exactly $n-1$ basic operations
 - Summing 5 numbers requires 4 summing operations

Time Complexity

- Average case:
 - Average number of basic operations performed by the algorithm on an input of a specific size.
 - Number of times the basic operation will be executed on typical input
 - NOT the average of worst and best case
 - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Time Complexity: Summary

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- Worst case: $C_{\text{worst}}(n)$ – maximum over inputs of size n
- Best case: $C_{\text{best}}(n)$ – minimum over inputs of size n
- Average case: $C_{\text{avg}}(n)$ – “average” over inputs of size n

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

or

Count actual number of basic operation's executions

- Analyze the empirical data

Example 1: Sequential search

ALGORITHM *SequentialSearch*($A[0..n - 1]$, K)

//Searches for a given value in a given array by sequential search

//Input: An array $A[0..n - 1]$ and a search key K

//Output: The index of the first element of A that matches K

// or -1 if there are no matching elements

$i \leftarrow 0$

while $i < n$ **and** $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

- Worst case
- Best case
- Average case

Example 2: Maximum element

ALGORITHM *MaxElement*($A[0..n - 1]$)

//Determines the value of the largest element in a given array

//Input: An array $A[0..n - 1]$ of real numbers

//Output: The value of the largest element in A

$maxval \leftarrow A[0]$

for $i \leftarrow 1$ **to** $n - 1$ **do**

if $A[i] > maxval$

$maxval \leftarrow A[i]$

return $maxval$

- Worst case
- Best case
- Average case

Example 3: Element uniqueness problem

ALGORITHM *UniqueElements*($A[0..n - 1]$)

//Determines whether all the elements in a given array are distinct

//Input: An array $A[0..n - 1]$

//Output: Returns “true” if all the elements in A are distinct

// and “false” otherwise

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

if $A[i] = A[j]$ **return false**

return true

- Worst case
- Best case
- Average case

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*($A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$)

//Multiplies two n -by- n matrices by the definition-based algorithm

//Input: Two n -by- n matrices A and B

//Output: Matrix $C = AB$

for $i \leftarrow 0$ **to** $n - 1$ **do**

for $j \leftarrow 0$ **to** $n - 1$ **do**

$C[i, j] \leftarrow 0.0$

for $k \leftarrow 0$ **to** $n - 1$ **do**

$C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

return C

- Worst case
- Best case
- Average case

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

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list

Brute-Force Sorting Algorithm

Selection Sort Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second element. Generally, on pass i ($0 \leq i \leq n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:

$$\begin{array}{ccccccc} A[0] & \leq & . & . & . & \leq & A[i-1] & | & A[i], & . & . & . & , & A[\textit{min}], & . & . & . \\ A[n-1] & & & & & & \uparrow & & & & & & & \uparrow & & & \end{array}$$

in their final positions

Example: 7 3 2 5

Brute-Force String Matching

- pattern: a string of m characters to search for
- text: a (longer) string of n characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Examples of Brute-Force String Matching

Pattern: 001011

Text: 10010101101001100101111010

Pattern: happy

Text: It is never too late to have a
happy childhood.

Exhaustive Search

A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- generate a list of all potential solutions to the problem in a systematic manner (see algorithms in Sec. 5.4)
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

Brute-Force Strengths and Weaknesses

- Strengths
 - wide applicability
 - simplicity
 - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)
- Weaknesses
 - rarely yields efficient algorithms
 - some brute-force algorithms are unacceptably slow
 - not as constructive as some other design techniques
- Note: In many cases, exhaustive search or its variation is the only known way to get exact solution

Designing Algorithms: Techniques/Strategies

- We shall get introduced to the rest of the algorithm's design strategies through examples
 - Problem dependent.

Our first case study

SORTING PROBLEM

Sorting Problem

- **The Sorting Problem (non-decreasing):**
 - **Input:** a sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$
 - **Output:** a permutation of the input sequence $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that $a'_1 \leq a'_2 \dots \leq a'_n$
- Sorting Algorithms
 - Techniques
 - Simplicity
- Determination of Best Sorting Algorithm
 - Time Complexity: Run Time
 - Space Complexity: Space or memory

Growth Rate
- Depend on the size of the problem
 - Linear, polynomial, exponential

Some Sorting Algorithms

- Insertion Sort
- Merge Sort
- Heap sort
- Quick sort
- Linear-time sorts
 - Counting Sort
 - Radix Sort
 - Bucket Sort

Sorting Problem

- Care must be taken on larger and larger example but not on small ones
- Any algorithm which scales exponentially is often not useful or will be very destructive

The End