



# CPE 11040830: Algorithms, Homework #0 Solution

**Dr. Khalil Ahmad Yousef**

(1) CLR 1.2-2

INSERTION-SORT beats MERGE-SORT when  $8n^2 < 64n \lg n$ ,  $\Rightarrow n < 8 \lg n$ ,  $\Rightarrow \frac{n}{8} < \lg n$ ,  $\Rightarrow 2^{n/8} < n$ , which is true when  $2 \leq n \leq 43$ .

(You can solve for  $n$  by trial and error using a calculator. Observe that  $5 \cdot 8 < n < 6 \cdot 8 = 48$ , since  $2^5 = 32 < 40$  and  $2^6 = 64 > 48$ , then narrow it down with the calculator.)

(2) CLR 1.2-3

ALGORITHM 1 will be faster when  $100n^2 < 2^n$ ,  $\Rightarrow n \geq 15$ .

(You can solve for  $n$  by trial and error using a calculator.)

(3) CLR 1.1

	1 second	1 minute	1 hour	1 day
$\lg n$	$2^{10^6}$	$2^{6 \times 10^7}$	$2^{3.6 \times 10^9}$	$2^{8.64 \times 10^{10}}$
$\sqrt{n}$	$10^{12}$	$3.6 \times 10^{15}$	$1.269 \times 10^{19}$	$7.4649 \times 10^{21}$
$n$	$10^6$	$6 \times 10^7$	$3.6 \times 10^9$	$8.64 \times 10^{10}$
$n \lg n$	$6.2746 \times 10^4$	$2.801417 \times 10^6$	$1.33378058 \times 10^8$	$2.755147513 \times 10^9$
$n^2$	$10^3$	$7.745 \times 10^3$	$6 \times 10^4$	$2.93938 \times 10^5$
$n^3$	100	391	1532	4420
$2^n$	19	25	31	36
$n!$	9	11	12	13

	1 month	1 year	1 century
$\lg n$	$2^{2.59 \times 10^{12}}$	$2^{3.1536 \times 10^{13}}$	$2^{3.1556736 \times 10^{15}}$
$\sqrt{n}$	$6.7081 \times 10^{24}$	$9.94519296 \times 10^{26}$	$9.95827587 \times 10^{30}$
$n$	$2.59 \times 10^{12}$	$3.1536 \times 10^{13}$	$3.1556736 \times 10^{15}$
$n \lg n$	$7.1870856404 \times 10^{10}$	$7.97633893349 \times 10^{11}$	$6.8654697441062 \times 10^{13}$
$n^2$	$1.609347 \times 10^6$	$5.615692 \times 10^6$	$5.6175382 \times 10^7$
$n^3$	13733	31593	146677
$2^n$	41	44	51
$n!$	15	16	17



(4) CLR 2.1-1

The operation of INSERTION-SORT can be illustrated by Figure 1

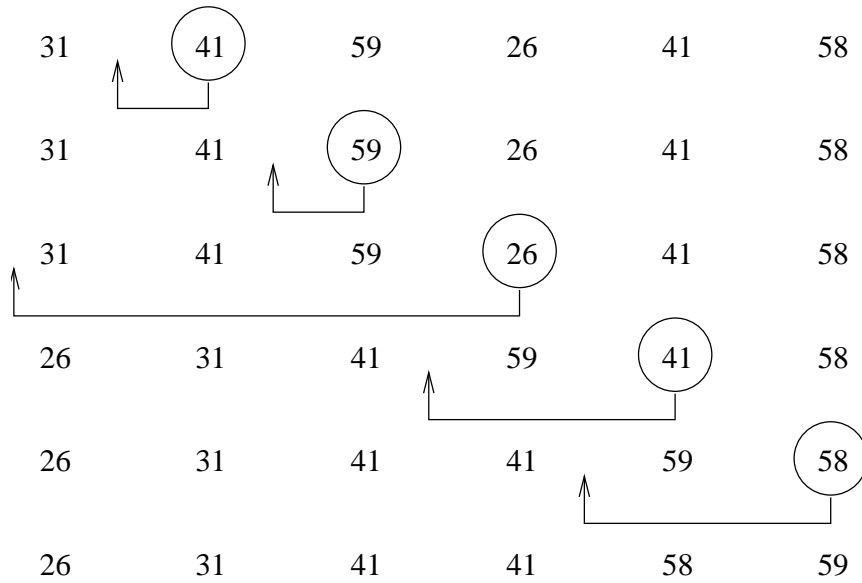


Figure 1: The operation of insertion sort for Problem CLR 2.1-1.

(5) CLR 2.1-2

NON-INCREASING-INSERTION-SORT(A)

1. **for**  $j \leftarrow 2$  to  $\text{length}[A]$
2.     **do**  $\text{key} \leftarrow A[j]$
3.          $\triangleright$  Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$
4.      $i \leftarrow j-1$
5.     **while**  $i > 0$  and  $A[i] < \text{key}$
6.         **do**  $A[i+1] \leftarrow A[i]$
7.          $i \leftarrow i-1$
8.      $A[i+1] \leftarrow \text{key}$

(6) CLR 2.1-3

The pseudocode is as follows:

LINEAR-SEARCH( $A, v$ )

1.  $j \leftarrow 1$
2. **while**  $j \leq \text{length}[A]$
3.     **if**  $A[j] = v$



4.           **then return**  $j$
5.           **else**  $j \leftarrow j + 1$
6. **return** NIL

We need to show the algorithm is correct by using a loop invariant in the algorithm. The loop invariant is as follows:

At the start of the  $j$ th loop iteration of the while loop in lines 2-5, the subarray  $A[1..j-1]$  consists of elements not equal to  $v$ .

Now we will show that the loop invariant satisfies the three properties mentioned in the book:

**Initialization:** When  $j = 1$ ,  $A[1..j-1]$  contains no elements; hence, there can be no elements in  $A[1..j-1]$  that are equal to  $v$ . Thus, the loop invariant holds.

**Maintenance:** If at the start of the  $j = i$  iteration, the subarray  $A[1..i-1]$  contains no elements equal to  $v$ , then there are two conditions to check. If  $A[j] = v$ , then we exit the loop and  $j$  is not incremented; hence, the loop invariant still holds. If  $A[j] \neq v$ , then  $j$  is incremented to  $i+1$  and  $A[1..i]$  contains no elements equal to  $v$  at the start of the  $j = i+1$  iteration; hence, the loop invariant still holds.

**Termination:** If the loop terminates with  $j = \text{length}[A] + 1$ , then  $A[1..\text{length}[A]]$  contains no elements equal to  $v$ , and the loop invariant holds. The algorithm will then return NIL at line 6, which is the correct response for not finding an element in  $A$  equal to  $v$ . If the loop terminates with  $j < \text{length}[A] + 1$ , then the algorithm has found an element  $A[j]$  which is equal to  $v$ , returning key  $j$  at line 4 without incrementing  $j$ . Note that it is still the case that no elements in  $A[1..j-1]$  are equal to  $v$ , and so the loop invariant holds.

Hence, the algorithm is correct when there is an element in  $A$  which is equal to  $v$  and when there is no such element.

#### (7) CLR 2.2-2

##### SELECTION-SORT( $A$ )

1.  $n \leftarrow \text{length}[A]$
2. **for**  $i \leftarrow 1$  to  $n - 1$
3.     **do**  $\text{min} \leftarrow \infty$
4.          $\triangleright$  Find smallest element of current portion of  $A[i..n]$
5.     **for**  $j \leftarrow i$  to  $n$
6.         **if**  $A[j] < \text{min}$
7.             **do**  $\text{min} \leftarrow A[j]$
8.              $\text{key} \leftarrow j$
9.          $\triangleright$  Swap the smallest  $A[\text{key}]$  with the element  $A[i]$  and work with  $A[i+1..n]$
10.     **exchange**  $A[i] \leftrightarrow A[\text{key}]$
11. **return**  $A$



The worst case running time for SELECTION-SORT is  $\Theta(n^2)$ , because the algorithm must find the next smallest element for each iteration. The best case could be  $O(n)$  if the array is sorted by adding a linear check to see if the array is already sorted; however, without this check the best case is  $\Theta(n^2)$ .

The loop invariant is that at the start of the  $i$ th iteration  $A[1..i-1]$  contains the smallest  $i-1$  elements of  $A$  in sorted order.

At the end of the  $(n-1)$ th iteration, the  $n$ th element of  $A$  is no smaller than the first  $n-1$  elements (by the loop invariant). And since  $A[1..n-1]$  is sorted at this point, the whole array is sorted. Thus, there is no need for the  $n$ th iteration.

(8) CLR 2.2-4

Just modify the algorithm to store a precomputed answer for some of the inputs. Check for those inputs at the outset, and return the corresponding answer if it is detected.

(9) CLR 2.3-3

**Base Case:** For  $n = 2$ ,  $T(2) = 2 \lg 2 = 2$

**Induction Hypothesis:** Assume for  $n = 2^k$ ,  $T(2^k) = 2^k \lg 2^k$ .

**Induction Step:** Show this is true for  $n = 2^{k+1}$ , that is,  $T(2^{k+1}) = 2^{k+1} \lg 2^{k+1}$ .

$$\begin{aligned} T(2^{k+1}) &= 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1} \\ &= 2T(2^k) + 2^{k+1} = 2(2^k \lg 2^k) + 2^{k+1} \\ &= 2^{k+1}(\lg 2^k + 1) = 2^{k+1}(\lg 2^k + \lg 2) \\ &= 2^{k+1}(\lg(2 \times 2^k)) = 2^{k+1} \lg(2^{k+1}) \end{aligned}$$

(10) CLR 2.3-4

The running time for the recursive version of INSERTION-SORT is the following:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + C(n) & \text{otherwise.} \end{cases}$$

where  $C(n)$  is the time to insert the  $n^{th}$  element into  $A[1..n-1]$ .

(11) CLR 2.3-7

FindMatch(S,x) ;return true if found, false if not

{

1. Mergesort(S,1,n) ;  $\Theta(n \cdot \log n)$



2. if  $x > S[n] + S[n-1]$  or  $x < S[1] + S[2]$  return false ; this line is optional, it can save some time by skipping the search for cases where  $x$  is too small or too large to be the sum of a pair of values in the sequence.
3. for  $i \leftarrow 1$  to  $n$
4.   do if (BinarySearch( $S, x - S[i]$ )=true);  $\Theta(\log n)$
5.    then return true
6. return false
- }

## (12) CLR 2-2

- (a) The output  $A'[1] \dots A'[n]$  should be a permutation of the input  $A[1] \dots A[n]$ .
- (b) The loop-invariant for the loop in lines 2-4 is that at the end of each iteration  $A[j-1] \leq A[k]$  for all  $k > j-1$ . This can be proven as follows:

**Initialization:** Before the first iteration either  $A[n-1] \leq A[n]$  or  $A[n-1] > A[n]$ . In the former case, the invariant will be true at the end of the first iteration because the **If** in line 3 evaluates to false and the exchange does not take place. In the latter case the **If** in line 3 evaluates to true, causing the last elements of the array  $A$  to be exchanged, and ensuring the invariant to hold true at the end of the iteration.

**Maintenance:** At the beginning of each iteration,  $A[j]$  is known to be no greater than  $A[k]$  for all  $k > j$ . During each iteration the elements  $A[j]$  and  $A[j-1]$  are compared and possibly exchanged such that the greater of the two is placed in  $A[j]$ . This ensures that at the end of the iteration  $A[j-1]$  is not greater than  $A[j]$ , and by transitivity also not greater than  $A[k]$  for all  $k > j-1$ .

**Termination:** At termination  $j = i + 1$ , so  $j - 1 = i$ . Because of the loop invariant we have that  $A[i] \leq A[k]$  for all  $k > i$ .

- (c) The loop-invariant for the loop in lines 1-4 is that at the end of each iteration the  $A[1] \leq A[2] \leq A[3] \dots \leq A[i]$ .

**Initialization:** At the end of the first iteration of the outer loop  $i = 1$  and the loop-invariant is trivially true.

**Maintenance:** The termination condition of the inner loop as proved in part(b) ensures that at the end of each iteration of the outer loop  $A[i] \leq A[k]$  for all  $k > i$ . In the next iteration the element that is brought into  $A[i+1]$  will be chosen from amongst  $A[k]$  for  $k > i$ . We are guaranteed from the termination condition of the inner loop in the previous iteration of the outer loop that this element  $A[i+1] \geq A[i]$ . Hence on each iteration of the outer loop the element brought into  $A[i]$  is such that  $A[i-1] \leq A[i]$ . The elements  $A[1] \dots A[i-1]$  are not disturbed during the  $i$ 'th iteration. So at the end of the  $i$ 'th iteration we have that  $A[1] \leq A[2] \leq A[3] \dots \leq A[i]$ .

**Termination:** At the end of the last iteration of the outer loop  $i = \text{length}[A]$  and the loop-invariant ensures that  $A[1] \leq A[2] \leq A[3] \dots \leq A[i]$ . Thus the algorithm terminates with the array  $A$  sorted in non-ascending order.



- (d) The outer loop iterates  $n = \text{length}[A]$  times. The inner loop iterates  $i$  times where  $i$  increases by 1 on each iteration of the outer loop. The running time of the algorithm can be summed as:

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n c_1$$

$$T(n) = \sum_{i=1}^n (n-i)c_1$$

$$T(n) = \sum_{i=1}^n nc_1 - \sum_{i=1}^n ic_1$$

$$T(n) = n^2c_1 - \frac{n(n+1)}{2}c_1$$

$$T(n) = \frac{1}{2}n^2c_1 - \frac{n}{2}c_1$$

$$T(n) = \Theta(n^2)$$

The running time for BUBBLESORT is the same as the worst-case running time for INSERTION-SORT

### (13) CLR 2-4

- (a) The input sequence,  $\langle 2, 3, 8, 6, 1 \rangle$ , has the following inversions:  $\{(1, 5), (2, 5), (3, 5), (4, 5), (3, 4)\}$ .
- (b) The array with the most inversions contains unique elements and is reverse sorted, for example,  $A = \{n, n-1, \dots, 1\}$ . In this case, there are the following number of inversions:

$$N_{inv} = (n-1) + (n-2) + \dots + (n-(n-1)) = \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- (c) Each element  $j$  of the array that is not in the correct position is shifted left until it reaches the element  $i$  such that  $A[i] > j$ . The number of shifts performed by insertion sort corresponds to the number of inversions of every element  $k$ ,  $j \leq k < i$  in the relation to  $i$  (i.e.,  $\text{inversion}(k, i)$ ). Each shift eliminates an inversion from the set.
- (d) The algorithm to determine the number of inversions in  $\Theta(n \lg n)$  uses MERGE-SORT with an enhancement in the MERGE procedure. When an array  $A_{low}$  ( $A$  with the lower index range) is merged with an array  $A_{high}$  ( $A$  with the greater index range, further from the index 1), if an element is picked from the top of  $A_{high}$ , the number of elements left in  $A_{low}$  should be added to form the total number of inversions at the end of the algorithm.

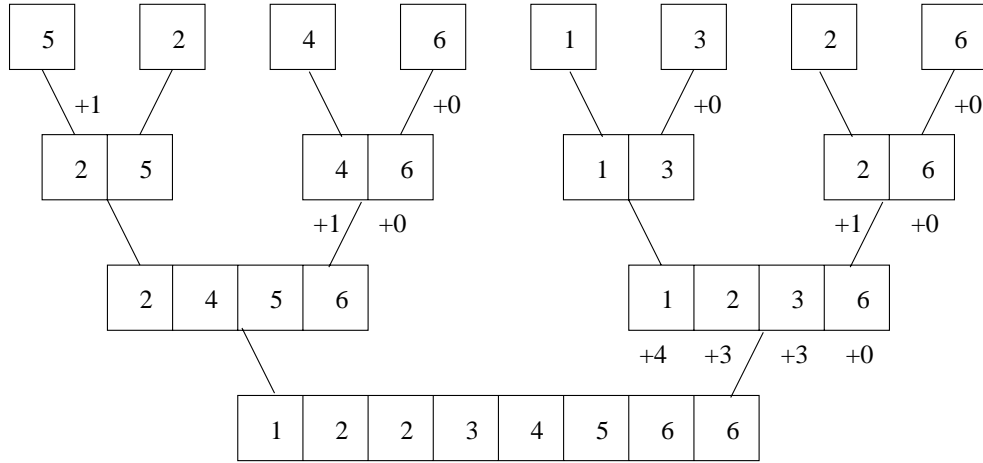


Figure 2: The MERGE-SORT enhancement for the example in Problem CLR 2-4(d).

For example,  $\langle 5, 2, 4, 6, 1, 3, 2, 6 \rangle$  has  $N_{inv} = 13$ , using the MERGE-SORT enhancement (see Figure 2).

MERGE-SORT can be modified to count inversions by adding line 18 to MERGE, as shown below. Note that *Num\_of\_inv* is a global variable initialized to zero that contains the total number of inversions when MERGE-SORT terminates. This will not change the asymptotic behavior of MERGE-SORT.

```

MERGE( $A, p, q, r$ )
1.  $n_1 \leftarrow q - p + 1$ 
2.  $n_2 \leftarrow r - q$ 
3. Create arrays  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$ 
4. for  $i \leftarrow 1$  to  $n_1$ 
5.   do  $L[i] \leftarrow A[p + i - 1]$ 
6. for  $j \leftarrow 1$  to  $n_2$ 
7.   do  $R[j] \leftarrow A[q + j]$ 
8.  $L[n_1 + 1] \leftarrow \infty$ 
9.  $L[n_2 + 1] \leftarrow \infty$ 
10.  $i \leftarrow 1$ 
11.  $j \leftarrow 1$ 
12. for  $k \leftarrow p$  to  $r$ 
13.   do if  $L[i] \leq R[j]$ 
14.     then  $A[k] \leftarrow L[i]$ 
15.        $i \leftarrow i + 1$ 
16.   else  $A[k] \leftarrow R[j]$ 
17.      $j \leftarrow j + 1$ 
18.      $Num\_of\_inv \leftarrow Num\_of\_inv + n_1 + 1 - i$ 

```



One method of solution uses a logarithm to convert the product to a sum:

$$\begin{aligned}
 \prod_{k=1}^n 2 \cdot 4^k &= 2^n \prod_{k=1}^n 4^k \\
 \lg(2^n \prod_{k=1}^n 4^k) &= \lg(2^n) + \lg(\prod_{k=1}^n 4^k) \\
 &= \lg(2^n) + \sum_{k=1}^n \lg(4^k) \\
 &= \lg(2^n) + \lg(4) \sum_{k=1}^n k \\
 &= \lg(2^n) + 2\lg(2) \frac{n(n+1)}{2} \\
 &= n\lg(2) + n(n+1)\lg(2) \\
 &= n(n+2)\lg(2) \\
 &= \lg(2^{n(n+2)}) \\
 \prod_{k=1}^n 2 \cdot 4^k &= 2^{n(n+2)}.
 \end{aligned}$$

An alternative method of solution uses product and exponent rules:

$$\begin{aligned}
 \prod_{k=1}^n 2 \cdot 4^k &= 2^n \cdot 4^{\sum_{k=1}^n k} \\
 &= 2^n \cdot 4^{\frac{n(n+1)}{2}} \\
 &= 2^n \cdot 2^{n(n+1)} \\
 &= 2^{n+n(n+1)} \\
 &= 2^{n^2+2n} \\
 &= 2^{n(n+2)}
 \end{aligned}$$