

**ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY**



**ASSIGNMENT 1**

**Submitted To : Sir Jamal**

**Submitted By Sayyada Alishba(CS3A)**

**Roll no : 14668**

**Dated : 31-10-24**

**<https://github.com/Alishba390s>**

**<https://github.com/Alishba390/assignment-.git>**

## CHAPTER ONE: THE ROLE OF ALGORITHMS IN COMPUTING

### QUESTION 1:

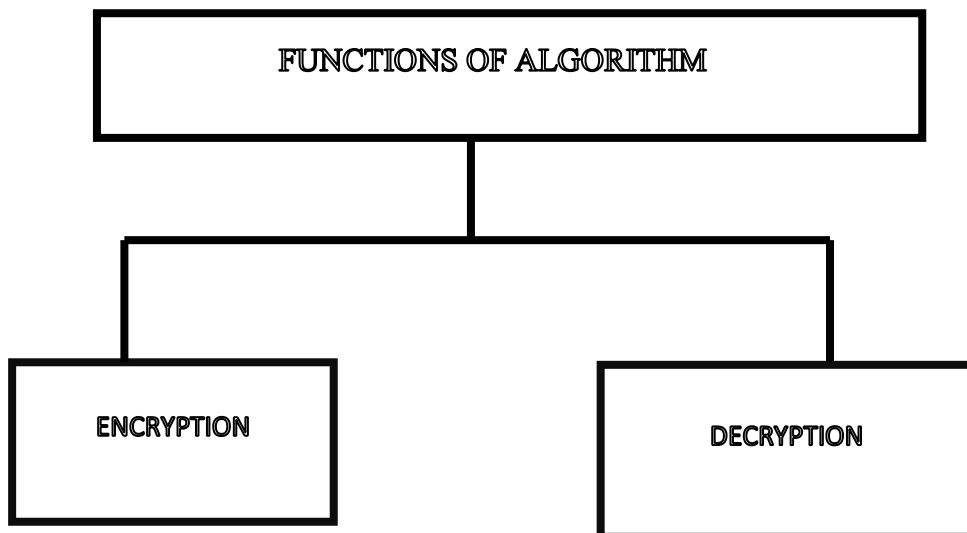
**Give the example of application that requires algorithmic content at the application level, and discuss the function of the algorithms involved .**

**Answer :**

One of the common application that requires algorithm content at the application level is encryption in secure messaging applications , such as WhatsApp or signals .encryption insures that communication between users is private and secure ,protecting data from unauthorised access. The algorithm used for this purpose is often a cryptographic algorithm , such as AES (ADVANCED ENCRYPTION STANDARDS ) or RSA (RIVEST – SHAMIR –ADLEMAN) .

### **FUNCTION OF THE ALGORITHM:**

In secure messaging, the algorithms primary functions are to encrypt and decrypt messages.



**Encryption (Symmetric or Asymmetric):**

When a user sends a message, the encryption algorithm transforms the readable message (plaintext) into an unreadable format (ciphertext) before it is transmitted.

### **AES (Symmetric Encryption):**

In symmetric encryption like AES, the same key is used to both encrypt and decrypt the message. AES works by breaking data into blocks and encrypting each block using the key.

### **RSA (Asymmetric Encryption):**

In asymmetric encryption like RSA, two keys are involved: a public key for encryption and a private key for decryption. This is widely used in scenarios where secure key sharing is important, such as exchanging keys over the internet.

### **Decryption:**

When the message is received by the intended recipient, the encryption algorithm reverses the process: it uses the key to convert the ciphertext back into the original readable message (plaintext).

In symmetric encryption (e.g., AES), the recipient uses the same shared key to decrypt the message.

In asymmetric encryption (e.g., RSA), the recipient uses their private key to decrypt the message, ensuring that only the intended recipient can read the message.

### **Question 2 :**

Suppose that for inputs of size  $n$  on a particular computer , insertion sort runs in  $8n^2$  steps and merge sort runs in  $64 n \lg n$  steps . For which values of  $n$  does inserton sort beat merge sort ?

### **Answer :**

Insertion sort takes  $=8n^2$

Merge sort takes  $=64 n \lg n$

We need to find when insertion sort eats merge sort

$$=8n^2 < 64 n \lg n$$

Dividing both sides by

$$= \frac{8n^2}{8n} < \frac{64n}{8n} \log_2 n$$

$$= n < 8 \log_2 n$$

Finding value of n :

Let suppose n = 1,2,3,5

When n=1

$$2 < 8 \log_2 (2)$$

$$= 8 \times 1$$

$$= 8$$

When n = 3

$$= 3 < 8 \log_2 (3)$$

$$= 8 \times 1.584$$

$$= 12.68$$

When n=5

$$= 5 < 8 \log_2 (5)$$

$$= 2.322 \times 8$$

$$= 1.857$$

When n= 15

$$= 15 < 8 \log_2 (15)$$

$$= 8 \times 3.9$$

$$= 31.25$$

**Insertion sort beats merge sort when n =15 which is equal to 31 .25**

### Question 3:

What is the smallest value of n such that an algorithm whose running time is  $100n^2$  runs faster than the algorithm whose running time is  $2^2$  on the same machine ?

**Answer :**

$$= 100n^2 < 2^n$$

We have to take some value of n

When n = 10

$$= 100(100^2)$$

$$= 100 \times 100$$

$$= 10,000$$

$$2^n$$

$$= 2^{10}$$

$$= 1024$$

$$100n^2 > 2^n$$

When n = 15

$$100n^2$$

$$= 100(15)^2$$

$$= 100 \times 225$$

$$= 22,500$$

$$2^n$$

$$= 2^{15}$$

$$= 32768$$

**When n = 15 which is sameless value where  $100n^2$  becomes less than  $2^n$ . Therefore smallest value of n such that the first algorithm runs faster than the second one is n = 15**

**Question 4 :**

**Describe your own real-world example that requires sorting. Describe one that requires the shortest distance between two points.**

**Answer :**

**Example : Real-World Sorting and Shortest Distance**

A library cataloging system requires sorting books by author, title or publication date to efficiently organize and retrieve books.

Google Maps calculates the shortest driving distance between two cities, providing the most efficient

**Question 5:**

**Answer :**

Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

**1. Memory Usage:**

The amount of memory an algorithm requires.

**2. Scalability:**

How well an algorithm performs with large inputs.

**3. Complexity:**

The simplicity or intricacy of the algorithm's design.

**4. Accuracy:**

The precision of the algorithm's results.

**5. Reliability:**

Consistency of performance under varying conditions.

**Question 6 :**

**Select a data structure that you have seen, and discuss its strengths and limitations.**

**Answer**

**Data Structure Analysis**

**1. Data Structure:**

Hash Table

**2. Strengths:**

Efficient search, insertion and deletion operations ( $O(1)$  average time).

Fast lookup and retrieval.

### **3. Limitations:**

Requires careful hash function selection.

Can be sensitive to input distribution.

May experience collisions.

### **Question 7 :**

**How are the shortest-path and traveling-salesperson problems given above similar?**

**How are they different?**

**Answer :**

### **Shortest-Path and Traveling Salesperson Problems**

#### **Similarities:**

Both involve finding optimal paths.

#### **Differences:**

**Shortest-Path:** Finds the shortest path between two points.

**Traveling Salesperson:** Finds the shortest possible tour that visits a set of cities and returns to the starting point.

### **Question 8 :**

**Suggest a real-world problem in which only the best solution will do. Then come up with one in which <approximately= the best solution is good enough.**

**Answer:**

### **Real-World Problems Requiring Best or Approximate Solutions**

#### **1. Best Solution:**

Financial portfolio optimization, where optimal asset allocation is crucial.

#### **2. Approximate Solution:**

Image or video compression, where near-optimal compression suffices.

### Question 8 :

Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advanced and arrives over time .

Answer :

#### Real-World Input Availability

##### 1. Entire Input Available:

Compiling code, where the entire program is available before compilation.

##### 2. Input Arrives Over Time:

Streaming video platforms, where data arrives continuously.

### Problem of chapter one :

#### Comparison of running times

For each function  $f(n)$  and time  $t$  in the following table , determine the largest size  $n$  of a problem that can be solved in time  $t$  , assuming that the algorithm to solve the problem takes  $f(n)$  microseconds .

Solution :

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^1 \times 10^6$	$2^6 \times 10^7$	$2^{3.6} \times 10^9$	$2^{8.64} \times 10^{10}$	$2^{2.592} \times 10^{12}$	$2^{3.1536} \times 10^{13}$	$2^{3.15576} \times 10^{15}$
$\sqrt{n}$	$1 \times 10^{12}$	$3.6 \times 10^{15}$	$1.29 \times 10^{19}$	$7.46 \times 10^{21}$	$6.72 \times 10^{24}$	$9.95 \times 10^{26}$	$9.96 \times 10^{30}$
$n$	$1 \times 10^6$	$6 \times 10^7$	$3.6 \times 10^9$	$8.64 \times 10^{10}$	$2.59 \times 10^{12}$	$3.15 \times 10^{13}$	$3.16 \times 10^{15}$
$n \lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	$6.86 \times 10^{13}$
$n^2$	1000	7745	60000	293938	1609968	5615692	56176151
$n^3$	100	391	1532	4420	13736	31593	146679
$2^n$	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

## CHAPTER 2 : GETTINGN STARTED

### EXERCISE :

#### Question 1 :



Using figure 2.2 as a model , illustrate th operation of insertion sort on an array initially containing the sequence (31, 41 ,59 ,26 ,41, 58)

Answer :

a)

31	41	59	26	41	58
----	----	----	----	----	----

b)

26	31	41	59	41	58
----	----	----	----	----	----

c)

26	31	41	41	59	58
----	----	----	----	----	----

d)

26	31	41	41	58	59
----	----	----	----	----	----

Question 2 :

Consider the procedure SUM -- ARRAY on the facing page . It computes the sum of the n numbers in array A[1: n] . State a loop variant for this procedure and use its initialization , maintenance , and termination properties to show that the SUM- ARRAY procedure returns the sum of the numbers in A[1: n].

SUM ARRAY (A ,n)

```
1  sum= 0
2  for i = 1 to n
3      sum = sum + a[i]
4  Return sum
```

Answer :

Loop Variant: i

Initialization:

At the start of the loop (line 2),  $i = 1$ , which is the first index of the array.

Maintenance:

In each iteration (lines 2-4),  $i$  increments by 1 (from  $i$  to  $i+1$ ), ensuring that every element in the array is processed exactly once.

**Termination:**

The loop terminates when  $i = n$  (line 2), meaning all  $n$  elements in the array have been processed.

**Loop Invariant:**

At the start of each iteration,  $\text{sum}$  is the sum of the elements in  $A[1:i-1]$

**Proof of Correctness:**

1. Initialization:  $\text{sum} = 0$  (line 1), which is the sum of the elements in  $A[1:0]$  (an empty array).
2. Maintenance: In each iteration,  $\text{sum}$  is updated to include the current element  $A[i]$  (line 3).
3. Termination: When  $i = n$ ,  $\text{sum}$  is the sum of all elements in  $A[1:n]$ .

Therefore, the SUM-ARRAY procedure correctly returns the sum of the numbers in  $A[1:n]$ .

**Correctness Theorem:**

The SUM-ARRAY procedure returns the sum of the numbers in  $A[1:n]$

By the loop invariant and termination properties,  $\text{sum}$  is the sum of all elements in  $A[1:n]$  when the loop terminates. The procedure returns this value (line 4), ensuring correctness.

```
def sum_array(A):  
    n = len(A)      # Length of the array  
    sum = 0         # Initialize sum to 0  
    for i in range(n):  
        sum += A[i] # Add each element in A to sum  
    return sum      # Return the final sum
```


```
A = [1, 2, 3, 4, 5]  
print(sum_array(A)) # Output: 15
```

**Question 3 :**

Rewrite the insertion sort procedure to sort into monotonically decreasing instead of monotonically increasing order .

```
def insertion_sort_descending(A):
    n = len(A)
    for j in range(1, n):
        key = A[j]
        i = j - 1
        # Move elements of A[0...j-1] that are less than the key to one
        position to the right
        while i >= 0 and A[i] < key:
            A[i + 1] = A[i]
            i -= 1
        A[i + 1] = key

# Example usage:
A = [5, 2, 4, 6, 1, 3]
insertion_sort_descending(A)
print(A) # Output: [6, 5, 4, 3, 2, 1]
```

 Copy code

```
[6, 5, 4, 3, 2, 1]
```

**Question 4 :**

Consider the searching problem

**Input :** A sequence of  $n$  numbers ( $a_1, a_2, a_3, \dots, a_n$ ) stored in array  $A[1:n]$  and a value  $x$

**Output :** an index  $i$  such that  $x$  equals  $A[i]$  or the special value NIL if  $x$  does not appear in  $A$  .

Write pseudo code for linear search , which scans through the array from the begin .....the three necessary properties .

**Answer :**

**Input :**

Two n element array A and B containing the binary digits of two numbers a and b

**Output :**

An ( n+1) element array C containing the binary digits of a+b

```
i = NIL
for j = 1 to A.length do
  if A[j] = v then
    i = j
    return i
  end if
end for
return i
```

---

**Question 5 :**

Consider the problem of adding two n –bits integers a and b .....along with the length n , and returns array C holding the sum .

```
carry = 0
for i=n to 1 do
  C[i+1] = (A[i] + B[i] + carry) (mod 2)
  if A[i] + B[i] + carry ≥ 2 then
    carry = 1
  else
    carry = 0
  end if
end for
C[1] = carry
```

**Carry =0**

**For i =n to 1 do**

**C[i+1]=(A[i] + B[j] +array )(mod 2)**

**If A[i] +B[i]+carry >2 then**

**Carry=1**

**Else**

**Carry =0**

**End if**

**End for**

C[1]=carry

**Question 6 :**

Express the function  $n^3/1000 + 100n^2 + 100n + 3$  in terms of theta notation

**Solution :**

$$\frac{n^3}{1000} + 100n^2 + 100n + 3 \in \theta(n^3)$$

**Question 7 :**

Consider sorting numbers sorted in array A[1 : n] by first finding .....best case running time any better ?

**Answer :**

**Input :** An n –element array A

**Output :** The array A with its elements rearranged into increasing order

The loop invariant of selection sort is as follows:

At each iteration of the for loop of lines 1 through 10, the subarray **A[1..i – 1]** contains the **i – 1** smallest elements of A in increasing order. After **n – 1** iterations of the loop, the **n – 1** smallest elements of A are in the first **n – 1** positions of A in increasing order, so then the element is necessarily the largest element. Therefore we do not need to run the loop a final time. The best-case and worst-case running times of selection sort are  $\Theta(n^2)$ . This is because regardless of how the elements are initially arranged, on the  $i^{th}$  iteration of the main for loop the algorithm always inspects each of the remaining **n–i** elements to find the smallest one remaining

---

```
for i = 1 to n – 1 do
    min = i
    for j = i + 1 to n do
        // Find the index of the  $i^{th}$  smallest element
        if A[j] < A[min] then
            min = j
        end if
    end for
    Swap A[min] and A[i]
end for
```

This yields a running time of

$$\sum_{i=1}^{n-1} n - i = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2 - n}{2} = \frac{n^2 - n}{2} = \Theta(n^2).$$

### Question 8 :

Consider linear search ..... justify your answer ?

**Answer:**

In linear search, we look through an array one element at a time until we find the target element or reach the end of the array. Let's analyze this process in terms of the number of comparisons needed on average and in the worst case, and then derive the running times.

#### Average Case

If the target element is equally likely to be at any position in an array of elements, on average, we would need to check half of the elements to find it. This is because the target could be at any position from the first to the last, and the average position of the target would be roughly in the middle.

Thus, the average number of comparisons is:

$$\frac{n}{2}$$

#### Worst Case

In the worst case, the target element is either not present in the array or is at the very last position. In this case, we have to check every single element, resulting in comparisons.

#### Running Time Analysis

Using Big-O notation:

Average-case time complexity: , because the average number of comparisons grows linearly with .

#### Worst-case time complexity:

Since in the worst case we must inspect all elements.

## Justification

Linear search has to check each element one by one, so in both the average and worst cases, the number of comparisons increases linearly with the array size.

## Question 9 :

**How can you modify any sorting algorithm to have a good best –case running time ?**

## Answer :

To modify a sorting algorithm for a better best-case running time, one strategy is to first check if the array is already sorted or partially sorted before proceeding with the full algorithm. Here's a general approach to achieve this:

### 1. Add a Pre-Sorted Check:

Before starting the sorting algorithm, perform a quick scan through the array to check if it's already sorted. This step takes  $O(n)$  time, where  $n$  is the number of elements in the array. If the array is already sorted, you can skip the sorting process entirely and return the array as is, achieving a best-case running time of  $O(n)$ .

### 2. Early Exit Conditions:

For algorithms like insertion sort, bubble sort, or quicksort, you can add checks within the main sorting loop to detect if the array becomes sorted before all iterations are complete. For example:

## Insertion Sort:

If no swaps are needed for a given pass, the algorithm can terminate early.

## Bubble Sort:

If a pass through the array makes no swaps, you can exit the loop, as the array is already sorted.

### 3. Adaptive Sorting Algorithms:

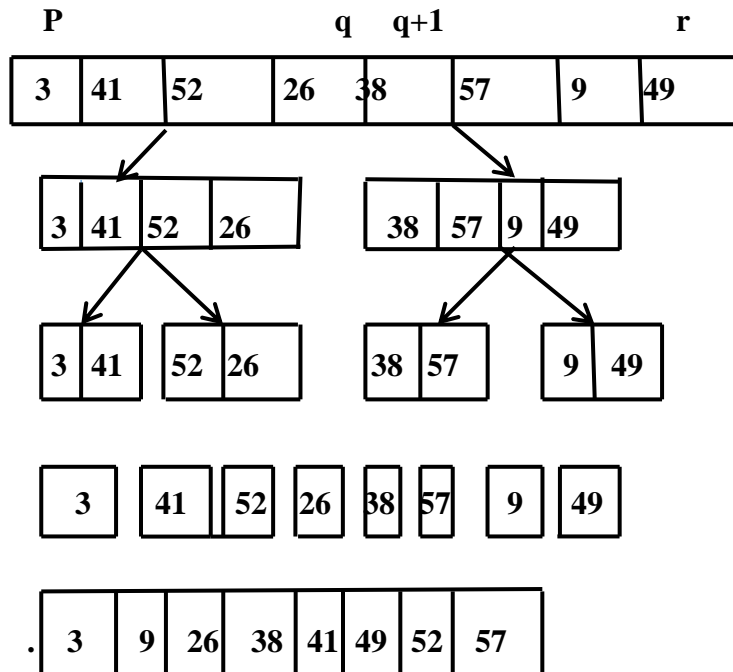
Some sorting algorithms are adaptive, meaning they adjust their behavior based on the initial order of the elements. Algorithms like TimSort (used in Python and Java) are designed to take advantage of existing order in the array, making them more efficient when the data is partially sorted.

By incorporating these modifications, you can improve the best-case running time for sorting algorithms to when the array is already sorted, while maintaining their original average and worst-case complexities. This approach leverages the fact that in practical applications, data is often partially sorted, so adaptive algorithms or pre-checks can yield significant efficiency gains.

## Question 11:

Using figure 2.4 as a model , illustrate the the operation of merge sort on an array initially containing the sequence ( 3 ,41,52,26,38,57,9,49)

Answer :



Question 12 :

The test in line 1 of the merge ..... Call as  $p > r$

Answer :

The following is a rewrite of MERGE which avoids the use of sentinels. Much like MERGE, it begins by copying the subarrays of A to be merged into arrays L and R. At each iteration of the while loop starting on line 13 it selects the next smallest element from either L or R to place into A. It stops if either L or R runs out of elements, at which point it copies the remainder of the other subarray into the remaining spots of A

Question 13 :

State a loop variant for the while loop of times 12 -18 .....MERGE procedure is correct .

Answer :



Since  $n$  is a power of two, we may write  $n = 2^k$ . If  $k = 1$ ,  $T(2) = 2 = 2 \log(2)$ . Suppose it is true for  $k$ , we will show it is true for  $k + 1$ .

$$\begin{aligned} T(2^{k+1}) &= 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1} = 2T(2^k) + 2^{k+1} = 2(2^k \lg(2^k)) + 2^{k+1} \\ &= k2^{k+1} + 2^{k+1} = (k+1)2^{k+1} = 2^{k+1} \lg(2^{k+1}) = n \lg(n) \end{aligned}$$

#### Question 14 :

Use mathematical induction to show that when .....  $T(n) = n \log n$

**Answer :**

Let  $T(n)$  denote the running time for insertion sort called on an array of size  $n$ . We can express  $T(n)$  recursively as

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

where  $I(n)$  denotes the amount of time it takes to insert  $A[n]$  into the sorted array  $A[1..n-1]$ . Since we may have to shift as many as  $n-1$  elements once we find the correct place to insert  $A[n]$ , we have  $I(n) = \Theta(n)$

```

 $n_1 = q - p + 1$ 
 $n_2 = r - q$ 
let  $L[1..n_1]$  and  $R[1..n_2]$  be new arrays
for  $i = 1$  to  $n_1$  do
     $L[i] = A[p + i - 1]$ 
end for
for  $j = 1$  to  $n_2$  do
     $R[j] = A[q + j]$ 
end for
 $i = 1$ 
 $j = 1$ 
 $k = p$ 
while  $i \neq n_1 + 1$  and  $j \neq n_2 + 1$  do
    if  $L[i] \leq R[j]$  then
         $A[k] = L[i]$ 
         $i = i + 1$ 
    else  $A[k] = R[j]$ 
         $j = j + 1$ 
    end if
     $k = k + 1$ 
end while
if  $i == n_1 + 1$  then
    for  $m = j$  to  $n_2$  do
         $A[k] = R[m]$ 
         $k = k + 1$ 
    end for
end if
if  $j == n_2 + 1$  then
    for  $m = i$  to  $n_1$  do
         $A[k] = L[m]$ 
         $k = k + 1$ 
    end for
end if
end if

```

### Question 15 :

You can also think of insertion sort as recursive algorithm ..... running time .

Answer :

**Recursive Insertion Sort:**

To sort an array **A[1:n]** :

- Sort **A[1:n -1]** recursively.
- Insert **A[n]** into the sorted subarray **A[1 : n -1 ]**

```

RecursiveInsertionSort(A, n):
    if n <= 1:
        return
    RecursiveInsertionSort(A, n - 1)
    Insert(A, n)

Insert(A, n):
    key = A[n]
    j = n - 1
    while j > 0 and A[j] > key:
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = key

```

### Recurrence for worst case :

- In the worst case , insertion takes  $O(n)$  time to insert the  $n$ -th elements
- The recurrence relation for the worst –case  $T(n)$  can be written as  
$$T(n) = T(n-1) + O(n)$$

### Question 16 :

Referring back to the search problem ..... subarray from further

### Answer :

```
BinarySearch(A, low, high, v):  
    if low > high:  
        return -1 # v is not present in the array  
    mid = (low + high) // 2  
    if A[mid] == v:  
        return mid  
    elif A[mid] > v:  
        return BinarySearch(A, low, mid - 1, v)  
    else:  
        return BinarySearch(A, mid + 1, high, v)
```

### Question 17 :

The while loop of lines 5 – 7 of the INSERTION –SORT ..... to theta ( $n \log n$ )?

### Answer :

By using binary search instead of linear search ,we could find the position for insertion in  $O(\log n)$  times . However the shift operation to insert the element still requires  $O(n)$  time in the worst case . Therefore , shift we replace the linear search with binary search , the worst –case running time remains  $O(n^2)$  as the cost of shifting elements dominates

### Question 18 :

Describe an algorithm that gives the set of  $S$  of  $n$  integers .....worst case

Answer :

#### 1. Algorithm :

- First sort the array  $S$  in  $O(n \log n)$
- Use the two pointer techniques (set one at the beginning and the other at the end of the sorted array )
- if sum of the both elements are true then return true
- if sum is less move the left pointer to the right
- if sum is greater then move right to the left
- Continue until the pointer meets

#### 2 . pseudocode :

```
TwoSum(S, x):
    S.sort() # Sorts the array in  $O(n \log n)$  time
    left = 0
    right = len(S) - 1
    while left < right:
        sum = S[left] + S[right]
        if sum == x:
            return True
        elif sum < x:
            left += 1
        else:
            right -= 1
    return False
```

#### 3 Running time analysis :

Sorting takes  $O(n \log n)$  times

The two pointers scans takes  $O(n)$

Thus , the overall running time is  $\theta(n \log n)$  , meeting the requirements

### Problems of chapter 2

#### Problem one :

Although merge sort runs in  $\theta(n \log n)$  worst case time and insertion sort runs in  $\theta(n^2)$  worst case time , the constant factors .....k in practice .

**Answer :**

**Insertion sort on small array in merge sort :**

- we have worst case time complexity of insertion sort on a single sub list of length  $k$  is  $\Theta(k^2)$  so  $\frac{n}{k}$  of them will take time  $\Theta(\frac{n}{k} k^2) = \Theta(nk)$

b) Suppose we have coarseness  $k$ . This means we can just start using the usual merging procedure, except starting it at the level in which each array has size at most  $k$ . This means that the depth of the merge tree is

$$\log(n) - \log(k) = \log(n/k).$$

Each level of merging is still time  $cn$ , so putting it together, the merging takes time

$$\Theta(n \log(n/k))$$

c) Viewing  $k$  as a function of  $n$ , as long as  $k(n) \in O(\lg(n))$ , it has the same asymptotics. In particular, for any constant choice of  $k$ , the asymptotics are the same

d) If we optimize the previous expression using our calculus 1 skills to get  $k$ , we have that  $c_1 n - \frac{nc_2}{k} = 0$  where  $c_1$  and  $c_2$  are the coefficients of  $nk$  and  $n \log(n/k)$  hidden by the asymptotics notation. In particular, a constant choice of  $k$  is optimal. In practice we could find the best choice of this  $k$  by just trying and timing for various values for sufficiently large  $n$

**Problem 2 :**

**Correctness of horner's rule**

**You get the coefficients .....the value of  $x$**

**HORNER(A , n, x)**

```
1  p = 0
2  for i = n downto 0
3      p = A[i] + x*p
4  return
```

**Answer :**

- a) If we assume that the arithmetic can all be done in constant time, then since the loop is being executed  $n$  times, it has runtime  $\Theta(n)$

b)

```

1:  $y = 0$ 
2: for  $i=0$  to  $n$  do
3:    $y_i = x$ 
4:   for  $j=1$  to  $n$  do
5:      $y_i = y_i x$ 
6:   end for
7:    $y = y + a_i y_i$ 
8: end for

```

- c) Initially,  $i = n$ , so, the upper bound of the summation is  $-1$ , so the sum evaluates to 0, which is the value of  $y$ . For preservation, suppose it is true for an  $i$ , then

$$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = a_i + x \sum_{k=1}^{n-i} a_{k+i} x^{k-1} = \sum_{k=0}^{n-i} a_{k+i} x^k$$

At termination,  $i = 0$ , so is summing up to  $n - 1$ , so executing the body of the loop a last time gets us the desired final result.

- d) We just showed that the algorithm evaluated  $\sum_{k=0}^n a_k x^k$ . This is the value of the polynomial evaluated at  $x$

### Problem 3 :

Let  $A[1 : n]$  be an array of  $n$  distinct number . if  $i < j$  and  $A[i] > A[j]$  then the pair  $(i, j)$  is called an inversion of  $A$

Answer :

- In the array  $[2, 3, 8, 6, 1]$  the five inversions are  $(3, 4), (1, 5), (2, 5), (3, 5), (4, 5)$
- The array  $[n, n-1, \dots, 1]$  has the most inversion s with  $\frac{n(n-1)}{2}$  inversions
- The running time of insertion sort is  $O(n+k)$  where  $k$  is the number of inversions . More inversions lead to longer sorting time with  $O(n^2)$  in the worst case
- Used a modify merge sort to count inversions in  $O(n \log n)$  time .During merging count when elements from the right subarray are less than those in the left

## CHAPTER 3 : ASYMPOTOTIC NOTATIONS

### Exercise :

**Modify the lower bound arguments for insertion sort to handle input sizes that are not necessary a multiple of 3 .**

### Answer :

The lower bound for comparison-based sorting algorithms can be shown to be  $\Omega(n \log n)$  by considering the number of possible permutations of an input of size  $n$  and the decision tree model. For insertion sort, this bound can still apply even if  $n$  is not a multiple of 3. The reasoning for the lower-bound argument doesn't change fundamentally; it's based on comparing elements to place them in the correct position, requiring  $\Omega(n^2)$  comparisons in the worst case.

Therefore, the lower bound for insertion sort for any arbitrary input size  $n$  remains  $\Omega(n^2)$ , regardless of whether  $n$  is a multiple of 3 or not.

### Question 2 :

**Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.**

### Answer

In selection sort, we repeatedly find the smallest element in the unsorted portion of the list and move it to the beginning. In terms of comparisons:

In the first pass, selection sort makes  $n-1$  comparisons.

In the second pass, it makes  $n-2$  comparisons, and so on, until the last pass, where it makes 1 comparison.

The total number of comparisons is:

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

### Question 3 :

**Suppose that  $\alpha$  is a fraction in the range  $(0, 1)$ . Show how to generalize the lower-bound argument for insertion sort to consider an input in which the  $\alpha n$  largest values start in the first positions. What additional restriction do you need to put on  $\alpha$ ? What value of  $\alpha$  maximizes the**

**number of times that the largest values must pass through each of the middle array positions?**

**Answer :**

To generalize the lower-bound argument for insertion sort with the largest values starting in the first positions, we consider the number of times these values must be moved to reach their correct positions. For each element in the largest values to reach the sorted position, it must pass through the  $(1 - \alpha)$  remaining positions. The values in the first positions need to be compared with all other elements to move into their correct positions, giving a lower bound on the number of comparisons.

To maximize the number of comparisons, should be close to 0.5, as this maximizes the interactions between the largest and smallest elements, increasing the amount of reordering needed. Hence, setting provides a scenario where the largest values must go through the middle positions the maximum number of times.