Abbottabad University Of Science And Technology (AUST)

Name Waleed Babar

Roll No 14830

Class BSCS 3D

Assignment No 1

Submitted To Sir Jamal Abdul Ahad

**Q.1 Describe your own real word example that require sorting. Describe one that requires finding the shortest distance between two points?**

ANS:

**SORTING EXAMPLE:**

Imagine a class teacher take a test from class. After test he give some marks to each student of class, for arranging the students marks in ascending order the teacher will use the sorting. So he can make a conclusion which students are good and which are in poor condition.

**FINDING THE SHORTEST DISTANCE:**

Imagine your are college student and you go to college every day. So, you need to find the shortest path among many path which goes to your college. So you need to calculate distance and assume which path is shortest, so you can save your time, using algorithm like Dijkstra’s.

**Q.2 Other than speed, what other measures of efficiency might you need to consider in**

**A real-world setting?**

ANS:

Other than speed following are important measures of efficiency :

**COST EFFICIENCY:**

Reducing the cost without compromising on the quality and productivity.

**ACCURACY:**

Complete a task without error or with minimum error to avoid from correction.

**Q.3 Select a data structure that you have seen ,and discuss it strength and limitations?**

ANS:

I select a data structure that I have seen, which is array.

**STRENGTHS OF ARRAY:**

1. Accessing any element by index is simple, O(1) time complexity
2. No additional memory required to store address

**LIMITATIONS OF ARRAY:**

1. Addition or removal of elements from any index but the last means re-arranging the whole list, O(n) time complexity.
2. Accessing an element by value means traversing the list, O(n) time complexity.
3. Needs contiguous memory.

**Q.4 how are the shortest path and traveling sales person problems given above similar?How are they different?**

ANS:

The shortest path problem and traveling salesperson both have similar problem because both are trying to find the shortest distance on the graph with aim to minimize the total distance.

**SHORTEST PATH PROBLEM:**

In this we find the shortest route between two points without visiting the all points. This is easier and easily solve using Dijkstra method.

**TRAVELING SALES-PERSON PROBLEM:**

Find the shortest route that visit every points at once and return to the starting point. It is much harder than previous due to its complexity.

**Q.5 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 available in advance and arrives over time**.

ANS:

A best example in which we do our best solution that is to giving a vaccine to a critical or in serious condition patient having a known cure .

An approximately best solution would be giving a chemotherapy to a person falling in

Cancer because cancer have no vaccine exists.

**Q.6 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 advance and arrives over time.**

ANS:

Real world example could be grocery shopping.

First situation: You have a complete shopping list before you leave home. You know exactly what to buy, so you can plan your trip and finish faster.

Second situation: You only know some items to buy, and while you're shopping, you get more calls or messages with additional items. You have to keep adding to your list and change your shopping path as new items come in.

Sometimes you can plan everything in advance, but other times, you have to adjust as new information comes while you're shopping.

**Exercise 1.2**

**Q.1 give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.**

ANS:

Example of an application that requires algorithmic content at the application level is google map.

**GOOGLE MAP:**

Google map is the example when we finding the route or distance between two places.

**INVOLVED ALGORITHM:**

Finding the shortest path algorithm which is known as Dijkstra method.

**FUNCTION:**

This algorithm or method is used to calculate the shortest and fastest route from one point to another.

**Q.2 Suppose that for inputs of size n on a particular computer, insertion sort cans in 8n2 steps and merge sort runs in 64n log n steps. For which values of n does insertion sort beat merge sort?**

ANS:

For insertion sort to beat merge sort for input of size n, 82 must be less than 64 n lg n.

8n2  < 64n lg n

= 8n.n <8n.8lg n

8n and 8n from both side cancel each other.

= n < 8 lg n

= n/8 < lg n

= 2n/8 < n

To find the required range of values of n ,

Note that for n < 8, 2n/8 will be a fraction. So let’s start with n=8 and then checks for values of n which are powers of 2.

n = 8 implies that 28/8 = 2 <n

n = 16 implies that 216/8 = 4 <n

n = 32 implies that 232/8 = 16 <n

n = 64 implies that 264/8 = 64 >n

We have found approximate range of values of n where merge sort start to beat insertion sort ; some where between 32 and 64. now trying the middle values which lies between 32 and 64.

n = 48 implies that 248/8 = 64 >n

n = 40 implies that 240/8 = 32 <n

n = 43 implies that 243/8 = 42.4 <n

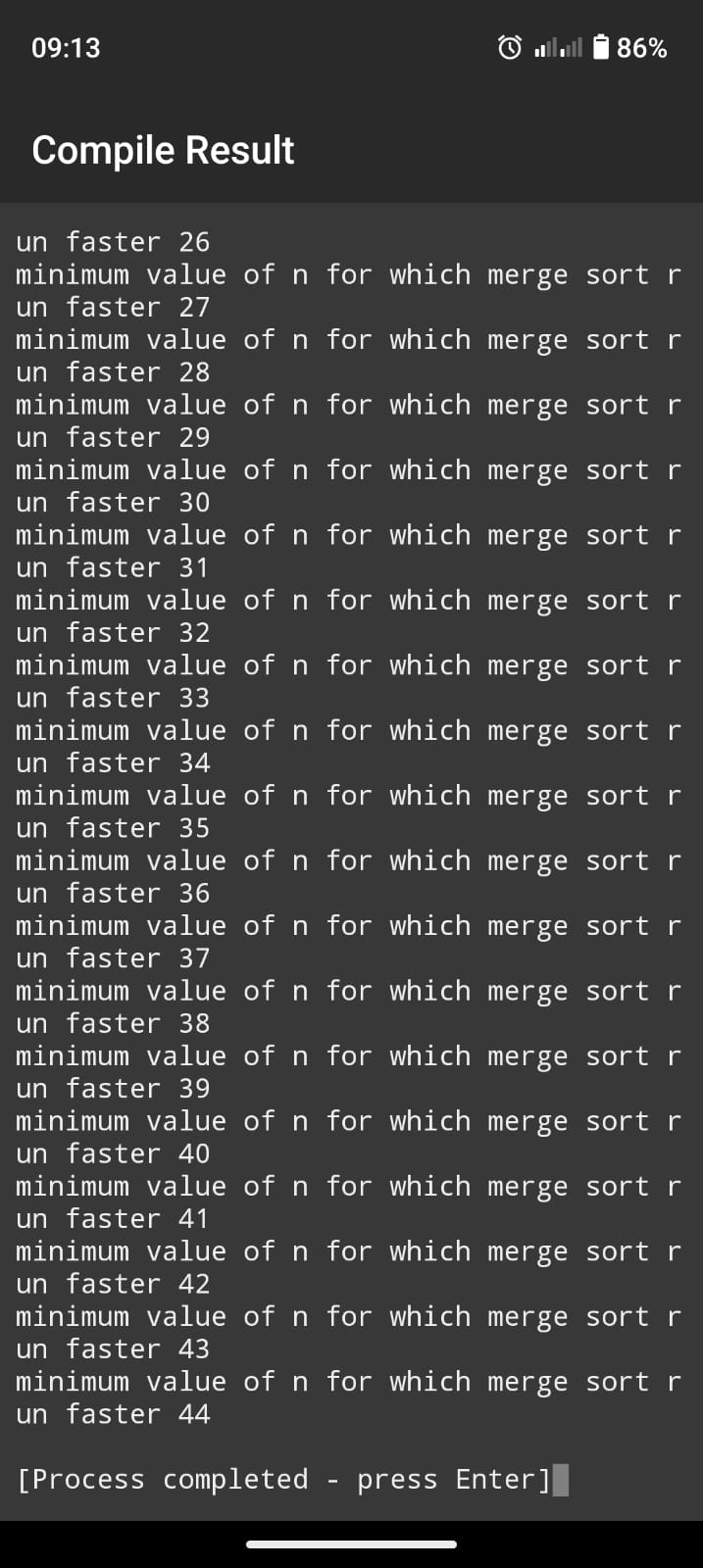
n = 44 implies that 244/8 = 44.8 >n

So at n=44 , merge sort start to beat insertion sort again therefore for 2<=n<=43 insertion sort beat merge sort.

**CODE:**



**OUTPUT**:



**Q.3 what is the smallest value of n such that an algorithm whose running time is 100 n2 runs faster than an algorithm whose running time is 2n on the same machine ?**

ANS:

For A to run faster than B , 100n2 must be smaller than 2n.

**CALCULATION:**

We can realize that A (quadratic time complexity ) will run much faster than (exponential time complexity) for every large values of n.

Let’s start from n=1 and go up for values of n which are power of 2 to see where that happens.

n = 1 implies that 100 \* 12 = 100 > 2n

n = 2 implies that 100 \* 22 = 400 > 2n

n = 4 implies that 100 \* 42 = 1600 > 2n

n =8 implies that 100 \* 82 = 6400 > 2n

n =16 implies that 100 \* 162 = 25600 < 2n

Between 8 and 16 , A starts run faster than B. Let’s we were going to try middle numbers which lies between 8 and 16.

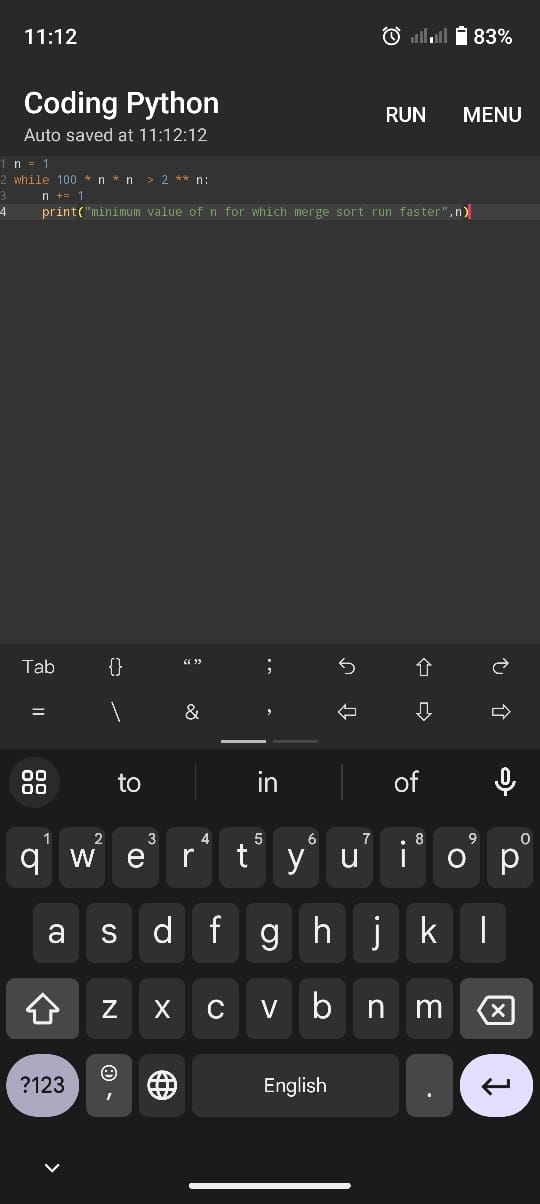
n = 12 implies that 100 \* 122 = 14400 > 2n

n = 14 implies that 100 \* 142 = 19600 > 2n

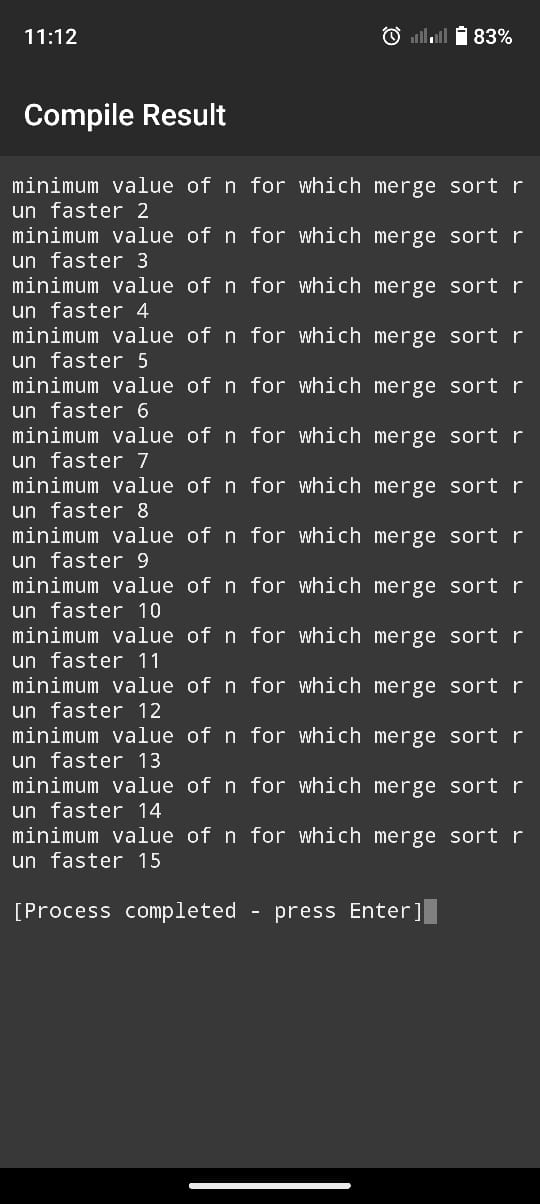
n = 15 implies that 100 \* 152 = 22500 > 2n

S0 at n=15 , A start runs faster than B.

**CODE:**



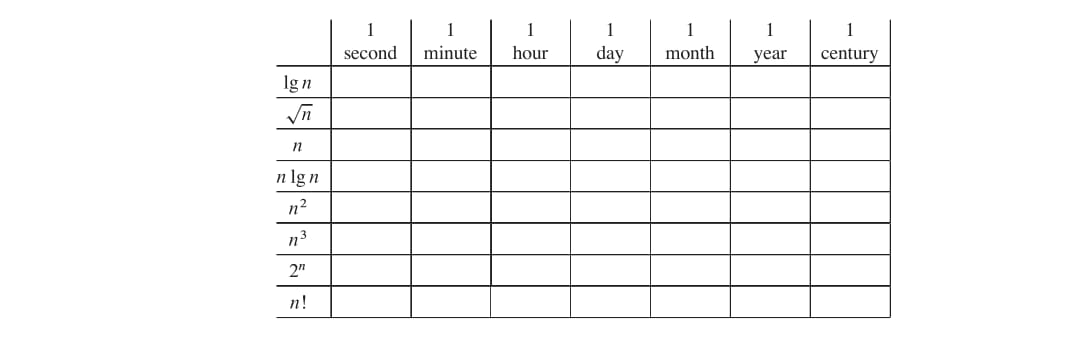
**OUTPUT**:



Problem 1.1

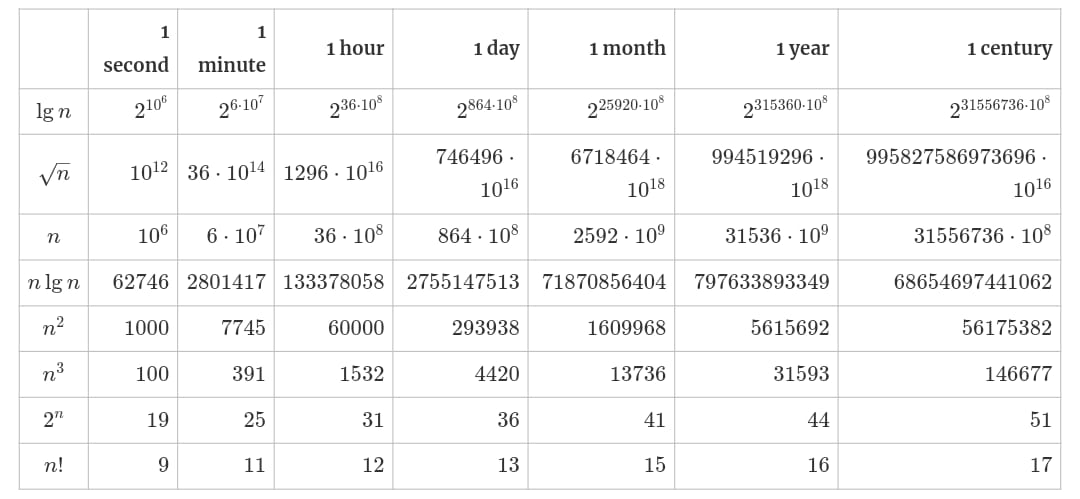
Comparison of Running Times

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 , assuming that the algorithm to solve the problem takes f(n) microseconds.



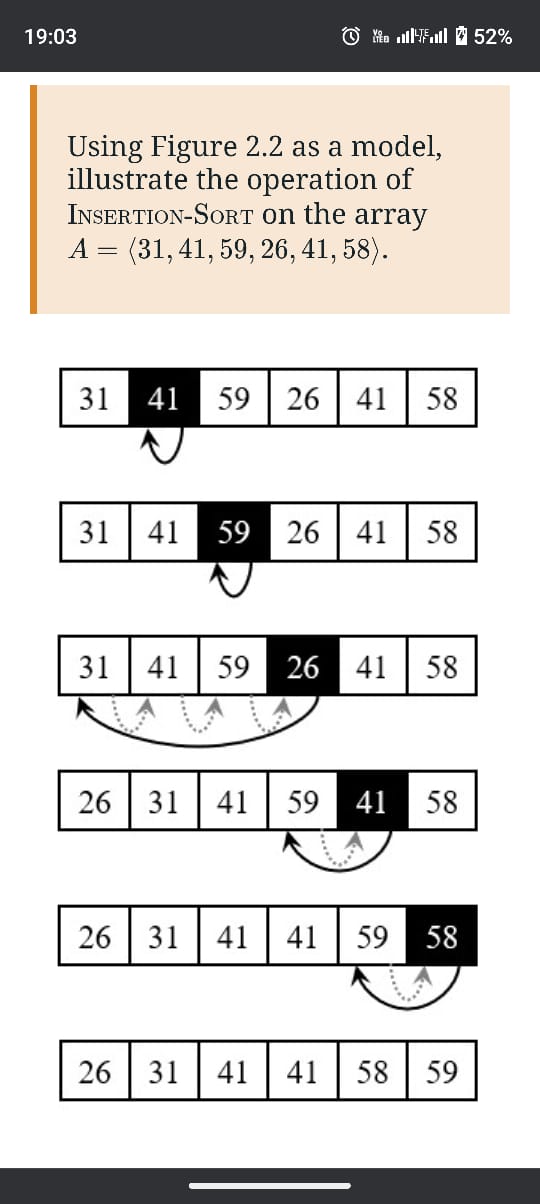
ANS:

For each function f(n) and time t in the fallowing table , determine the largest sizen of a pro blem that can be solved in time t, assuming that the algorithm to solve the problems takes f(n) microseconds.



**EXERCISE 2.1**

**Q.1 Using figure 2.2 as a model , illustrate the operation of insertion sort on a array initially containing the sequence (31, 41, 59, 26 ,41, 48,).**

:

**Q.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 invariant 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} ?**

ANS:

SUM-ARRAY(A, n)

sum = 0

for i = 1 to n

sum = sum + A[i]

return sum

The procedure accepts and Array (A) and the length of the array(n) Using loop invariant.

**Initialization:**

By assigning i to 1 the loop is initialized, executing the vode inside of the body of the loop. The code takes the current value of i in this case 1 as an index in the Array A. It then pulls the value at that index and add it to the last value of the sum in this case zero, before reassigning that value to sum itself.

**Maintenance:**

The loop is maintained by incrementing the value of i, which in turn activates the body of the loop, pulling the value from the array at the index i, adding it with the previous sum, then storing it in the sum variable.

**Termination:**

The loop terminates when i is equal to n(the length of the array). At the termination of the loop, all the items in the Array (A) must have been processed in the body of the loop. The body of the loop stores the sum of A[1: i] by evaluating the sum of A[1 : i - 1] (i.e the sum of previous elements) and the value of A[i] . At the end of the loop, sum = SUM(A[1:n]) The algorithm is correct.

**Q.3**

**SUM-ARRAY(A, n)**

**sum = 0**

**for i = 1 to n**

**sum = sum + A[i]**

**return sum**

**Rewrite the NSERTION-SORT procedure to sort into monotonically decreasing in-**

**stead of monotonically increasing order?**

ANS:

We just need to reverse the comparison of A[i] and key in line 5 as fallows ,

INSERTION-SORT(A)

1 for j = 2 to length[A]

2 key = A[j]

3 i = j - 1

4 while i > 0 and A[i] < key

5 A[i + 1] = A[i]

6 i = i - 1

7 A[i + 1] =

Q,4 **Consider the searching problem:**

**Input: A sequence of n numbers ( a1,a2,………….an) 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 pseudocode for linear search, which scans through the array from beginning to end, looking for x . Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properti**es.

ANS:

For linear search we just need to scan the array from the beginning till the end , index 1 to index n and check the entry at the equal position equal to V or not. The pseudocode can be written as..

LINEAR-SEARCH(A, n, x)

1. for i = 1 to n

2. if A[i] == x

3. return i

4. return NIL

**LOOP INVARIENT:**  
And here is how the three necessary properties hold for the loop invariant:  
****Initialization:**** Initially the subarray is empty. So, none of its’ elements are equal to v.  
****Maintenance:**** In i*i*-th iteration, we check whether A[i] is equal to v or not. If yes, we terminate the loop or we continue the iteration. So, if the subarray A[1..i−1] did not contain v before the i*i*-th iteration, the subarray A[1..i] will not contain v before the next iteration (unless i*i*-th iteration terminates the loop).  
****Termination:**** The loop terminates in either of the following cases,  
We have reached index i*i* such that v = A[i], or  
We reached the end of the array, i.e. we did not find v*v* in the array A*A*. So, we return NIL.  
In either case, our algorithm does exactly what was required, which means the algorithm is correct.  
  
  
**Q.5 Consider the problem of adding two n-bit binary integers, a and b, stored in two n-element arrays and , where each element is either 0 or 1. The integers a and b are represented as:**

**a = \sum\_{i=0}^{n-1} A[i] \cdot 2^i**

**b = \sum\_{i=0}^{n-1} B[i] \cdot 2^i ]**

**The sum of the two integers should be stored in binary form in an -element array , where:**

**c = \sum\_{i=0}^{n} C[i] \cdot 2^i**

**Write a procedure ADD\_BINARY\_INTEGERS that takes as input arrays and , along with the length , and returns the array , which holds the binary sum of the two integers.**

ANS:

The problem can be formally stated as…

****Input:**** Two n bit binary integers stored in two n*n* element array of binary digits (either 0 or 1) A=⟨a1,a2,...,an⟩ and B=⟨b1,b2,...,bn⟩.

****Output:**** A (n+1)  bit binary integer stored in (n+1) element array of binary digits (either 0 or 1) C=⟨c1,c2,...,cn+1)  such that C=A+B

We also assume the binary digits are stored with least significant bit first, i.e. from right to left, first bit in index 1, second bit in index 2, and so on. Why we are doing this is discussed after the pseudocode.

ADD-BINARY-INTEGERS(A, B, n)

1. Let C[0..n] be a new array of size n+1 (initialized to 0)

2. carry = 0

3. for i = 0 to n-1 do

4. sum = A[i] + B[i] + carry

5. C[i] = sum mod 2 // sum bit

6. carry = sum // 2 // carry bit

7. end for

8. C[n] = carry // final carry (most significant bit)

9. return C

**EXERCISE 2.2**

**Q.1Express the function n3 /1000 + 100n2\_  100n +3 in terms of theta notation.**

ANS:  
The highest order of n term of the function ignoring the constant coefficient is n3. So, the function in Θ-notation will be Θ(n3).  
  
  
**Q.2 Consider sorting n number stored in a array A by first finding the smallest element of A and exchanging it with the element in A[ 1] .Then find the second smallest element of A and exchange it with A[2] .Continue in this manner for the first n-1elements of A . Write psecudocode for this algorithm which is known as selection sort .what loop invarient does this algorithm maintain. Why does it need to run for only the first n-1elements , rather than all n elements ? Give the first best case and worst case running time of selection sort in 0(theta) notation .**  
ANS:  
Selection-Sort(A)  
 n = length(A)  
 for i = 1 to n - 1  
 min\_index = I  
 for j = i + 1 to n  
 if A[j] < A[min\_index]  
 min\_index = j  
 if min\_index ≠ I  
 swap A[i] with A[min\_index]  
  
  
**Loop Invariant**

The loop invariant maintained by this algorithm is:

* At the start of each iteration of the outer loop (for each index i), the subarray A[1] to A[i-1] contains the smallest i-1 elements of the array A, sorted in non-decreasing order.

### Reason for Running Only n-1 Elements

The algorithm runs for only the first n-1 elements because, after placing the smallest n-1 elements in their correct positions, the last remaining element (the nth element) must be the largest and will naturally be in its correct position. Therefore, there is no need to perform an exchange for the last element.

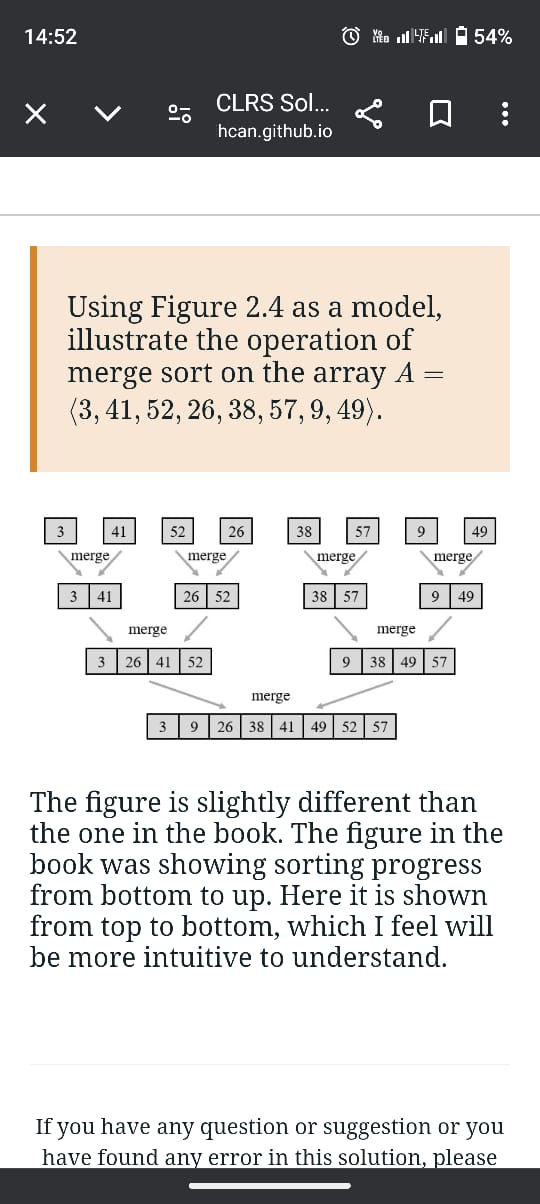
### Running Time of Selection Sort

* **Best Case:** Θ(n2)
* **Worst Case:** Θ(n2 )

In both cases, the selection sort algorithm performs n−1 iterations of finding the minimum element from the unsorted portion of the array, which involves scanning the remaining elements. Each scan takes linear time, resulting in a quadratic time complexity overall.

**Q.3 Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.**ANS:  
On average half the elements in the array A will be checked before v (represent key element) is found in it.And in the worst case ( v is not present in A ), all the elements needs to be checked.  
  
In either case , the running time will be proportional to n( Θ(n)).

**Q.4 How can you modify any sorting algorithm to have a good best-case running time?**ANS:  
We can design any algorithm to treat its best-case scenario as a special case and return a predetermined solution.  
  
For example, for selection sort, we can check whether the input array is already sorted and if it is, we can return without doing anything. We can check whether an array is sorted in linear time. So, selection sort can run with a best-case running time of Θ(n)

**EXERCISE 2.3**  
  
Q.1  **Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence (3, 41, 52, 26, 38, 57, 9, 49).  
ANS:**  
  
the figure is slightly different than the book .The figure in the book is showing the sorting progress bottom to up . here it is shown the progress from top to bottom, which I feel will be more easy to understand

**Q.2 The test in line 1 of the MERGE-SORT procedure reads if p > r rather than if p != r . If MERGE-SORT is called with p > r , then the subarray A[p: r] is empty.Argue that as long as the initial call of MERGE-SORT(.A, 1, n) has n >= 1, the test if p!= r suffices to ensure that no recursive call has p > r .**ANS:  
In the MERGE-SORT algorithm, the condition p ≠ r (which is equivalent to p < r in this context) is used to ensure that the recursion only continues when the subarray being sorted has more than one element. Let's break down why this condition suffices to prevent the situation where p > r (which would imply an invalid subarray):  
  
**. Base Case and Recursion Logic:**  
When MERGE-SORT(A, p, r) is called, it is intended to sort the subarray A[p..r].If p = r, the subarray has exactly one element, which is trivially sorted, and no further recursive calls are made. If p ≠ r, the subarray has more than one element, so the algorithm splits the subarray in half by computing the midpoint q = ⌊(p + r) / 2⌋ and recursively calling MERGE-SORT(A, p, q) and MERGE-SORT(A, q + 1, r).

Preventing p > r:  
For the recursion to continue, the subarray must be split into two valid subarrays. In the recursive calls, the range is split such that p ≤ q and q + 1 ≤ r. This guarantees that in the recursive calls, p ≤ q for the left half and q + 1 ≤ r for the right half.Since the split always maintains valid bounds (p ≤ q and q + 1 ≤ r), it prevents a situation where a recursive call would occur with p > r.  
  
. **Initial Call Assumption**:  
The problem assumes that MERGE-SORT(A, 1, n) is initially called with n ≥ 1. This means that as long as n ≥ 1, the subarray has at least one element, and no recursive call will be made with an empty or invalid subarray (p > r).In other words, if p = 1 and r = n in the initial call, and since n ≥ 1, the condition p ≤ r will hold throughout the recursion.  
  
Thus, the condition p ≠ r (or equivalently, p < r) is sufficient to ensure that no recursive call is made with p > r. This condition ensures that the recursion only occurs when the subarray contains more than one element, and it prevents invalid recursive calls.

**Q,3 State a loop invariant for the while loop of lines 12 -18 of the MERGE procedure.Show how to use it, along with the while loops of lines 20 - 23 and 24 - 27, to prove that the MERGE procedure is correct.**ANS:  
  
To analyze the MERGE procedure and prove its correctness using loop invariants, let's establish a loop invariant for the main while loop (lines 12-18) and show how it interacts with the other loops (lines 20-23 and 24-27).  
Loop Invariant for the Main While Loop (Lines 12-18)  
**Loop Invariant:**  
 At the start of each iteration of the while loop (lines 12-18), the elements in the merged array ( C[1:k] ) are sorted, and all elements in the input arrays ( A[p:q] ) and ( A[q+1:r] ) that have not yet been merged into ( C ) are still in their original order.  
  
Using the Loop Invariant  
**Initialization (Before the first iteration)**  
 Initially, no elements have been added to ( C ) (i.e., ( k = 1 )).  
 The merged array ( C ) is empty, which is trivially sorted.  
 The elements in ( A[p:q] ) and ( A[q+1:r] ) are still in their original sorted order.  
  
**Maintenance (During iterations of the while loop)** In each iteration, the loop compares the current elements of the two subarrays (from ( A[p:q] ) and ( A[q+1:r] )).  
 The smaller of the two elements is added to ( C ), and the index of the corresponding subarray is incremented.,. After each iteration, the elements in ( C ) remain sorted because:  
 The smallest element from the two subarrays is always appended.  
 The remaining elements in both subarrays remain in order.

**Termination (When the loop exits**)  
 The loop terminates when either of the subarrays has been completely processed.  
At this point, since the loop invariant holds and one subarray is exhausted, all remaining elements from the other subarray can simply be appended to ( C ).  
 This preserves the sorted order of ( C ) because the remaining elements are already sorted.  
  
 Additional While Loops (Lines 20-23 and 24-27)  
  
**While Loop (Lines 20-23):** This loop handles any remaining elements from the first subarray ( A[p:q] )  
 The loop invariant still holds because we are appending sorted elements to ( C ).  
 After this loop, all elements from ( A[p:q] ) are included in ( C ) and maintain their sorted order.  
  
**While Loop (Lines 24-27):**  
 This loop deals with any remaining elements from the second subarray ( A[q+1:r] s).  
 Again, the loop invariant continues to hold since we append these remaining elements to ( C ) while they are already sorted.  
 Once this loop completes, all elements from both subarrays are now in ( C ).  
  
 Conclusion  
By establishing and utilizing the loop invariant in the main while loop and ensuring it holds through the subsequent loops, we can conclude that:  
  
Initialization: The invariant holds before the first iteration.  
Maintenance:The invariant holds through each iteration of the main while loop.  
Termination: Upon exiting the loop, we can append any remaining elements while maintaining sorted order.  
  
Thus, the entire MERGE procedure correctly merges two sorted arrays into one sorted array, confirming its correctness.

**Q.4 Use mathematical induction to show that when n >= 2 is an exact power of 2, the**

solution of the recurrence

T(n)= { 2 if n=2

{2T(n/2)=n if n>2

Is T(n)= n lg n.

ANS:   
 **Base Case**

When n=2*n*, *T*(2)=2=2lg2. So, the solution holds for the initial step.  
  
**Inductive Step**

Let’s assume that there exists a k*k*, greater than 1, such that *T*(2*k*)=2*k*lg2*k* We must prove that the formula holds for *k*+1  too, i.e. *T*(2*k*+1)=2*k*+1lg2*k*+1.  
From our recurrence formula,  
*T*(2*k*+1)​=2*T*(2*k*+1/2)+2*k*+1  
=2*T*(2*k*)+2⋅2*k*=2.2*k*lg2*k*+2⋅2*k*=2⋅2*k*(lg2*k*+1)  
=2*k*+1(lg2*k*+lg2)  
=2*k*+1lg2*k*+1​  
  
This completes the inductive step.

Since both the base case and the inductive step have been performed, by mathematical induction, the statement *T*(*n*)=*n*lg*n* holds for all n that are exact power of 2.

**Q.5 We can express insertion sort as a recursive procedure as follows. In order to sort A[1..n] we recursively sort A[1..n−1] and then insert A[n] into the sorted array A[1..n−1]. Write a recurrence for the running time of this recursive version of insertion sort.**

**ANS:**

There are two steps in this recursive sorting algorithm:

1. Sort the sub-array A[1..n−1]
2. Insert A[n] into the sorted sub-array from step 1 in proper position

For n=1, step 1 doesn’t take any time as the sub-array is an empty array and step 2 takes constant time, i.e. the algorithm runs in Θ(1)) time.

For *n*>1, step 1 again calls for the recursion for n−1 and step 2 runs in Θ(n)) time.

So, we can write the recurrence as:

T(n)= {  Θ(1) if n=1

{T(n - 1) +Θ(1) if n>1

**Q.6**

**Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further.**

ANS:

Here is the psecudocode if you prefer iterative solution

BinarySearch(arr, target)

1. low ← 0

2. high ← length(arr) - 1

3. while low ≤ high

4. mid ← (low + high) / 2

5. if arr[mid] == target

6. return mid // Target found

7. else if arr[mid] < target

8. low ← mid + 1 // Target is in the right half

9. else

10. high ← mid - 1 // Target is in the left half

11. return -1 // Target not found

Recursive one…

BinarySearchRecursive(arr, target, low, high)

1. if low > high

2. return -1 // Target not found

3. mid ← (low + high) / 2

4. if arr[mid] == target

5. return mid // Target found

6. else if arr[mid] < target

7. return BinarySearchRecursive(arr, target, mid + 1, high) // Search in the right half

8. else

9. return BinarySearchRecursive(arr, target, low, mid - 1) // Search in the left half

Intuitively, in worst case, i.e. when v is not at all present in A, we need to search over the whole array to return NIL. In other words, we need to repeatedly divide the array by 2 and check either half for v. So the running time is nothing but how many times the input size can be divided by 2, i.e. nlg*n*.

For recursive case, we can write the recurrence as follows

T(n)= {  Θ(1) if n=1

{T(n - 1) +Θ(1) if n>1

**Q.7** **The while loop of lines 5-7 of the I NSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray A[1 : j - 1]. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to ‚Θ(.n lg n)?**

ANS:

Let’s take a look at the loop in question:

while i > 0 and A[i] < key

A[i + 1] = A[i]

i = i - 1

This loop serves two purposes:

1. A linear search to scan (backward) through the sorted sub-array to find the proper position for key.
2. Shift the elements that are greater than key towards the end to insert key in the proper position.

Although we can reduce the number of comparisons by using binary search to accomplish purpose 1, we still need to shift all the elements greater than key towards the end of the array to make space for key. And this shifting of elements runs at Θ(n) time, even in average case (as we need to shift half of the elements). So, the overall worst-case running time of insertion sort will still be Θ(n2)).

**Q.8 Describe an algorithm that, given a set S of n integers and another integer x , de-termines whether S contains two elements that sum to exactly x . Your algorithm should take ‚.n lg n/ time in the worst case.**

ANS:

If the running time constraint was not there, we might have intuitively used the brute-force method of picking one element at a time and iterating over the set to check if there exists another element in the set such that sum of them is x. Even in average case, this brute-force algorithm will run at Θ(*n*2) time (as we have to iterate over the set for each element).

But we have to think of a Θ(*n*lg*n*)-time algorithm.

This problem can be solved in another way which still uses a Θ(*n*lg*n*) sorting procedure but instead of using Binary-Search, it uses a **two-way search**, i.e. simultaneous search from both end of the array, to check if two elements sums up to expected sum, *x*.

function findPairWithSum(S, n, x):

Sort S using a sorting algorithm (O(n log n))

left = 0

right = n - 1

while left < right:

sum = S[left] + S[right]

if sum == x:

return (S[left], S[right]) // Pair found

else if sum < x:

left = left + 1 // Increase the sum

else:

right = right - 1 // Decrease the sum

return false // No pair found

 we can sort the array with merge sort Θ(*n*lg*n*) and then for each element S[i]in the array, we can do a binary search for x−S[i] on the sorted array Θ(*n*lg*n*)). So, the overall algorithm will run atΘ(*n*lg*n*) time.

**Problems**

## Q.1 Insertion sort on small arrays in merge sort

**Although merge sort runs in Θ(lg*n*) worst-case time and insertion sort runs in Θ(*n*2) worst case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to **coarsen** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.**

1. **Show that insertion sort can sort the n/k*n*/*k* sublists, each of length k*k*, in Θ(*nk*) worst-case time.**
2. **Show how to merge the sublists in Θ(*n*lg(*n*/*k*)) worst-case time.**
3. **Given that the modified algorithm runs in Θ(*nk*+*n*lg(*n*/*k*)) worst-case time, what is the largest value of k as a function of n*n* for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?**
4. **How should we choose k in practice?**

ANS:

### 1. Sorting sublists

This is simple enough. We know that sorting each list takes *ak2*+*bk*+*c* for some constants *a*, *b* and *c*. We have *n*/*k* of those, thus:

*K/n* (*ak2*+*bk*+*c*) = *ank*+*bn*+*cn/K* = Θ(*nk*)

### 2.Merging sublists

This is a bit trickier. Sorting a*a* sublists of length k each takes:

T(a)= {  0 if a=1

{2T (a / 2) +ak if a=2p , if p>1

This makes sense, since merging one sublist is trivial and merging a*a* sublists means splitting dividing them in two groups of *a*/2 lists, merging each group recursively and then combining the results in *ak* steps, since have two arrays, each of length *a/2.k*

I don't know the master theorem yet, but it seems to me that the recurrence is actually *ak* lg *a*. Let's try to prove this via induction:

****Base****. Simple as ever:

T(1)=1klg1=k . 0=0  
  
****Step****. We assume that *T*(*a*)=*ak*lg*a* and we calculate *T*(2*a*):

*T*(2*a*)​=2*T*(*a*)+2*ak*=2(*T*(*a*)+*ak*)  
=2(*ak*lg*a*+*ak*)  
=2*ak*(lg*a*+1)  
=2*ak*(lg*a*+lg2)  
=2*ak*lg(2*a*)​

This proves it. Now if we substitue the number of sublists n/k*n*/*k* for a*a*:

*T*(*n*/*k*)=*n/k .k* lg *n/k*=*n*lg(*n*/*k*)

While this is exact only when n/k*n*/*k* is a power of 2, it tells us that the overall time complexity of the merge is Θ(*n*lg(*n*/*k*)).

### **3. The largest value of k**

The largest value is *k*=lg*n*. If we substitute, we get:

Θ(*n*lg*n*+*n*lglg*nn*​)=Θ(*n*lg*n*)

### If *k*=*f*(*n*)>lg*n*, the complexity will be Θ(*nf*(*n*)), which is larger running time than merge sort. 4. The value of k in practice

It's constant factors, so we just figure out when insertion sort beats merge sort, exactly as we did in [exercise 1.2.2](https://ita.skanev.com/01/02/02.html), and pick that number for *k*.

**Q.2 Correctness of bubblesor**

**Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. Bubblesort ( A )**

**Bubblesort (A, n)**

**for i = 1 to n - l**

**for j = n . length down to i + 1**

**if A [ j ] < A [ j − 1 ]**

**exchange A [ j ] with A [ j − 1 ]**

1. **​ Let A ′ A ′ denote the output of Bubblesort(A) Bubblesort(A) . To prove that Bubblesort Bubblesort is correct, we need to prove that it terminates and that**

**A ′ [ 1 ] ≤ A ′ [ 2 ] ≤ ⋯ ≤ A ′ [ n ]. In order to show that Bubblesort Bubblesort actually sorts, what else do we need to prove? The next two parts will prove inequality**

1. **. State precisely a loop invariant for the for loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter. C), C).Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.**
2. **. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?**

ANS:

#### **A. Required Proof of Correctness**

We also need to prove that A′ consists of the elements of A but in sorted order.

#### **B. Loop Invariant for Inner Loop**

The loop invariant for the for loop in lines 2–4 can be stated as follows:

At the start of each iteration of the for loop, the subarray A[j…n] consists of the elements originally in A[j…n] before entering the loop but possibly in a different order and the first element is the smallest among them.

And here is how the three necessary properties hold for the loop invariant:

****Initialization:**** Initially the subarray contains only the last element A[n] and this is the smallest element of the subarray.

****Maintenance:**** In every step we compare A[j] with A[j−1] and make A[j−1] the smallest among them. So, after the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.

****Termination:**** The loop terminates when j=i+1. At that point also the length of the subarray increases by one and the first element is the smallest of the subarray as we swap A[i+1] with A[i].

#### **C. Loop Invariant for the Outer Loop**

The loop invariant for the for loop in lines 1–4 can be stated as follows:

At the start of each iteration of the for loop, the subarray A[1…i−1] consists of the elements that are smaller than the elements in the subarray A[i…n] in sorted order.

And here is how the three necessary properties hold for the loop invariant:

****Initialization:**** Initially the subarray A[1…i−1] is empty and trivially this is the smallest element of the subarray.

****Maintenance:**** From part **(b)**, after the execution of the inner loop, A[i] will be the smallest element of the subarray A[i…n]. And in the beginning of the outer loop, A[1…i−1]consists of elements that are smaller than the elements of A[i…n], in sorted order. So, after the execution of the outer loop, subarray A[1…i] will consists of elements that are smaller than the elements of A[i+1…n], in sorted order.

****Termination:**** The loop terminates when *i*=*A*.*length*. At that point the array *A*[1…*n*] will consists of all elements in sorted order.

#### **D. Running Time of Bubblesort**

In the worst-case (reverse sorted array), bubblesort will iterate over the whole array for each element, i.e. for each element bubble sort will perform n*n* comparisons and swaps. Therefore, worst-case running time of bubblesort is Θ(*n*2).

Although insertion also runs at Θ(n2) worst-case time, the number of assignments (swaps) performed in bubblesort is way more than that of insertion sort. So, the constant factors in the running time will be much larger for bubblesort compared to that of insertion sort. This means, for the same input size, insertion sort will run faster than bubblesort.

**Q.3 Correctness of Horner’s Rule**

**The following code fragment implements Horner’s rule for evaluating a polynomial**

**P ( x ) = ∑ k = 0 n a k x k = a 0 + x ( a 1 + x ( a 2 + ⋯ + x ( a n − 1 + x a n ) ⋯ ) )**

**given the coefficients a 0 , a 1 , … , a n and a value for x:**

**y = 0**

**for i = n downto 0**

**y = a i + x . y**

1. **In terms of Θ notation, what is the asymptotic running time of this code fragment for Horner’s rule?**
2. **Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner’s rule?**
3. **Consider the following loop invariant: At the start of each iteration of the for loop of lines 2-3, y = ∑ k = 0 n − ( i + 1 ) a k + i + 1 x k Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, y = ∑ k = 0 n a k x k**
4. **Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients a 0 , a 1 , ⋯ , a n a 0 ,a 1 ,⋯,a n**

ANS:

#### **A. Asymptotic Running Time**

From the pseudocode of Horner’s Rule, the algorithm runs in a loop for all the elements, i.e. it runs at Θ(n)Θ(*n*) time.

#### **B. Comparison with Naive Algorithm**

We can write the pseudocode as follows, where A is an array of

NEW-Horner (*A*,*x*)

1 y=0

2 for i=1 to A.length

3 m=1

4 for j=1 to i−1

5 m=m⋅x

6 y=y+A[i]⋅m

The above algorithm runs with a ****for**** loop of (n−1) elements (lines 4-5) inside another ****for**** loop (lines 2-6) of n elements. Therefore, the algorithm runs at Θ(n2) time.

This algorithm is obviously worse than Horner’s rule which runs at linear time.

#### **C. Loop Invariant Analysis**

****Initialization:**** At the start of the first iteration, there are no terms in the summation, so the sum is zero.

****Maintenance:**** From the loop invariant, for any arbitrary 0≤*i*<*n*, at the start of the i-th iteration of the ****For**** loop of lines 2-3, y=∑n−(i+1)k=0 ak+i+1xk

Now, after the i-th iteration, as we are iterating from n*n* ****downto**** 00, we will have*i*=*i*−1. So, to prove the maintenance of the loop invariant, we’ll need to show that after the i*i*-th iteration, we will have

 y=∑k=0n−((i−1)+1)ak+(i−1)+1xk

=∑k=0n−iak+ixk*y*

This can be shown as follows…

y=ai+x

∑k=0n−(i+1)ak+i+1xk

=aix0+ai+1x1+ai+2x2+⋯+anxn−i=∑k=0n−iak+ixk

****Termination:**** When the loop terminates, we have i=−1. So,

y=∑k=0n−(i+1)ak+i+1xk

=∑k=0n−(−1+1)ak−1+1xk

=∑k=0nakxk*y*​

This is precisely what we wanted to calculate.

#### **D. Correctness Argument**

When Horner’s rule terminates it successfully evaluates the polynomial as it intended to. This means the algorithm is correct.

**Q.4 Inversions**

**Let A[1…n]*A*[1…*n*] be an array of n*n* distinct numbers. If i<j and A[i]>A[j], then the pair (i,j) is called an inversion of A.**

1. **List the five inversions of the array ⟨2,3,8,6,1⟩.**
2. **What array with elements from the set {1,2,…,n} has the most inversions? How many does it have?**
3. **What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.**
4. **Give an algorithm that determines the number of inversions in any permutation on n*n* elements in )Θ(*n*lg*n*) worst-case time. (Hint: Modify merge sort.)**

ANS:

#### **A. List of Inversions**

Inversions in the given array are: (1, 5), (2, 5), (3, 4), (3, 5), and (4, 5). (Note: Inversions are specified by indices of the array, not by values.)

#### **B. Array With Most Inversions**

The array with elements from the set 1,2,…,n with the most inversions will have the elements in reverse sorted order, i.e. ⟨n,n−1,…,2,1⟩⟩.

As the array has n*n* unique elements in reverse sorted order, for every unique pair of (i,j), there will be an inversion. So, total number of inversion = number of ways to choose 2 distinct integer from the set 1,2,…,n = nC2​ = n(n−1)2​.

Another way to find number of inversions in such an array, is to notice that there will be *n*−1 inversions with the first index as the first element of the inversion pair,*n*−2 inversions with the second index as the first element of the inversion pair, and so on. And finally, zero inversions with the n*n*-th index as the first element of the inversion pair.

In other words, total number of inversions = (*n*−1)+(*n*−2)+⋯+1=*n*(*n*−1)/2​.

#### **C. Relationship With Insertion Sort**

If we take a look at the pseudocode for insertion sort with the definition of inversions in mind, we will realize that more the number of inversions in an array, the more times the inner ****while**** loop will run.

This is also in line with our findings in sub-problem **b**. Maximum number of inversions are possible when the array is reverse sorted.

So, the higher the number of inversions in an array, the longer insertion sort will take to sort the array.

#### **D. Algorithm to Calculate Inversions**

Although a hint to modify merge sort is already given, without that also we should think of divide-and-conquer algorithms whenever we see running time with lg*n* term.

As was done in merge sort, we need to recursively divide the array into halves and count number of inversions in the sub-arrays. This will result in lg*n* steps and Θ(*n*) operations in each step to count the inversions. All in all a Θ(*n*lg*n*) algorithm.

The problem did not specifically asked to write pseudocode, but we can do that as well for the sake of completion.

We can rewrite Merge-SortMerge-Sort as follows to repeatedly subdivide the array and count number of inversions in each half.

Count-Inversions (*A*,*p*,*r*)

1 if p≥ r

2 return 0

3 q=⌊(p+r)/2⌋

4 left=Count-Inversions(A,p,q)

5 right=Count-Inversions(A,q+1,r)

6 inversions=left+right+Merge(A,p,q,r)

7 return inversions

And here is the modified Merge-Sort pseudocode that actually counts the number of inversion in linear time.

Merge (A,p,p,r)

1 n1=q−p+1

2 n2=r−q

3 let L[1..n1] and R[1..n2] be  new  arrays

4 for i=1 to n1

5 L[i]=A[p+i−1]

6 for j=1 to n2

7 R[j]=A[q+j]

8 L[n1+1]=∞

9 R[n2+1]=∞

10 i=1

11 j=1

12 inversions=0

13 for k=p to r

14 if L[i]≤R[j]

15 A[k]=L[i]

16 i=i+1

17 else

18 inversions=inversions+(n1−i+1)

19 A[k]=R[j]

20 j=j+1

21 return inversions

**EXERCISE 3.1**

**Q.1 Modify the lower-bound argument for insertion sort to handle input sizes that are**

**not necessarily a multiple of 3.**

ANS;

To handle input sizes that aren’t a multiple of 3 in the lower-bound argument for insertion sort, we can adjust by grouping elements as closely as possible:

Divide the Input:

If isn’t a multiple of 3, divide it into groups of 3 wherever possible, leaving a remainder of 1 or 2 elements.

Calculate Comparisons:

The lower bound for insertion sort is still Ω(n2) because each element in the worst case requires up to n comparisons to find its correct position. The remainder (1 or 2 elements) doesn’t change this bound.

Thus, for any n , the lower bound remains Ω(n2).

**Q.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.**

ANS:

#### **Running Times**

For both the best-case (sorted array) and worst-case (reverse sorted array), the algorithm will anyway take one element at a time and compare it with all the other elements. In other words, each of the n elements will be compared with rest of the n−1 elements. So, the running times for both scenario will be Θ(n2).

1. Total comparisons:

(n - 1) + (n - 2) +……… + 1 = n(n-1) / 2 = O(n2)

2. Total swaps:

n - 1

3. Overall time complexity:

O(n2)

**Q.3 Suppose that ˛ is a fraction in the range 0 < α < 1. Show how to generalize the lower-boundargument for insertion sort to consider an input in which the ˛n largest values start in the FIrst α˛n positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the α n largest values must pass through each of the middle (1 - 2α)n array positions?**

ANS:

1. **Input Structure**:

Insertion sort handles an array where the largest values are in the first positions and the remaining

**(1 - α)n**  values are in the last positions.

1. **Pass-Through Count:**

Each of the **(1 - α)n**  smaller values must pass through the **α n**  largest values during insertion, leading to interactions.

1. **Additional Restriction:**

The value of **α** must satisfy **0 < α < 0.5**  for valid configurations.

1. **Maximizing Pass-Throughs:**

The value **α** of that maximizes the number of pass-throughs of the largest values through the middle **(1 - 2α)n** positions is:

**α = 0.5**

Summary

Restriction on **α**  : **0 < α < 0.5**

Maximized Value **α**: **α = 0.5**  maximizes interactions during sorting.