

***ASSIGNMENT 1:***

**Submitted by：Fatima Bibi:**

**Submitted To: sir Jamal Abdul ahad:**

**Roll No: 14646:**

**Course: Data structure:**

**Assignment: 01:**

**Submission Date: October 31, 2024**

***Chapter 01:***

***The Role of Algorithms in Computing***

***Question 1:***

Describe your own real-world example that requires sorting. Describe one that

requires ûnding the shortest distance between two points.

***Answer:***

***Sorting:***

**it** is the process of arranging items in a specific order based on certain criteria. In simple terms, it's like putting things in a sequence that makes sense, such as from smallest to largest, oldest to newest, or alphabetically.

### Example:

Think of sorting as cleaning up your desk. You have papers, books, and pens all scattered around. You might sort:

* Books by size (from smallest to largest)
* Papers by date (from newest to oldest)
* Pens by color (red, blue, black)

The goal of sorting is to make things easier to find, use, or organize.

### Why Sorting is Useful:

* **Finding things quickly**: If your clothes are sorted in your closet by color, you can easily find a red shirt when you need it.
* **Organizing data**: On your computer, files are often sorted by name or date so that you can quickly locate the one you need.
* **Making decisions**: For example, sorting prices from lowest to highest helps you choose the best deal when shopping online.

### Types of Sorting:

1. **Ascending order**: Sorting from smallest to largest or A to Z (like a list of numbers: 1, 2, 3… or words: apple, banana, cherry…).
2. **Descending order**: Sorting from largest to smallest or Z to A (like numbers: 100, 50, 10… or words: zebra, monkey, apple…).

### Real-world Example that Requires Sorting by Roll Number

To organize the student roster for a class, the teacher decides to sort the names **alphabetically** to make it easier to take attendance and manage records. The list of students includes Alice, Bob, Charlie, David, Emma, Frank, Grace, Hannah, Ian, and Jack. After sorting, the roster appears as follows: Alice, Bob, Charlie, David, Emma, Frank, Grace, Hannah, Ian, and Jack. This alphabetical arrangement allows the teacher to quickly locate a student's name during roll call, ensuring an efficient and organized process for managing the class.

### Real-world Example that Requires Finding the Shortest Distance:

**Scenario: Delivering Packages in a City**

Imagine you work for a local delivery service, and you have a list of packages that need to be delivered to different locations around the city. Your first stop is a café downtown, then a bookstore a few blocks away, followed by a bakery on the outskirts of town. To minimize driving time and fuel costs, you need to calculate the **shortest distance** between these delivery points. You look at a map and determine the best route: starting at the café, moving to the bookstore, and finally heading to the bakery. By finding this efficient route, you can ensure that the packages are delivered quickly, allowing you to serve more customers and save on expenses.

***Question 2:***

Other than speed, what other measures of efûciency might you need to consider in

a real-world setting?

* ***Answer:***
* **Resource Use**:
* This looks at how well resources like time, money, materials, and people are used. Efficient systems make the most out of these resources without wasting them.
* **Cost-effectiveness**: It’s important to check if the results are worth the money spent. An efficient process should provide good results without costing too much.
* **Quality of Work**: Efficiency isn’t just about how fast something is done; it’s also about how well it’s done. High efficiency should lead to high-quality results that meet expectations.
* **Flexibility**: This refers to how easily a process can change to meet new needs or challenges. An efficient system should adapt quickly when things change.
* **User Experience**: For businesses, making sure customers have a good experience is vital. Efficient systems should make it easy and pleasant for users or customers.
* **Teamwork and Communication**: In group settings, how well team members communicate and work together can greatly affect productivity. Efficient systems should encourage good communication among team members.
* **Quality of Output**: The effectiveness of a process is not just about how quickly it is done but also about the quality of the results. High efficiency should also lead to high-quality outcomes that meet or exceed standards

***Question 3:***

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

### *Data Structure:*

### *****Array:*****

*****Strengths:*****

* **Fast Access**: Arrays provide O(1) time complexity for accessing elements using their index. This means you can quickly retrieve any element if you know its position.
* **Simple Structure**: Arrays are easy to understand and implement. They have a straightforward linear structure that makes them intuitive to work with.
* **Memory Efficiency**: Arrays store elements in contiguous memory locations, which can lead to better cache performance. This means that accessing consecutive elements is generally faster.
* **Fixed Size**: Since arrays have a fixed size, they can be beneficial when you know the number of elements in advance, allowing for efficient memory allocation.

*****Limitations:*****

* **Fixed Size**: Once an array is created, its size cannot be changed. This limitation can lead to wasted space if the array is too large or insufficient space if the array is too small.
* **Insertion and Deletion**: Adding or removing elements from the middle of an array requires shifting elements, leading to O(n) time complexity. This makes dynamic data management less efficient compared to other data structures like linked lists.
* **Type Restrictions**: In many programming languages, arrays can only hold elements of the same type. This restriction can limit their flexibility in certain applications.
* **Memory Wastage**: If the array is larger than necessary to hold the elements, the unused space can lead to memory wastage, which is inefficient in resource-constrained environments.

***Question 4:***

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

How are they different?

***Answer:***

The **Shortest-Path Problem** and the **Traveling Salesperson Problem (TSP)** are both classic problems in graph theory and optimization, and while they share some similarities, they also have significant differences.

### *Similarities:*

**Graph-Based Representation**:

Both problems can be represented using graphs, where vertices (or nodes) represent locations (e.g., cities or points of interest) and edges represent the connections between these locations with associated weights (e.g., distances or costs).

**Optimization Objective**:

Both problems aim to find the most efficient route based on a defined criterion. The shortest-path problem seeks the minimum distance from one node to another, while the TSP seeks the shortest possible route that visits each node exactly once and returns to the starting point.

**Real-World Applications**:

Both problems have practical applications in fields like logistics, transportation, and network design. They are used for route planning, delivery scheduling, and optimizing travel paths.

### *Differences:*

* **Problem Definition**:

**Shortest-Path Problem**: The goal is to find the shortest path from a single source node to one or more target nodes. It may involve multiple starting and ending points, but each path is independent.

**Traveling Salesperson Problem (TSP)**: The goal is to find a single tour that visits each node exactly once and returns to the starting node. This means all nodes must be included in the solution, making it a more complex problem.

* ****Complexity**:**

**Shortest-Path Problem**: Typically solvable in polynomial time using algorithms like Dijkstra’s or Bellman-Ford, making it efficient even for large graphs.

* **Path Constraints**:

**Shortest-Path Problem**: There are no constraints on how many times nodes can be visited (except in specific variations). You can traverse nodes multiple times if it leads to a shorter path.

**Traveling Salesperson Problem (TSP)**: Each node must be visited exactly once (except for the starting point), which adds to the complexity of finding an optimal solution.

***Question 5:***

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:***

### Problem Requiring the Best Solution:

### ****Airline Flight Scheduling:****

**Scenario**: Airlines often face complex scheduling challenges, including coordinating multiple flights, crews, aircraft availability, and airport slots. In this context, even a small inefficiency can lead to significant delays, increased operational costs, and poor customer satisfaction.

* **Why Only the Best Solution Will Do**:
* Flight schedules need to be optimized to ensure on-time departures and arrivals, maximize aircraft utilization, and minimize operational costs. Any inefficiency can result in domino effects, causing delays for connecting flights, dissatisfied customers, and potential loss of revenue.
* For instance, if an airline has a flight scheduled that misses a connecting flight for many passengers, it can lead to a poor reputation and loss of future business. Therefore, finding the optimal schedule that accommodates all these factors is crucial

### Problem Allowing for an Approximate Solution:

### ****Route Planning for Delivery Trucks:****

**Scenario**: A logistics company needs to plan delivery routes for its trucks that serve multiple locations throughout a city. The objective is to deliver packages efficiently while considering factors such as traffic conditions and delivery windows.

**Why Approximately the Best Solution is Good Enough**:

* In this case, the company may have flexibility in the delivery times. While the goal is to minimize fuel costs and travel time, minor inefficiencies in the route may not significantly impact overall performance, especially if there are multiple deliveries scheduled throughout the day.
* For example, if a truck takes a slightly longer route due to traffic or road construction but still delivers packages within an acceptable time frame, this may be acceptable. The company may prioritize flexibility over strict optimization to accommodate unexpected delays or changing delivery requirements.

***Conclusion:***

In summary, airline flight scheduling is a scenario where only the best solution will suffice due to the critical nature of on-time performance and customer satisfaction. In contrast, route planning for delivery trucks allows for approximate solutions, as minor deviations from the optimal route may still meet the operational requirements without significant penalties.

***Question 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?

***Answer:***.

**All Input Available in Advance:**

Sometimes, a customer fills out a detailed support form online, listing all the issues or questions in advance. This form includes all the information necessary for the agent to resolve the issue without additional data coming in over time. For example, a customer submits a full report of a technical issue along with all relevant device details. The support agent can review all this information before even contacting the customer, allowing for a single, more efficient support call or email.

**Input Arriving Over Time:**

Other times, customers call in directly and explain their issue in real-time. In this scenario, additional information might emerge gradually through back-and-forth interaction. The agent might need to ask clarifying questions or wait for the customer to retrieve specific details about their device or problem as the call progresses. Here, the agent doesn’t have the full "input" from the start and must process the incoming information dynamically as it arrives.

***Question 7:***

Give an example of an application that requires algorithmic content at the applica

tion level, and discuss the function of the algorithms involved.

***Answer:***

A good example of an application that relies on algorithms at the application level is a **navigation app**, like Google Maps or Waze. These apps help users find the fastest route to a destination, avoid traffic, and locate points of interest, such as restaurants, gas stations, and hotels.

### Key Functions of the Algorithms Involved:

* **Route Planning Algorithm**:  
  This is one of the main algorithms that calculates the best route from the user’s starting point to their destination. It uses shortest path algorithms, like Dijkstra's or A\*, to find the quickest way to reach the destination based on factors like distance and time.
* **Traffic Prediction Algorithm**:  
  Navigation apps use data from various sources, including real-time user data, to detect and predict traffic. Machine learning algorithms analyze this data to identify patterns, such as peak traffic times, and can even suggest alternative routes to avoid traffic jams.
* **Location Search and Recommendation Algorithm**:  
  When users search for nearby places, like restaurants or gas stations, the app uses recommendation algorithms to suggest popular or highly-rated locations. It also uses spatial search algorithms to find places within a certain distance.

These algorithms work together to provide a seamless user experience, allowing users to get directions, avoid traffic, and find places easily.

Conversely, in other situations, the input is not fully available until the delivery process is already underway. For example, if a customer places an order for a delivery service during the day, or if unexpected traffic conditions arise (like accidents or road construction), the delivery service may need to adapt its routes on the fly. Drivers may receive new delivery requests while they’re already on the road, requiring real-time adjustments to their planned routes to accommodate new stops and avoid delays.

***Question 8:***

Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n2

steps and merge sort runs in 64 n lg n steps. For which values of n does insertion

sort beat merge sort?

***Answer:***

For insertion sort to beat merge sort for inputs of size n*n*, 8n28*n*2 must be less than 64nlg n64*n*lg*n*.

**8n2<64nlogn**

**⇒8n⋅n<8n⋅8lgn**

**⇒n<8lgn⇒**

**n/8<lgn**

**⇒2n/8<n**

This is not a purely polynomial equation in n*n*. To find the required range of values of n*n*, there are a few different methods we can use…

* Manually calculate the values of these expressions for different values of n*n*
* Plot these functions and find their intersections
* Write a piece of code to found the values.

#### **Calculation:**

It is obvious that insertion sort runs at quadratic time which is definitely worse than merge sort’s linearithmic time for very large values of n*n*. We know for n=1*n*=1, merge sort beats insertion sort. But for values greater than that, insertion sort beats merge sort. So, we will start checking from n=2*n*=2 and go up to see for what value of n*n* merge sort again starts to beat insertion sort.

Notice that for n<8*n*<8, 2n/82*n*/8 will be a fraction. So, let’s start with n=8*n*=8 and check for values of n*n* which are powers of 2.

n=8⇒28/8=2<n

n=16⇒216/8=4<n

n=32⇒232/8=16<n

n=64⇒264/8=256>n​

Note that we don’t need to continue anymore as we have found an approximate range of values for n*n* where merge sort starts to beat insertion sort; somewhere between 32 and 64. Let’s do what we were doing but now we are going to try middle value of either range, repeatedly (or in other words, binary search, if you have been reading ahead).​

n=48⇒248/8=64>n

n=40⇒240/8=32<n

n=44⇒244/8=44.8>n

n=42⇒242/8=38.4<n

n=43⇒243/8=42.4<n​

So, at n=44*n*=44, merge sort starts to beat insertion sort again. Therefore, for 2≤n≤432≤*n*≤43, insertion sort beats merge sort.

***Question 9:***

What is the smallest value of n such that an algorithm whose running time is 100n2

runs faster than an algorithm whose running time is 2 n on the same machine?

***Answer:***

For A to run faster than B, 100n2100*n*2 must be smaller than 2n2*n*.

#### **Calculation**

Intuitively we can realize that A (quadratic time complexity) will run much faster than B (exponential time complexity) for very large values of n*n*.

Let’s start checking from n=1*n*=1 and go up for values of n*n* which are power of 22 to see where that happens.

n=1⇒100×12=100>2n

n=2⇒100×22=400>2n

n=4⇒100×42=1600>2n

n=8⇒100×82=6400>2n

n=16⇒100×162=25600<2n​

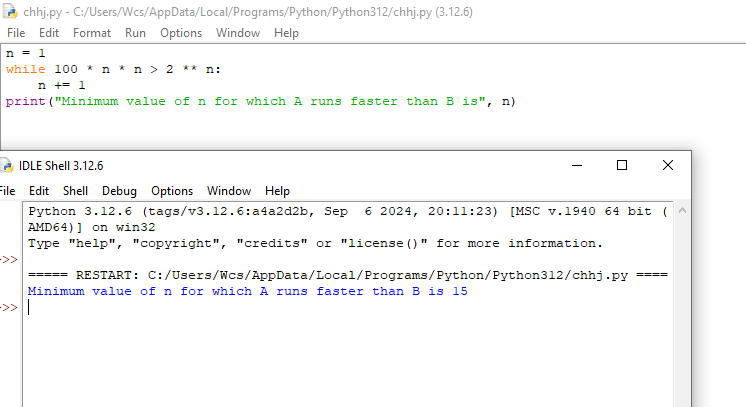
Somewhere between 8 and 16, A starts to run faster than B. Let’s do what we were doing but now we are going to try middle value of the range, repeatedly (binary search).

n=12⇒100×122=14400>2n

n=14⇒100×142=19600>2n

n=15⇒100×152=22500<2n

So, at n=15*n*=15, A starts to run faster than B.



***CHPTER 02:***

***Getting Started***

***Question 1:***

Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an

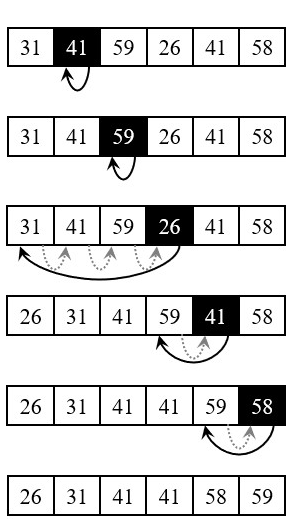
array initially containing the sequence (31; 41; 59; 26; 41; 58).

***Answer:***

1. Start with **31** and **41**: already sorted as [31,41][31, 41][31,41].
2. Insert **59**: remains as [31,41,59][31, 41, 59][31,41,59].
3. Insert **26**: shifts **31, 41, 59** right, result [26,31,41,59][26, 31, 41, 59][26,31,41,59].
4. Insert **41**: shifts **59** right, result [26,31,41,41,59][26, 31, 41, 41, 59][26,31,41,41,59].
5. Insert **58**: shifts **59** right, result [26,31,41,41,58,59][26, 31, 41, 41, 58, 59][26,31,41,41,58,59].

### Final Sorted Array

[26,31,41,41,58,59][26, 31, 41, 41, 58, 59][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 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]

.

Program:

SUM -ARRAY (A,n)

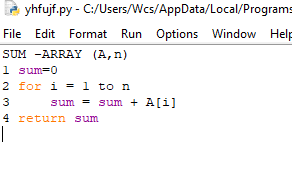
1 sum=0

2 for i = 1 to n

3 sum = sum + A[i]

4 return sum`?

***Answer:***



* ***Loop invariant:*** at each iteration of the loop, the variable sum contains the running sum of numbers, from A[0] to A[i-1]
* ***Initialization:*** before the first loop iteration,when i=1, A[i-1]=A[0] contains no elements,so the sum of no elements is 0.
* ***Maintenance:*** for each iteration of the loop, the value of A[i] is added to the variable sum, line 3.
* ***Termination:*** the loop terminates at i=n+1, and according to our invariant we have the sum of all elements from A[0] to A[n]

***Question 3:***

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

stead of monotonically increasing order?

***Answer:***

INSERTION\_SORT\_NONINCREASING(A)

for i = 2 to A.length

key = A[i]

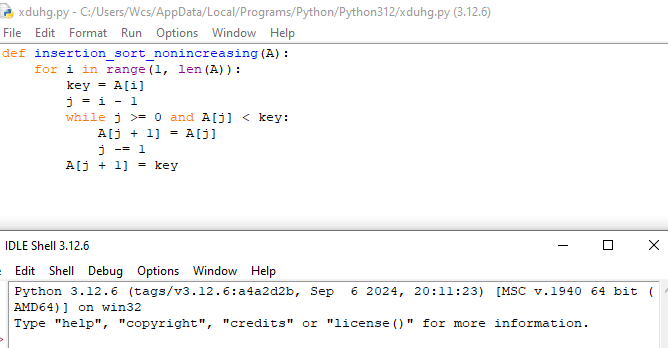
j = i - 1

while j >= 1 and A[j] < key

A[j + 1] = A[j]

j = j - 1

A[j + 1] = key



***Question 4***:

Consider the searching problem:

Input: A sequence of n numbers ha1; a2; : : : ; ani 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 begin

ning to end, looking for x. Using a loop invariant, prove that your algorithm is

correct. Make sure that your loop invariant fulûlls the three necessary properties?

***Answer:***

***Program:***

LINEAR\_SEARCH (A,x)

for i = 1 to A.length()

if A[i] == x

return i

return NIL

#### **Loop Invariant:**

At the start of the each iteration of the ****for**** loop of lines 1-3, the subarray A[1..i−1]*A*[1..*i*−1] does not contain the value v*v*.

****Initialization:****

 Initially the subarray is empty. So, none of its’ elements are equal to v*v*.

****Maintenance:****

In i*i*-th iteration, we check whether A[i]*A*[*i*] is equal to v*v* or not. If yes, we terminate the loop or we continue the iteration. So, if the subarray A[1..i−1]*A*[1..*i*−1] did not contain v*v* before the i*i*-th iteration, the subarray A[1..i]*A*[1..*i*] will not contain v*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*v* = A[i]*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.

***Question 5:***

Consider the problem of adding two n-bit binary integers a and b, stored in two

n-element arrays A[0:n-1] and B[0:n-1], where each element is either 0

or 1,

n

iD  0 1

AŒi�  2 i , and b D P

n

iD  0 1

BŒi�  2 i . The sum c D a C b of the

two integers should be stored in binary form in an .n C 1/-element array C Œ0 W n�,

where c D P

n

iD0

C Œi�  2 i . Write a procedure ADD-BINARY-INTEGERS that takes

as input arrays A and B, along with the length n, and returns array C holding the

sum.

***Answer:***

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

let C = new array of size n + 1

carry = 0

for i = 0 to n - 1

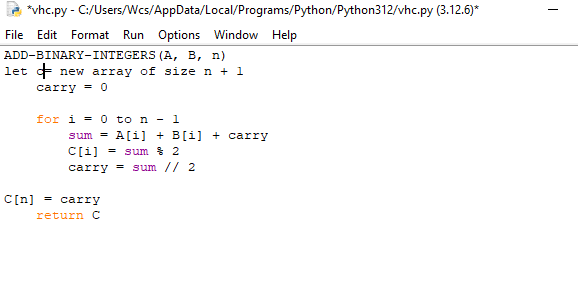
sum = A[i] + B[i] + carry

C[i] = sum % 2

carry = sum // 2

C[n] = carry

return C



***Question 6:***

Express the function n 3=1000 + 100n2 - 100n + 3 in terms of ‚Θ notation.

***Answer:***

The function f(n)=n31000+100n2−100n+3f(n) = \frac{n^3}{1000} + 100n^2 - 100n + 3f(n)=1000n3​+100n2−100n+3 contains terms with different growth rates:

* 1000n3​: This term grows as n3
* 100n2: This term grows as n2
* −100n: This term grows as n.
* +3: This is a constant term.

***Apply Theta Notation****:*

* We can ignore the lower-order terms 100n2100n^2100n2, −100n-100n−100n, and +3+3+3 because they grow slower than n3n^3n3 for large n.
* The constant coefficient 11000\frac{1}{1000}10001​ does not affect the asymptotic growth rate in Big Theta notation.

thus, we express the function in Big Theta notation as:

f(n)=Θ(n3)

**::**The function f(n)=n3/1000+100n2−100n+3 is Θ(n3)is the dominant term as n grows large.

***Question 7:***

Consider sorting n numbers stored in array A[1:n] by ûrst ûnding the smallest

element of A[1:n] and exchanging it with the element in A[1]. Then find the

smallest element of A[2:n], and exchange it with A[2]. Then find the smallest

element of A[3:n], and exchange it with A[3]. Continue in this manner for the

First( n -1) elements of A. Write pseudocode for this algorithm, which is known

as selection sort. What loop invariant does this algorithm maintain? Why does it

need to run for only the first (n1) elements, rather than for all n elements? Give the

worst-case running time of selection sort in ‚Θ notation. Is the best-case running

time any better?

SELECTION\_SORT(A)

for i = 1 to A.length - 1

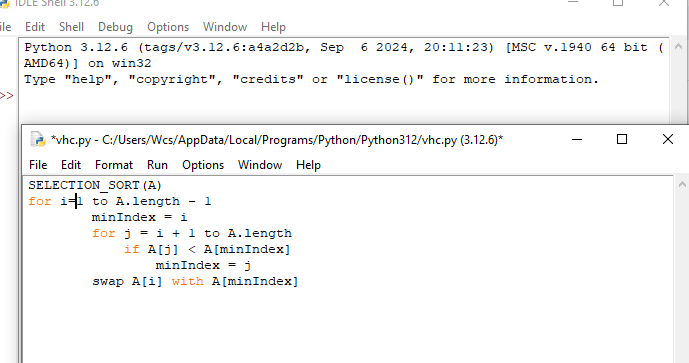
minIndex = i

for j = i + 1 to A.length

if A[j] < A[minIndex]

minIndex = j

swap A[i] with A[minIndex]



#### **Loop Invariant**

At the start of the each iteration of the outer ****for**** loop of lines 1-6, the subarray A[1..i−1]*A*[1..*i*−1] consists of i−1*i*−1 smallest elements of A*A*, sorted in increasing order.

#### **Why only first n - 1 elements**

The algorithm needs to run for only the first n−1*n*−1 elements, rather than for all n*n* elements because the last iteration will compare A[n]*A*[*n*] with the minimum element in A[1..n−1]*A*[1..*n*−1] in line 4 and swap them if necessary. So, there is no need to continue the algorithm for all the way to the last element.

#### **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*n* elements will be compared with rest of the n−1*n*−1 elements. So, the running times for both scenario will be same complaxity Θ(n2)Θ(*n*2).

The above reasoning should be sufficient to understand or convey why the runtime would be Θ(n2)Θ(*n*2). However, for the sake of completeness, an exhaustive mathematical proof is given below.

#### **Runtime Analysis**

Let’s assume the inner for loop in line 3-5 is executed for tj*tj*​ times for j=2,3,…,n*j*=2,3,…,*n*, where n=A.length*n*=*A*.*length*. Now note that, line 5 will be executed less than tj−1*tj*​−1 times in the average case, but it’ll still be of the order of n*n*.

For the sake of simplicity let’s assume the worst case, i.e. a reverse sorted array, when it’ll be executed exactly tj−1*tj*​−1 times. Note, this assumption is only for that particular line, which is not going to change our overall analysis, it will only make our calculation easier.

We can now calculate the cost and times for individual lines of the pseudocode as follows …

| **Line** | **Cost** | **Times** |
| --- | --- | --- |
| 1 | c1*c*1​ | n*n* |
| 2 | c2*c*2​ | n−1*n*−1 |
| 3 | c3*c*3​ | ∑j=2ntj∑*j*=2*n*​*tj*​ |
| 4 | c4*c*4​ | ∑j=2n(tj−1)∑*j*=2*n*​(*tj*​−1) |
| 5 | c5*c*5​ | ∑j=2n(tj−1)∑*j*=2*n*​(*tj*​−1) |
| 6 | c6*c*6​ | n−1*n*−1 |

Now, for any arbitrary value of j*j*, the inner for loop (line 3-5) compares the previously computed minimum value with all elements in the subarray A[j..n]*A*[*j*..*n*]. So the inner for loop executes n−j+1*n*−*j*+1 times, i.e. tj=(n−j+1)*tj*​=(*n*−*j*+1) for j=2,3,…,n*j*=2,3,…,*n*. So, t2=n−1,t3=n−2,…tn=1*t*2​=*n*−1,*t*3​=*n*−2,…*tn*​=1. We can calculate the summations for line 3-5 as follows …

∑j=2ntj=(n−1)+(n−2)+⋯+1=n(n−1)2∑j=2n(tj−1)=∑j=2ntj−∑j=2n1=n(n−1)2−(n−1)=(n−3)(n−2)2*j*=2∑*n*​*tj*​*j*=2∑*n*​(*tj*​−1)​=(*n*−1)+(*n*−2)+⋯+1=2*n*(*n*−1)​=*j*=2∑*n*​*tj*​−*j*=2∑*n*​1=2*n*(*n*−1)​−(*n*−1)=2(*n*−3)(*n*−2)​​

***Question 8:***

Consider linear search again (see Exercise 2.1-4). How many elements of the input

array 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?

Using ‚-notation, give the average-case and worst-case running times of linear

search. Justify your answers.

***Answer:***

### 1. Average-Case Analysis

Assuming the element being searched for is equally likely to be any element in the array, we can derive the average number of comparisons as follows:

* **Array Size:** Let nnn be the number of elements in the array.
* **Possible Outcomes:** The target can be found in any of the nnn positions (or not found at all).
* **Average Comparisons:**
  + If the target is at position 1, it takes 1 comparison.
  + If the target is at position 2, it takes 2 comparisons.
  + ...
  + If the target is at position nnn, it takes nnn comparisons.

To calculate the average, we sum the number of comparisons for all possible positions of the target and divide by nnn:

Average Comparisons=1+2+3+…+nn=n(n+1)2n=n+12\text{Average Comparisons} = \frac{1 + 2 + 3 + \ldots + n}{n} = \frac{\frac{n(n + 1)}{2}}{n} = \frac{n + 1}{2}Average Comparisons=n1+2+3+…+n​=n2n(n+1)​​=2n+1​

So, on average, we need to check approximately n+12\frac{n + 1}{2}2n+1​ elements.

### 2. Worst-Case Analysis

In the worst-case scenario for linear search:

* The target element is either the last element in the array or is not present at all.
* This means we have to check all nnn elements.

Thus, in the worst case, we need to check nnn elements.

### 3. Running Times Using Onotation

Using :O-notation, we can summarize the running times as follows:

* **Average Case:** The average-case running time is O(n)O(n)O(n). This is because, on average, we check about n2\frac{n}{2}2n​ elements, which scales linearly with the size of the array.
* **Worst Case:** The worst-case running time is also O(n)O(n)O(n) since we need to check all nnn elements in the worst case.

### Justification

* In both cases, the linear search involves checking elements one by one, leading to a linear relationship between the size of the input n and the number of checks performed.
* Therefore, both average-case and worst-case scenarios yield a linear time complexity of O(n)O(n)O(n).

In conclusion, for a linear search algorithm, the average-case and worst-case running times are both O(n).

***Question 9:***

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

***Answer:***

· **Hybrid Approaches**: Use Insertion Sort for small or nearly sorted subarrays.

· **Flags in Bubble Sort**: Detect no swaps to exit early, achieving O(n)O(n)O(n) time.

· **Optimized Merge Sort**: Merge already sorted segments efficiently.

· **Randomized Quick Sort**: Use random pivots for better average performance.

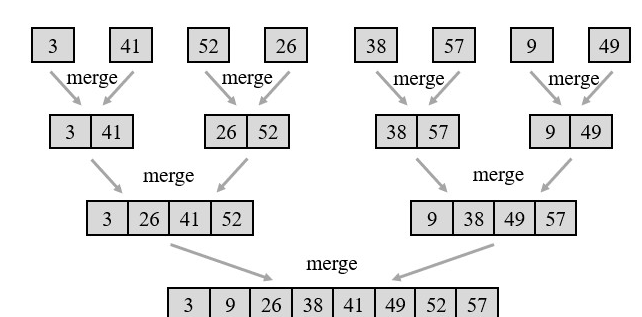
· **Timsort**: Leverage existing order in data for efficiency.

· **Non-comparison Sorts**: Use Counting or Radix Sort for limited ranges, achieving lin

***Question 10:***

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}



### Step 1: Initial Array

[3,41,52,26,38,57,9,49][3, 41, 52, 26, 38, 57, 9, 49][3,41,52,26,38,57,9,49]

### Step 2: Divide

1. Split into two halves: [3,41,52,26]and[38,57,9,49][3, 41, 52, 26] \quad \text{and} \quad [38, 57, 9, 49][3,41,52,26]and[38,57,9,49]
2. Keep splitting until single elements: [3],[41],[52],[26],[38],[57],[9],[49][3], [41], [52], [26], [38], [57], [9], [49][3],[41],[52],[26],[38],[57],[9],[49]

### Step 3: Merge

1. Merge pairs: [3,41],[26,52],[38,57],[9,49][3, 41], [26, 52], [38, 57], [9, 49][3,41],[26,52],[38,57],[9,49]
2. Merge into sorted arrays: [3,26,41,52]and[9,38,49,57][3, 26, 41, 52] \quad \text{and} \quad [9, 38, 49, 57][3,26,41,52]and[9,38,49,57]
3. Final merge: [3,9,26,38,41,49,52,57][3, 9, 26, 38, 41, 49, 52, 57][3,9,26,38,41,49,52,57]

### Final Sorted Array

[3,9,26,38,41,49,52,57][3, 9, 26, 38, 41, 49, 52, 57][3,9,26,38,41,49,52,57]

***Question 11:***

The test in line 1 of the MERGE-SORT procedure reads <if p  r= rather than < 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 = rsuffices to ensure that no recursive call has p > r.

***Answer:***

***inductive case***:

ifp≤rifp<rq=⌊(p+r)/2⌋q≥pq+1≤rMERGE−SORT(A,p,q)MERGE−SORT(A,q+1,r)else returnno recursive call has p>rp≤r holds in new recursive call*ifp*≤*rifp*<*rq*=⌊(*p*+*r*)/2⌋*q*≥*pq*+1≤*rMERGE*−*SORT*(*A*,*p*,*q*)*MERGE*−*SORT*(*A*,*q*+1,*r*)else returnno recursive call has p>r*p*≤*r* holds in new recursive call​

**base case:**

MERGE−SORT(A,1,n)n≥1p≤rholds*MERGE*−*SORT*(*A*,1,*n*)*n*≥1*p*≤*rholds*​

"if p≠r*p*=*r*" suffices to ensure that no recursive call has p>r*p*>*r*.

***Question 12:***

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.

***Answer:***

### Loop Invariant for the MERGE Procedure (Lines 12-18)

The **MERGE** procedure is used to combine two sorted subarrays into a single sorted array. Here’s a loop invariant for the primary while loop in lines 12-18:

**Loop Invariant:**  
At the beginning of each iteration of the while loop (lines 12-18):

1. The elements in A[p..k−1]A[p..k-1]A[p..k−1] are in sorted order.
2. These elements in A[p..k−1]A[p..k-1]A[p..k−1] contain the smallest elements from the two original sorted subarrays LLL (left subarray) and RRR (right subarray).

### Proof of Correctness Using Loop Invariants

#### 1. Initialization

* **Before the first iteration**:
  + k=pk = pk=p, i=1i = 1i=1, and j=1j = 1j=1.
  + The subarrays LLL and RRR are sorted, and no elements have been added to AAA yet.
  + The invariant holds because A[p..k−1]A[p..k-1]A[p..k−1] is initially empty.

#### 2. Maintenance

* **During each iteration of the loop (lines 12-18)**:
  + Compare L[i]L[i]L[i] and R[j]R[j]R[j]:
    - If L[i]≤R[j]L[i] \leq R[j]L[i]≤R[j], copy L[i]L[i]L[i] to A[k]A[k]A[k] and increment iii.
    - If R[j]<L[i]R[j] < L[i]R[j]<L[i], copy R[j]R[j]R[j] to A[k]A[k]A[k] and increment jjj.
  + kkk is then incremented to extend the sorted portion of A[p..k]A[p..k]A[p..k].
* After each iteration, the smallest element from LLL or RRR is added to A[p..k]A[p..k]A[p..k], maintaining sorted order. Thus, the loop invariant continues to hold.

#### 3. Termination

* **When the while loop (lines 12-18) terminates**:
  + Either i>n1i > n\_1i>n1​ (all elements in LLL have been copied) or j>n2j > n\_2j>n2​ (all elements in RRR have been copied).
  + The elements already in A[p..k−1]A[p..k-1]A[p..k−1] are in sorted order.

#### 4. Remaining Loops (Lines 20-23 and 24-27)

* After the main loop ends, one of the subarrays LLL or RRR may have remaining elements:
  + **Lines 20-23**: If there are remaining elements in LLL, they are copied to AAA directly. Since LLL is sorted, these elements will also be in sorted order.
  + **Lines 24-27**: If there are remaining elements in RRR, they are copied to AAA directly, maintaining sorted order.

### Conclusion

Using the loop invariant for each section, we ensure that:

1. All elements in A[p..r]A[p..r]A[p..r] are sorted after merging.
2. The **MERGE** procedure correctly combines two sorted subarrays into a single sorted array.

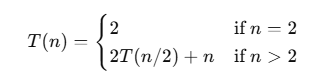
***Question 13***:

Use mathematical induction to show that when n  2 is an exact power of 2, the

solution of the recurrence

T .n/ D ( 2 2T .n=2/ C n if if n n > 2 D 2 ;

is T .n/ D n lg n.



### Base Case:

Let’s check the base case where For n=2

T(2)=2.

Now, we compute the right-hand side:

T(2)=2log2=2.1=2

Thus, the base case holds true, as T(2)=2

### Inductive Step:

Assume the statement is true for all powers of 2 less than or equal to n, where n=2k for some integer k. That is, assume:

T(n)=nlog2n

for n=2k holds true. Now, we need to show that it holds for n=2k+1.

#### Inductive Step

Now, consider n=2k+1n = 2^{k+1}n=2k+1:

T(2k+1)=2T(2k)+2k+1

By the inductive hypothesis, we substitute T(2k):

T(2k+1)=2(2k⋅k)+2k+1T(2^{k+1})

Simplifying this expression:

T(2k+1)=2k+1k+2k+1

T(2k+1)=2k+1(k+1)

Now, we want to show that this matches the formula for n=2k+1:

T(2k+1)=2k+1log2​(2k+1)=2k+1(k+1)

### Conclusion

Thus, we have shown that if T(n)=nlog2​n holds for n=2kthen it also holds for n=2k+1n

Since both the base case and the inductive step have been verified, by the principle of mathematical induction, we conclude that for all nnn which are exact powers of 2, the solution to the recurrence is:

T(n)=nlog2n

***Question 14:***

You can also think of insertion sort as a recursive algorithm. In order to sort

A[1:n], recursively sort the subarray A[1: n-1] and then insert A[n] into the

sorted subarray A[1:n-1]. Write pseudocode for this recursive version of inser

tion sort. Give a recurrence for its worst-case running time.

***Answer:***

To write a recursive version of **Insertion Sort**, we can define it to sort an array A[1…n]A[1 \dots n]A[1…n] by recursively sorting the subarray A[1…n−1]A[1 \dots n-1]A[1…n−1], then inserting the last element A[n]A[n]A[n] into its correct position within the sorted subarray.

### Recursive Insertion Sort Pseudocode

plaintext

Copy code

RECURSIVE-INSERTION-SORT(A, n)

1. if n > 1

2. RECURSIVE-INSERTION-SORT(A, n - 1)

3. INSERT(A, n)

INSERT(A, n)

1. key = A[n]

2. i = n - 1

3. while i > 0 and A[i] > key

4. A[i + 1] = A[i]

5. i = i - 1

6. A[i + 1] = key

### Explanation of the Pseudocode

**RECURSIVE-INSERTION-SORT(A, n)**:

* + If n>1n > 1n>1, the function recursively sorts the subarray A[1…n−1]A[1 \dots n-1]A[1…n−1].
  + Then, it calls the **INSERT(A, n)** function to insert A[n]A[n]A[n] into the correct position within the sorted subarray A[1…n−1]A[1 \dots n-1]A[1…n−1].

**INSERT(A, n)**:

* + This function places A[n]A[n]A[n] in its correct position in the sorted subarray A[1…n−1]A[1 \dots n-1]A[1…n−1].

### Recurrence for the Worst-Case Running Time

Let T(n)T(n)T(n) denote the worst-case running time of **RECURSIVE-INSERTION-SORT** on an array of size nnn.

1. **Recursive Call**: Sorting A[1…n−1]A[1 \dots n-1]A[1…n−1] takes T(n−1)T(n-1)T(n−1) time.
2. **Insertion**: The worst-case time to insert A[n]A[n]A[n] into the sorted subarray is O(n)O(n)O(n), as it may need to shift up to n−1n-1n−1 elements.

Thus, the recurrence relation for the worst-case running time is:

T(n)=T(n−1)+O(n)T(n) = T(n - 1) + O(n)T(n)=T(n−1)+O(n)

With base case T(1)=O(1)T(1) = O(1)T(1)=O(1).

This recurrence solves to T(n)=O(n2)T(n) = O(n^2)T(n)=O(n2), which matches the worst-case time complexity of the iterative version of insertion sort.

***Question 15:***

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 Problems for Chapter 2 45

consideration. The binary search algorithm repeats this procedure, halving the

size of the remaining portion of the subarray each time. Write pseudocode, either

iterative or recursive, for binary search. Argue that the worst-case running time of

binary search is ‚.lg n/. ?

***Answer:***

Iterative-Binary-Search (*A*,*v*)

*low*=*A*[1]

*high*=*A*[*A*.*length*]

**while***low*≤*high*

*mid*=⌊(*low*+*high*)/2⌋

**if***v*==*A*[*mid*]

**return***mid*

**elseif***v*>*A*[*mid*]

Low=mid+1

Else

High =mid-1

Return Nill

Recursive-Binary-Search (*A*,*v*,*low*,*high*)

**if***low*>*high*

**return** NIL

*mid*=⌊(*low*+*high*)/2⌋

**if***v*==*A*[*mid*]

**return***mid*

*Elseifu >A[mid]*

Recursive-Binary-Search (*A*,*v*,mid+1,*high*)

Else

Recursive-Binary-Search (*A*,*v*,*low*,mid-1)

​

***Question 16:***

The while loop of lines 537 of the INSERTION-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 ‚.n lg n/?

### Analysis of Insertion Sort with Binary Search

In the **Insertion Sort** algorithm, for each element A[j]A[j]A[j] (where j=2,3,…,nj = 2, 3, \dots, nj=2,3,…,n), we search for the correct position in the sorted portion A[1…j−1]A[1 \dots j-1]A[1…j−1] to insert A[j]A[j]A[j]:

1. **Binary Search for Position**: Using binary search to find the correct position for A[j]A[j]A[j] within A[1…j−1]A[1 \dots j-1]A[1…j−1] would take O(log⁡j)O(\log j)O(logj) time.
2. **Shifting Elements**: After finding the position, we still need to shift the elements in A[1…j−1]A[1 \dots j-1]A[1…j−1] to make space for A[j]A[j]A[j]. This shifting step requires O(j)O(j)O(j) operations in the worst case.

### Total Running Time

Let’s calculate the worst-case running time for the modified **Insertion Sort** with binary search:

* For each jjj, binary search takes O(logj)O(\log j)O(logj) time to find the position.
* Shifting elements takes O(j) time.

This is because O(j)) dominates O(logj), leading to:

T(n)=O(∑j=2nj)=O(n2)

### Conclusion

Even with binary search, the overall worst-case running time remains O(n2)due to the O(j) shifting cost. Therefore, using binary search does not reduce the worst-case time complexity of insertion sort to Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn).

***Question 17:***

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.

### Algorithm Steps

**Sort the Array**:

* 1. Sort S in ascending order. This takes Θ(nlogn) time.

**Use Two Pointers**:

* 1. Initialize two pointers:
     1. left at the start (index 0).
     2. right at the end (index n−1n ).
  2. While left < right, do the following:
     1. Calculate the sum: sum=S[left]+S[right]\text{sum} = S[\text{left}] + S[\text{right}]sum=S[left]+S[right]
     2. If sum=x\text{sum} = xsum=x, return **true**.
     3. If sum<x\text{sum} < xsum<x, increment left.
     4. If sum>x\text{sum} > xsum>x, decrements right.
  3. If no pair is found, return **false**.

### Pseudo code

plain text

Copy code

TWO-SUM(S, x)

1. Sort S

2. left = 0

3. right = n - 1

4. while left < right

5. sum = S[left] + S[right]

6. if sum == x

7. return true

8. else if sum < x

9. left = left + 1

10. else

11. right = right - 1

12. return false

### Time Complexity

* Sorting: Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn)
* Two-pointer search: O(n)O(n)O(n)

**Overall**: Θ(nlogn)time complexity.

he total worst-case time complexity of the algorithm is:

Θ(nlog⁡n)+O(n)=Θ(nlog⁡n)

***Chapter 3***

***Characterizing Running Times***

***Question 1:***

Modify the lower-bound argument for insertion sort to handle input sizes that are

not necessarily a multiple of 3.

***Answer:***

**Worst-Case Scenario**:

* 1. Insertion Sort performs worst when the input is in reverse order, requiring each new element to be compared with all previously sorted elements.

**Comparison Counting**:

* 1. For an input size n, the total number of comparisons in the worst case is: C(n)=0+1+2+…+(n−1)=(n−1)n2≈n22C(n) = 0 + 1 + 2 + \ldots + (n-1) = \frac{(n-1)n}{2} \approx \frac{n^2}{2}C(n)=0+1+2+…+(n−1)=2(n−1)n​≈2n2​

**Lower Bound**:

* 1. This implies that the worst-case time complexity of Insertion Sort is Θ(n2), applicable to any n, regardless of whether it’s a multiple of 3.

### Conclusion

Thus, the argument shows that Insertion Sort has a lower bound of Ω(n2) for any input size nnn, confirming its inefficiency for larger inputs.

***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:***

### election Sort Running Time Analysis

**Selection Sort** repeatedly selects the smallest element from the unsorted portion of the array and places it at the beginning. Here's a simple analysis of its running time:

**Outer Loop**:

* 1. The algorithm runs n times (where n is the number of elements).

**Inner Loop**:

* 1. For each pass, it searches through the unsorted elements to find the minimum:
     1. In the first pass, it checks n elements.
     2. In the second pass, it checks n−1n - 1n−1 elements.
     3. This continues until the last element.

**Total Comparisons**:

* 1. The total number of comparisons is: C(n)=(n−1)+(n−2)+…+1+0≈n22C(n) = (n - 1) + (n - 2) + \ldots + 1 + 0 \approx \frac{n^2}{2}C(n)=(n−1)+(n−2)+…+1+0≈2n2​

**Time Complexity**:

* 1. Therefore, the running time of Selection Sort is Θ(n2) for all cases (best, average, worst).

### Conclusion

Selection Sort is not efficient for large arrays, as it takes about n2n^2n2 comparisons regardless of the input order.

the lower-bound argument for insertion sort to consider an input in which the ˛n

largest values start in the ûrst ˛n positions. What additional restriction do you

need to put on ˛? What value of ˛ maximizes the number of times that the ˛n

largest values must pass through each of the middle .1  2˛/n array positions?

### Generalization of the Lower-Bound Argument

**Understanding the Input Structure**:

* 1. Assume an array A of size nnn.
  2. The largest αn\alpha nαn values are in the first αn\alpha nαn positions, and the smallest (1−α)nvalues are in the last (1−α)npositions.

**Insertion Sort Process**:

* 1. Insertion Sort will iterate through the array, and for each of the smallest (1−α)n(1 - \alpha)n(1−α)n elements, it will need to compare and potentially move it past the αn\alpha nαn largest elements.
  2. Each of the smallest elements may have to move past all αn\alpha nαn largest values before being placed in the sorted portion of the array.

**Counting Comparisons**:

* 1. For each of the (1−α)n smallest elements, it may need to be compared to each of the αn\alpha nαn largest elements, leading to a number of comparisons equal to: Total Comparisons=(1−α)n×αn=α(1−α)n2\text{Total Comparisons} = (1 - \alpha)n \times \alpha n = \alpha(1 - \alpha)n^2Total Comparisons=(1−α)n×αn=α(1−α)n2

**Lower Bound:**

* 1. This means that the number of comparisons in this case is at least Ω(n2), confirming that Insertion Sort's lower bound still holds even with this specific arrangement of input.

### Additional Restriction on α\alphaα

* For this argument to be valid, we need 0<α<0.50 < \alpha < 0.50<α<0.5. This ensures that the number of smallest elements is greater than the number of largest elements, leading to significant interactions during sorting.

### Maximizing the Number of Passes

* The number of times the αn\alpha nαn largest values must pass through the middle (1−2α)n(1 - 2\alpha)n(1−2α)n positions is maximized when α\alphaα is set to 0.50.50.5 because:
  + This balance allows for the largest values to interact with the maximum number of smaller elements (half of the array).

### Conclusion

In summary:

* The lower bound for Insertion Sort remains Ω(n2)\Omega(n^2)Ω(n2) when the largest αn\alpha nαn values start in the first αn\alpha nαn positions, provided 0<α<0.50 < \alpha < 0.50<α<0.5.
* Setting α\alphaα to 0.50.50.5 maximizes the number of passes through the middle (1−2α)n(1 - 2\alpha)n(1−2α)n positions, leading to the most significant interaction in the sorting process.