ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ASSINGMENT 1

CLASS: BSCS-3A

INSTRUCTOR: SIR JAMAL ABDUL AHAD

SUBJECT: DATA STRUCTURE AND ALGORITHM

SUBMITTED BY :KHADIJA ALI

ROLL NO :14653

Q1: Solve the questions from exercises from chapter 1, 2 and 3.

CHAPTER #1:

EXERCISE 1.1

Q1.Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

ANSWER:

Step 1

As we know, the sorting technique is used in many fields or in society. If we should describe one that requires determining the shortest distance between two points, it can be maps applications. When we open the map application and write the destination address, it will show us some possible vehicles to reach the destination. They can be these options:

- On foot
- By bus or train
- By car

Step 2

If we choose by bus or train option it will show the shortest distance to the destination address according to the bus and train routes. It gives priority to the routes according to the time spent on travel, not the shortest distance.

Step 3

But if we choose on foot or by car options, it shows the shortest distance according to the way on the map. The process of choosing the best route happens in this way:

Map application starts to track the route to the destination address. If the road is divided into multiple ways, the application will track all of them.

These divided roads can be also divided in some ways. This application tracks all these possibilities till it reaches the destination address.

After reaching the destination, this map application compares all these possibilities and chooses the best one according to the shortest distance.

Q2. Other than speed, what other measures of efficiency might you need to consider in a real-world setting.

ANSWER:

In real-world situations, efficiency isn't just about speed. Here are some other ways to think about it:

Cost Efficiency: Sometimes, saving money is more important than being the fastest. For example, when shipping products, a slower, cheaper option might be better for the budget than a costly express service.

Resource Efficiency: Making the most out of limited resources—like materials, energy, or manpower—is essential. In a factory, for instance, using less energy per product or reducing waste can improve overall efficiency.

Energy Efficiency: In energy-intensive settings like data centers or industrial sites, using less power can be a huge win. It saves on costs and can be more environmentally friendly.

Space Efficiency: Making good use of physical space can be crucial, especially in crowded environments like warehouses or urban areas. When space is limited, a well-organized setup can help everything run smoother.

Reliability: Efficiency can also mean having a system that runs smoothly with minimal errors. Whether it's software or machinery, reliable performance reduces downtime, maintenance needs, and overall hassle.

Simplicity and Ease of Use: For tools and technology, efficiency often means that they're simple and easy to use. A system that's quick to learn and navigate can save time and reduce frustration.

Environmental Impact: Efficiency can also mean reducing emissions, waste, or other environmental harm. Many companies prioritize this as part of a push toward sustainability, finding ways to cut down on their environmental footprint.

Q3. Select a data structure that you have seen, and discuss its strengths and limitations.

ANSWER:

Let's look at hash tables as an example data structure—they're incredibly popular in programming for quick data retrieval.

Strengths of Hash Tables:

1. **Fast Data Access**: Hash tables are super-fast when you need to look up data, usually in constant time, or O(1). Say you have an app that tracks store inventory; with a hash table, you can pull up an item's info instantly using its unique ID, making it easy to check availability in real time.
2. **Good for Large Datasets**: Hash tables can handle large amounts of data effectively, provided that the hash function distributes data evenly. For example, if you're storing user profiles for a large social media platform, hash tables can help you retrieve user information quickly without slowing down.
3. **Ideal for Key-Value Pairs:** Hash tables store data in key-value pairs, making them great for tasks like creating a cache to store website data, where URLs are the keys and the website content is the value.

Limitations of Hash Tables:

1. **Collisions**: Even the best hash function can't prevent all collisions (where two keys hash to the same slot), which slows down performance. For instance, if a hash table is storing many usernames, two similar usernames could end up in the same slot, complicating retrieval.
2. **No Ordering**: Hash tables don't keep items in order, so if you need data sorted—like arranging a leaderboard from highest to lowest scores—a hash table isn't ideal. For that, you might want a data structure like a binary search tree or a sorted array.
3. **Memory Overhead:** Hash tables can be memory-intensive because they reserve extra slots to reduce collisions. This can lead to wasted space, especially if the dataset is small or fluctuates in size.
4. **Dependency on a Good Hash Function:** The performance of a hash table relies on a well-designed hash function. Designing one that's both efficient and spreads data evenly isn't always easy, and a poor hash function can lead to poor performance.

In short, hash tables are fantastic for quick lookups in large, unordered datasets—like retrieving a customer's details in a customer service app—but they can be memory-intensive and need careful collision management.

Q4. How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

ANSWER:

The **shortest-path** and **traveling salesperson** (TSP) problems are both classic optimization problems in graph theory, but they have some key similarities and differences:

Similarities:-

1. **Graph-Based**: Both problems are typically represented using graphs, where nodes represent locations, and edges represent the paths (distances) between them.
2. **Optimization Goals**: Each problem aims to find the "best" path according to a specific criterion. In the shortest-path problem, the goal is to find the minimum distance between two points, while in TSP, the goal is to find the shortest possible route covering all nodes.
3. **Applications in Routing and Navigation**: Both problems are commonly used in navigation and routing tasks. For instance, logistics companies use shortest-path algorithms for deliveries, and TSP algorithms for planning delivery routes that involve multiple stops.

Differences:-

1. **Objective:** The shortest-path problem focuses on finding the minimum path between two specific points (start and end). In contrast, TSP aims to find the shortest round-trip route that visits all nodes (locations) and returns to the starting point.
2. **Number of Nodes Visited**: Shortest-path only requires travel between two nodes, not necessarily covering all nodes in the graph. TSP requires visiting all nodes exactly once, covering the entire graph.
3. **Complexity**: The shortest-path problem is generally easier to solve and has efficient algorithms, like Dijkstra's or A* (polynomial time for most implementations). TSP, on the other hand, is **NP-hard**, meaning there’s no known efficient solution, and it becomes computationally infeasible as the number of nodes increases.

4. **Practical Use Cases**: Shortest-path is useful for point-to-point navigation (e.g., finding the fastest route to a destination), whereas TSP is suited for circuit-based tasks, like optimizing delivery routes or planning sales calls across multiple stops.

In summary, while both are graph-based optimization problems, shortest-path focuses on the most efficient route between two specific points, and TSP is about finding the shortest complete route that visits every location exactly once.

Q5. 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 Only the Best Solution:

Aviation Flight Scheduling: In the aviation industry, flight schedules must be optimized to minimize delays, maximize fuel efficiency, and ensure passenger safety. For example, when planning flight paths, airlines need to calculate the best route to avoid bad weather, air traffic congestion, and other disruptions. Here, only the best solution will suffice because even minor inefficiencies can lead to significant delays, increased operational costs, and a poor customer experience.

Problem Allowing for Approximately the Best Solution:

Product Recommendation Systems: In e-commerce platforms like Amazon, recommendation algorithms suggest products to users based on their browsing and purchase history. The goal is to provide suggestions that are likely to lead to sales, but the exact product recommendations don't need to be perfect. As long as the recommendations are reasonably aligned with user preferences—based on a mix of popularity, user ratings, and browsing behavior—this can lead to customer satisfaction and increased sales. In this scenario, an approximately optimal solution is usually good enough, as the system can always refine recommendations based on user feedback and engagement.

Q6. 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:

Real-World Problem: Traffic Management System

In the context of a traffic management system, the availability of input data can vary significantly:

Entire Input Available in Advance:

In some scenarios, traffic management systems can analyze historical data to predict traffic patterns for specific events, like a concert or a sports game. For instance, if a major event is scheduled, planners can review past traffic data to anticipate road usage, prepare detour routes, and deploy traffic officers effectively. In this case, the entire input (historical traffic data, event timings, expected attendees) is available in advance, allowing for comprehensive planning and real-time adjustments based on predicted congestion.

Input Arriving Over Time:

Conversely, during regular commuting hours, traffic data comes in real-time from various sources like GPS data from vehicles, traffic cameras, and sensors embedded in the roads. In this case, the traffic management system must respond to changing conditions as information arrives. For instance, accidents or road closures can occur suddenly, leading to changes in traffic flow that require immediate adjustments to traffic signals, rerouting, or informing drivers through electronic signs. Here, the input is not fully available beforehand, necessitating dynamic decision-making based on incoming data.

EXERCISE 1.2

Q1. Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

ANSWER:

Navigation and Mapping Apps (e.g., Google Maps, Waze)

Navigation apps require sophisticated algorithms to provide users with optimal routes, estimated travel times, and real-time traffic updates. These algorithms work at the application level, directly affecting user experience and functionality.

Key Algorithms and Their Functions:

1. **Shortest Path Algorithms** (e.g., Dijkstra's Algorithm, A*):
   - **Function:** These algorithms calculate the shortest or fastest route from a starting point to a destination, considering factors like distance, speed limits, and road conditions.
   - **Application:** When a user enters a destination, the app uses these algorithms to generate a route that minimizes travel time or distance, helping the user reach their destination efficiently.
2. **Traffic Prediction and Congestion Detection:**
   - **Function**: Algorithms use historical traffic data and real-time inputs from user devices to detect current congestion levels and predict future traffic flow.
   - **Application:** If the app detects heavy traffic along the primary route, it can suggest alternative paths. This function relies on machine learning and statistical algorithms to estimate delays based on current conditions.
3. **Route Optimization for Multiple Stops:**
   - **Function:** When users have multiple destinations, route optimization algorithms (often variants of the Traveling Salesperson Problem) are used to find the most efficient sequence to visit each location.
   - **Application**: This helps delivery drivers or users with multiple errands find the shortest possible round-trip route, reducing travel time and fuel costs.
4. **User Behavior Prediction and Recommendations:**
   - **Function:** Machine learning algorithms analyze user patterns, such as preferred routes, departure times, and common destinations, to make route suggestions or notify users of estimated departure times.

- **Application:** For example, if a user commutes at a regular time each day, the app might preemptively suggest their usual route, considering current traffic conditions.

5. **Real-Time Data Integration and Update**:
    - **Function:** Algorithms process incoming data from GPS, sensors, and user reports to provide live updates on accidents, road closures, or new traffic patterns.
    - **Application**: This ensures that users receive timely alerts and route adjustments, enhancing safety and reliability.

In summary, navigation apps are powered by a combination of algorithms that process vast amounts of data to provide optimal routes, real-time updates, and personalized suggestions, creating a seamless navigation experience.

Q2. 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:

To find the values of n for which insertion sort is faster than merge sort, we need to compare the two functions given:

- Insertion sort takes 8n^2 steps.
- Merge sort takes 64n log n steps.

We want to find values of n such that:

$$8n^2 < 64 n \log n$$

Dividing both sides by 8n (assuming n>0) simplifies to:

$$n<8\log n$$

This inequality can be solved by testing values of n to find when n exceeds 8log n. I'll calculate it step-by-step to identify when the inequality no longer holds.

For values of n≤43, insertion sort is faster than merge sort. For n>43, merge sort becomes more efficient.

Q3. 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:

To find the smallest value of n for which an algorithm with a running time of 100n^2 runs faster than an algorithm with a running time of 2^n, we need to solve:

100n^2 < 2^n

I'll calculate approximate values to identify the smallest n that satisfies this inequality.

The smallest value of n for which an algorithm with a running time of 100 n ^2 runs faster than an algorithm with a running time of 2^n is n=15.

We want that 100n^ 2 < 2^ n. note that if n = 14, this becomes 100(14)^2 = 19600 > 2^1 4 = 16384. For n = 15 it is 100(15)^2 = 22500 < 2^ 15 = 32768. So, the answer is n = 15.

PROBLEM:

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.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

ANSWER:

We assume a 30 days month and 365 days year.

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n \lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{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

EXERCISE 2.1:-

Q1. 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:

31	41	59	26	41	58

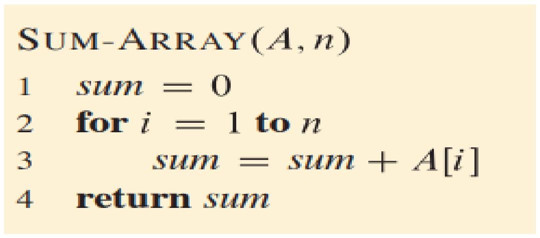
31	41	26	59	41	58

26	31	41	59	41	58

26	31	41	41	59	58

26	31	41	41	58	59

Q2. 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].



```
SUM-ARRAY($A, n$)
sum = 0
for $i$ = 1 to $n$
    sum = sum + $A[i]$
return sum
```

ANSWER:

Algorithm 1 Non-increasing Insertion-Sort(A)

```
for j = 2 to A . length do
    key = A[j]
```

```
3:     // Insert A[j] into the sorted sequence A[1..j − 1].
4:      i = j – 1
5:      while i > 0 and A[i] < key do
6:        A[i + 1] = A[i]
7:        i = i – 1
8:      end while
9:      A[i + 1] = key
10: end for
```

Q3. Rewrite the I NSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

ANSWER:

To modify the Insertion Sort algorithm to sort elements in monotonically decreasing order, we simply need to adjust the comparison so that each element is inserted where it's greater than (rather than less than) the previous elements.

Here's the modified Insertion Sort algorithm:

```
def insertion _ sort _descending(arr):
   for j in range(1, len(arr)):
      key = arr[j]
      # Move elements of arr[0...j-1] that are less than key
      # to one position ahead of their current position
      i = j - 1
      while i >= 0 and arr[i] < key:
         arr[i + 1] = arr[i]
         i -= 1
      arr[i + 1] = key

# Example usage
arr = [5, 2, 9, 1, 5, 6]
insertion _ sort _descending(arr)
print("Sorted array in decreasing order:", arr)
```

Explanation:

1. We start with the element at index j = 1 and consider it the key.
2. We then move through the sorted part of the array (from j-1 down to 0) and shift elements to the right as long as they are less than the key.
3. When we find an element that is greater than or equal to key, we insert key at that position, thus building a sorted array in descending order.

This modified algorithm now sorts the array in monotonically decreasing order.

Q4. Consider the searching problem:
Input: A sequence of n numbers (a_1, a_2, ……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 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 properties.

ANSWER:

Pseudocode for Linear Search:-

Here is the pseudocode for a linear search algorithm that scans through the array from beginning to end to find the value x:

LINEAR-SEARCH(A, x)

```
1. for i = 1 to n
2.    if A[i] == x
3.        return i
4. return NIL
```

- Input: An array A of n numbers and a value xxx to search for.
- Output: The index i where A[i]=x, or NIL if x is not found.

Loop Invariant

Loop Invariant Statement

At the start of each iteration of the loop (from line 2 onward), the subarray A[1:i−1] has been checked, and if x appears in this subarray, the function has already returned the index where x was found. If not, x does not appear in A[1:i−1], and the loop will continue to check A[i] and onward.

Proof of Correctness Using the Loop Invariant

To prove the algorithm is correct, we will demonstrate that our loop invariant fulfills the three necessary properties: Initialization, Maintenance, and Termination.

Initialization:

- Before the loop begins (at the start of line 2), no elements have been checked, so A[1: i−1] is an empty set.
- The loop invariant holds trivially, as no values have been found or ruled out yet

.

Maintenance:

- At the start of each iteration, we assume that the loop invariant is true, meaning that x has not been found in A[1: i−1].
- During the iteration, we check if A[i]=x:
    - If A[i]=x, then the function returns i, which satisfies the problem's requirements.
    - If A[i]≠x, then the loop continues to the next iteration, preserving the loop invariant because now A[1: i]has been checked, and x does not appear in A[1: i].
- Therefore, the loop invariant holds after each iteration.

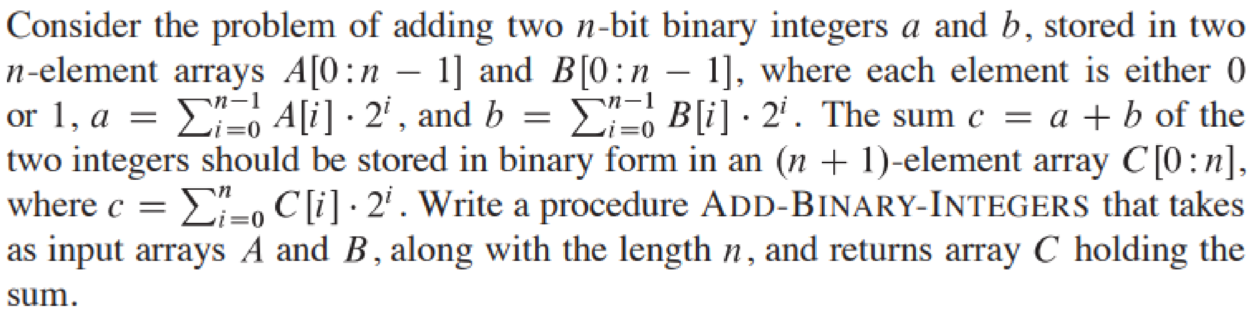
Termination:

- The loop terminates when i =n+1, meaning we have checked all elements in A[1:n].
- If x was found during the iterations, the function would have already returned the index i.
- If the loop completes without finding x, the function returns NIL, indicating that x is not present in the array.

Conclusion

Since the loop invariant holds true throughout the algorithm, and upon termination, the algorithm provides the correct output (either an index where x is found or NIL if x is not found), we conclude that the linear search algorithm is correct.

Q5.



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, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. 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:

This problem is about adding two n-bit binary integers stored in two arrays, A and B, where each element is either 0 or 1. The sum of these integers, c=a+ b, should be stored in an (n+1)element array C to account for a possible carry-over.

Procedure for ADD-BINARY-INTEGERS

We'll create a function **ADD-BINARY-INTEGERS** that takes arrays A and B and the length n, then returns the array C holding the sum.

Pseudocode

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

  1. Initialize array C[0:n] with zeros
  2. carry = 0
  3. for i = n-1 down to 0
  4.    sum = A[i] + B[i] + carry
  5.    if sum >= 2
  6.       C[i+1] = sum - 2
  7.       carry = 1
  8.    else
  9.       C[i+1] = sum
  10.       carry = 0
  11. C[0] = carry
  12. return C
```

Explanation

1. **Initialization**: We initialize an array CCC of size n+1n+1n+1 to store the sum, with each element initially set to 0.
2. **Carry Tracking**: We set a carry variable to 0 initially, as there is no carry at the start.
3. **Loop Through Bits**: We iterate from the least significant bit to the most significant bit (from n−1n-1n−1 to 0).
   - For each bit position iii, calculate the sum of A[i]A[i]A[i], B[i]B[i]B[i], and carry.
   - If the result is 2 or 3, set C[i+1]C[i+1]C[i+1] to sum - 2 and set carry to 1. This accounts for binary overflow (since 2 in binary is 10 and 3 is 11).
   - If the result is less than 2, set C[i+1]C[i+1]C[i+1] to sum and set carry to 0.

4. **Final Carry**: After the loop, we set C[0]C[0]C[0] to carry, which handles any overflow from the addition.
5. **Return Result**: The array CCC now holds the binary sum of AAA and BBB.

This algorithm correctly adds two binary numbers stored in arrays by simulating the binary addition process.

EXERCISE 2.2:-

Q1. Express the function n 3/1000 + 100n2 - 100 n + 3 in terms of Θ -notation**.**

ANSWER:

$n^3/1000 - 100n^2 - 100n + 3 \in \Theta(n\ 3)$

Q2. How can you modify any sorting algorithm to have a good best-case running time?

ANSWER:

For a good best-case running time, modify an algorithm to first randomly produce output and then check whether or not it satisfies the goal of the algorithm. If so, produce this output and halt. Otherwise, run the algorithm as usual. It is unlikely that this will be successful, but in the best-case the running time will only be as long as it takes to check a solution. For example, we could modify selection sort to first randomly permute the elements of A, then check if they are in sorted order. If they are, output A. Otherwise run selection sort as usual. In the best case, this modified algorithm will have running time Θ(n).

EXERCISES:-

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).

ANSWER:

If we start with reading across the bottom of the tree and then go up level.

By level:

3	41	52	26	38	57	9	49
3	41	26	52	38	57	9	49
3	26	41	52	9	38	49	57
3	9	26	38	41	49	52	57

2.

The test in line 1 of the MERGE-SORT procedure reads "**if** $p \geq r$" rather than "**if** $p \neq r$." 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 $n \geq 1$, the test "**if** $p \neq r$" suffices to ensure that no recursive call has $p > r$.

ANSWER:

In this problem, the focus is on the base case check in the MERGE-SORT algorithm. The base case is typically used to stop further recursion when the subarray has only one element or none, meaning it's already "sorted" by definition.

The question asks why the condition if p >= r is used instead of if p ≠ r, and why this still ensures that we don't encounter cases where p>r, as long as the initial call has n≥1.

Explanation

1. **Initial Call:**
    - When MERGE-SORT(A, 1, n) is called with n≥1, this means that p=1 and r=n, so p≤ r.
    - As long as n≥1, the array has at least one element.
2. **Recursive Division:**
    - In MERGE-SORT, the array is recursively divided into two halves. For each recursive call, the midpoint q is calculated as:

      q =[p + r/2]

    - Then, the array is split into two subarrays: A[p: q] and A[q+1:r].
3. **Ensuring Valid Recursive Calls:**
    - Since the midpoint q is always between p and r, each recursive call will have bounds where p≤ q≤ r.
    - When a subarray has only one element, we reach a case where p=r, meaning no further division is possible.
    - With the condition if p >= r, the recursion stops when p=r, ensuring no calls are made with p>r.

4. **Why if p ≠ r is Unnecessary:**
    - Since MERGE-SORT is designed to split the array down to single elements, we never encounter a case where p>r due to the way q is calculated.
    - Therefore, the condition if p >= r is sufficient to stop recursion in the base case, and p ≠ r is not required because no recursive call will produce p>r.

Conclusion

The if p >= r condition is sufficient to handle the base case in MERGE-SORT because it stops recursion when a single element is reached. This ensures no invalid recursive calls occur where p>r, making if p ≠ r unnecessary.

<u>3</u>. 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.

<u>ANSWER:</u>

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

$$\mathbf{T(2^{k+1}) = 2T\ (\ 2^{k+1}/\ 2\ ) + 2^{k+1} = 2T\ (2^k)\ \ + 2^{k+1} = 2(2^k\ lg(2^k\ )) + 2^{k+1}}$$

$$\mathbf{= k2^{k+1} + 2^{k+1} = (k + 1)2^{k+1} = 2^{k+1}\ lg(2^{k+1}) = n\ lg(n)}$$

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

$$T(n) = \begin{cases} 2 & \text{if } n = 2\,, \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is $T(n) = n \lg n$.

<u>ANSWER:</u>

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) = θ(n)

Algorithm 5 Merge(A, p, q, r)

```
$n_1 = q - p + 1$
$n_2 = r - q$
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
```

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

ANSWER:

The following recursive algorithm gives the desired result when called with a = 1 and b = n.

```
1: BinSearch(a,b,v)
2: if then a > b
3:    return NIL
4: end if
5: m = b a +b 2 c
6: if then m = v
7:    return m
8: end if
9: if then m < v
10:   return BinSearch (a, m, v)
11: end if
12: return BinSearch(m+1,b,v)
```

Note that the initial call should be BinSearch(1, n, v). Each call results in a constant number of operations plus a call to a problem instance where the quantity b − a fall by at least a factor of two. So, the runtime satisfies the recurrence T(n) = T(n/2) + c. So, T(n) ∈ Θ(lg(n))

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 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:

A binary search wouldn't improve the worst-case running time. Insertion sort has to copy each element greater than key into its neighboring spot in the array. Doing a binary search would tell us how many how many elements need to be copied over, but wouldn't rid us of the copying needed to be done.

7. The while loop of lines 5-7 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 , Θ n lg n/?

ANSWER:

Using a binary search to find the position to insert the current element in **Insertion Sort** would not improve its overall worst-case running time to Θ(n log n).

Here's why:

Analysis

1. **Binary Search for Finding the Insertion Position**:
   - In the original **Insertion Sort**, a linear search is used to find the correct position in the sorted subarray A[1:j−1].
   - Replacing the linear search with a **binary search** would indeed reduce the time to find the correct position from O(j) to O(log j).
2. **Shifting Elements**:
   - Even though finding the correct position using binary search is faster, **Insertion Sort** still needs to shift all elements after the insertion point to the right to make space for the new element.
   - In the worst case, this shifting step requires O(j) operations for each element A[j], just like in the original insertion sort.
3. **Total Time Complexity**:
   - The worst-case running time of Insertion Sort is dominated by the cost of shifting elements rather than finding the insertion position.

- Therefore, even if binary search reduces the time to locate the position to O(log j), the shifting step still requires O(j) time per iteration.
- The total time complexity remains $O(n^2)$, because shifting elements across all iterations (from 1 to n) adds up to $O(n^2)$ work.

Conclusion

Using binary search for finding the insertion point in **Insertion Sort** does not improve its overall worst-case time complexity. The algorithm would still require $O(n^2)$ time in the worst case due to the need to shift elements. Thus, the worst-case running time would not improve to Θ(n log n).

8. Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take ,Θ(n lg n)time in the worst case.

ANSWER:

To solve this problem in Θ(n log n) time, we can use a **two-pointer technique** combined with sorting. Here's the approach:

Algorithm

1. **Sort the Set S**: First, sort the set S of n integers. Sorting takes Θ(n log n) time.
2. **Use Two Pointers to Search for the Sum**:
   - Initialize two pointers: one at the beginning of the sorted array S (let's call it left) and one at the end (call it right).
   - While left is less than right, do the following:
     - Calculate the sum of the elements at left and right pointers.
     - If this sum equals x, return True (indicating that two elements that sum to x have been found).
     - If the sum is less than x, increment the left pointer to increase the sum.
     - If the sum is greater than x, decrement the right pointer to decrease the sum.
3. **Return False if No Pair is Found**:
   - If the left pointer meets the right pointer and no such pair is found, return False.

Pseudocode:

TWO-SUM(S, x):

```
1. Sort S in ascending order
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.     elif sum < x:
9.         left = left + 1
10.    else:
11.        right = right - 1
12. return False
```

Explanation of the Algorithm

1. **Sorting** the array takes Θ(n log n)time.
2. The **two-pointer search** runs in O(n) time because each pointer only moves in one direction, and we perform at most n comparisons.

Overall Complexity

- Sorting the array: Θ(n log n).
- Two-pointer traversal: O(n).
- Thus, the overall time complexity is Θ(n log n) which meets the requirement.

PROBLEMS:-

1. Insertion sort on small arrays in merge sort

Although merge sort runs in ,Θ (n 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.

a. Show that insertion sort can sort the n=k sublists, each of length k, in ,Θ(nk) worst-case time.

b. Show how to merge the sublists in ,Θ(n lg(n/k)) worst-case time.

c. 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 for which the modified algorithm has the same running time as standard merge sort, in terms of ,-notation?

d. How should you choose k in practice?

ANSWER:

a. The time for insertion sort to sort a single list of length k is Θ(k 2), so, n/k of them will take time Θ(n/k k^2) = Θ(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 lg(n) − lg(k) = lg(n/k). Each level of merging is still time cn, so putting it together, the merging takes time Θ(n lg(n/k)).

c. Viewing k as a function of n, as long as k(n) ∈ 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 c1n− nc_2/ k = 0 where c1 and c2 are the coeffients of nk and n lg(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.

CHAPTER 3

Exercises 3.1:

1. Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

ANSWER:

Since we are requiring both f and g to be asymptotically non-negative, suppose that we are past some n_1 where both are non-negative (take the max of the two bounds on the n corresponding to both f and g). Let $c_1 = .5$ and $c_2 = 1$.

$$0 \le .5(f(n) + g(n)) \le .5(\max(f(n), g(n)) + \max(f(n), g(n)))$$

$$= \max(f(n), g(n)) \le \max(f(n), g(n)) + \min(f(n), g(n)) = (f(n) + g(n))$$

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

ANSWER:

1. **Problem Setup**: Given an array of n elements where the largest αn values are in the first αn positions, we want to analyze the performance of **Insertion Sort** in terms of moves required to sort the array.

2. **Additional Restriction on α\alphaα**: To ensure there's a middle section in the array that the largest αn elements must pass through, we need α<1/2. This ensures the middle section has size (1−2α)n.

3. **Maximizing Moves Through the Middle**: To maximize the number of moves through the middle section, we balance the size of the middle section with the number of large elements. The optimal value of α for this is α=1/4, which maximizes the product α(1−2α).

4. **Conclusion**: With α=1/4 , the largest n/4 elements must pass through a middle section of size n/2, maximizing the number of comparisons. This configuration reinforces the $\Omega(n^2)$ lower bound for Insertion Sort.