Abbottabad University of Science and Technology

Department of Computer Science

SUBMITTED TO

Sir Jamal

SUBMITTED BY

Maryum Qaiser

ROLL NUMBER

14660

SUBJECT

Data Structure

DATE OF SUBMISSION

31/10/2024

Assignment: 1

CHAPTER 1 QUESTIONS:

QUESTION.1

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

ANSWER:

Real-World Example Requiring Sorting:

Organizing Student Grades

Imagine you are a teacher managing a class of students and you need to organize their grades from highest to lowest to determine class rankings. For example, you have a list of students with their scores from a recent exam, and you want to sort them to see who performed best. Sorting is essential here to quickly and easily analyze the distribution of grades, identify top performers, and even assign ranks like "first position" or "top 10 students."

This is a real-world case where sorting algorithms like **Quicksort** or **Merge_sort** can be used to arrange the data efficiently, making it easier to review and manage.

Real-World Example Requiring the Shortest Distance between Two Points:

Navigating a City Using GPS

When you are traveling in a new city and use a GPS app (like Google Maps), you want to find the **shortest route** from your current location to your destination, whether it's a restaurant, hotel, or a friend's house. The app needs to calculate the shortest possible path, considering road networks, traffic, and any potential obstacles.

This problem can be solved using algorithms like **Dijkstra's Algorithm** or *A Search Algorithm**, which are designed to find the shortest path between two points on a map, optimizing both time and distance. It's a real-world application that millions of people rely on daily for commuting and navigation.

QUESTION.2

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

ANSWER: In a real-world setting, efficiency isn't just about speed; there are several other important measures to consider, depending on the context. Some of these include:

1. Memory Usage (Space Efficiency)

- **Why it matters**: In environments with limited resources (like embedded systems or mobile devices), how much memory an algorithm or process consumes is crucial. An algorithm might be fast but could consume a lot of memory, making it impractical for certain applications.

- **Example**: Sorting algorithms like MergeSort use more memory than QuickSort, even though they may have similar speed performance. In a memory-constrained system, minimizing memory usage could be more critical than speed.

2. Scalability

- **Why it matters**: An efficient solution should work well not only on small datasets but also on large-scale inputs. Scalability ensures that as the problem size grows, the system continues to perform efficiently.
- **Example**: A database that handles a few thousand users might perform well initially, but what happens when it needs to handle millions of users? The solution must be designed to scale up without significantly degrading in performance.

3. Energy Consumption

- **Why it matters**: In mobile devices, IoT devices, or even large data centers, power consumption is a key concern. A process or algorithm that consumes less energy is more efficient, especially for battery-operated systems.
- **Example**: Efficient algorithms designed for smartphones should minimize battery drain, so even if an algorithm runs slightly slower, it may be preferred if it conserves energy.

4. Latency

- **Why it matters**: Latency refers to the time delay between a request and a response. In real-time systems, like video streaming or online gaming, minimizing latency is crucial for maintaining a good user experience.
- **Example**: In a video conferencing application, data must be transmitted and received with minimal delay to prevent lag. Even if a process is fast in general, high latency could make it inefficient in this context.

5. Throughput

- **Why it matters**: Throughput measures how much data or how many operations can be processed within a given timeframe. In high-traffic environments, such as network servers, throughput is more important than just the speed of individual tasks.
- **Example**: A web server handling thousands of requests per second needs to maximize throughput to serve as many users as possible simultaneously.

6. Reliability

- **Why it matters**: A system or algorithm that is fast but prone to failure is not efficient. Efficiency must account for how well a system can handle errors or continue functioning under unexpected conditions.
- **Example**: In cloud systems, fault tolerance is critical. A solution that may take slightly longer but can recover from faults without data loss is more efficient in the long run.

7. Ease of Maintenance

- **Why it matters**: Efficient systems are not just about performance but also about how easy they are to maintain, update, and debug. A more complex but fast system may end up being inefficient if maintenance becomes costly and time-consuming.

- **Example**: Code that's readable, well-documented, and modular is easier to maintain, even if it's slightly slower, compared to a highly optimized but convoluted system that's difficult to understand.

8. Cost Efficiency

- **Why it matters**: Cost is a critical factor in any real-world system. Efficiency should also be measured in terms of resource costs, such as hardware, storage, or even human resources required to manage the system.
- **Example**: A cloud-based solution that optimizes computing power to reduce server costs is more efficient financially than a faster but more expensive solution.

9. Security

- **Why it matters**: A process might be fast and efficient in terms of speed and memory, but if it compromises security, it's not efficient in a real-world setting. Efficient systems must also protect data and prevent vulnerabilities.
- **Example**: Encryption algorithms may slow down the system slightly but are necessary to ensure data security in applications like online banking or e-commerce. The balance between speed and security is key.

QUESTION.3

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

ANSWER:

The **Array** data structure, which is one of the most commonly used.

Strengths of Arrays:

1. **Simple and Easy to Use**: Arrays provide a straightforward way to store multiple values of the same type in a single, ordered collection.
2. **Efficient Access (O(1))**: You can access elements directly by their index in constant time, which is very efficient for read operations.
3. **Memory Efficiency**: Arrays are stored contiguously in memory, which makes them space-efficient and often improves cache performance.
4. **Static Size (Fixed Size)**: Arrays are useful when you know the number of elements in advance, allowing you to allocate memory accordingly.
5. **Iteration**: Arrays are easy to loop through for tasks like searching or updating all elements.

Limitations of Arrays:

1. **Fixed Size**: Once an array is created, its size cannot change. If you need more space or want to shrink it, you would need to create a new array and copy the data.
2. **Inefficient Insertions and Deletions (O(n))**: Adding or removing elements, especially in the middle of the array, requires shifting other elements, making these operations slower.

3. **Homogeneous Data**: Arrays generally store elements of the same type, which can be restrictive when you need to store a mix of different data types (though this depends on the language).
4. **Contiguous Memory Requirement**: Arrays require a block of contiguous memory, which may not always be available, especially for very large arrays.

Arrays are great for scenarios requiring fast access by index, but they are less efficient for dynamic or flexible data storage.

QUESTION.4

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

ANSWER:

The shortest-path and traveling salesperson problem **(TSP)** both deal with finding optimal routes, but they differ in their complexity and scope. Here's how they are similar and different:

Similarities:

1. **Graph Representation**: Both problems can be visualized as graphs where locations are represented as nodes and the paths between them as edges, with distances or costs attached to these edges.
2. **Optimization Objective**: In both cases, the objective is to minimize the total distance, cost, or time required to travel between points.
3. **Pathfinding**: Both involve determining an efficient route, whether it's between specific locations or multiple stops.

Differences:

1. **Nature of the Problem:**
    - **Shortest-Path Problem**: Focuses on finding the shortest route between two points, say from A to B, possibly passing through intermediate points.
    - **TSP**: Involves finding the shortest route that visits every point exactly once and returns to the starting location.
2. **Computational Complexity**:
    - **Shortest-Path**: Can be solved efficiently with algorithms like Dijkstra's or Bellman-Ford, and is typically easier to compute.
    - **TSP**: Is an NP-hard problem, meaning it becomes computationally difficult as the number of points increases. Solving it optimally requires considering all possible routes, which is complex.
3. **Return to Start**:
    - **Shortest-Path**: Does not require returning to the starting point; the path ends at the destination.

- **TSP**: Requires forming a circuit, returning to the starting point after visiting all locations.

4. **Number of Points**:
   - **Shortest-Path**: Focuses on finding the best route between two specific points, possibly through some intermediate nodes.
   - **TSP**: Must cover **a**ll points on the graph and return to the start, making it a more comprehensive problem.

QUESTION.5

Suggest a real-world problem in which only the best solution will do. Then come up with one in which

ANSWER:

Real-World Problem Where Only the Best Solution Will Do

Building a Spacecraft for a Manned Mission to Mars:

In this case, only the best possible solution is acceptable because the safety and success of the mission depend on it.

Critical Considerations:

Engineers must consider every detail: from the spacecraft's structure, fuel efficiency, life-support systems, to trajectory planning. Any small error could result in catastrophic failure, so perfection in design, planning, and execution is critical.

Stakes Involved:

In this scenario, there's no room for compromises or suboptimal solutions because human lives and billions of dollars are at stake.

Real-World Problem Where an Approximate Solution is Acceptable

Routing Delivery Trucks for Daily Deliveries:

For companies like Amazon or UPS, finding the **optimal route** for delivery trucks can save time and fuel, but it's often unnecessary to find the absolute best route each day.

Acceptable Solutions

Instead, an **approximate solution** that's close to optimal will do the job efficiently enough. Factors like changing traffic conditions, unpredictable roadblocks, or weather mean that even if the perfect solution is found initially, real-world disruptions might alter the plan.

Practical Approach

Therefore, a "good enough" route that saves time and cost without exhaustive calculation is practical for day-to-day operations.

QUESTION.6

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

ANSWER:

Applications Online Recommendation System

The algorithmic content that application layer recommendation systems rely on online — such as those used by Netflix, Amazon or Spotify to recommend things you may like based on your previous behavior. These systems apply the study of patterns to recognition tasks in which high dimensional data is input, examples are automatic speech clustering, image segmentation and object recognition. They analyze user behavior (previous elements purchased/viewed/clicked on), preferences (user profile/user ratings), resources (e.g., songs or products with large sales numbers; YouTube videos trending worldwide); item characteristics (attributes directly associated with items such as Arabic movie genre for movies) so that would recommend product/movies/music etc.

Algorithms Involved

Function: This algorithm provides suggestions as per the user behavior and interests. It works under the assumption that if two users have similar tastes, then they will like different items. The two main types of collaborative filtering are:

User-Based Collaborative Filtering: It identifies other users who are same with the target content/user and create recommendations using items found in each user.

Item-Based Collaborative Filtering: This method focuses on using item similarity to recommend other items the current user might like.

For instance:

If User A and User B rated a number of movies the same way so that revenant data set would have been pretty, when user watched a movie he liked but B did not (in his training example dataset).

QUESTION.7

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

ANSWER:

To figure out when insertion sort is faster than merge sort, let's break down the steps simply:

1. Compare Steps for Each Algorithm:

Insertion sort takes \ (8n^2 \) steps to run, while merge sort takes \ (64n \lg n \) steps. Here, \ (n \) represents the size of the input, and \ (\lg n \) is the logarithm of \ (n \) (logarithms grow slowly as \ (n \) gets larger).

2. Set Up an Inequality:

We want insertion sort to be faster than merge sort, so we set up an inequality:

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

3. Simplify the Inequality:

Divide both sides by 8 and \ (n \) (assuming \ (n > 0 \)) to make it simpler:

$$n < 8 \lg n$$

This means we're looking for values of \ (n \) where \ (n \) is smaller than \ (8 \lg n \).

4. Estimate Values of \ (n \):

For small values of \(n \), we can see that \(8 \lg n \) is larger than \(n \), which makes the inequality true. However, as \(n \) gets bigger, \(n \) starts to grow faster than \(8 \lg n \).

5. Find the Cutoff Point:

By testing values, we find that the inequality holds for values of \ (n \) up to around 43. So, insertion sort is faster than merge sort when \ (n \) is less than about 4

Mathematically:

8n2<64nlgn

0<n<8lgn

1<2n<n8

2≤n≤43

QUESTION.8

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

ANSWER:

To determine the smallest value of \ (n \) for which an algorithm with a running time of \ (100n^2 \) becomes faster than an algorithm with a running time of \ (2^n \), we can approach it this way:

1. Identify the Growth Patterns:

The term \ (100n^2 \) grows quadratically (more slowly), while \ (2^n \) grows exponentially (much faster). This means that for smaller values of \ (n \), \ (100n^2 \) will likely be larger. However, as \ (n \) increases, \ (2^n \) will eventually outgrow \ (100n^2 \).

2. Set Up a Comparison:

We are looking for the smallest \ (n \) where \ (100n^2 \) is less than \ (2^n \). So we test small values of \ (n \) until we find a point where \ (100n^2 < 2^n \) holds.

3. Check Values Incrementally:

By testing values of \(n \), we find that initially, \(100n^2 \) is larger than \(2^n \), but at \(n = 15 \), \(2^n \) becomes larger. Thus, the smallest \(n \) where \(100n^2 \) is faster than \(2^n \) is \(n = 15 \).

100*n*2<2*n*

***n*≥15.**

QUESTION.9

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

ANSWER:

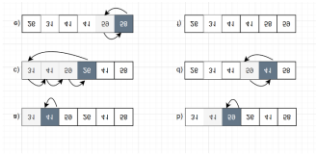
Function	1 second	1 minute	1 hour	1 day	1 month	1 year
lgn	2^{10^6}	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.59\times10^{12}}$	$2^{3.15\times10^{13}}$
$\sqrt{n}$	10^{12}	3.6×10^{15}	1.3×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}
n	10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}
$nlgn$	6.24×10^4	2.8×10^6	1.33×10^8	2.76×10^9	7.19×10^{10}	7.98×10^{11}
n^2	1,000	7,745	60,000	293,938	1,609,968	5,615,692
n^3	100	391	1,532	4,420	13,736	31,593
2^n	19	25	31	36	41	44
$n!$	9	11	12	13	15	16

Chapter:2

QUESTION.1

Exercises 2.1-1 Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence h31;41;59;26;41;58 i .

Figure 2.2:



ANSWER:

As shown in the figure above, the array A change as follow:

A=⟨26, 31, 41, 59, 41, 58⟩

A=⟨26, 31, 41, 59, 41, 58⟩

A=⟨26, 31, 41, 59, 41, 58⟩

A=⟨26, 31, 41, 41, 59, 58⟩

A=⟨26, 31, 41, 41, 58, 59⟩

A=⟨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 . 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 AŒ1 W n .

ANSWER:

SUM-ARRAY (A, n)

Sum = 0

For i = 1 to n

Sum = sum + A [i]

Return sum

- **Loop invariant**: At the start of the *i*th iteration of the **for** loop, the equation sum=∑j=1i−1A[j]*sum*=∑*j*=1*i*−1*A*[*j*] holds.
- **Initialization**: Before the first iteration (i=1), sum=∑j=10A[j] =0*sum*=∑*j*=10*A*[*j*] =0 holds.
- **Maintenance**: During the *i*th iteration, sum = sum + A[i] results sum=∑j=1i−1A[j]+A[i]=∑j=1iA[j], at the end of this iteration, Incrementing *i* for the next iteration of the for loop then preserves the loop invariant.
- **Termination**: The loop terminates when i>n*i*>*n*, since *i* increase 1 at each iteration, i=n+1*i*=*n*+1 at termination.sum=∑j=1i−1A[j]=∑j=1nA[j]*sum*=∑*j*=1*i*−1*A*[*j*]=∑*j*=1*n* *A*[*j*] holds.

QUESTION.3

Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

ANSWER:

INSERTION_SORT_NONINCEASE (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]
 A [j+1] = key
```

QUESTION.4

Consider the problem of adding two n-bit binary integers aa and bb, stored in two nn-element arrays A[0:n−1]A[0:n−1] and B[0:n−1]B[0:n−1], where each element is either 00 or 11. a=∑i=0n−1A[i]·2i$a=\sum_{i=0}^{n-1}A[i]\cdot 2^i$ and b=∑i=0n−1B[i]·2i$b=\sum_{i=0}^{n-1}B[i]\cdot 2^i$. The sum c=a+b$c=a+b$ of the two integers should be stored in binary form in an (n+1n+1)-element array C[0:n]C[0:n], where c=∑i=0nC[i]·2i$c=\sum_{i=0}^{n}C[i]\cdot 2^i$. Write a procedure ADD-BINARY-INTEGER**S** that takes as input arrays AA and BB, along with the length n, and returns array **C** holding the sum.

ANSWER:

```
ADD_BINARY_INTEGERS (A, B, n)
   C [0: n]
   Carry=0
   for i = 1 to n
   C[i-1]=(A[i]+B[i]+carry)%2
   carry = (A[i]+B[i]+carry)/2
   C[n] = carry
   return C;
```

QUESTION.5

Express the function n3/1000+100n2−100n+3n3/1000+100n2−100n+3 in terms of ΘΘ-notation.

ANSWER:

The function n3/1000+100n2−100n+3n3/1000+100n2−100n+3 in terms of ΘΘ-notation is:

Θ(*n*3)

QUESTION.6

Consider sorting n*n* numbers stored in array A[1:n]*A*[1:*n*] by first finding the smallest element of A[1:n]*A*[1:*n*] and exchanging it with the element in A[1]*A*[1]. Then find the smallest element of A[2:n]*A*[2:*n*], and exchange it with A[2]*A*[2]. Then find the smallest element of A[3:n]*A*[3:*n*], and exchange it with A[3]*A*[3]. Continue in this manner for the first n−1*n*−1 elements of A*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*n*−1 elements, rather than for all n*n* elements? Give the worst-case running time of selection sort in ΘΘ-notation. Is the best-case running time any better?

ANSWER:

- **pseudocode**

SELECTION_SORT(A)

for i = 1 to A.length()-1

minindex = i

for j = i to A.length()

if A[j]<A[minindex]

minindex = j

swap(A[i],A[minindex])

- **loop invariant:** At the start of i*i*th iteration, A[1:i−1]*A*[1:*i*−1] consist of the i−1*i*−1 elements of A[1:n]*A*[1:*n*] and it is at sort order.
  - When the i-1 smallest elements are sort in the subarray A[1:n−1],thebiggestelementmustbe*A*[1:*n*−1],*thebiggestelementmustbe*A[n]$, and then A[1:n] are sort.
- **Best case:** the same as worst case since in each iteration in inner **for** loop, j must go from i to n to get the minded.

QUESTION.7

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 ΘΘ-notation, give the average-case and worst-case running times of linear search. Justify your answers

ANSWER:

- average-case: running time∑i=1ni/n=(n+1)/2=Θ(n)running time∑*i*=1*ni*/*n*=(*n*+1)/2=Θ(*n*)
- worst-case: running time=n=Θ(n)running time=*n*=Θ(*n*)

Explanation:

Suppose that every entry has a fixed probability p of being the element looked for. A different interpretation of the question is given at the end of this solution. Then, we will only check k elements if the previous k − 1 positions were not the element being looked for, and the kth position is the desired value. This means that the probability that the number of steps taken is k is p(1−p) k . The last possibility is that none of the elements in the array match what we are looking for, in which case we look at all A.length many positions, and it happens with probability (1 − p). By multiplying the number of steps in each case by the probability that that case happens, we get the expected value of: E(steps) = A.length(1 − p) A.length + A.length X k=1 k(1 − p) k−1 p The worst case is obviously if you have to check all of the possible positions, in which case, it will take exactly A.length steps, so it is Θ(A.length). Now, we analyze the asymptotic behavior of the average-case. Consider the following manipulations, where first, we rewrite the single summation as a double summation, and then use the geometric sum formula twice.

QUESTION.8

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

ANSWER:

To improve the best-case running time of a sorting algorithm, we can apply a few key modifications:

1. Check if the Array is Already Sorted:

Add a preliminary pass to check if the list is sorted before running the main algorithm. If the list is already sorted, skip sorting entirely. This check has a time complexity of \(O(n) \), which makes the best-case time \(O(n) \) for algorithms that are typically slower.

2. Use Early Exit Conditions

Modify the algorithm to detect if no more sorting is needed partway through. For example:

Bubble Sort: Track if any swaps were made in a pass; if not, exit early.

Insertion Sort: Naturally achieves \(O(n) \) in the best case when the list is already sorted, so no modification is needed here.

3. Implement Adaptive and Hybrid Techniques

Use adaptive algorithms or hybrid approaches that adjust based on input structure. For instance, Time sort (used in Python) merges sorted runs in the data and performs \(O(n) \) for mostly sorted lists.

Quicksort Hybrid: Switch to insertion sort for smaller subarrays, improving efficiency on small or nearly sorted data.

4. Choose an Efficient Pivot for Quicksort

In quicksort, selecting the pivot carefully can improve performance on sorted data. Using the median of the first, middle, and last elements (median-of-three) avoids worst-case splits on sorted inputs.

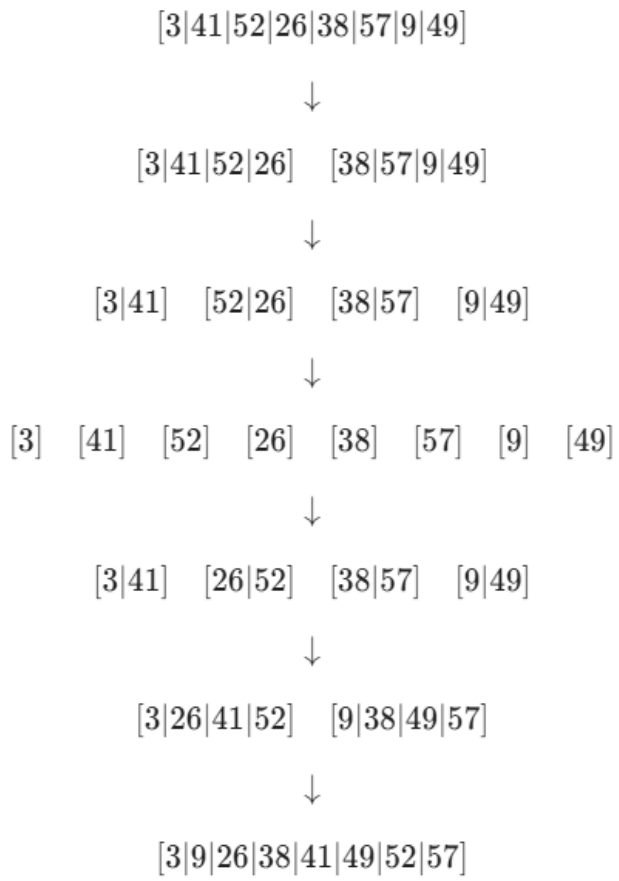
These modifications can make sorting algorithms achieve better best-case running times by adapting to sorted or partially sorted input conditions.

2.3 exercise:

QUESTION.1

sing 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⟩⟨3,41,52,26,38,57,9,49⟩.

ANSWER:



QUESTION.2

The test in line 1 of the MERGE-SORT procedure reads "if p≥r*p*≥*r*" rather than "if p≠r*p*□=*r*." If MERGE-SORT is called with p>r*p*>*r*, then the subarray A[p:r]*A*[*p*:*r*] is empty. Argue that as

long as the initial call of MERGE-SORT(A, 1, n) has n≥1$n \geq 1$, the test "if p≠r$p \neq r$" suffices to ensure that no recursive call has p>r$p > r$.

ANSWER:

In the MERGE-SORT algorithm, the purpose of the condition `if $p \neq r$` (instead of `if $p \geq r$`) is to ensure that the array has more than one element, meaning that the subarray `A[p:r]` can be split into smaller parts for further sorting. Let's examine why `$p \neq r$` works effectively without causing the issue of `$p > r$` in recursive calls, given that the initial call is made with `MERGE-SORT(A, 1, n)`

Key Points in the Argument

1. Initial Condition:

- The first call to `MERGE-SORT` is with `MERGE-SORT(A, 1, n)`, where `1` is the starting index `p` and `n` is the ending index `r`, and `n` is at least `1`.
- This setup ensures that `p` is always less than or equal to `r` initially, so the subarray `A[1:n]` is non-empty.

2. Behavior of Recursive Calls:

- When `$p \neq r$`, the algorithm proceeds to split the array by calculating the midpoint `q = (p + r) / 2` and then recursively calls `MERGE-SORT` on the subarrays `A[p:q]` and `A[q+1:r]`.
- Since `q` is always between `p` and `r`, the recursive calls will always split into subarrays where `$p \leq q$` and `$q + 1 \leq r$`.
- The values of `p` and `r` are adjusted in each recursive call, but because they are based on integer calculations of midpoints, there will never be a case where `p` becomes greater than `r`.

3. Termination Condition:

- The test `if $p \neq r$` (or equivalently `if $p < r$`) is sufficient to ensure that recursion only proceeds when there is more than one element in the subarray.
- If `$p = r$`, the subarray has only one element, and no further splitting occurs. This condition naturally stops further recursive calls from being made, as the subarray cannot be divided further.
- Therefore, we never reach a state where `$p > r$` because the recursion terminates at `$p = r$`.

Conclusion:

The condition `if $p \neq r$` is enough to prevent any case where `$p > r$` in recursive calls. This ensures that the algorithm only operates on valid, non-empty subarrays and terminates correctly when the subarray has been divided down to single elements, satisfying the merge-sort process without violating array bounds.

QUESTION.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.

Answer:

- loop invariant: at the start of each iteration, A[p:k−1]*A*[*p*:*k*−1] consist of the smallest elements of union of L*L* and R*R* at sort order.
- when the *while* loop of lines 12-18 finish A[p:k−1]*A*[*p*:*k*−1] consist of the smallest elements of union of L*L* and R*R* at sort order, either *while* loop of lines 20–23 or 24–27 will be run, L*L* or R*R* has the biggest rest elements in sort order.when the second *while* loop finish, they are added to the end of A, and A consist of all elements at sort order.

QUESTION.4

Use mathematical induction to show that when n≥2*n*≥2 is an exact power of 2, the solution of the recurrence.

Answer

To prove this by mathematical induction, we need to verify that for \(n \geq 2 \), where \(n \) is an exact power of 2, the solution of the recurrence relation satisfies the given property.

Let's define the recurrence relation \(T(n) \) as follows:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

where

$$\(T(2) = c \) for some constant \(c \).$$

We want to show that \(T(n) = n \log_2 n + cn \) holds for all \(n = 2^k \), with \(k \geq 1 \), by using mathematical induction.

Step 1: Base Case

Let's verify for the base case, \(n = 2 \):

$$T(2) = c$$

According to the formula

$$(T(n) = n \log_2 n + cn :$$

$$\boldsymbol{T(2) = 2 \cdot log2(2) + c \cdot 2 = 2 \cdot 1 + 2c = 2c}$$

QUESTION.5

You can also think of insertion sort as a recursive algorithm. In order to sort A[1:n]*A*[1:*n*], recursively sort the subarray A[1:n−1]*A*[1:*n*−1] and then insert A[n]*A*[*n*] into the sorted subarray A[1:n−1]*A*[1:*n*−1]. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

Answer

The pseudocode for a recursive version of insertion sort that sorts an array \(A[1:n] \) by recursively sorting the subarray \(A[1:n-1] \) and then inserting \(A[n] \) into the sorted subarray.

Recursive Insertion Sort Pseudocode:

RECURSIVE-INSERTION-SORT(A, n)

1. if $n \leq 1$
2. return // Base case: a single element is already sorted
3. RECURSIVE-INSERTION-SORT(A, n - 1) // Sort the subarray A[1:n-1]
4. INSERT(A, n) // Insert A[n] into the sorted subarray A[1:n-1]

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] // Shift element one position to the right
5. i = i - 1
6. A[i + 1] = key // Place the key in its correct position

Explanation

- RECURSIVE-INSERTION-SORT** takes an array \(A \) and the size \(n \) as inputs.
- The base case is when \(n \leq 1 \), as an array with one or no elements is already sorted.
- For \(n > 1 \), it first recursively sorts the subarray \(A[1:n-1] \).
- After the recursive call, the subarray \(A[1:n-1] \) is sorted, and we then call INSERT to place \(A[n] \) in its correct position within this sorted subarray.

Worst-Case Running Time Analysis:

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

- **Recursive call:** Sorting the subarray \(A[1:n-1] \) takes \(T(n - 1) \).
- **INSERT function**: In the worst case, INSERT might shift all \(n - 1 \) elements in \(A[1:n-1] \) to the right, taking \(O(n - 1) \) time.

So, the recurrence relation for \(T(n) \) is:

$$[T(n) = T(n-1) + O(n-1)]$$

In (\Theta \) notation, we have:

$$[T(n) = T(n-1) + \backslash Theta(n)]$$

Solving the Recurrence

Expanding the recurrence:

$$T(n) = T(n-1) + n$$
$$= T(n-2) + (n-1) + n$$
$$= T(1) + 2 + 3 + \dots\dots\dots + n$$

The sum \(2 + 3 + \dots + n \) is an arithmetic series with a sum of \(\frac{n(n + 1)}{2} - 1 \), which simplifies to \(\Theta(n^2) \).

Thus, the worst-case running time of the recursive insertion sort is:

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

QUESTION.6

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

Answer:

Recursive Insertion Sort:

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

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

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

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

Recurrence for worst case :

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

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

Question7

Referring back to the search problem …….. Subarray from further .

Answer :

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

Question 8 :

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

Answer:

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

---

Question 9 :

Describe an algorithm n that give the set of S of n integers …….worst case
Answer :

1. Algorithm :

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

2 . pseudocode :

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

3 Running time analysis :

Sorting takes **O (n log n)** times

The two pointers scans takes **O (n)**

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

---

Problems of chapter 2

Question:1

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

Answer:

Insertion sort on small array in merge sort:

a) we have worst case time complexity of insertion sort on a single sub list of length k is theta (k^2) so $\frac{n}{k}$ of them will take time **theta** $(\frac{n}{k}\ k^2)$ **theta (nk)**
b) Suppose we have coarseness k. This means we can just start using the usual merging procedure, except starting it at the level in which each array has size at most k. This means that the depth of the merge tree is **log(n) − log(k) = log(n/k).**
c) Each level of merging is still time cn, so putting it together, the merging takes time

Θ (n log(n/k))

a) 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. In particular, for any constant choice of k, the asymptotics are the same
b) If we optimize the previous expression using our calculus 1 skills to get k, we have that $c_1 n$ -

$\frac{nc_2}{k}$ **= 0** where c1 and c2 are the coeffients of nk and **n log(n/k)** hidden by the asymptotics notation. In particular, a constant choice of k is optimal. In practice we could find the best choice of this k by just trying and timing for various values for sufficiently large n

---

Problem 2 :

Correctness of Horner's rule You get the coefficients …….the value of x

HORNER (A , n, x)

1 p =0

2 for i = n downto 0

3 p =A[i] + x-p

4 return

Answer :

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

b)

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

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

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

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

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

---

Problem 3:

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

Answer:

a) In the array [2 ,3 ,8,6,1] the five insertions are

(3 ,4) , (1, 5) , (2, 5) , (3 ,5) ,(4 ,5)

b) The array [n ,n-1 ,…..1] has the most inversion s with $\frac{n(n-1)}{2}$ inversions

c) The running time of insertion sort is O (n+k) where k is the number of inversions. More inversions lead to longer sorting time with O (n^2) in the worst case

Used a modify merge sort to count inversions in O (n log n) time .During merging count when elements from the right subarray are less than those in the left

---

Chapter # 3:

Question: 3.1-1

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

Answer:

To change the argument for insertion sort, here's a simple explanation:

1. **What Insertion Sort Does**:
   Insertion sort arranges a list by picking one item at a time and putting it in the right place among the already sorted items. It works worst when the list is sorted in reverse order.
2. **Counting Comparisons**:
   For a list with n items, the worst-case number of comparisons is:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$$

OR

$$1 + 2 + 3 + \cdots + (n-1) = \frac{(n-1)n}{2}$$

This means it takes about $O(n^2)$comparisons.

3. **Any Size of n**:
   This argument works whether n is a multiple of 3 or not. Even if there are leftover items, insertion sort still needs to compare each new item with the sorted ones.
4. **Conclusion**:
   So, insertion sort always takes about $O(n^2)$ time, no matter how many items there are. This shows that insertion sort is not good for large lists.

In summary, insertion sort needs $O(n^2)$ time for any input size, making it inefficient for big lists.

Question: 3.1-2

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:

Selection Sort Algorithm:
Selection sort arranges a list by repeatedly selecting the smallest element from the unsorted portion and moving it to the front. Here's how it works:

1. Start with the first element and assume it is the smallest.

2. Compare this element with all other elements to find the smallest one.
3. Swap the smallest found element with the first element.
4. Move to the next position and repeat until the entire array is sorted.

Pseudocode:

$for\ i\ from\ 0\ to\ n-1\ do$:

  $min\_index\ =\ i$

  $for\ j\ from\ i+1\ to\ n-1\ do$:

    $if\ array[j]\ <\ array[min\_index]\ then$

      $min\_index\ =\ j$

  $swap\ array[i]\ with\ array[min\_index]$

Running Time Analysis:

1. **Comparisons**:
   - In the first pass, it compares n elements.
   - In the second pass, it compares n−1n - 1n−1 elements, and this continues until it compares just 1 element.
   - The total number of comparisons is:

$$1 + 2 + 3 + \cdots + (n-1) = \frac{(n-1)n}{2}$$

   This simplifies to $O(n^2)$.
2. **Swaps**:
   - There is one swap for each of the $n-1$ passes, so the number of swaps is also $O(n)$, but it is not the dominant factor.

Conclusion:

The overall running time of the selection sort algorithm is:

$$O(n^2)$$

This means selection sort is not efficient for large lists because its running time increases significantly as the number of elements grows.

Question: 3.1-3

Suppose that ¸ is a fraction in the range $0 < \alpha < 1$. 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 α maximizes the number of times that the αn largest values must pass through each of the middle $(1-2\alpha)n$ array positions?

Answer:

To analyze the lower-bound argument for insertion sort with the αn\alpha nαn largest values starting in the first αn positions, we can follow these steps:

Generalizing the Argument:

1. **Input Arrangement**:
    - We have an array of size n.
    - The first αn\alpha nαn positions contain the largest αn\alpha nαn values.
    - The remaining $(1-\alpha)n$ values are mixed and can be smaller.
2. **Insertion Sort Process**:
    - Insertion sort processes the array from left to right, comparing each element with those that are already sorted.
    - Since the largest αn values start at the front, they will need to be placed correctly among the smaller values that follow.

Analyzing the Passes:

3. **Passes Through the Middle Positions**:
    - The middle positions consist of the values in the range from αn to $(1-\alpha)n$.
    - As the αn largest values move to their final positions, they must pass through the middle $(1-2\alpha)n$ positions.

Restrictions on α\alphaα:

4. **Restrictions**:
    - To ensure that there are actual middle positions for the largest values to pass through, we need:

$$1-2\alpha > 0 \Longrightarrow \alpha < \frac{1}{2}$$

Maximizing Passes:

5. **Maximizing the Number of Passes**:
    - The number of times the αn\alpha nαn largest values pass through the middle positions is maximized when α\alphaα is as large as possible, while still satisfying $\alpha < \frac{1}{2}$.
    - Therefore, the best value for α\alphaα to maximize the number of passes through the middle positions is:

$$\alpha \to \frac{1}{2}$$

Conclusion:

In summary, the lower-bound argument for insertion sort shows that when the αn largest values start in the first αn positions, they will still require multiple comparisons to find their correct place in the array. The restriction on α is that it must be less than $\frac{1}{2}$, and the value that maximizes the passes through the middle positions is α approaching $\frac{1}{2}$.
