Chapter 1

Exercises-1.1

Question-1

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

Answer:

Example of Sorting: Consider rearranging our music playlist according to the songs that were played most recently. This involves arranging the songs in order of their play date so that we can listen to the ones that we haven't heard in a while.

Example of Shortest Distance: If we aim to locate the fastest way to reach a friend's house, we would utilize a map to identify the shortest distance between our location and theirs, especially if we're trying to avoid traffic.

Quetion-2

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

Answer:

Additional Efficiency Metrics:

Memory Usage: Certain algorithms might prioritize speed but require more memory, posing a challenge for systems with limited resources.

Energy Efficiency: Energy consumption is a crucial factor, especially for mobile or embedded devices.

Scalability: The performance of an algorithm as the input size increases is an important consideration.

Ease of Implementation and Maintenance: The complexity of coding or maintaining some algorithms can impact development time and costs.

Question-3

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

Answer:

Arrays are a type of data structure.

They are easy to understand and utilize, making them ideal for situations requiring fast data retrieval by index.

Limitation: They have a fixed size, so adding or removing items can be slow or require creating a new array.

Question-4

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

Answer:

Both the shortest-path problem and the traveling-salesperson problem involve determining the optimal route on a map.

Shortest path: The main objective is to travel from one point to another in the fastest way possible.

Traveling salesperson: The goal is to visit multiple points and then return to the starting point, all while aiming to minimize the overall trip distance.

Questin-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:

Requiring the best solution: For instance, in the case of heart surgery, it is essential for the doctor to adhere to the most optimal plan to guarantee the safety of the patient, leaving no room for a "nearly correct" solution.

Accepting an approximate solution: In the context of purchasing a flight ticket, it is not always necessary to secure the absolute lowest price. Something that comes close to it is often good enough.

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:

When considering completing our taxes let's say, typically all the necessary documents (such as income and expenses) are accessible when we begin filling out the forms.

As we respond to real-time text messages, we do not have all the messages in advance. We must reply as new messages continue to arrive.

Exercise-1.2

Question-1

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

Answer:

An example of an application that requires algorithmic content is a navigation system (e.g., Google Maps). The primary algorithm involved in this system is Dijkstra's Algorithm or the A (A-star) Algorithm* for finding the shortest path between two locations on a map.

Function of the Algorithms:

Dijkstra's Algorithm: This is a classic algorithm used for finding the shortest path from a source node to all other nodes in a graph. In the case of a navigation system, locations are treated as nodes, and roads or paths between them are treated as edges with weights representing distance or time. Dijkstra's algorithm ensures that the navigation system can quickly compute the optimal route from one location to another.

Algorithm*: This is a more efficient pathfinding algorithm than Dijkstra's in certain scenarios. It uses a heuristic to estimate the distance to the goal and prioritizes exploring nodes that are

more likely to lead to the goal quickly. It's often used in real-time navigation systems to improve performance and provide faster routes by limiting the number of nodes explored.

Question-2

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 :

Solve the function: 8n^2 <= 64nlgn, so n <= 43

n	8n^2	64nlgn
2	32	128
3	72	304
4	128	512
10	800	2126
20	3200	5532

Question-3

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:

We must determine the smallest nn so that:

$$100n2<2n$$

This inequality can be solved by testing values of n:

- For n=10: $100n^2$=100×10^2=10000 and 2^{10} =1024 (False)
- For n=15: $100n^2$=100× 15^2=22500 and 2^{15} = 32768 (True)
- For n=20:$100n^2$=100×20^2=40000 and 2^{20}= 1048576 (True)

So, the smallest value of n where 100n2100n^2100n2 runs faster than 2n2^n2n is **15**.

CHAPTER : 2

Exercise 2.1

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:

Let's walk through how **Insertion Sort** works step-by-step on the array:

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

Insertion Sort Algorithm:

1. Start from the second element (index 1) and compare it with the first element.

2. Insert the current element in its correct position among the already sorted elements to its left.

3. Repeat this process for each element, moving through the array from left to right.

Initial Array:

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

Step 1: Compare 41 with 31

- 31 is already in the correct position.
- 41 is larger than 31, so no changes are made.

Array after step 1:

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

Step 2: Compare 59 with 41

- The elements 31 and 41 are sorted.
- 59 is larger than 41, so it stays in its position.

Array after step 2:

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

Step 3: Compare 26 with 59, 41, 31

- 26 is smaller than 59, so it should be inserted before 59.
- 26 is smaller than 41, so it should be inserted before 41.
- 26 is smaller than 31, so it should be inserted before 31.

After shifting 31, 41, and 59 to the right, we insert 26 at the beginning.

Array after step 3:

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

Step 4: Compare 41 with 59, 41, 31, 26

- 41 is smaller than 59, so it should be inserted before 59.

- 41 is equal to 41, so it stays in its place after shifting 59 to the right.

Array after step 4:

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

Step 5: Compare 58 with 59, 41, 41, 31, 26

- 58 is smaller than 59, so it should be inserted before 59.
- 58 is larger than 41, so it stays after the second 41.

After shifting 59 to the right, we insert 58 before it.

Array after step 5 (Final Sorted Array):

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

Final Sorted Array:

[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 AŒ1 W n.? State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUMARRAY procedure returns the sum of the numbers in AŒ1 W n?

Answer:

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].

Pseudo code:

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

My solution:

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 instead of monotonically increasing order.

Answer:

InsertionSort (A) (Pseudo code)

```
1   for j=2 to A.length
2    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
6   A[i+1]=A[i]
7    i=i−1
8    A[i+1]=key
```

Question-4

Consider the searching problem: Input: A sequence of n numbers ha1; a2; : : : ; ani stored in array AŒ1 W n and a value x. Output: An index i such that x equals AŒi 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 fulûlls the three necessary properties.

Answer:

For linear search, we just need to scan the array from the beginning till the end, index 11 to index nn, and check if the entry at that position equal to v or not. The pseudocode can be written as follows…

LINEAR-SEARCH (A,v) (Pseudocode)

```
1  for i=1 to A.length
```

2 if A[i]==v

3 return I

4 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 vv.

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

Initialization: Initially the subarray is empty. So, none of its' elements are equal to vv.

Maintenance: In i-th iteration, we check whether A[i] is equal to U 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 U before the ii-th iteration, the subarray A[1..i]A[1..i] will not contain U before the next iteration (unless i-th iteration terminates the loop).

Termination: The loop terminates in either of the following cases,

- We have reached index ii such that U = A[i]A[i], or
- We reached the end of the array, i.e. we did not find v in the array A. So, we return NIL.

In either case, our algorithm does exactly what was required, which means the algorithm is correct.

Exercise 2.2

Question-1

Express the function n 3=1000 C 100n2 ⍰ 100n C 3 in terms of ,-notation

Answer:

To express the function f(n) = $\frac{n^3}{1000} + 100n^2 - 100n + 3$ in terms of ThetaΘ-notation, we analyze the leading term, which will dominate the growth of the function as nnn becomes large.

1. **Identify the leading term:** The term $n^3/1000$ is the highest order term in the polynomial.

2. **Ignore lower-order terms:** The other terms $100n^2 + 100n$,and 3 grow slower than n^3 for large n.
3. **Express in Theta Θ-notation:** We can focus on the leading term:

   f(n) = $\frac{n^3}{1000} + 100n^2 - 100n + 3 \sim n^3/1000$

   As n approaches infinity, f(n) behaves like $n^3/1000$.

Thus, we can conclude that:

f(n) = Θ(n^3)

This indicates that the function grows cubically with n.

Question-2

Consider sorting n numbers stored in array AŒ1 W n] by ûrst ûnding the smallest element of AŒ1 W n] and exchanging it with the element in AŒ1]. Then ûnd the smallest element of AŒ2 W n], and exchange it with AŒ2]. Then ûnd the smallest element of AŒ3 W n], and exchange it with AŒ3]. Continue in this manner for the ûrst 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 ûrst n1 elements, rather than for all n elements? Give the worst-case running time of selection sort in ,-notation. Is the best-case running time any better?

Answer:

SELECTION-SORT(A, n) (Pseudo code)

1. for i = 1 to n-1
2. min Index = i
3. for j = i+1 to n
4. if A[j] < A[min Index]
5. min Index = j
6. swap A[i] with A[min Index]

Explanation:

1. The outer loop runs from i=1i = 1i=1 to n−1n-1n−1, which iterates over each element in the array.

2. For each iteration, the algorithm finds the smallest element in the unsorted portion of the array A[i…n]A[i \dots n]A[i…n].
3. Once the smallest element is found, it is swapped with the element at index iii.
4. This process repeats for all the elements, except for the last one. Once the first n−1n-1n−1 elements are sorted, the remaining element must be the largest, so the algorithm doesn't need to check the last element separately.

Question-3

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:

On average, half the the elements in array A will be checked before v is found in it. And in the worst case (v is not present in A), all the elements needs to be checked.

In either case, the running time will be proportional to n, i.e. Θ(n)Θ(n).

Question-4

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

Answer:

We can design any algorithm to treat its best-case scenario as a special case and return a predetermined solution.

For example:

for selection sort, we can check whether the input array is already sorted and if it is, we can return without doing anything. We can check whether an array is sorted in linear time. So, selection sort can run with a best-case running time of Θ(n).

Exercise-2.3

Question-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence h3; 41; 52; 26; 38; 57; 9; 49i.

Answer

To illustrate the operation of Merge Sort on the array \[3, 41, 52, 26, 38, 57, 9, 49\], we will break it down step by step using a divide-and-conquer approach.

Initial Array:

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

Step 1: Divide

We divide the array into two halves until each subarray has only one element.

1. Split:

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

2. Split each half again:

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

3. Continue splitting:

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

Now each subarray contains only one element, and we start merging.

Step 2: Conquer (Merge)

1. Merge [3] and [41] → [3, 41]

Merge [52] and [26] → [26, 52]

Merge [38] and [57] → [38, 57]

Merge [9]and [49] → [9, 49]

After the first round of merging:

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

2. Merge [3, 41] and [26, 52] → [3, 26, 41, 52]

Merge [38, 57] and [9, 49] → [9, 38, 49, 57]

After the second round of merging:

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

3. Merge [3, 26, 41, 52] and [9, 38, 49, 57] → [3, 9, 26, 38, 41, 49, 52, 57]

Final Sorted Array:

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

This is the result of the merge sort process.

Question-2

The test in line 1 of the MERGE-SORT procedure reads r, then the subarray AŒp W r□ is empty. Argue that as long as the initial call of MERGE-SORT.A; 1; n/ has n □ 1, the test r.

Answer:

To understand why the condition "if p ≠ r " suffices to ensure that no recursive call in the MERGE-SORT algorithm has p > r , we can analyze the behavior of the recursive calls made by the MERGE-SORT procedure.

The algorithm generally follows these steps:

1. **Base Case**: If p ≥ r (or alternatively p≠r), then the array section is either empty or has one element, which is already sorted.
2. **Recursive Case**: If p< r, the procedure calculates the midpoint q and recursively sorts the two halves:
    - Sort the left half: MERGE-SORT(A, p, q)
    - Sort the right half: MERGE-SORT(A, q + 1, r)

Therefore, as long as the initial call to MERGE-SORT(A, 1, n) is valid (with n≥1), the condition "if p≠ r" suffices because it prevents any recursive calls from having p> r. This prevents the algorithm from attempting to sort an empty subarray and guarantees that every subarray processed is valid and well-defined, ensuring the algorithm behaves correctly.

Question-3

State a loop invariant for the while loop of lines 12318 of the MERGE procedure. Show how to use it, along with the while loops of lines 20323 and 24327, to prove that the MERGE procedure is correct.

Answer:

To establish a loop invariant for the while loop of lines 12-18 of the MERGE procedure, we first need to clarify the general purpose of the MERGE procedure, which is to combine two sorted arrays into a single sorted array. Here's a typical structure of such a procedure:

Loop Invariant for Lines 12-18

Invariant: At the start of each iteration of the while loop from lines 12-18, the subarrays A[p..i] and B[q..j] are both sorted, and all elements from A[p..i] and B[q..j] have been placed into the merged array C[0..k].

Explanation:

- **Initialization**: Before the first iteration, both subarrays are empty, and C has no elements. This is trivially true.

- **Maintenance:** During each iteration, we compare elements from A and B, adding the smaller one to C, thus preserving the sorted order in C.
- **Termination:** When the loop terminates, all elements from A and B will have been added to C, maintaining the overall sorted order.

Proving Correctness of the MERGE Procedure

1. **Lines 12-18 (First While Loop):**
    - The loop runs while there are remaining elements in both A and B.
    - The loop invariant ensures that the merged array C contains the smallest elements in sorted order.
2. **Lines 20-23 (Second While Loop for Remaining A):**
    - After the first loop, if elements remain in A, they are all greater than the last added element in C (due to the invariant).
    - The second loop appends the remaining elements of A to C, maintaining the sorted order.
3. **Lines 24-27 (Third While Loop for Remaining B):**
    - Similarly, if elements remain in B, they are also greater than the last added element in C.
    - This loop adds the remaining elements of B to C, again preserving the sorted order.

Question-4

Use mathematical induction to show that when n □ 2 is an exact power of 2, the solution of the recurrence T .n/ D (2 if n D 2 ; 2T .n=2/ C n if n > 2 is T .n/ D n lg n.

Answer:

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

T(2^{k+1}) = 2T ($\frac{2^{k+1}}{2}$) + 2^{k+1}

= 2T (2^k) + 2^{k+1}

= 2(2^klg(2^k)) + 2^{k+1}

= $k2^{k+1} + 2^{k+1}$

= $(k+1)2^{k+1}$

= 2^{k+1} lg(2^{k+1})

= n lg(n)

Question-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:

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) = Θ(1) if n ≤ c

T(n − 1) + I(n) otherwise

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)

1: n1 = q − p + 1

2: n2 = r − q

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

4: for i = 1 to n1 do

5: L[i] = A[p + i − 1]

6: end for

7: for j = 1 to n2 do

8: R[j] = A[q + j]

9: end for 10: i = 1

11: j = 1

12: k = p

13: while i ≠ n1 + 1 and j ≠ n2 + 1 do

14: if L[i] ≤ R[j] then

15: A[k] = L[i]

16: i = i + 1

17: else A[k] = R[j]

18: j = j + 1

19: end if

20: k = k + 1

21: end while

22: if i == n1 + 1 then

23: for m = j to n2 do

24: A[k] = R[m]

25: k = k + 1

26: end for

27: end if

28: if j == n2 + 1 then

29: for m = i to n1 do

30: A[k] = L[m]

31: k = k + 1

32: end for

33: end if

Question-7

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 AŒ1 W 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/?

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.

Question-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 ,.n lg n/ time in the worst case.

Answer:

If the running time constraint was not there, we might have intuitively used the brute-force method of picking one element at a time and iterating over the set to check if there exists another element in the set such that sum of them is xx. Even in average case, this brute-force algorithm will run at Θ(n2)Θ(n2) time (as we have to iterate over the set for each element).

But we have to think of a Θ(nlg⁡n)Θ(nlgn)-time algorithm.

Sum-Search (S,x)

i. Merge-Sort(S,1,S.length)
ii. for i=1 to S.length
iii. index=Binary-Search(S,x−S[i])
iv. if index≠NIL and index≠i
v. return true
vi. return false

Note the additional conditional check for index not being equal to ii in line 4. This is necessary for avoiding cases where the expected sum, xx, is twice of any element. An algorithm without this conditional check, will wrongly return true in such cases. This was pointed out by Ravi in the comments.

CHAPTER-3

Exercise-3.1

Question-1

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

Answer:

To modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3, adjust the start and end indices of the outer loop.

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:

Running Time Analysis

1. **Outer Loop**: The outer loop runs n−1 times, where n is the number of elements in the array.
2. **Finding the Minimum**: For each iteration of the outer loop at position iii:
    - The inner loop runs from i to n−1, checking each element to find the minimum. This takes O(n−i) time.
    - The total time spent finding the minimum across all iterations can be calculated as follows:
        - For i =0 : n−0 = n
        - For i =1 : n−1
        - For i =2 : n−2
        - ...
        - For i= n−2 : n−(n−2) = 2
        - For i= n−1 : n−(n−1) = 1

    The total time for finding the minimum over all iterations is:

    (n)+(n−1)+(n−2)+…+2+1 = n(n−1)/2

    This simplifies to O(n^2).

3. **Swapping**: Each swap operation is constant time, O(1), and since we perform at most n−1 swaps, this contributes an O(n) factor, which is negligible compared to O(n^2).

Conclusion

Putting it all together, the dominant factor in the running time of selection sort is the process of finding the minimum element, which takes O(n^2) time. Thus, the overall time complexity of selection sort is:

O(n^2)

Selection sort, therefore, has a quadratic time complexity for all cases (best, average, and worst), making it inefficient for large lists compared to more advanced algorithms like quicksort or merge sort.

Question-3

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

Answer:

The best running time for an algorithm is the smallest amount of time it takes to run on inputs of a specific size. To improve the best-case running time of a sorting algorithm, you can use a loop invariant.

Here are the best-case running times for some sorting algorithms:

- Bubble Sort: O(n)
- Quick Sort: O(n log n)
- Merge Sort: O(n log n)
- Insertion Sort: O(n)

The bubble sort algorithm is useful for determining if a list is sorted. It has a best-case running time of O(n), and it requires little memory and only a few lines of code.