Abbottabad University of Science and Technology

Department of Computer Science

SUBMITTED TO

Sir Jamal Abdul Ahad

SUBMITTED BY

Bibi Hajra

ROLL NUMBER

14640

SUBJECT

Data Structures

DATE

31/10/2024

ASSIGNMENT #1

Chapter #1

EXERCISE #1

1.1-1 Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

ANSWER

Library Bookshelf Organization

A librarian needs to organize books on a shelf alphabetically by author's last name, then by title. The books are currently in random order.

Sorting Requirements:

1. Sort books by author's last name (A-Z).
2. For books with the same author, sort by title (A-Z).

Benefits:

1. Easy location of specific books.
2. Efficient browsing for readers.
3. Simplified maintenance and updates.

1.1-2 Other than speed, what other measures of eficiency might you need to consider in a real-world setting?

ANSWER

These factors ensure a comprehensive understanding of performance in real-world settings which are efficiency measures beyond speed:

1. Resource usage
2. Cost
3. Quality/Accuracy
4. Scalability
5. User experience
6. Reliability
7. Security

8. Environmental impact

9. Social impact

1.1-3 Select a data structure that you have seen, and discuss its strengths and limitations.

ANSWER

Strengths Of Linked Lists:

1. Simpler addition and removal of elements, O(1)$O(1)$ time complexity.
2. Does not need contiguous memory space.
3. New element can be easily inserted in any location.

Limitations Of Linked Lists :

1. Accessing an element by index or by value means traversing the list, O(n)$O(n)$ time complexity.
2. Additional memory is required for storing the address (pointer) of the next/previous element.

Strengths Of Array:

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

Limitations Of Array:

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

1.1-4 How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

ANSWER

They are similar since both problems can be modeled by a graph with weighted edges and involve minimizing distance, or weight, of a walk on the graph. They are different because the shortest path problem considers only two vertices, whereas the traveling salesman problem considers minimizing the weight of a path that must include many vertices and end where it began.

1.1-5 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 enough.

ANSWER:

Problem requiring the best solution:

1. Medical diagnosis
2. Financial transactions
3. GPS navigation
4. Air traffic control
5. Space exploration

Approximate Solution :

1. Image filters
2. Music recommendations
3. Weather forecasts
4. Product suggestions
5. Social media feeds

1.1-6 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:

A real-world example of this kind of problem is **scheduling appointments in a hospital or clinic**:

Entire Input Available in Advance:
In some cases, a hospital may have all scheduled appointments for a particular day or week booked well in advance. This allows the hospital staff to plan resources, assign rooms, and manage staff schedules effectively, as they have all the required information at the start.

Input Arrives Over Time:
At other times, new appointments or emergency cases are added throughout the day. This means the schedule is constantly changing, requiring staff to adjust plans, reallocate resources, and accommodate new patients as they arrive. Here, the input is incomplete at the beginning, and the hospital has to make scheduling decisions on the fly as new information arrives.

1.2 Algorithms as a technology

Exercises

1.2-1 Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

ANSWER:

Example:

A **weather forecasting system** requires algorithms to predict future weather patterns.

Function of Algorithms:

- **Data collection algorithms** gather information from sensors, satellites, and other sources to create a picture of current weather.
- **Predictive algorithms** analyze historical and real-time data to forecast future conditions, considering patterns and trends.
- **Climate modeling algorithms** simulate different scenarios to improve prediction accuracy over time.

These algorithms help provide reliable forecasts, which are crucial for safety, agriculture, and day-to-day planning.

1.2-2 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:

To determine for which values of nnn the inequality 8n2<64nlog⁡2(n)8n^2 < 64n \log_2(n)8n2<64nlog2(n) holds, we can simplify the inequality:

1. Start with the inequality:

   $8n^2$<64nlog_2 (n)

2. Divide both sides by 8:

   n^2<8nlog_2(n)

3. Rearranging gives:

   n<8$\log_2$(n)

Now, we need to find the values of n for which this holds. By testing values, we find that:

- The inequality holds true for $n \leq 43$

In summary, **insertion sort is faster than merge sort for sorting up to 43 items**. For larger values of n, merge sort becomes the more efficient choice.

1.2-3 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 find the smallest n such that an algorithm with a runtime of $100n^2$runs faster than an algorithm with a runtime of $2n^2$, we set up the inequality:

$$\mathbf{100n^2 < 2^n}$$

Through testing, we find that the smallest value of **n** that satisfies this inequality is approximately **n=15**, since $100 \times 15^2 = 22500$, while $2^{15} = 327682$. For values of n greater than 15, 2^n grows much faster than $100n^2$, making the quadratic algorithm run faster.

Problems

1-1 Comparison of running times

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

ANSWER:

We assume a 30 day month and 365 day year.

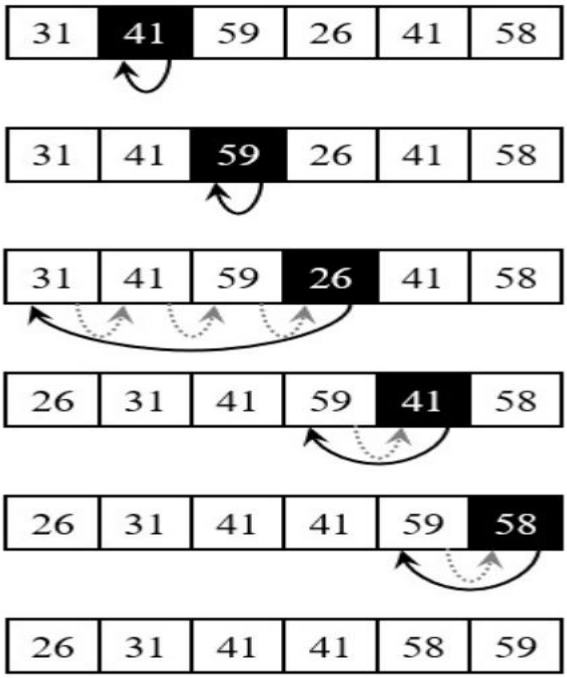
	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	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

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 {31; 41; 59; 26; 41; 58}.

ANSWER:



2.1-2 Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array A [1: n]. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in A [1: n].

ANSWER:

Loop Invarient for SUM-ARRAY:

Loop Invariant: At the start of each loop iteration, `total` contains the sum of elements A[1] to A[i−1].

PROOF:

1. **Initialization**:
   - Before the loop starts (when i=1), `total` is initialized to 0. The sum of A[1]to A[0]is 0, so the invariant holds.
2. **Maintenance**:

- Assume the invariant is true for i=k (i.e., `total` is the sum of A[1] to A[k−1]).
- In the next iteration (i=k+1), we add A[k] to `total`, so now `total` is the sum of A[1]to A[k]. The invariant still holds.

3. **Termination**:
   - The loop finishes after processing A[n] (when i=n+1). At this point, `total` holds the sum of A[1] to A[n].

CONCLUSION:

The **SUM-ARRAY** procedure correctly returns the sum of the numbers in A[1:n].

2.1-3 Rewrite the I NSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

ANSWER:

The rewritten INSERTION-SORT procedure to sort in monotonically decreasing order:

INSERTION-SORT(A)

1. for i = 2 to A.length

2. key = A[i]

3. j = i - 1

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

5. A[j + 1] = A[j]

6. A[j + 1] = key

2.1-4 Consider the searching problem:

Input: A sequence of n numbers {a1; a2; : : : ; an}stored in array A[1:n]and a value x.

Output: An index i such that x equals A[i] or the special value NIL if x does not appear in A.

Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

ANSWER:

Algorithm: Linear-Search (A, v)

1: i = NIL

2: for j = 1 to A.length do

3: if A[j] = v then

4: i = j

5: return i

6: end if

7: end for

8: return i

2.1-5 Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays A[0:n-1]and B[0:n-1], where each element is either 0 or 1, a =$\sum_{i=0}^{n-1} A[i] .2^i$, and b =$\sum_{i=0}^{n-1} B[i] . 2^i$. 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 =$\sum_{i=0}^{n-1} C[i]$.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:

Input: two n-element arrays A and B containing the binary digits of two numbers a and b.

Output: an (n + 1)-element array C containing the binary digits of a + b.

Algorithm: Adding n-bit Binary Integers

1. carry = 0
2. for i=n to 1 do
3. C[i + 1] = (A[i] + B[i] + carry) (mod 2)
4. if A[i] + B[i] + carry ≥ 2 then
5. carry = 1
6. else
7. carry = 0
8. end if
9. end for
10. C[1] = carry

2.2 Analyzing algorithms

2.2-1 Express the function n^3=1000 +100n^2- 100n + 3 in terms of Θ –notation.

ANSWER:

The highest order of *n* term of the function ignoring the constant coefficient is n3n3. So, the function in Θ-notation will be Θ (n3).

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

.

2.2-2 Consider sorting n numbers stored in array A[1:n] by ûrst ûnding the smallest element of A[1:n] and exchanging it with the element in A[1:n]. Then ûnd the smallest element of A[1:n], and exchange it with A[1:n]. Then ûnd the smallest element of A[1:n], and exchange it with A[1:n]. Continue in this manner for the first n-1 elements of A. Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first n-1 elements, rather than for all n elements? Give the worst-case running time of selection sort in , Θ-notation. Is the best-case running time any better ?

ANSWER:

Input: An n-element array A.

Output: The array A with its elements rearranged into increasing order.

The loop invariant of selection sort is as follows: At each iteration of the for loop of lines 1 through 10, the subarray A [1: i − 1] contains the i − 1 smallest elements of A in increasing order. After n − 1 iterations of the loop, the n − 1 smallest elements of A are in the first n − 1 positions of A in increasing order, so the n^{th}element is necessarily the largest element. Therefore we do not need to run the loop a final time. The best-case and worst-case running times of selection sort are Θ (n^2). This is because regardless of how the elements are initially arranged, on the i^{th} iteration of the main for loop the algorithm always inspects each of the remaining n−i elements to find the smallest one remaining.

Algorithm for Selection Sort:

1. for i = 1 to n − 1 do
2. min = i
3. for j = i + 1 to n do
4. // Find the index of the i^{th} smallest element if A[j] < A[min] then
5. min = j
6. end if
7. end for
8. Swap A[min] and A[i]
9. end for

This yields a running time of

$\sum_{i=1}^{n-1} n(n-1) \sum_{i-1}^{n-1} i = n^2$-n$-\frac{n2-n}{2} = \frac{n2-n}{2} = \Theta\ (\ n^2)$

2.2-3 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 , Θ -notation, give the average-case and worst-case running times of linear search. Justify your answers.

ANSWER:

In a linear search, we look for a target element in an unsorted array by checking each element in sequence until we find it or reach the end of the array.

AVERAGE CASE ANALYSIS:

If the target element is equally likely to be any element in the array, then, on average, we expect to find it halfway through the array. If there are n elements in the array, then on average, we would check n/2 elements before finding the target.

Thus, in the average case, the running time of linear search is proportional to n, which we express in Θ-notation as:

Average-case running time=Θ (n)

WORST CASE:

In the worst case, the element might be at the very end of the array or might not be present in the array at all. In this case, we would have to check all n elements in the array.

Therefore, in the worst case, the running time of linear search is also proportional to n, which we express in Θ-notation as:

Worst-case running time=Θ (n)

JUSTIFICATION:

- **Average case**: Since each element is equally likely to be the target, we check about half the array elements on average, leading to an average-case complexity of Θ (n).
- **Worst case**: In the worst scenario, we check all n elements, which also leads to a worst-case complexity of Θ (n).

Thus, both the average-case and worst-case running times for linear search are Θ (n).

2.2-4 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.

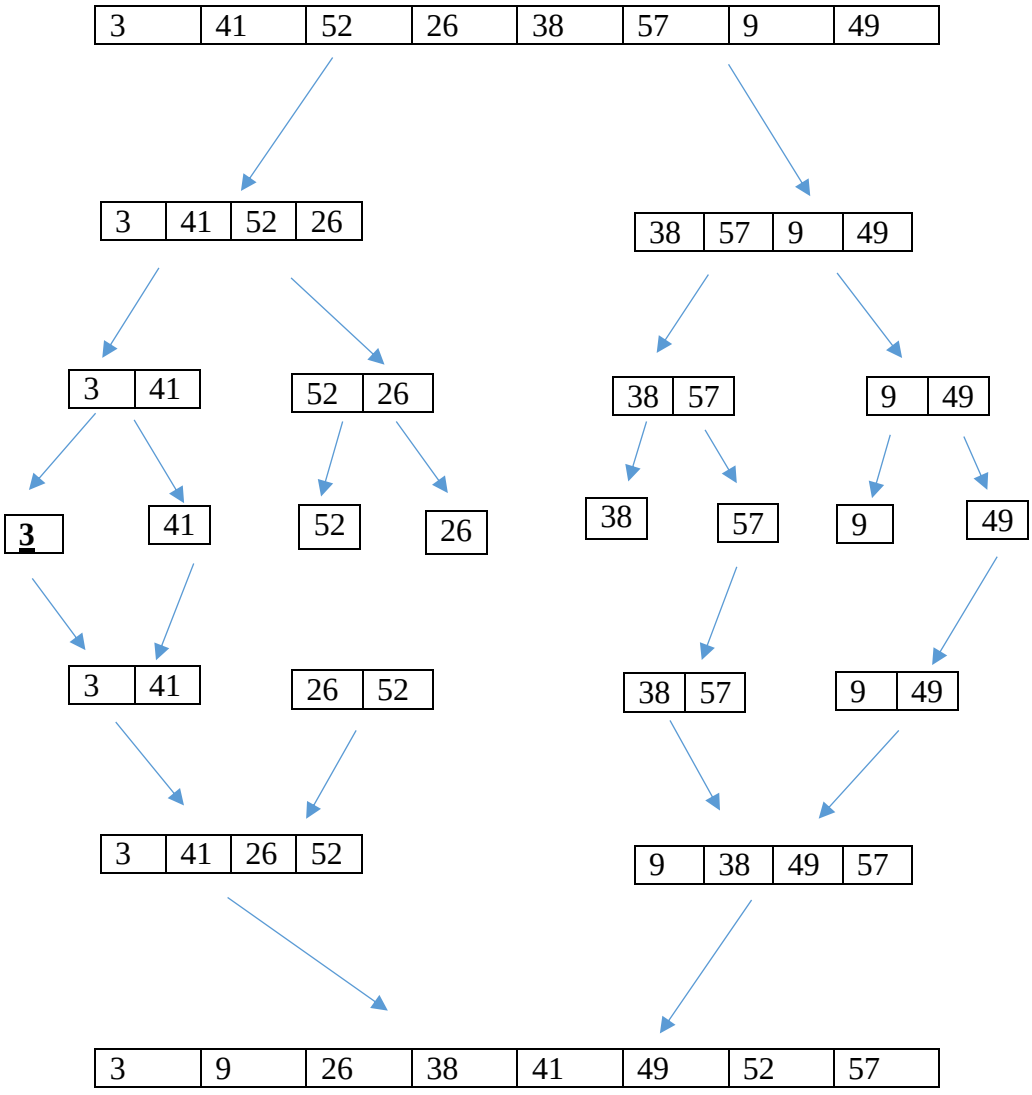
For example: we could modify selection sort to first randomly permute the elements of A, then check if they are in sorted order. If they are, output A. Otherwise run selection sort as usual. In the best case, this modified algorithm will have running time Θ (n).

2.3 Designing algorithms

Exercises

2.3-1 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:



2.3-2 The test in line 1 of the MERGE-SORT procedure should read "if $p < r$" instead of "if $p \neq r$." If MERGE-SORT is invoked with $p > r$, then the subarray A [p.. r] is empty. It can be argued that as long as the initial call to MERGE-SORT (A, 1, n) has $n \geq 1$, the condition "if $p < r$" is sufficient to guarantee that no recursive call will have $p > r$.

ANSWER:

In the MERGE-SORT algorithm, the condition should be "if $p<r$ "instead of "if $p\neq r$" This is because:

- If p=r, the subarray has only one element and doesn't need sorting.
- If p>r, the subarray is empty and can be ignored.

Using "if $p<r$ is enough to ensure that the recursion works correctly without empty subarrays, as long as the initial call has $n\geq1$. This condition guarantees that recursion stops when subarrays reach a single element, avoiding any invalid ranges.

2.3-3 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:

A loop invariant is a condition that holds true before and after each iteration of a loop. It's a wa y to ensure the correctness of an algorithm by making sure that certain properties remain unch anged throughout the execution of the loop.

Key Points:

- **Initialization**: The invariant must be true before the loop starts.
- **Maintenance:** The invariant must hold true before and after each iteration of the loop.
- **Termination:** When the loop terminates, the invariant can be used to show that the algori thm has correctly solved the problem.

Loop Invariant for Lines 12-18 of the MERGE Procedure

Loop Invariant: At the start of each iteration of the while loop (lines 12-18), the subarray M[1…k] contains the k smallest elements from the combined arrays L[1…n] and R[1…m].

Proof of Correctness Using the Loop Invariant

1. **Initialization**:
    - Before the first iteration (k=0k = 0k=0), MMM is empty, which trivially contains the smallest 0 elements from LLL and RRR. The invariant holds.

2. **Maintenance**:
    - During each iteration, one element from either L or R is added to M. Since both L and R are sorted, the invariant continues to hold after each addition.
3. **Termination**:
    - The loop terminates when either L or R is exhausted. Lines 20-23 and 24-27 handle copying any remaining elements from the other array into M. These remaining elements will be greater than those already in M, preserving the sorted order.

Conclusion:

The loop invariant proves that the MERGE procedure correctly merges two sorted arrays into one sorted array.

2.3-4 Use mathematical induction to show that when n ≥2 is an exact power of 2, the solution of the recurrence

$$\boldsymbol{T(n) = \begin{cases} 2 & \textit{if } n = 2 \\ 2T(n/2) & \textit{if } n > 2 \end{cases}}$$

is T(n)=n lg n.

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

$$\boldsymbol{T(n) = \begin{cases} \Theta(1) & \textit{if } n \leq c \\ T(n-1) + I(n) & \textit{otherwise} \end{cases}}$$

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 : 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 6= n1 + 1 and j 6= 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

2.3-5 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 = $\left\lceil \frac{a+b}{2} \right\rceil$

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 falls 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)).

2.3-6 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 Θ 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.

2.3-7 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:

We can see that the while loop gets run at most O(n) times, as the quantity j−i starts at n−1 and decreases at each step. Also, since the body only consists of a constant amount of work, all of lines 2-15 takes only O(n) time. So, the runtime is dominated by the time to perform the sort, which is Θ(n lg(n)). We will prove correctness by a mutual induction. Let $m_{i,j}$ be the proposition A[i] + A[j] < S and $M_{i,j}$be the proposition A[i] + A[j] > S. Note that because the array is sorted, $m_{i,j} \Rightarrow \forall k < j$, $m_{i,k}$, and $M_{i,j} \Rightarrow \forall k > i$, $M_{i,k}$. Our program will obviously only output true in the case that there is a valid i and j. Now, suppose that our program output false, even though there were some i, j that was not considered for which A[i] + A[j] = S. If we have i > j, then swap the two, and the sum will not change, so, assume i ≤ j. we now have two cases: Case 1 ∃k,(i, k) was considered and j < k. In this case, we take the smallest

1. Use Merge Sort to sort the array A in time Θ(n lg(n))
2. i = 1
3. j = n
4. while i < j do
5. if A[j] + A[j] = S then
6. return true
7. end if
8. if A[i] + A[j] < S then
9. i = i + 1
10. end if
11. if A[i] + A[j] > S then
12. j = j − 1
13. end if
14. end while
15. return false

Such k. The fact that this is nonzero meant that immediately after considering it, we considered (i+1,k) which means $m_{i,k}$this means $m_{i,j}$.

Case 2 ∃k,(k, j) was considered and k < i. In this case, we take the largest such k. The fact that this is nonzero meant that immediately after considering it, we considered (k,j-1) which means $M_{k,j}$ this means $M_{i,j}$ Note that one of these two cases must be true since the set of considered points separates {(m, m') : m ≤ m' < n} into at most two regions. If you are in the region that contains (1, 1)(if nonempty) then you are in Case 1. If you are in the region that contains (n, n) (if non-empty) then you are in case 2.

2.3-8 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:

Algorithm description for finding two elements in a set S that sum to a given integer x, with a time complexity of Θ (n log n):

1. **Sort S**: Begin by sorting the set SSS in ascending order. Sorting takes Θ(nlogn) time.
2. **Use Two Pointers**: Initialize two pointers — one at the start and one at the end of the sorted set.
3. **Iterate and Check**:
    - Calculate the sum of the elements at the two pointers.
    - If the sum equals x, return **true** (the pair is found).
    - If the sum is less than x, move the left pointer one step to the right to increase the sum.
    - If the sum is greater than x, move the right pointer one step to the left to decrease the sum.
4. **End Condition**: If the pointers cross without finding a pair, return **false**.

This approach ensures Θ (n log n) time complexity due to the initial sort, followed by a linear Θ (n) pass for the two-pointer search.

1. **End Condition**: If the pointers cross without finding a pair, return **false**.

This approach ensures Θ (n log n) time complexity due to the initial sort, followed by a linear Θ (n) pass for the two-pointer search.

CHAPTER # 3

Characterizing Running Times

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:

To modify the lower-bound argument for insertion sort for input sizes that are not multiples of 3:

1. **Insertion Sort Basics**: Insertion sort builds a sorted array by inserting elements one at a time, requiring comparisons with already sorted elements.
2. **Worst-Case Scenario**: In the worst case (e.g., the array is sorted in reverse), each new element can require comparisons with all previously sorted elements.
3. **Total Comparisons**: For an array of size n, the number of comparisons is:

   $T(n) = 0+1+2+\ldots+(n-1) = \frac{(n-1)n}{2} = \Theta(n^2)$

   Non-Multiples of 3: The worst-case scenario applies regardless of whether n is a multiple of 3, so the lower bound remains Ω (n^2).

Conclusion: The lower bound for insertion sort is Θ (n^2) for any input size n, including those not divisible by 3.

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:

The reasoning behind the selection sort's Θ (n^2) running time comes from the way the algorithm repeatedly scans through unsorted elements to find the smallest (or largest) one and then places it in the correct position. Here's the breakdown:

1. **Finding the Minimum Element**: In each pass, selection sort goes through all the unsorted elements to find the minimum. For an array of n elements:
    - On the first pass, it compares n−1n - 1n−1 elements.
    - On the second pass, it compares n−2n - 2n−2 elements.
    - This continues until only one element remains.

    The total comparisons in all passes add up to:

    $(n-1) + (n-2) + \cdots + 1 = \frac{\mathbf{n(n-1)}}{2} = \Theta(n2)$

2. **Swapping the Minimum Element**: After each pass, selection sort swaps the minimum element found with the first unsorted element. This happens once per pass, so there are n−1n - 1n−1 swaps in total, or Θ (n).

    Total Complexity

Since the comparisons dominate, selection sort's overall time complexity is Θ (n2), the same in best, worst, and average cases because it always performs the same number of comparisons.

3.1-3 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 ¸n largest values start in the αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the α n largest values must pass through each of the middle (1-2 α)n array positions?

ANSWER:

To generalize the lower-bound argument for insertion sort, assume the αn\alpha nαn largest elements start in the last αn positions, where $0<\alpha<\frac{1}{2}$These largest elements need to move through the middle (1−2α)n positions to reach their sorted positions.

The number of moves is roughly proportional to:

$$\alpha\, n \times (1-2\alpha)$$

Maximizing this product gives $\alpha=\frac{1}{4}$,which maximizes the passes each large element must make through the middle section.