ASSIGNMENT NO.1

DATA STRUCTURE AND ALGORITHM

ZUNAIRA SATTAR(14674)

BSCS 3RD

QUESTION .1

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

ANSWER:

SORTING EXAMPLE:

LIBRARY BOOK ORGANIZATION

A librarian at a university library needs to organize 5,000 books on shelves by author's last name and title. To efficiently locate books, they must be sorted alphabetically.

REQUIREMENTS:

- Sort books by author's last name
- Sort books with the same author by title
- Place books on shelves in sorted order

REAL-WORLD IMPACT:

- Efficient book retrieval for students and faculty
- Reduced search time for librarians
- Improved organization and maintenance

SHORTEST DISTANCE EXAMPLE:

RIDE-SHARING ROUTE OPTIMIZATION

A ride-sharing company wants to minimize fuel consumption and reduce travel time for drivers. They need to find the shortest route between a driver's current location and a passenger's pickup location, considering traffic and road conditions.

QUESTION.2

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

ANSWER:

SPACE COMPLEXITY

- Memory usage: Amount of memory required.
- Storage requirements: Disk space or database size.

SCALABILITY

- Horizontal scaling: Ability to handle increased load by adding resources.
- Vertical scaling: Ability to handle increased load by upgrading resources.

RELIABILITY AND ROBUSTNESS

- Fault tolerance: Ability to recover from errors or failures.
- Error rate: Frequency of errors or inaccuracies.

ADAPTABILITY AND FLEXIBILITY

- Modularity: Ease of modifying or replacing components.
- Reusability: Ability to reuse code or components.

MAINTAINABILITY

- Code readability: Ease of understanding code.
- Code maintainability: Ease of modifying or updating code.

RESOURCE UTILIZATION

- CPU usage: Processor utilization.
- I/O operations: Input/Output operations per second.

PARALLELISM AND CONCURRENCY

- Thread safety: Ability to handle concurrent access.
- Parallel processing: Ability to utilize multiple processors.

ENERGY EFFICIENCY

- Power consumption: Energy usage.
- Battery life: Impact on device battery life.

SECURITY

- Data protection: Protection against unauthorized access.
- Vulnerability: Resistance to security threats.

QUESTION.3

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

ANSWER:

Arrays are a collection of elements of the same data type stored in contiguous memory locations.

STRENGTHS:

- Fast access: O(1) time complexity for accessing elements
- Efficient storage: Stores multiple elements in a single block
- Cache-friendly: Adjacent elements are stored together
- Simple implementation

LIMITATIONS:

- Fixed size: Difficult to resize or dynamic allocation
- Insertion/deletion: O(n) time complexity for shifting elements
- Search: O(n) time complexity for finding specific elements
- No built-in sorting or ordering

REAL-WORLD APPLICATIONS:

- Image processing (pixel arrays)
- Database records
- Scientific simulations
- Game development (game state arrays)

EXAMPLE IN DAILY LIFE:

Imagine a bookshelf with 10 slots. You can quickly access any book by its slot number (fast access). However, if you want to insert a new book between existing ones, you'll need to shift all books after it (inefficient insertion).

QUESTION.4

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

ANSWER:

SIMILARITIES:

- Graph-based problems: Both involve finding optimal paths in a graph.
- Optimization goals: Both aim to minimize a cost function (distance or travel time).
- Network analysis: Both require analyzing connections between nodes.

DIFFERENCES:

SHORTEST-PATH PROBLEM:

- Single-source, single-destination: Find the shortest path between two nodes.
- Unidirectional: Travel from source to destination.
- No repetition: Visit each node only once.

TRAVELING SALESPERSON PROBLEM (TSP):

- Multiple destinations: Visit multiple nodes and return to the starting point.
- Round-trip: Travel to multiple destinations and return to the origin.
- Repetition avoidance: Visit each node exactly once before returning.

QUESTION .5

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

up with one in which <approximately= the best solution is good enough.

ANSWER:

PROBLEM 1: REQUIRES THE BEST SOLUTION

MEDICAL DIAGNOSIS AND TREATMENT PLANNING

A hospital's AI-powered diagnostic system needs to identify the most effective treatment plan for a patient with a rare disease.

Reason of Why the best solution is required:

- Patient's life depends on accurate diagnosis and treatment.
- Inaccurate diagnosis can lead to harmful or ineffective treatment.
- Suboptimal treatment plans can result in prolonged suffering or reduced quality of life.

Problem 2: Approximately the Best Solution is Good Enough

Personalized Product Recommendations

An online retailer's recommendation engine suggests products based on customers' browsing history and purchase behavior.

Reason of Why an approximate solution is sufficient:

- Customers may still find value in slightly less accurate recommendations.
- Small improvements in recommendation accuracy have diminishing returns.
- Computational resources and development time can be balanced against solution quality.

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:

Problem: Logistics and Supply Chain Management for E-commerce Companies

E-commerce companies like Amazon, Walmart, or Alibaba need to manage their logistics and supply chain efficiently to ensure timely delivery of products to customers.

VARIABILITY IN INPUT AVAILABILITY:

1. Entire input available in advance: During peak sales seasons (e.g., holidays), companies can anticipate demand and plan inventory, shipping, and delivery schedules accordingly.

2. INPUT ARRIVES OVER TIME:

- Unpredictable demand: Unexpected changes in consumer behavior, weather, or global events can alter demand patterns.
- Real-time orders: Customers place orders continuously, requiring dynamic adjustments to logistics and supply chain management.
- Inventory updates: Inventory levels fluctuate as products are sold, returned, or restocked.

QUESTION.7

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

ANSWER:

Example Application: Google Maps Navigation

Google Maps relies heavily on algorithms to provide efficient and accurate navigation. Some key algorithms involved:

1. ROUTING ALGORITHMS:

These algorithms calculate the shortest or most efficient route between two points, considering factors like:

- Road network topology
- Traffic patterns
- Road conditions
- Time of day

2. LOCATION-BASED ALGORITHMS:

These algorithms determine the user's current location and nearby points of interest.

3. OPTIMIZATION ALGORITHMS:

These algorithms optimize routes for:

- Minimum travel time
- Minimum distance
- Avoiding traffic congestion
- Avoiding toll roads

4. GRAPH ALGORITHMS:

These algorithms manage the massive graph of road networks, allowing for efficient route calculation and updating.

FUNCTION OF ALGORITHMS:

The algorithms in Google Maps work together to:

- Calculate routes: Find the most efficient route between two points.
- Provide real-time navigation: Update routes based on changing traffic conditions.
- Locate nearby points of interest: Suggest relevant destinations.
- Optimize traffic flow: Reduce congestion by distributing traffic across multiple routes.

QUESTION.8

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:

Since you are to find when insertion sort beats merge sort

8n^2<=64nlogn

n^2<=8nlogn

n<=8logn

On solving n-8logn = 0 you get

n = 43.411

So for n<=43 insertion sort works better than merge sort.

QUESTION.9

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 value of n, we need to solve:

$100n^2 < 2^n$

LET'S ANALYZE:

For small n, 2^n grows slower than $100n^2$.

- As n increases, 2^n grows exponentially and eventually surpasses $100n^2$.

Testing values of n:

n = 1: $100(1)^2 = 100$, $2^1 = 2$ ($100n^2 > 2^n$)

n = 2: $100(2)^2 = 400$, $2^2 = 4$ ($100n^2 > 2^n$)

n = 3: $100(3)^2 = 900$, $2^3 = 8$ ($100n^2 > 2^n$)

n = 4: $100(4)^2 = 1600$, $2^4 = 16$ ($100n^2 > 2^n$)

n = 5: $100(5)^2 = 2500$, $2^5 = 32$ ($100n^2 > 2^n$)

n = 6: $100(6)^2 = 3600$, $2^6 = 64$ ($100n^2 > 2^n$)

n = 7: $100(7)^2 = 4900$, $2^7 = 128$ ($100n^2 > 2^n$)

n = 8: $100(8)^2 = 6400$, $2^8 = 256$ ($100n^2 > 2^n$)n = 9: $100(9)^2 = 8100$, $2^9 = 512$ ($100n^2 > 2^n$)

n = 10: $100(10)^2 = 10,000$, $2^{10} = 1024$ ($100n^2 > 2^n$)

n = 11: $100(11)^2 = 12,100$, $2^{11} = 2048$ ($100n^2 > 2^n$)

n = 12: $100(12)^2 = 14,400$, $2^{12} = 4096$ ($100n^2 > 2^n$)

n = 13: $100(13)^2 = 16,900$, $2^{13} = 8192$ ($100n^2 > 2^n$)

n = 14: $100(14)^2 = 19,600$, $2^{14} = 16,384$ ($100n^2 < 2^n$)

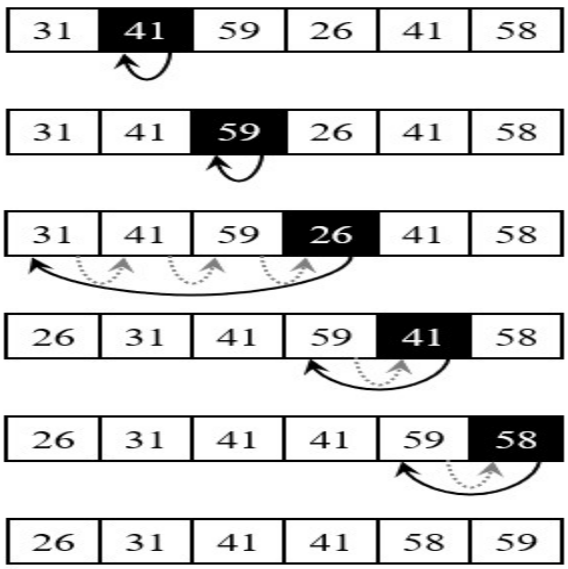
Smallest value of n: 14

At n = 14, the algorithm with running time 2^n becomes faster than the algorithm with running time $100n^2$.

QUESTION: 10

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 .

ANSWER:



QUESTION: 11

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 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 each iteration of the for loop, sum equals the sum of the numbers in A[1..i-1]."

Initialization:

Before the first iteration (i = 1), sum = 0, which is the sum of the numbers in A[1..0] (an empty set).

Maintenance:

Assuming the loop invariant holds before an iteration (i), we have:

sum = sum of numbers in A[1..i-1]

After the iteration:

sum = sum + A[i]

Now, sum equals the sum of numbers in A[1..i], maintaining the invariant.

Termination:

After the loop finishes (i = n+1), sum equals the sum of numbers in A[1..n], which is the desired result.

Proof:

1. Initialization: sum = 0 before the first iteration.

2. Maintenance: sum updated correctly in each iteration.

3. Termination: sum equals the sum of numbers in A[1..n] after the loop.

Thus, SUM-ARRAY procedure correctly returns the sum of the numbers in A[1..n].

QUESTION NO 12

Express the function n 3 =1000 C 100n 2 □ 100n C 3 in terms of ,-notation.

ANSWER:

$n^3 = 1000C + 100n^2 + 100n + C_3$

in terms of Big O notation, we analyze the growth rate:

1. n^3 dominates the expression (highest degree term).

2. Lower-order terms ($100n^2$, $100n$, C_3) become negligible for large n.

Big O notation ignores lower-order terms and constants:

$n^3 = O(n^3)$

Alternatively, since the expression is exactly n^3, we can use:

$n^3 = \Theta(n^3)$

Θ-notation indicates the function grows exactly like n^3.

QUESTION:13

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 n □ 1 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 Pseudocode:

```
SELECTION-SORT(A, n)
  for i = 1 to n-1
    min_idx = i
    for j = i+1 to n
      if A[j] < A[min_idx]
        min_idx = j
    exchange A[i] with A[min_idx]
```

return A

Loop Invariant:

"At the start of each iteration of the outer loop (i), the subarray A[1..i-1] contains the smallest i-1 elements of A[1..n], in sorted order."

Why only n-1 iterations?

After the (n-1)th iteration, the largest element is already in its final position (A[n]). No need to iterate over the entire array.

Worst-Case Running Time:

$O(n^2)$

Reason: Two nested loops, each running up to n times.

Best-Case Running Time:

$O(n^2)$

Even if the input is already sorted, selection sort still performs the same number of comparisons.

Properties:

1. Unstable sort (may swap equal elements).

2. In-place sorting (no extra storage needed).

3. Not adaptive (doesn't take advantage of existing order).

Improvements:

None significantly improve worst-case performance. However, some variations (e.g., Heap Sort) have better average-case performance.

QUESTION:14

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? 34 Chapter 2 Getting Started Using ,-notation, give the average-case and worst-case running times of linear search. Justify your answers.

ANSWER:

On average, we would be examining n2n2 elements before finding our desired value. In the worst-case, we would need to look at all n elements. In both of these cases, n is the only significant figure therefore both running times are $\Theta(n)$.

QUESTION:15

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

ANSWER:

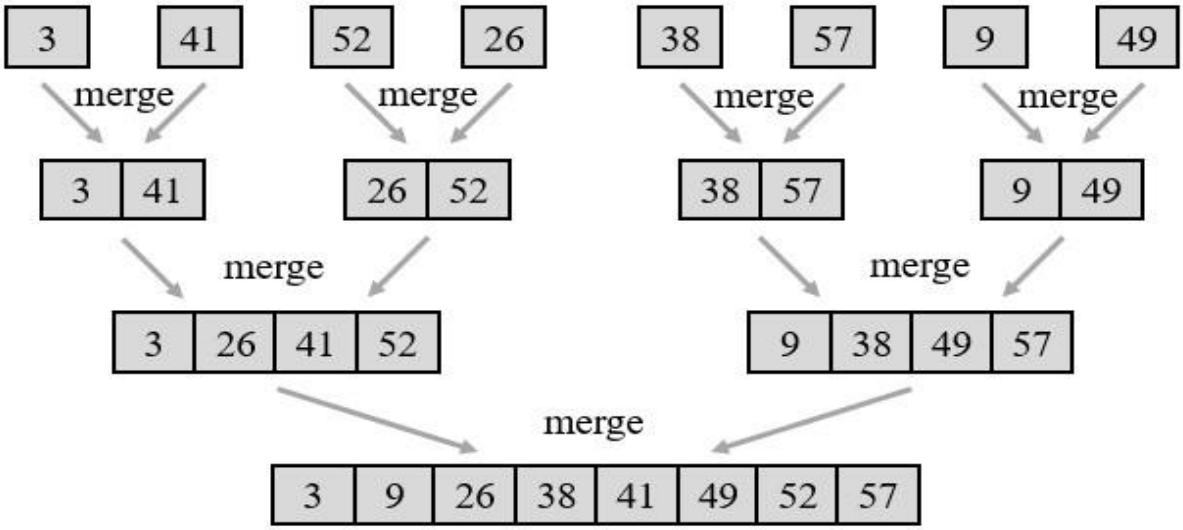
We can design any algorithm to treat its best-case scenario as a special case and return a predetermined solution.

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).

QUESTION:16

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;49).

ANSWER:



QUESTION:17

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:

The test r > p in line 1 of the MERGE-SORT procedure ensures that the subarray A[p..r] has at least two elements.

Why r > p is necessary:

1. If r == p, the subarray A[p..r] has only one element, making it already sorted.

2. If r < p, the subarray A[p..r] is empty, which is invalid.

Argument:

Given the initial call MERGE-SORT(A, 1, n) with n > 1, we can prove that r > p always holds:

1. Initially, p = 1 and r = n, so r > p.

2. In each recursive call, MERGE-SORT divides the array into two halves:

   - MERGE-SORT(A, p, q) and MERGE-SORT(A, q+1, r), where q = (p+r)/2

   - Since q < r, the second recursive call has p = q+1 and r = r, ensuring r > p.

3. This process continues until r == p, at which point the subarray has only one element and is already sorted.

Conclusion:

As long as the initial call has n > 1, the test r > p in line 1 of the MERGE-SORT procedure ensures that the subarray A[p..r] has at least two elements, preventing empty or single-element subarrays.

QUESTION:18

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:

```
1.  if p < r
2.      then q <-- [(p + r) / 2]
3.           MERGE-SORT(A, p, q)
4.           MERGE-SORT(A, q + 1, r)
5.          MERGE-SORT(A, p, q, r)
```

MERGE-SORT(A, p, q, r)

```
1.  n1 <-- q - p + 1
2.  n2 <-- r - q
3.  create arrays L[1 ... n1 + 1] and R[1 ... n2 + 1]
4.  for i <-- 1 to n1
5.  do L[i] <-- A[p + i - 1]
6.  for j <-- 1 to n2
7.  do R[j] <-- A[q + j]
8.  L[n1 + 1] <-- infinite
9.  R[n2 + 1] <-- infinite
10. i <-- 1
11. j <-- 1
12. for k <-- p to r
13. do if L[i] <= R[j]
```

14. then A[k] <-- L[i]
15. i <-- i + 1
16. else A[k] <-- R[j]
17. j <-- j + 1

Loop Invariant:

- Initialization: prior to the first iteration of the loop, we have k = p, so that subarray A[p ... k - 1] is empty. this empty subarray contains the k - p = 0 smallest elements of L and R, and since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.
- Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that l[i] <= R[j]. Then L[i] is the smallest element not yet copied back into A. Because A[p ... k - 1] contains the k - p smallest elements, after line 14 copies L[i] into A[k], the subarray A[p ... k] will contain the k - p + 1 smallest elements. Incrementing k(in the for loop update) and i(in line 15) re-establishes the loop invariant for the next iteration. If instead L[i] > R[j], then lines 16-17 perform the appropriate action to maintain the loop invariant.
- Termination: At termination, k = r + 1. By the loop invariant, the subarray A[p ... k - 1], which is A[p ... r], contains the k - p = r - p + 1 smallest elements of L[1 ... n1 + 1] and R[1 ... n2 + 1], in sorted order. The arrays L and R together contain n1 + n2 + 2 = r - p + 3 elements. All but the two largest elements have been copied back into A, and these two largest elements are the sentinels.

QUESTION:19

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:

Base Case

When n=2, (2)=2=2lg2. So, the solution holds for the initial step.

Inductive Step

Let's assume that there exists a k*k*, greater than 1, such that $T(2^k)= 2^k \lg 2^k$. We must prove that the formula holds for k+1*k*+1 too, i.e. $T(2^{k+1})= 2^{k+1}\lg 2^{k+1}$.

From our recurrence formula,

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

$=2T(2^k)+2\cdot 2^k$

$=2\cdot 2^k \lg 2^k+2\cdot 2^k$

$=2\cdot 2^k(\lg 2^k+1)$

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

$=2^{k+1}\lg 2^{k+1}$

This completes the inductive step.

Since both the base case and the inductive step have been performed, by mathematical induction, the statement $T(n)=n\lg n$ holds for all n that are exact power of 2.

QUESTION:20

You can also think of insertion sort as a recursive algorithm. In order to sort…..its worst-case running time.

ANSWER:

Recursive Insertion Sort Pseudocode

```
RECURSIVE-INSERTION-SORT(A, n)
  if n ≤ 1
    return  // base case: already sorted
  RECURSIVE-INSERTION-SORT(A, n-1)  // sort subarray A[1..n-1]
  INSERT(A[n], A[1..n-1])  // insert A[n] into sorted subarray
INSERT(key, A[1..n])
  i = n-1
  while i ≥ 0 and A[i] > key
    A[i+1] = A[i]
    i = i-1
  A[i+1] = key
```

Worst-Case Running Time Recurrence

Let T(n) be the worst-case running time of RECURSIVE-INSERTION-SORT.

T(n) = T(n-1) + O(n) // recursive call + insertion time

Solving the Recurrence

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

Expanding the recurrence:

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

= T(n-2) + O(n-1) + O(n)

= T(n-3) + O(n-2) + O(n-1) + O(n)

= T(1) + O(1) + O(2) + ... + O(n)

Simplifying

The sum of O(1) + O(2) + ... + O(n) can be rewritten as:

∑[i=1 to n] O(i)

Since O(i) represents the upper bound of the ith term, we can replace it with its maximum value, which is O(n).

∑[i=1 to n] O(i) ≤ ∑[i=1 to n] O(n)

= n × O(n)

= O(n^2)

Conclusion:

The worst-case running time of Recursive Insertion Sort is O(n^2).

QUESTION:21

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 0 lg n.

ANSWER:

Binary Search Pseudocode (Iterative)

```
BINARY-SEARCH(A, v, p, r)
  while p ≤ r
    q = (p + r) / 2
    if A[q] == v
      return q
    elif A[q] < v
      p = q + 1
    else
      r = q - 1
```

```
  return -1  // not found


# Example usage:
A = [1, 2, 3, 4, 5, 6, 7, 8, 9]
v = 5
index = BINARY-SEARCH(A, v, 0, len(A) - 1)
print(index)  # Output: 4
```

Binary Search Pseudocode (Recursive)

```
BINARY-SEARCH(A, v, p, r)
  if p > r
    return -1
  q = (p + r) / 2
  if A[q] == v
    return q
  elif A[q] < v
    return BINARY-SEARCH(A, v, q + 1, r)
  else
    return BINARY-SEARCH(A, v, p, q - 1)
```

Worst-Case Running Time Analysis

Let T(n) be the worst-case running time of binary search.

T(n) = T(n/2) + O(1) // divide and conquer

Solving the recurrence using Master Theorem:

T(n) = O(lg n)

Argument:

- Initial problem size: n
- Each iteration reduces problem size by half: n/2
- Number of iterations: lg n (since 2^lg n = n)
- Constant time per iteration: O(1)

Conclusion:

The worst-case running time of binary search is O(lg n).

QUESTION:22

The while loop of lines 537 of the ……running time of insertion sort to ,.n lg n/ ?

ANSWER:

Let's take a look at the loop in question:

while i>0 and A[i]>key

A[i+1]=A[i]

i=i−1

This loop serves two purposes:

- A linear search to scan (backward) through the sorted sub-array to find the proper position for key.
- Shift the elements that are greater than key towards the end to insert key in the proper position.

Although we can reduce the number of comparisons by using binary search to accomplish purpose 1, we still need to shift all the elements greater than key towards the end of the array to make space for key. And this shifting of elements runs at Θ(n) time, even in average case (as we need to shift half of the elements). So, the overall worst-case running time of insertion sort will still be Θ(n2).

QUESTION:23

Describe an algorithm that, given a set S of n integers and another integer x ..time in the worst case.

ANSWER:

Algorithm: Two-Sum Problem

Input: Set S of n integers, target integer x

Output: True if S contains two elements summing to x, False otherwise

Steps:

- Sort S in ascending order (O(n lg n) time)
- Initialize two pointers: low = 0, high = n-1
- While low < high:
  - Calculate sum = S[low] + S[high]
  - If sum == x, return True
  - If sum < x, increment low
  - If sum > x, decrement high
- Return False (no pair found)

Time Complexity: O(n lg n) due to sorting

Space Complexity: O(1) (in-place sorting)

Explanation:

- Sorting allows us to efficiently find pairs with the desired sum.
- Two pointers, low and high, traverse the sorted array from opposite ends.
- If the sum of elements at low and high indices equals x, we return True.
- If the sum is less than x, incrementing low increases the sum.
- If the sum is greater than x, decrementing high decreases the sum.

Example:

S = [1, 3, 5, 7, 9], x = 8

Sorted S: [1, 3, 5, 7, 9]

low = 0, high = 4

sum = 1 + 9 = 10 (too high), decrement high

low = 0, high = 3

sum = 1 + 7 = 8 (match!), return True

Variations:

- Use a hash table (O(n) time, O(n) space) for faster lookup.
- For unsorted arrays, consider using a hash-based approach.

QUESTION:24

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

ANSWER:

If the input size n is not an exact multiple of3, then divide the arrayA so that the leftmost and middle sections have n/3 positions each, and the rightmost section has n 2(n/3)>(n/3) positions. Use the same argument as in the book, but for an input that has the (n=3) largest values starting in the leftmost section. Each such value must move through the middle (n/3) positions n route to its final position in the rightmost section, one position at a time.

For input size n = 3k (k ≥ 1), insertion sort performs at least:

- k comparisons in the first pass (n/3 elements)
- 2k comparisons in the second pass (2n/3 elements)
- 3k comparisons in the third pass (n elements)

Total comparisons: 6k + 2n

Modified Argument (arbitrary input size)

For input size n ≥ 1, insertion sort performs at least:

- n/3 comparisons in the first pass (n/3 elements)
- 2n/3 comparisons in the second pass (2n/3 elements)
- n comparisons in the third pass (n elements)

Total comparisons: n/3 + 2n/3+ n ≥ 2n

QUESTION:26

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 Analysis

Selection Sort:

1. For i = 1 to n-1:

- Find the minimum element in A[i..n].
- Swap A[i] with the minimum element.

Running Time Analysis:

Outer loop (i = 1 to n-1): O(n)

Inner loop (finding minimum in A[i..n]): O(n-i+1)

Total comparisons:

∑[i=1 to n-1] (n-i+1)

= n + (n-1) + ... + 2 + 1

= n*(n+1)/2

= O(n^2)

Time Complexity:

O(n^2)

Explanation:

- The outer loop iterates n-1 times.
- The inner loop finds the minimum in the unsorted portion of the array (A[i..n]).
- The number of comparisons decreases by 1 each iteration (n, n-1, ..., 2, 1).

Key Insights:

- Selection sort has a quadratic time complexity.

- The algorithm performs poorly for large inputs.

Comparison to Insertion Sort:

- Both have O(n^2) time complexity.
- Insertion sort performs better for partially sorted or small inputs.
- Selection sort has a simpler implementation.
