ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY
Department of Computer Science
ASSIGEMENT No 1 – Fall 2024

Name: Muhammad Haseeb **Class: BSCS-3rd (A)**

Roll No: 14861 **Subject: DSA**

Chapter 1

Exercise 1.1-1

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

ANSWER

Sorting Example:

Imagine you have a big box of crayons. You want to sort them by color—red, blue, green, and so on. Once they're in color groups, you might arrange each group from light to dark. This makes it easy to find the color you want when you're drawing.

Shortest Distance Example:

Think about a pizza delivery driver who has to drop off pizzas at several houses. The driver wants to find the quickest way to deliver all the pizzas, starting from the pizza shop. They would figure out the best route to take so they drive the least distance and save time.

These examples show how sorting and finding the shortest path help us in everyday tasks!

Exercise 1.1-2

ANSWER

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

1. **Resource Utilization:** Assess how effectively resources (human, financial, materials) are used. High efficiency often means minimal waste.
2. **Cost Efficiency:** Evaluate the relationship between the cost incurred and the output produced. This involves balancing expenses with quality and quantity.
3. **Energy Efficiency:** Look at how much energy is consumed relative to the output or service provided. Lower energy consumption often indicates higher efficiency.
4. **Quality of Output:** Consider the consistency and reliability of the output. Higher quality often leads to less rework and waste.
5. **Flexibility:** Measure how adaptable a process is to changes in demand or conditions without significant loss of performance.
6. **Time Management:** Beyond speed, assess how well time is allocated across different tasks, ensuring that priority activities are completed effectively.
7. **Employee Productivity:** Analyze how effectively employees are performing their tasks, which can involve looking at output per hour worked or task completion rates.
8. **Customer Satisfaction:** Consider how efficiently a process meets customer needs, as higher satisfaction often reflects effective service delivery.
9. **Supply Chain Efficiency:** Evaluate the flow of goods and services, focusing on minimizing delays and optimizing logistics.
10. **Sustainability**: Assess the environmental impact and sustainability of processes, as efficient operations often consider long-term ecological effects.

Balancing these measures can lead to a more comprehensive understanding of overall efficiency in a given context.

Exercise 1.1-3

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

Strengths:

1. **Efficient Search, Insertion, and Deletion:** In a balanced BST, these operations can be performed in O(log n) time, making it efficient for dynamic data sets.
2. **Sorted Order:** BSTs maintain sorted order, which makes it easy to perform in-order traversal to access data in a sorted sequence.

3. **Efficient Search, Insertion, and Deletion:** In a balanced BST, these operations can be performed in O(log n) time, making it efficient for dynamic data sets.
4. **Sorted Order:** BSTs maintain sorted order, which makes it easy to perform in-order traversal to access data in a sorted sequence.
5. **Dynamic Size:** Unlike arrays, BSTs can grow and shrink dynamically without needing to allocate or deallocate memory explicitly.

Limitations:

1. **Imbalance Issues:** If the BST becomes unbalanced (e.g., inserting sorted data), the time complexity can degrade to O(n) for operations, similar to a linked list.
2. **Complexity of Balancing:** To maintain balance, additional algorithms (like AVL or Red-Black Trees) must be implemented, increasing complexity.
3. **Memory Overhead:** Each node in a BST typically requires additional memory for pointers, which can be significant for large data sets.

In summary, while binary search trees offer efficient operations and maintain order, they require careful management to avoid imbalance and can incur additional memory costs.

Exercise 1.1-4

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

The shortest-path problem and the traveling salesperson problem (TSP) are both fundamental problems in graph theory, but they have distinct characteristics**. Similarities:**

1. Graph-Based: Both problems can be represented using graphs, where nodes represent locations and edges represent the distances or costs between those locations.
2. Optimization: They both aim to find an optimal solution—either the shortest distance or the least cost.

Difference

1. Objective:
    - Shortest-Path Problem: Seeks the shortest path from a single source node to a target node, typically finding the minimum distance or cost between these two points.
    - Traveling Salesperson Problem (TSP): Involves visiting a set of cities (nodes) exactly once and returning to the starting point, minimizing the total distance traveled.
2. Complexity:
    - Shortest-Path Problem: Generally solvable in polynomial time (e.g., using Dijkstra's or Bellman-Ford algorithms).
    - TSP: NP-hard problem, meaning no known polynomial-time solution exists for all instances, making it much more computationally intensive as the number of nodes increases.
3. Path Requirements:
    - Shortest-Path Problem: Can traverse nodes multiple times if needed and focuses on two specific nodes.
    - TSP: Requires visiting each node exactly once, except for the return to the starting node.

These differences make each problem unique in its applications and approaches for finding solutions.

Exercise 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 good enough.

Problem Requiring the Best Solution:

Emergency Response Routing: In situations like natural disasters or medical emergencies, response teams need to find the quickest route to reach those in need. In this case, only the best solution (i.e., the fastest route) will ensure timely assistance, which can be critical for saving lives.

Problem Where Approximately the Best Solution is Good Enough:

Product Delivery Optimization: For e-commerce companies, optimizing delivery routes can significantly reduce costs. However, if the delivery times are close (within a few minutes or hours) and customers are generally satisfied, a near-optimal solution (like a route that saves on fuel or time but isn't the absolute best) is often sufficient. Here, slightly longer routes might still meet service level agreements, making an approximate solution acceptable.

Exercise 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.

Real-World Problem: Traffic Management
Entire Input Available: In situations like planned events (e.g., concerts, sports games), city planners can access data in advance about expected attendance, local traffic patterns, and road closures. They can analyze this

complete input to develop a comprehensive traffic management plan to optimize flow and minimize congestion.

Input Arriving Over Time: Conversely, during unexpected incidents (like accidents or natural disasters), traffic data comes in real-time as incidents unfold. In these cases, planners must adapt to changing conditions, such as new road closures or unexpected traffic spikes, making decisions based on partial information and adjusting their strategies as more data becomes available.

This duality reflects the challenges in managing traffic efficiently, highlighting the need for both proactive planning and reactive adaptability.

Exercise 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.

Application: Online Recommendation Systems

Algorithms Involved:

1. **Collaborative Filtering**: Recommends items based on user similarities, leveraging past behaviors of similar **users.**
2. **Content-Based Filtering:** Suggests items similar to those a user has liked, focusing on item attributes.
3. **Matrix Factorization:** Decomposes user-item interactions to identify latent factors that influence preferences.
4. **Deep Learning:** Utilizes neural networks to analyze complex interactions and improve recommendation accuracy.

Function:

These algorithms personalize user experiences, enhancing engagement and driving sales.

Exercise 1.2-2

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?

We wish to determine for which values of n the inequality $8n^2 < 64n\log_2(n)$ holds. This happens when $n < 8\log_2(n)$, or when $n \leq 43$. In other words, insertion sort runs faster when we're sorting at most 43 items. Otherwise merge sort is faster.

Exercise 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?

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

Chapter 2

Exercise 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).

31	41	59	26	41	58
31	41	59	26	41	58
31	41	59	26	41	58
26	31	41	59	41	58
26	31	41	41	59	58
26	31	41	41	58	59

Exercise 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] 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 A [1: n].

SUM-ARRAY(A, n):

```
sum = 0

for i = 1 to n:

   sum = sum + A[i]

return sum
```

Loop Invariant

Loop Invariant: At the start of each iteration of the loop (for each index i), the variable sum contains the sum of the elements A[1] through A[i-1].

Initialization

- **Before the first iteration** (i = 1):
  - The variable sum is initialized to 0.
  - There are no elements to sum before A[1], so the sum of the elements from A[1] to A[0] is indeed 0.
- Therefore, the loop invariant holds true at initialization.

Maintenance

- **Assume the loop invariant holds true at the beginning of iteration i**:
  - That is, sum is equal to A[1] + A[2] + ... + A[i-1].
- **During the iteration**:
  - The line sum = sum + A[i] updates sum to now include A[i], so:
  - After the update, sum becomes A[1] + A[2] + ... + A[i-1] + A[i], which means the loop invariant holds true at the start of the next iteration (i + 1).

Termination

- **When the loop terminates**:
    - The loop runs until i = n + 1, meaning the last executed iteration was when i = n.
    - At this point, according to our loop invariant, sum contains A[1] + A[2] + ... + A[n].
- The procedure then returns sum, which correctly reflects the total sum of the array from index 1 to n.

Conclusion

Since the loop invariant holds true at initialization, is maintained during each iteration, and leads to a correct result upon termination, we can conclude that the SUM-ARRAY procedure returns the correct sum of the numbers in A[1:n].

Exercise 2.1-3

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

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

```
            j -= 1
        arr[j + 1] = key
```

```
# Example usage:
array = [5, 2, 9, 1, 5, 6]
insertion_sort_decreasing(array)
print(array)  # Output: [9, 6, 5, 5, 2, 1]
```

Exercise 2.1-4

Consider the *searching problem*:

Input: A sequence of n numbers [a_1,a_2 ,,,,,, a_n] i stored in array A [1: n] W 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 fulûlls the three necessary properties.

Pseudocode for ADD Procedure

1. function linearsearch(a, n, x)
2. for i from 1 to n do
3. if a[i] = x then

4. return i
5. // found x, return the index

---

6. end for
7. return nil
8. // x not found
9. end function

Loop Invariant

Loop Invariant: At the start of each loop (when checking the element at index i), all previous elements (from 1 to i-1) have been checked, and none are equal to x.

Proof of the Loop Invariant

1. **Initialization**:
    - Before the first loop (when i = 1), we haven't checked any elements yet. This is fine because there are no checked elements, so the invariant holds.
2. **Maintenance**:
    - Suppose we've checked all elements up to i-1, and none were x.
    - In the current loop, we check A[i].
        - If A[i] is equal to x, we return i because we found it.
        - If A[i] is not x, we go to the next index (i + 1).
    - Now, we've checked A[1] to A[i], and since A[i] wasn't x, the invariant still holds.

3. **Termination**:

   - The loop ends when i goes beyond n. This means we've checked every element from A[1] to A[n].
   - If we haven't returned an index during the loop, it means x isn't in the array, so we return NIL.

Conclusion

Since we've shown that the loop invariant is true at the start, is kept true during the loop, and is true when we finish, we can say the linear search works correctly. It will either find x and return its position or return NIL if x isn't there.

2.1-5

Exeericse

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 .

Pseudocode for ADD Procedure

```
1. function add(a, b, n)
2.   // initialize the carry to 0
3.       carry = 0
4.          // loop through each bit from 0 to n-1
5.               for i from 0 to n-1 do
6.                    // compute the sum of a[i], b[i], and
    carry
7.                              sum = a[i] + b[i] + carry
8.                     // the resulting bit is the sum modulo 2
9.               c[i] = sum mod 2
10.                 // the new carry is the sum divided by 2
11.               carry = sum div 2
12.                    end for
13.                           // store the final carry in c[n]
14.       c[n] = carry
15.                end function
```

Explanation

1. **Initialization**: We start with a carry of 0, which will be used to handle cases where the sum of two bits exceeds 1 (i.e., results in a carry to the next higher bit).

2. **Loop through each bit**: We iterate from 0 to n-1, processing each bit:

    - We calculate the total sum of the current bits from A and B, plus any carry from the previous iteration.

    - The result for the current bit (C[i]) is obtained using sum MOD 2, which gives us the binary value (either 0 or 1).

- We update the carry using sum DIV 2, which determines if there will be a carry for the next iteration.

3. **Final carry**: After processing all bits, we store any remaining carry in C[n].

Result

The array C will now contain the binary sum of the two integers, with C[0] to C[n-1] representing the sum, and C[n] representing any final carry.

Exercise 2.2-1

Express the function $n^3/1000 + 100n^2 - 100n + 3$ in terms of ,(~)notation

$n^3/1000 - 100n^2 - 100n$

$+ 3 \in \Theta(n^3)$

Exercise 2.2-2

Consider sorting n numbers stored in array A[1:n]by first finding the smallest element of A[1:n] and exchanging it with the element in A[1] Then find the smallest element of A[2:n] and exchange it with A[2]. Then find the smallest element of A[3:n], and exchange it with A[3]. 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 ,-notation. Is the best-case running time any better?

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 $\Theta(n^2)$. 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.

CODE:

```
SelectionSort(A):
  n = length(A)
  for i from 1 to n - 1 do:
    minIndex = i
    for j from i + 1 to n do:
      if A[j] < A[minIndex] then:
        minIndex = j
    // Swap A[i] with A[minIndex]
```

if minIndex ≠ i then:

temp = A[i]

A[i] = A[minIndex]

A[minIndex] = temp

This yields a running time of

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

.

Exercise 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 (~)-notation, give the average-case and worst-case running times of linear search. Justify your answers.

Average Case Analysis

For linear search, we assume that each element in the array is equally likely to be the target value. Let's say we have an array of size n.

- **Average Case**:
  - If we consider that the target element is equally likely to be at any position in the array, on average, the target will be found halfway through the list

    Thus, if the target is present, it will be found after checking approximately $\frac{n}{2}$
  - elements.

- If the target element is not present, we will check all n elements before concluding that it is not in the array.

So, the average number of elements checked is:

$$\text{Average elements checked} = \frac{1+2+3\ldots\ldots\ldots+n}{n} = \frac{n+1}{2} \approx \frac{n}{2}$$

In big-O notation, we express this as:

$$O(n)$$

However, when considering average-case complexity more formally, we can denote it as:

$$\Theta(n)$$

Worst Case Analysis

- **Worst Case**:
  - The worst-case scenario occurs in two situations: either the target element is not present in the array or it is the last element in the array. In both cases, we

need to
check all n
elements

- Thus, in the worst case, the number of elements checked is:

Worst case elements che
cked=n

In big-O notation, the worst-case running time of linear search is:

$O(n)$

And in terms of tight bounds, we express it as:

$\Theta(n)$

Exercise 2.2-4

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

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 bestcase 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 $\Theta(n)$.

Exercise 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).

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

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

Exercise 2.3-2

The test in line 1 of the MERGE-SORT procedure reads "<if p ≥ r= rather than "<if

p ≠ r.= If MERGE-SORT is called with p > r, then the subarray a[p:r] is empty. Argue that as long as the

---

initial call of MERGE-SORT (A; 1; n) has n ≥ 1, the test <if p ≠ r= suffices to ensure that no recursive call has p > r.

1. **What MERGE-SORT Does:** MERGE-SORT splits an array into smaller parts, sorts them, and then merges them back together.
2. **How It Decides to Sort:** The check <if p ≠ r> means "only sort if there's more than one element." If p equals r, that means there's just one element (or none).
3. **Initial Call:** When you first call MERGE-SORT, you use indices like 1 to n (assuming n ≥ 1), so p starts at 1 and r at n. Here, p is always less than or equal to r.
4. **What MERGE-SORT Does:** MERGE-SORT splits an array into smaller parts, sorts them, and then merges them back together.
5. **How It Decides to Sort:** The check <if p ≠ r> means "only sort if there's more than one element." If p equals r, that means there's just one element (or none).

6. **What MERGE-SORT Does:** MERGE-SORT splits an array into smaller parts, sorts them, and then merges them back together.

7. **How It Decides to Sort:** The check <if p ≠ r> means "only sort if there's more than one element." If p equals r, that means there's just one element (or none).

8. **Initial Call:** When you first call MERGE-SORT, you use indices like 1 to n (assuming n ≥ 1), so p starts at 1 and r at n. Here, p is always less than or equal to r.

9. **Dividing the Array:** When the array is divided, it always keeps p less than or equal to r. For example, when you split it, you create two new ranges that still follow this rule.

10. **No Empty Ranges:** Because of how the algorithm splits the array, p will never be greater than r in the recursive calls.

In short, as long as the starting call has at least one element, the condition <if p ≠ r> ensures that you won't ever try to sort an empty range.

Exercise 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.

Loop Invariant for the While Loop (Lines 12-18)

Loop Invariant: **At the start of each iteration**:

1. All elements from (L[1]) to (L[i]) are in their final position in the merged array (A).
2. All elements from (R[1]) to (R[j]) are in their final position in (A).
3. The remaining elements of (L) (from \(L[i+]) to (L[m])) and \(R) (from(R[j+1]) to \(R[n])) have not yet been considered.

Proof of Correctness

1. **Initialization:** Before the first iteration, no elements have been merged, so the invariant holds trivially.

2. **Maintenance:** During each iteration, the smallest element from (L) or (R) is added to (A), maintaining the invariant.

3. **Termination:** When the loop ends (either (i > m \) or \(j > n)), the remaining elements from the non-empty subarray are added directly to (A), preserving the sorted order.

Conclusion:

All elements from both subarrays are merged in sorted order, confirming the correctness of the MERGE procedure.

Exercise 2.3-4

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

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

is T(n)= n log.

Let $T(n)$ denote the running time for insertion sort called on an array of size n. We can express $T(n)$ recursively as

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

where $I(n)$ denotes the amount of time it takes to insert $A[n]$ into the sorted array $A[1..n-1]$. Since we may have to shift as many as $n-1$ elements once we find the correct place to insert $A[n]$, we have $I(n) = \theta(n)$.

Algorithm5 *Merge*(*A,p,q,r*)

1: $n = q - p + 1$
2: $n_2 = r - q$
3: let $L[1,..n_1]$ and $R[1..n_2]$ be new arrays
4: **for** $i = 1$ to n_1 **do**
5: $L[i] = A[p + i - 1]$

6: **end for**
7: **for** $j = 1$ to n_2 **do**

8: $R[j] = A[q + j]$
9: **end for**
10: $i = 1$
11: $j = 1$ 12:
$k = p$
13: **while** i 6= $n_1 + 1$ and j 6= $n_2 + 1$ **do**
14: **if** $L[i] \leq R[j]$ **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** $i == n_1 + 1$ **then**
23: **for** $m = j$ to n_2 **do**
24: $A[k] = R[m]$
25: $k = k + 1$
26: **end for**
27: **end if**
28: **if** $j == n_2 + 1$ **then**
29: **for** $m = i$ to n_1 **do**
30: $A[k] = L[m]$
31: $k = k + 1$
32: **end for** 33: **end if**

Exercise 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.

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

Note that the initial call should be *BinSearch*(1,*n*,*v*). Each call results in a constant number of operations plus a call to a problem instance where the quantity $b - a$ falls

by at least a factor of two. So, the runtime satisfies the recurrence $T(n) = T(n/2) + c$. So, $T(n) \in \Theta(\lg(n))$

Exercise 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 (~) (lg n).

Pseudocode for Binary Search

```
1. function binarysearch(a, v, low, high)
2. while low ≤ high do
3. mid = (low + high) // 2
4. if a[mid] = v then
5. return mid // found v at index mid
6. else if a[mid] < v then
7. low = mid + 1 // search in the right half
8. else
9. high = mid - 1 // search in the left half
10.    end while
11.    return -1 // v not found
```

Explanation of the Worst-Case Running Time

1. **Halving the Search Space**:
    - Each time we check the middle element (A[mid]), we can determine whether the target value v is in the left half or the right half of the current subarray. This allows us to eliminate half of the remaining elements.
2. **Number of Iterations**:
    - Starting with n elements, after the first iteration, you are left with at most n/2. After the second iteration, at most n/4, then n/8, and so on.
    - This halving continues until the size of the search space reduces to 1 or you find the value.
3. **Mathematical Representation**:
    - The number of times you can halve n until you get down to 1 can be expressed mathematically as

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1$$

The number of halvings is given by the equation:

$$2k=n \Longrightarrow k=\log 2(n)$$

Thus, it takes at most $\log_2(n)$ iterations to reduce the problem size to 1.

Conclusion:

- Therefore, the worst-case running time of binary search is O(logn) because the algorithm performs a

---

logarithmic number of comparisons in relation to the number of elements in the array.

Exercise 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/?**

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 $\Theta(n\lg(n))$. 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 $M_{i,j} \Rightarrow \forall k > i, M_{k,j}$.

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 $i \leq j$. we now have two cases:

Case 1 $\exists k,(i,k)$ was considered and $j < k$. In this case, we take the smallest

1: Use Merge Sort to sort the array A in time $\Theta(n\lg(n))$
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 $\exists k,(k,j)$ 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 $\{(m,m^0) : m \leq m^0 < n\}$ 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.

Name Muhammad Haseeb

Roll no 14861

Subject DSA Chapter 3

Exercise 3.1-1

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

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

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

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

Exercise 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.

Let $c = 2^b$ and $n_0 \geq 2a$. Then for all $n \geq n_0$ we have $(n+a)^b \leq (2n)^b = cn^b$ so $(n+a)^b = O(n^b)$. Now let $n_0 \geq \frac{-a}{1-1/2^{1/b}}$ and $c = \frac{1}{2}$. Then $n \geq n_0 \geq \frac{-a}{1-1/2^{1/b}}$ if and only if $n - \frac{n}{2^{1/b}} \geq -a$ if and only if $n + a \geq (1/2)^{a/b}n$ if and only if $(n+a)^b \geq cn^b$. Therefore $(n+a)^b = \Omega(n^b)$. By Theorem 3.1, $(n+a)^b = \Theta(n^b)$.

Exercise

3.1-3

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

There are a ton of different funtions that have growth rate less than or equal to n^2. In particular, functions that are constant or shrink to zero arbitrarily fast. Saying that

you grow more quickly than a function that shrinks to zero quickly means nothing.

Generalized Lower-Bound Argument for Insertion Sort

1. **Setup**: Let 0<α<10 < \alpha < 10<α<1. The largest αn\alpha nαn values are in the first αn\alpha nαn positions of the array.
2. **Insertion Process**:
    - When inserting these αn\alpha nαn largest values, they must find their correct positions among the (1−α)n(1 - \alpha)n(1−α)n smaller values in the array.
3. **Total Comparisons**:
    - Each of the αn\alpha nαn largest values may need to compare against and shift past up to (1−α)n(1 - \alpha)n(1−α)n smaller values.
    - Thus, the total comparisons for the αn\alpha nαn largest values is:

Additional Restrictions on α\alphaα

- α\alphaα must remain in the range 0<α<10 < \alpha < 10<α<1 to ensure that both the largest and the smaller values exist.

Maximizing Comparisons

1. **Function to Maximize**:

   $$f(\alpha)=\alpha(1-\alpha)$$

2. **Finding the Maximum**:
   - Taking the derivative:

     $$f'(\alpha)=1-2\alpha$$

   - Setting the derivative to zero:

     $$1-2\alpha=0 \Longrightarrow \alpha=1/2$$

Conclusion

- The value of α\alphaα that maximizes the number of comparisons is $\alpha=1/2$ .This ensures that the αn largest values interact most with the remaining values during insertion.

Exercise 3.2-1

Let f (n)and g(n) be asymptotically nonnegative functions. Using the basic definition of Θ notation, prove that max {f(n), g (n)}= Θ f(n), g (n)

Let $n_1 < n_2$ be arbitrary. From f and g being monatonic increasing, we know $f(n_1)$
$< f(n_2)$ and $g(n_1) < g(n_2)$. So

$$f(n_1) + g(n_1) < f(n_2) + g(n_1) < f(n_2) + g(n_2)$$

Since $g(n_1) < g(n_2)$, we have $f(g(n_1)) < f(g(n_2))$. Lastly, if both are nonegative, then, $f(n_1)g(n_1) = f(n_2)g(n_1) + (f(n_2) - f(n_1))g(n_1)$

$= f(n_2)g(n_2) + f(n_2)(g(n_2) - g(n_1)) + (f(n_2) - f(n_1))g(n_1)$

Since $f(n_1) \geq 0$, $f(n_2) > 0$, so, the second term in this expression is greater than zero. The third term is nonnegative, so, the whole thing is$< f(n_2)g(n_2)$.

Exercise 3.2-2

Explain why the statement, <The running time of algorithm A is at least $O(n^2)$ is meaningless.

$a^{\log_b(c)} = a^{\frac{\log_a(c)}{\log_a(b)}} = c^{\frac{1}{\log_a(b)}} = c^{\log_b(a)}$.

Exercise 3.2-3

Is $2^{n+1.} = 0(2^n)$? Is $= 0\ (2^n)$?

As the hint suggests, we will apply stirling's approximation

$$\lg(n!) = \lg\left(\sqrt{(2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)\right)$$

$$= \frac{1}{2}\lg(2\pi n) + n\lg(n) - n\lg(e) + \lg\left(\Theta\left(\frac{n+1}{n}\right)\right)$$

Note that this last term is $O(\lg(n))$ if we just add the two expression we get when we break up the lg instead of subtract them. So, the whole expression is dominated by $n\lg(n)$. So, we have that $\lg(n!) = \Theta(n\lg(n))$.

$$\lim_{n\to\infty}\frac{2^n}{n!}=\lim_{n\to\infty}\frac{1}{\sqrt{2\pi n}(1+\Theta(\frac{1}{n}))}\left(\frac{2e}{n}\right)^n\leq\lim_{n\to\infty}\left(\frac{2e}{n}\right)^n$$

If we restrict to $n > 4e$, then this is

$$\leq\lim_{n\to\infty}\frac{1}{2^n}=0$$

$$\lim_{n\to\infty}\frac{n^n}{n!}=\lim_{n\to\infty}\frac{1}{\sqrt{2\pi n}(1+\Theta(\frac{1}{n}))}e^n=\lim_{n\to\infty}O(n^{-.5})e^n\geq\lim_{n\to\infty}\frac{e^n}{c_1\sqrt{n}}$$

$$\geq\lim_{n\to\infty}\frac{e^n}{c_1 n}=\lim_{n\to\infty}\frac{e^n}{c_1}=\infty$$

Exercise 3.2-4

Prove Theorem 3.1.

The function dlog$n$e! is not polynomially bounded. If it were, there would exist constants c, a, and n_0 such that for all $n \geq n_0$ the inequality dlog$n$e! $\leq cn^a$ would hold. In particular, it would hold when $n = 2^k$ for $k \in$ N. Then this becomes $k! \leq c(2^a)^k$, a contradiction since the factorial function is not exponentially bounded.

We'll show that dloglog$n$e! $\leq n$. Without loss of generality assume $n = 2^{2k}$.

Then this becomes equivalent to showing $k! \leq 2^{2k}$, or $1 \cdot 2 \cdots (k-1) \cdot k \leq$

$4 \cdot 16 \cdot 2^8 \cdots 2^{2k}$, which is clearly true for $k \geq 1$. Therefore it is polynomially bounded.

Proof of Ω(n2)\Omega(n^2)Ω(n2) Lower Bound for Insertion Sort

1. **Input Arrangement**:
    - Consider an array A of n elements.

- Assume the largest n/ values are in the first n3frac{n}{3}n/3 positions.

2. **Movement of Values**:
   - Each of the n/3n largest values must be moved to their correct positions in the last n/3n positions.
   - To do this, they must pass through the middle n/3n positions.
3. **Total Comparisons**:
   - Each of the n/3n largest values makes at least n/3 comparisons.
   - Thus, the total number of comparisons is:
   - $=Total\ Comparisons = (3n) \times (3n) = 9n2$
4. **Conclusion**:
   - Since the number of comparisons is at least n29\frac{n^2}{9}9n2, we have:

$$T(n)=\Omega(n^2)$$

This shows that the worst-case time complexity of insertion sort is $\Omega(n^2)$.

Exercise 3.2-5

Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Proof

(1) If T(n)=Θ(g(n))

- By definition, T(n)=Θ(g(n)) means:
  - There exist constants c1,c2>0 and n0 such that for al0n≥n0
- This implies:
  - T(n)=O(g(n))T(n) = O(g(n))T(n)=O(g(n)) (from the upper bound).
  - T(n)=Ω(g(n))T(n) = \Omega(g(n))T(n)=Ω(g(n)) (from the lower bound).

(2) If T(n)=O(g(n))T(n) = Ω(g(n))

- From T(n)=O(g(n)) there exist constants c2>0 and n0 such that: T(n)≤c2g(n)for all n≥n0
- From T(n)=Ω(g(n)) there exist constants c1>such that:
- T(n)≥c1g(n)for all n≥0
- Combining these, we have:
- c1g(n)≤T(n)≤c2g(n)
- Thus, T(n)=Θ(g(n)).

Conclusion

We have shown that:

T(n)=Θ(g(n)) ⟺
T(n)=O(g(n)) and T(n)=Ω(g(n))

Exercise 3.2-6

Prove that o(g(n)) ∩ ω(g(n)) is the empty set.

$$\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = 1 + \frac{1+\sqrt{5}}{2} = 1+\phi$$

$$\hat{\phi}^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{6-2\sqrt{5}}{4} = 1 + \frac{1-\sqrt{5}}{2} = 1+\hat{\phi}$$

Exercise 3.2-7

We can extend our notation to the case of two parameters n and m tha
∞ independently at different rates. For a given function $g(n,m)$, we d

$O(g(n;\ m))$ ***the set of functions***

$O(g(n,m))$

$= \{f(n,m)$ ***there exist positive constants*** $c, n0,$ ***and*** $m0$

such that $0 \leq f(n,m) \leq cg(n,m)$

for all $n \geq n0$ ***or*** $m \geq m0\}$:

Give corresponding deû***nitions for*** $(g(n,m))$ ***and*** $(g(,m))$.

Definitions

1. **Big-Omega Notation**:

Ω(g(n,m))={f(n,m)|there exist positive constants c,n0,m0 such t hat 0≤c·g(n,m)≤f(n,m) for all n≥n0 and m≥m0

Theta Notation:

Θ(g(n,m))={f(n,m)|there exist positive constants c1,c2,n0,m0 su ch that c1·g(n,m)≤f(n,m)≤c2·g(n,m) for all n≥n0 and m≥m0

Summary

- **O(g(n,m))O(g(n, m))O(g(n,m))**: Upper bound on f(n,m)f(n, m)f(n,m).
- **Ω(g(n,m))\Omega(g(n, m))Ω(g(n,m))**: Lower bound on f(n,m)f(n, m)f(n,m).

- **Θ(g(n,m))\Theta(g(n, m))Θ(g(n,m))**: Tight bound on f(n,m)f(n, m)f(n,m) (both upper and lower bounds).

These definitions allow us to analyze the behavior of functions with two parameters independently.

Exercise 3.3-1

Show that if f (n) and g(n) are monotonically increasing functions, then so are the functions f (n) + g(n) and f (g(n)), and if f (n) and g(n) are in addition nonnegative, then f (n) * g(n) is monotonically increasing.

As f(n) and g(n) are monotonically increasing functions,

$$m \leq n \Longrightarrow f(m) \leq f(n) \quad \textbf{(1)}$$

$$m \leq n \Longrightarrow g(m) \leq g(n) \quad \textbf{(2)}$$

Therefore, f(m)+g(m)≤f(n)+g(n) i.e. monotonically increasing.
Also, combining (1) and (2), f(g(m)≤f(g(n))
Therefore, f((g(n)) is also monotonically increasing.
If f(n)and g(*n*) are nonnegative we can multiply inequalities (1) and (2), to say:

$$f(m) \cdot g(m) \leq f(n) \cdot g(n)$$

Therefore, f(n)·g(n) is also monotonically increasing.

Exercise 3.3-2

To prove that bαnc+d(1−α)n =n for any integer n and real number α in the range 0≤α≤10 ?

Definitions:

- $b\alpha n c$: the greatest integer less than or equal to αn.
- $d(1-\alpha)n = (1-\alpha)n$.

Decompose n:

$$N = b\alpha n c + d(1-\alpha)n + r$$

where r is the remainder from αn.

Express $b\alpha n c$:

$$b\alpha n c = \alpha n - r$$

with $0 \leq r < 10$.

Substituting:

$$b\alpha n c + d(1-\alpha)n = (\alpha n - r) + (1-\alpha)n$$

Simplify:

$$= \alpha n + (1-\alpha)n - r = n - r$$

Final equality:

$$n = b\alpha n c + d(1-\alpha)n + r$$

Thus, the equation $b\alpha n c + d(1-\alpha)n = n$ holds true.

3.3-3

Use equation (3.14) or other means to show that $(n+0(n))^k = \Theta(n^k)$ for all real constant k. Conclude that $(n^k) = \Theta(n^k)$ and $(n^k) = \Theta(n^k)$.

The notation o(n)o(n)o(n) means that a function grows slower than nnn. More formally, f(n)=o(n)f(n) if

$$\lim_{n\to\infty}\left(\frac{f(n)}{n}\right)^{\square} = 0$$

Step 2: Expanding (n+o(n))^k

Using the binomial expansion, we can expand (n+o(n))^k:

(n+o(n))^k = n$^k\left(1+\frac{0(n)}{n}\right)k$

Step 3: Analyzing $\left(\mathbf{1}+\frac{\mathbf{0}(\boldsymbol{n})}{\boldsymbol{n}}\right)\boldsymbol{k}$

Now, let's analyze the term $\left(\mathbf{1}+\frac{\mathbf{0}(\boldsymbol{n})}{\boldsymbol{n}}\right)\boldsymbol{k}.\boldsymbol{since}\ \frac{\mathbf{0}(\boldsymbol{n})}{\boldsymbol{n}} \rightarrow\ 0\ as\ n \rightarrow \infty$ *we can use the limit*:

$$\lim_{n\to\infty}\left(1+\frac{0(n)}{n}\right)^{k} = 1k\ =\ 1.$$

Step 4: Concluding (n+o(n))^k=Θ(n^k)

Thus, for large n:

(n+o(n))^k=n^k·(1+o(1)),

where o(1)→0 as n→∞n →∞. This shows that:

(n+o(n))^k=n^k+o(n^k),

indicating that (n+o(n))k =Θ(nk.

Step 6: Concluding the Results

1. (n+o(n))k =Θ(nk)
2. n^k=Θ(n^k)

EXERICSE

3.3-5 Is the function (lg n)! polynomially bounded? Is the function (lg lg n)! polynomially bounded?

Answer

To determine whether the functions (lg n)! and (lglgn)! are polynomially bounded, we need to analyze the growth rates of these factorials.

1. **For (lgn)!**:

- The logarithm function lgn grows very slowly compared to n. As n increases, lgn approaches infinity, and thus (lgn)! will also grow, but we need to analyze its growth in relation to polynomial functions.

- Using Stirling's approximation, we have:Type equation here.

3.3-6

Which is asymptotically larger: lg.(lg*n) or l g* (lg n)?

Note that $\lg^*(2^n) = 1 + \lg^*(n)$, so

lg.(lg*n) lg(lg*(2n))

$$\lim_{n\to\infty} \frac{\lg(\lg(n))}{\lg(1+\lg^*(n))} = \lim_{n\to\infty} \frac{\lg^*(2^n)}{\lg^*(n)} = \lim_{n\to\infty} \frac{n}{1+n} = 0$$

lim

3.3-7 Show that the golden ratio ∅ and its conjugate ∅ˆy both satisfy the equation
$X^2 = x+1$.

Answer
To show that the golden ratio ϕ and its conjugate ψ satisfy the equation $x^2=x+1$, we first need to define the golden ratio and its conjugate.

The golden ratio ϕ is defined as:

$$\phi=\frac{1+\sqrt{5}}{2}$$

Its conjugate ψ is:

$$\psi=\frac{1-\sqrt{5}}{2}$$

Step 1: Verify that ϕ satisfies the equation
Substituting ϕ into the equation $x^2=x+1$:

1. Calculate ϕ^2:

$$\phi=(\frac{1+\sqrt{5}}{2})^2=\frac{(1+\sqrt{5})2}{4}=\frac{(1+2\sqrt{5}+5)}{4}=\frac{(6+2\sqrt{5})}{4}\ \frac{(3+\sqrt{5})}{2}$$

2 Calculate ϕ+1:

$$\phi+1=\frac{1+\sqrt{5}}{2}+1=\frac{1+\sqrt{5}}{2}+\frac{2}{2}=\frac{(3+\sqrt{5})}{2}$$

3 Compare $\phi^2$and ϕ+1**:**

$$\phi^2=\frac{(3+\sqrt{5})}{2} \; and \; \phi+1=\frac{(3+\sqrt{5})}{2}$$

Thus, $\phi^2=\phi+1$.

Step 2: Verify that ψ satisfies the equation

Now we substitute ψ into the equation:

1. Calculate ψ^2:

   $$\psi^2=(\frac{1\_\sqrt{5}}{2})^2=\frac{(1\_\sqrt{5})2}{4}=\frac{(1\_2\sqrt{5}+5)}{4}=\frac{(6\_2\sqrt{5})}{4}\;\frac{(3\_\sqrt{5})}{2}$$

2. Calculate ψ+1:

   $$\frac{1\_\sqrt{5}}{2}+1=\frac{1\_\sqrt{5}}{2}+\frac{2}{2}=\frac{(3\_\sqrt{5})}{2}$$

3. Compare ψ^2 and ψ+1:

   $$\psi^2=\frac{(3\_\sqrt{5})}{2} \; and \; \psi+1=\frac{(3\_\sqrt{5})}{2}$$

   Thus, $\psi^2=\psi+1$.

   Final Result

   Both ϕ and ψ satisfy the equation $x^2=x+1$.

EXERICE

3.3-8

Prove by induction that the ith Fibonacci number satisûes the equation

**Fi = .�i □ □ □

inductive Step

Now assume that the formula holds or i=n and i=n−1:

$Fi = \frac{\Phi n\_\psi n}{\sqrt{5}}$ and Fn−1= $\frac{\Phi n\_1\_\psi n\_1}{\sqrt{5}}$

We need to show that it holds for i=n+1:

$$F_{n+1=}F_{n=} + F_{n\_1=}$$

$F_i = F_{i-1} + F_{i-2} = \frac{\phi^{i-1}+\phi^{i-2}-(\hat{\phi}^{i-1}+\hat{\phi}^{i-2})}{\sqrt{5}} = \frac{\phi^i-\hat{\phi}^i}{\sqrt{5}}$

Exercise 3.3-9

Show that k lg k =. (~)n() implies k =(~)(n/ lg n).

Let c1 and c2 be such that c1n ≤ k ln k ≤ c2n. Then we have ln c1 + ln n =
ln(c1n) ≤ ln(k ln k) = ln k + ln(ln k) so ln n = O(ln k). Let c3 be such that
ln n ≤ c3 ln k. Then
$\frac{n}{\ln n} \geq \frac{n}{c3 \ln k} \geq \frac{k}{c2\ c3}$
so that n
ln n = Ω(k). Similarly, we have ln k + ln(ln k) = ln(k ln k) ≤ ln(c2n) =

ln(c2) + ln(n) so ln(n) = Ω(ln k). Let c4 be such that ln n ≥ c4 ln k. Then

$$\frac{n}{\ln n} \geq \frac{n}{c3 \ln k} \geq \frac{k}{c2\ c3}$$

so that $\frac{n}{\ln n}$ = O(k). By Theorem 3.1 this implies $\frac{n}{\ln n}$ = Θ(k). By symmetry,
k = $\Theta \frac{n}{\ln n}$.