**Abbottabad University of Science and Technology**



**Department: Computer Science**

**Name: Awais Zaib**

**Roll No: 14707**

**Class: BSCS 3rd (A)**

**Subject: Data Structure and Algorithm**

**Submitted To: Mr. Jamal Abdul Ahad**

**Assignment: 01**

**Date: 31/10/2024**

**Chapter # 1**

**Exercises**

* + 1. **Describe your own real-world example that requires sorting. Describe one that requires ûnding the shortest distance between two points.**

1. **Sorting Example:** Consider the registrar's office at a university compiling a list of graduating students. To assign honors or choose seating for the ceremony, the list must be sorted by the students' GPA. The registrar can effectively classify students by honors levels (such as cum laude, magna cum laude, and summa cum laude) and expedite the graduation process by ordering students in descending order of GPA.
2. **Shortest Distance Example :** In order to optimize routes, a logistics company that oversees a fleet of delivery trucks frequently has to determine the shortest distance between a warehouse and several delivery locations. With on-time delivery, this not only saves gasoline and time but also increases customer satisfaction. The organization can determine the best routes dynamically based on traffic and distance in real time by utilizing techniques such as the A\* algorithm or Dijkstra's for single-source shortest paths.
   * 1. **Other than speed, what other measures of efûciency might you need to consider in a real-world setting?**

**Other Efficiency Measures**

1. **Memory utilization**: Because low-power devices (such Internet of Things sensors) have limited storage and processing capacity, memory utilization is crucial for programs running on these devices.
2. **Scalability:** In order to provide fast load times and responsive interactions, algorithms for applications such as social media platforms must be able to process billions of data points without seeing appreciable performance degradation.
3. **Energy Consumption:** To extend battery life in smartphones, apps and algorithms need to be energy-efficient. Excessive energy use can quickly deplete the battery, which reduces user happiness.
4. **Maintainability:** Because financial system algorithms frequently change to meet new requirements or enhance efficiency, they must be extremely maintainable.
   * 1. **Select a data structure that you have seen, and discuss its strengths and limitations.**

**Data Structure Example**

1. **Strengths of Arrays:**

**Constant Time Access:** For situations requiring fast lookups, such as a leaderboard, arrays provide constant-time access (O(1)) to elements by index, which is very effective.  
**Predictable Memory Usage:** Arrays' memory footprint is predictable due to their constant size, which might be helpful in applications with limited memory.

1. **Limitations of Arrays:**

**Lack of Size Flexibility:** Because static arrays have set sizes, handling dynamic data without resizing becomes challenging.  
**Inefficient Insertions/Deletions**: Because they frequently call for shifting elements, insertions and deletions are inefficient. Because of this, arrays may not be the best choice for tasks requiring frequent changes, like text editing.

* + 1. **How are the shortest-path and traveling-salesperson problems given above similar? How are they different?**

**Shortest Path vs. Traveling Salesperson Problem (TSP)**

1. **Comparables:**  
   **Graph Representation:** Both issues are commonly represented as graphs, in which nodes stand in for places and edges for the separations between them.  
   **Optimization Goal:** Both are crucial in routing and logistics applications since they seek to reduce cost or distance.
2. **Differences:**

**Shortest Path**: entails determining the quickest path between two designated nodes, such as a city and another city. Algorithms such as Dijkstra's can be used to efficiently find solutions.

**Traveling Salesperson Problem**: entails determining the shortest path that makes exactly one stop at each node before returning to the beginning node. Since it becomes impractical to obtain the exact solution with big data sets, approximation algorithms or heuristics are typically used to solve this NP-hard problem.

* + 1. **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.**

**Examples of Solution Requirements**

1. **Best Solution Needed: Air Traffic Control**

**Why the Best Solution Is Needed:** To prevent crashes, air traffic management requires precise flight path calculations. A less-than-ideal or rough solution could have disastrous consequences.  
**Using Precise Algorithms:** Air traffic algorithms use intricate, precise algorithms to assure safety, prioritizing accuracy and dependability over speed.

1. **Approximately Best Solution: Digital Marketing Ad Targeting**

**Why Approximation Is Acceptable:** As long as digital advertisements reach the right audience, precise targeting is less important. Targeting by age and general region, for instance, is usually "good enough."  
**Used Algorithms:** For real-time ad targeting, heuristics and approximate matching algorithms that value speed above perfect accuracy are appropriate.

* + 1. **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.**

1. **Entire Input Available Example: Music Festival Planning**

**Example Context:** Planning a music festival involves a lot of work, including scheduling, venue layouts, and performer lineups. A structured plan is made possible by the fact that most of the input (such as confirmed artist schedules and venue specifics) is known ahead of time.  
**Advantages of Advanced Input:** Without having to make last-minute adjustments, organizers may assign resources, make thorough timetables, and notify the public.

1. **Input Arriving Over Time Example: Real-Time Navigation**

**Example Context:** Road closures, user positions, and traffic statistics are constantly updated in real-time navigation tools such as Google Maps. When fresh information (like a traffic congestion) is received, the app recalculates routes and displays directions based on the most recent input.  
**Advantages of Input in Real Time**: The service is more responsive to current conditions since real-time updates based on incoming data guarantee users receive the most recent routes. Such algorithms depend on dynamic recalculations and ongoing data integration.

**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.**

**Solution:**

Google Maps is one instance of an application that needs algorithmic content at the application level. Fundamentally, Google Maps depends largely on algorithms to perform a number of tasks, including determining the quickest or shortest path between two points.  
Dijkstra's method (also known as the A\* algorithm, which is an optimization of Dijkstra's) is the main algorithm used for route optimization. In a weighted graph, where each road or pathway has a cost or distance associated with it, these algorithms determine the shortest path between two points. The user's trip time is optimized by this route-finding feature by:

1. **Finding the shortest path:** Minimizes travel distance or time.
2. **Real-time traffic updates:** Adjusts the route dynamically if there’s traffic congestion, using a combination of historical and real-time data.
3. **Multiple destinations:** Allows for pathfinding with multiple stops, which is typically handled by variations of the Traveling Salesman Problem.

These algorithms enhance the user experience by providing efficient and quick routes, making complex navigational tasks feasible and accessible.

**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?**

We wish to determine for which values of n the inequality 8n2 < 64nlog2(n) holds. This happens when n < 8log2(n), or when n ≤ 43. In other words, insertion sort runs faster when we’re sorting at most 43 items. Otherwise merge sort is faster.

**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 2n on the same machine?**

**Solution:**  
To find when 100n2100n^2100n2 is less than 2n2^n2n, we need to solve the inequality:

100n2<2n100n^2 < 2^n100n2<2n

Because 2n2^n2n grows exponentially, it will eventually surpass 100n2100n^2100n2, which grows polynomially. We can estimate this by trying values of nnn until 100n2100n^2100n2 becomes smaller than 2n2^n2n.

Testing values for nnn:

* **For n=10n = 10n=10:**  
  100×102=10000100 \times 10^2 = 10000100×102=10000 and 210=10242^{10} = 1024210=1024 (false)
* **For n=15n = 15n=15:**  
  100×152=22500100 \times 15^2 = 22500100×152=22500 and 215=327682^{15} = 32768215=32768 (true)
* **For n=14n = 14n=14:**  
  100×142=19600100 \times 14^2 = 19600100×142=19600 and 214=163842^{14} = 16384214=16384 (false)

Thus, the smallest value of nnn such that 100n2<2n100n^2 < 2^n100n2<2n is **n=15n = 15n=15**.

**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 h31; 41; 59; 26; 41; 58i.**

**Initial Array**

The initial array is:

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

**Step-by-Step Insertion Sort**

1. **Iteration 1** (Element = 41, i = 1):
   * Since 41 is already greater than 31, no shifting is required.
   * **Array remains:** [31,41,59,26,41,58][31, 41, 59, 26, 41, 58][31,41,59,26,41,58]
2. **Iteration 2** (Element = 59, i = 2):
   * 59 is greater than 41, so no shifting is required.
   * **Array remains:** [31,41,59,26,41,58][31, 41, 59, 26, 41, 58][31,41,59,26,41,58]
3. **Iteration 3** (Element = 26, i = 3):
   * Compare 26 with 59: shift 59 to the right.
   * Compare 26 with 41: shift 41 to the right.
   * Compare 26 with 31: shift 31 to the right.
   * Place 26 at the beginning of the array.
   * **Array now:** [26,31,41,59,41,58][26, 31, 41, 59, 41, 58][26,31,41,59,41,58]
4. **Iteration 4** (Element = 41, i = 4):
   * Compare 41 with 59: shift 59 to the right.
   * Place 41 in the correct position.
   * **Array now:** [26,31,41,41,59,58][26, 31, 41, 41, 59, 58][26,31,41,41,59,58]
5. **Iteration 5** (Element = 58, i = 5):
   * Compare 58 with 59: shift 59 to the right.
   * Place 58 in the correct position.
   * **Array now:** [26,31,41,41,58,59][26, 31, 41, 41, 58, 59][26,31,41,41,58,59]

**Final Sorted Array**

After all iterations, the sorted array is:

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

**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):**

o The variable sum is initialized to 0.

o 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:

o That is, sum is equal to A[1] + A[2] + ... + A[i-1].

• **During the iteration:**

o The line sum = sum + A[i] updates sum to now include A[i], so:

o 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:

o The loop runs until i = n + 1, meaning the last executed iteration was when i = n.

o 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].

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

def insertion\_sort\_decreasing(arr):

for i in range(1, len(arr)):

key = arr[i] # The element to be inserted

j = i - 1

while j >= 0 and arr[j] < key:

arr[j + 1] = arr[j] # Shift element right

j -= 1

arr[j + 1] = key # Place the key in the correct position

data = [12, 11, 13, 5, 6]

insertion\_sort\_decreasing(data)

print("Sorted array in decreasing order:", data)

# 2.1-4 Consider the *searching problem*:

# Input: A sequence of n numbers [a1,a2 ,,,,,, an ] 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.

FUNCTION LinearSearch(A, n, x)

INPUT: An array A[1:n], an integer n, and a value x

OUTPUT: An index i such that A[i] = x or NIL if x is not in A

FOR i FROM 1 TO n DO

IF A[i] = x THEN

RETURN i

END IF

END FOR

RETURN NIL

END FUNCTION

### Proof of Correctness Using Loop Invariant

To prove that the linear search algorithm is correct, we can use a loop invariant. A loop invariant is a condition that holds true before and after each iteration of the loop.

**Loop Invariant: At the start of each iteration of the loop (for iii from 1 to nnn), the following statement is true:**

"If *xxx* appears in the array *AAA*, then it must appear in the subarray *A[1:i−1]A[1:i-1]A[1:i−1]* or at the current index *iii*."

#### Three Necessary Properties of the Loop Invariant

1. **Initialization (Base Case):**
   * Before the first iteration of the loop (when i=1i = 1i=1), the invariant holds true because we have not examined any elements yet. If xxx is in AAA, it is not in A[1:0]A[1:0]A[1:0] (an empty subarray). Thus, the invariant holds.
2. **Maintenance (Inductive Step):**
   * Assume the invariant holds at the beginning of the iii-th iteration. During this iteration, we check if A[i]=xA[i] = xA[i]=x:
     + If A[i]=xA[i] = xA[i]=x, we return iii, and thus we have found the index where xxx appears.
     + If A[i]≠xA[i] \neq xA[i]=x, then xxx must still be in the remaining subarray A[i+1:n]A[i+1:n]A[i+1:n] or it may have already appeared in A[1:i−1]A[1:i-1]A[1:i−1]. Hence, after checking A[i]A[i]A[i], the invariant still holds for the next iteration (i.e., the start of the (i+1)(i + 1)(i+1)-th iteration).
3. **Termination (Conclusion):**
   * The loop terminates when iii exceeds nnn. At this point, if xxx has not been found (i.e., we have not returned an index), then xxx is not in the entire array A[1:n]A[1:n]A[1:n]. Thus, the algorithm correctly returns NIL, satisfying the output condition.

### Conclusion

Since the loop invariant holds true at initialization, is maintained during each iteration, and leads to a correct conclusion when the loop terminates, we conclude that the linear search algorithm is correct. The algorithm guarantees that if xxx exists in the array AAA, it will find its index; otherwise, it will return NIL, as required.

**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 𝒏−𝟏 𝒊=𝟎 A[i]. 2i , , and b=∑𝒏−𝟏 ⬚ B[i]. 2i . 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=∑𝒏−𝟏 ⬚ C[i]. 2i . 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.**

FUNCTION ADD-BINARY-INTEGERS(A, B, n)

INPUT: Two arrays A[0:n-1] and B[0:n-1] representing binary integers, and an integer n

OUTPUT: An array C[0:n] representing the sum of the two binary integers

DECLARE C[0:n]

C[0] ← 0

C[1:n] ← 0

DECLARE carry ← 0

FOR i FROM 0 TO n-1 DO

sum ← A[i] + B[i] + carry

C[i] ← sum MOD 2

carry ← sum DIV 2

END FOR

C[n] ← carry

RETURN C

END FUNCTION

**Explanation**

1. **Input and Initialization:**
   * The function takes two binary arrays AAA and BBB of length nnn and initializes a new array CCC of length n+1n + 1n+1 to store the result.
   * The first element C[0]C[0]C[0] is initialized to 0 for the carry, and the rest of the elements in CCC are initialized to 0.
2. **Iterating through the Bits:**
   * The algorithm iterates from 0 to n−1n-1n−1, processing each bit of AAA and BBB along with the carry.
   * For each bit index iii, it calculates the total sum of the corresponding bits A[i]A[i]A[i] and B[i]B[i]B[i] and the carry from the previous iteration.
3. **Storing the Result:**
   * The least significant bit of the sum is stored in C[i]C[i]C[i] using sum MOD 2.
   * The carry for the next iteration is updated using sum DIV 2.
4. **Final Carry:**
   * After processing all bits, if there is still a carry left, it is stored in the last element C[n]C[n]C[n].
5. **Return Value:**
   * Finally, the function returns the array CCC, which contains the binary representation of the sum c=a+bc = a + bc=a+b.

**Complexity**

* **Time Complexity:** O(n)O(n)O(n), as the algorithm processes each bit exactly once.
* **Space Complexity:** O(n)O(n)O(n), for the output array CCC.

**Exercises**

**2.2-1 Express the function n 3=1000 C 100n2  100n C 3 in terms of ‚-notation.**

*n*3*/*1000 − 100*n*2 − 100*n*

+ 3 ∈ Θ(*n*3)

**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?**

### Selection Sort

Selection-Sort(A)

1. for i from 1 to n - 1 do

2. min\_index = i

3. for j from i + 1 to n do

4. if A[j] < A[min\_index] then

5. min\_index = j

6. end if

7. end for

8. Exchange A[i] with A[min\_index]

9. end for

**Loop Invariant**

The loop invariant for this algorithm is:

**At the start of each iteration of the outer loop (for each i from 1 to n-1), the subarray A[1:i-1] is sorted, and contains the i-1 smallest elements of the array.**

**Explanation of the Loop Invariant**

* **Initialization:** Before the first iteration (i = 1), the subarray A[1:0] is considered sorted (as it contains no elements).
* **Maintenance:** During each iteration of the outer loop, the smallest element from the remaining unsorted subarray (A[i:n]) is found and placed in position A[i]. After this exchange, A[1:i] becomes sorted.
* **Termination:** After the last iteration (i = n-1), the entire array A[1:n] is sorted since the last element A[n] will naturally be the largest element.

**Why Run for Only the First n-1 Elements**

* The algorithm only needs to run for the first n-1 elements because by the time we have placed the first n-1 elements in their correct positions, the last element will automatically be in its correct position. This is because it is the only remaining element left in the array, which must be the largest if all previous elements are sorted.

**Worst-case Running Time**

The worst-case running time of selection sort can be analyzed as follows:

1. The outer loop runs n-1 times (for i from 1 to n-1).
2. For each iteration of the outer loop, the inner loop runs n - i times to find the minimum element.

Thus, the total number of comparisons made can be calculated as:

(n−1)+(n−2)+…+1=(n−1)⋅n2(n - 1) + (n - 2) + \ldots + 1 = \frac{(n - 1) \cdot n}{2}(n−1)+(n−2)+…+1=2(n−1)⋅n​

This gives us a worst-case running time of:

O(n2)O(n^2)O(n2)

**Best-case Running Time**

The best-case running time for selection sort is also:

O(n2)O(n^2)O(n2)

This is because, even in the best-case scenario where the array is already sorted, the algorithm still performs the same number of comparisons to find the minimum element for each position, as it does not take advantage of the existing order in the array.

**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 iequally 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 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:

Average elements checked=1+2+3……..+𝑛 =

n

𝑛+1 ≈ 𝑛

2 2

In big-O notation, we express this as:

O(n)

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

Θ(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 checked=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:

Θ(n)

**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 14 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).

**Exercises**

**2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence h3; 41; 52; 26; 38; 57; 9; 49i.**

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 |

**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.**

**MERGE-SORT Overview**

The MERGE-SORT algorithm is a recursive sorting algorithm that works by:

1. Dividing the array into two halves.
2. Recursively sorting each half.
3. Merging the sorted halves back together.

**The Recursive Structure**

In the MERGE-SORT procedure, the array is typically divided at a midpoint q, where:

q=⌊(p+r)/2⌋q = \lfloor (p + r) / 2 \rfloorq=⌊(p+r)/2⌋

The recursive calls for sorting the two halves of the array would look like this:

* **Left half**: MERGE-SORT(A, p, q)
* **Right half**: MERGE-SORT(A, q + 1, r)

**Understanding the Base Condition**

The test statement <if p ≠ r> is used to determine when the algorithm should stop recursing. This test acts as a base case for the recursion. If p = r, it means that the subarray has only one element (or none, if we consider an empty array).

The key points to note are:

1. **No Subarray with p > r**: The indices p and r denote the boundaries of the subarray being sorted. For the base case:
   * If p = r, the array is either a single element or empty.
   * If p > r, the array is considered invalid and should not be processed.
2. **Ensuring Valid Indices**: When the recursive calls are made, we always split the array at the midpoint. Since the indices are updated based on the current values of p and r:
   * For the left subarray, the new call is made with p and q, where p remains the same and q is the midpoint.
   * For the right subarray, the new call is made with q + 1 and r.

**Argument Against p > r**

Now, let's argue why <if p ≠ r> is sufficient to ensure that no recursive call results in p > r:

* **Initial Call**: When MERGE-SORT is initially called with (A, 1, n), we have 1 ≤ n. This means p starts at 1 and r at n.
* **Recursive Calls**: The recursive calls involve splitting the indices:
  + For any valid p and r, q is always computed such that:

p≤q≤rp \leq q \leq rp≤q≤r

* + Consequently, for the left recursive call MERGE-SORT(A, p, q), p remains the same and q is less than or equal to r. This keeps p ≤ q ≤ r, ensuring that p ≤ r in this case.
  + For the right recursive call MERGE-SORT(A, q + 1, r), since q is at most r, q + 1 is always at most r + 1. Therefore, p is updated to q + 1, and we still have:

p≤rp ≤ rp≤r

**Conclusion**

Because of the way the recursive calls are structured, starting with a valid range defined by 1 ≤ n, and since each subsequent recursive call maintains the property that p cannot exceed r, it follows that:

* The only time p could equal r or stay valid is if the algorithm reaches its base case, which is handled by the condition <if p ≠ r>.
* Thus, if the condition holds, p > r cannot occur, and the algorithm will not attempt to sort an invalid or empty subarray.

Therefore, the test <if p ≠ r> is indeed sufficient to ensure that no recursive call has p > r, as it guarantees that the algorithm will only proceed with valid subarray ranges.

**2.3-3 State a loop invariant for the while loop of lines 12318 of the MERGE procedure. Show how to use it, along with the while loops of lines 20323 and 24327, to prove that the MERGE procedure is correct.**

# 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.

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

**𝟐, 𝒊𝒇 𝒏 = 2,**

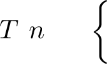
**𝑻(𝒏) = {**

**𝟐𝑻(𝒏/𝟐), 𝒊𝒇 𝒏 > 𝟐**

**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

Θ(1) if *n* ≤ *c*

( ) =

*T*(*n* − 1) + *I*(*n*) otherwise

where *I*(*n*) denotes the amount of time it takes to

insert *A*[*n*] into the sorted array *A*[1*..n* − 1]. Since we may have to shift as many as *n* − 1 elements once we find the correct place to insert *A*[*n*], we have *I*(*n*) = *θ*(*n*).

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

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*] ≤ *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**

**2.3-5 You can also think of insertion sort as a recursivalgorithm. 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.**

**Recursive Insertion Sort**

function RecursiveInsertionSort(A, n):

if n <= 1:

return

RecursiveInsertionSort(A, n - 1)

key = A[n]

i = n - 1

while i > 0 and A[i] > key:

A[i + 1] = A[i]

i = i - 1

A[i + 1] = key

**Recurrence for Worst-Case Running Time**

In the worst-case scenario, when the array is sorted in reverse order, the insertion of the nth element requires shifting all previous elements. The running time can be expressed as follows:

* Let T(n)T(n)T(n) be the worst-case running time for sorting an array of size nnn.
* The time to sort the first n−1n-1n−1 elements is T(n−1)T(n-1)T(n−1).
* In the worst case, inserting the nth element requires O(n)O(n)O(n) comparisons and shifts.

Thus, the recurrence relation for the worst-case running time is:

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

**Solving the Recurrence**

To solve the recurrence, we can expand it:

T(n)=T(n−1)+O(n)

=(T(n−2)+O(n−1))+O(n)

=T(n−2)+O(n−1)+O(n)

=T(n−3)+O(n−2)+O(n−1)+O(n)

Continuing this expansion leads to:

=T(1)+O(2)+O(3)+…+O(n)

Since T(1)T(1)T(1) is a constant O(1), we can ignore it for large n:

T(n)=O(1)+O(2)+O(3)+…+O(n)

=O(n(n+1)/2​)=O(n2)

### Conclusion

The worst-case running time of the recursive insertion sort algorithm is O(n2).

**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).**

**Binary Search (Iterative)**

Binary Search (A, v, low, high):

# 1. while low ≤ high:

# 2. mid ← (low + high) / 2

# 3. if A[mid] == v:

# 4. return mid

# 5. else if A[mid] < v:

# 6. low ← mid + 1

# 7. else:

# 8. high ← mid - 1

# 9. return -1 // v is not in A

# Binary Search (Recursive)

BinarySearch(A, v, low, high):

1. if low > high:

2. return -1 // v is not in A

3. mid ← (low + high) / 2

4. if A[mid] == v:

5. return mid

6. else if A[mid] < v:

7. return BinarySearch(A, v, mid + 1, high)

8. else:

9. return BinarySearch(A, v, low, mid - 1)

**Explanation of Worst-Case Running Time O(log n)**

1. **Divide and Conquer Approach**: In each iteration (or recursive call), the binary search algorithm splits the array into two halves. This division reduces the size of the search space by half each time.
2. **Number of Steps**: If the initial array has nnn elements, then:
   * After the first check, the search space becomes n/2.
   * After the second check, it becomes n/4.
   * This continues until the search space reduces to 1.
3. **Logarithmic Complexity**: The maximum number of times you can divide nnn by 2 until reaching 1 is log log2​n. Therefore, the worst-case time complexity is O(log n).

# 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 Θ(nlg(n)). We will prove

correctness by a mutual induction. Let mi,j be the proposition A[i]+A[j] < S and Mi,j be the proposition

A[i]+A[j] > S. Note that because the array is sorted, mi,j ⇒

∀k < j,mi,k, and Mi,j ⇒ ∀k > i,Mk,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 ≤ 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(nlg(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 mi,k this means mi,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 Mk,j this means Mi,j

Note that one of these two cases must be true since the set of considered points separates {(m,m0) : m ≤ m0 < 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 ‚.n lg n/ time in the worst case.**

**Algorithm**

1. **Sort the set SSS of nnn integers** in O(nlog⁡n)O(n \log n)O(nlogn) time.
2. **Initialize two pointers**: one pointer left\text{left}left at the beginning of the sorted list and another pointer right\text{right}right at the end of the list.
3. **Iterate**:
   * Calculate the sum of the elements at the positions of the two pointers, sum=S[left]+S[right]\text{sum} = S[\text{left}] + S[\text{right}]sum=S[left]+S[right].
   * If sum==x\text{sum} == xsum==x, return **True** (you've found two elements that sum to xxx).
   * If sum<x\text{sum} < xsum<x, move the left\text{left}left pointer one position to the right (increment left\text{left}left by 1) to increase the sum.
   * If sum>x\text{sum} > xsum>x, move the right\text{right}right pointer one position to the left (decrement right\text{right}right by 1) to decrease the sum.
4. **Continue** this process until the left\text{left}left pointer is no longer less than the right\text{right}right pointer. If no such pair is found, return **False**.

**Explanation of Time Complexity**

1. **Sorting the set** takes O(nlog n) time.
2. **Using two pointers** to check each pair of elements takes O(n) time, as each pointer only moves in one direction and meets the other in at most nnn steps.

Thus, the overall time complexity of this algorithm is O(nlog n), which meets the requirement.

**Example**

Suppose S={1,4,5,7,11,12}S = \{1, 4, 5, 7, 11, 12\}S={1,4,5,7,11,12} and x=15x

Sort SSS (already sorted in this case).

Initialize left= = 0 (points to 1) and right=5 (points to 12).

Check the sum S[left]+S[right]=1+12=13

Since 13<15, increment left to point to 4.

Check the sum S[left]+S[right]=4+12=16

Since 16>15, decrement right to point to 11.

Check the sum S[left]+S[right]=4+11=15

Since 15= =15 , return **True**.

**Chapter# 3**

**Exercises 3.1-1 Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.**

The lower-bound argument for insertion sort generally states that insertion sort has a worst-case time complexity of Ω(n2) because, in the worst case, each element in an unsorted list of n elements has to be compared with each of the previous elements to find its correct position.

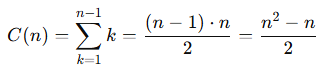
Here's how we can generalize this argument for input sizes that are not necessarily a multiple of 3:

**Understanding the Lower Bound**

In insertion sort, every element (except the first one) needs to be compared with the elements that have already been sorted to determine its position in the sorted section. Therefore, in the worst case (when the input list is in reverse order), the iii-th element will have to be compared with each of the i−1 previous elements. This results in a sum of comparisons that grows as n2.

**Modifying the Argument for Arbitrary nnn**

We can calculate the total number of comparisons in the worst case as follows:



This formula holds regardless of whether n is a multiple of any specific number (such as 3). It’s simply the total number of comparisons required for insertion sort when all elements are in the worst possible order.

**Conclusion**

Since this formula and the resulting time complexity, Ω(n2), do not depend on whether nnn is a multiple of 3, no modification is required to handle arbitrary input sizes. Thus, the lower-bound argument for insertion sort still applies directly to any n, confirming that the time complexity remains Ω(n2) for any input size.

**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 = 2b and n0 ≥ 2a. Then for all n ≥ n0 we have (n+a)b ≤ (2n)b = cnb

so (n+a)b = O(nb). Now let and . Then

if and only if if and only if n + a ≥ (1/2)a/bn if and only if (n+a)b ≥ cnb. Therefore (n+a)b = Ω(nb). By Theorem 3.1, (n+a)b = Θ(nb).

**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 growthrate 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.

# 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.

# 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(α)=α(1−α)

# Finding the Maximum:

* + Taking the derivative:

f′(α)=1−2α

* + Setting the derivative to zero: 1−2α=0⟹α=1/2

# Conclusion

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

**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) ≥ 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).

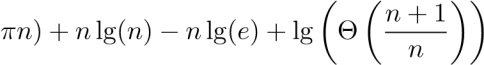
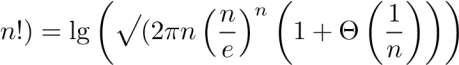
# 3.2-2 Explain why the statement, <The running time of algorithm A is at least O(n2) is meaningless.

The statement "The running time of algorithm A is at least O(n2)O(n^2)O(n2)" is meaningless for the following reasons:

1. **Misinterpretation of Big O**: Big O notation describes an upper bound on an algorithm's running time. Saying "at least O(n2)" confuses the concept of upper bounds with lower bounds.
2. **Correct Terminology**: To express a lower bound, one should use Ω(n2). The phrase "at least" does not clearly communicate the algorithm's performance.
3. **Ambiguity**: The statement lacks clarity and may mislead about the algorithm's efficiency, implying it is inherently complex when it may not be.

# 3.2-3 Is 2n+1. =0(2n)? Is = 0 (2n)?

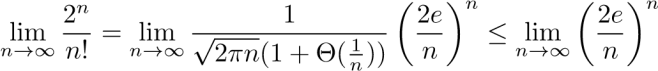
 As the hint suggests, we will apply stirling’s approximation lg(



= lg(2

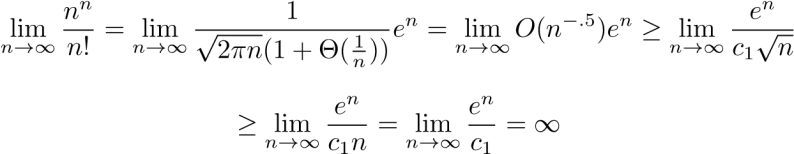
Note that this last term is *O*(lg(*n*)) if we just add the two expression we get when we break up the lg instead of

subtract them. So, the whole expression is dominated by

*n*lg(*n*). So, we have that lg(*n*!) = Θ(*n*lg(*n*)).

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





**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* ≥ *n*0 the inequality dlog*n*e! ≤ *cna* would hold. In particular, it would hold when *n* = 2*k* for *k* ∈ N. Then this becomes *k*! ≤ *c*(2*a*)*k*, a contradiction since the factorial function is not exponentially bounded.

We’ll show that dloglog*n*e! ≤ *n*. Without loss of generality assume *n* = 22*k*.

Then this becomes equivalent to showing *k*! ≤ 22*k*, or 1 · 2···(*k* − 1)

· *k* ≤

4 · 16 · 28 ···22*k*, which is clearly true for *k* ≥ 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.

# 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.

# 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) × (3𝑛) = 9𝑛2⬚

# Conclusion:

* + Since the number of comparisons is at least n29\frac{n^2}{9}9n2, we have:

T(n)=Ω(n2)

This shows that the worst-case time complexity of insertion sort is Ω(n2).

# 3.2-5 Prove that the running time of an algorithm is Θ (g(n)) if and only if its worst-case running time is O(g(n)) and its best-case running time is Ω (g(n)).

**Proof**

# 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).

o 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))

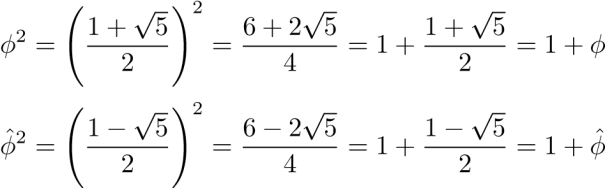
* + From T(n)=O(g(n)) there exist constants c2>0 and n0 such that: T(n)≤c2g(n)for all n≥n0
  + From T(n)=Ω(g(n)) there exist constants c1>such that:
  + T(n)≥c1g(n)for all n≥0
  + Combining these, we have:
  + c1g(n)≤T(n)≤c2g(n)
  + Thus, T(n)=Θ(g(n)).

# Conclusion

We have shown that:

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

# 3.2-6 Prove that o(g(n)) ⩀ ⩀(g(n)) is the empty set.



**3.2-7** 𝑾𝒆 𝒄𝒂𝒏 𝒆𝒙𝒕𝒆𝒏𝒅 𝒐𝒖𝒓 𝒏𝒐𝒕𝒂𝒕𝒊𝒐𝒏 𝒕𝒐 𝒕𝒉𝒆 𝒄𝒂𝒔𝒆 𝒐𝒇 𝒕𝒘𝒐 𝒑𝒂𝒓𝒂𝒎𝒆𝒕𝒆𝒓𝒔 𝒏 𝒂𝒏𝒅 𝒎 𝒕𝒉 ∞ 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 ≤ 𝒇 (𝒏, 𝒎) ≤ 𝒄𝒈 (𝒏, 𝒎)

𝒇𝒐𝒓 𝒂𝒍𝒍 𝒏 ≥ 𝒏𝟎 𝒐𝒓 𝒎 ≥ 𝒎𝟎} ∶

𝑮𝒊𝒗𝒆 𝒄𝒐𝒓𝒓𝒆𝒔𝒑𝒐𝒏𝒅𝒊𝒏𝒈 𝒅𝒆û𝒏𝒊𝒕𝒊𝒐𝒏𝒔 𝒇𝒐𝒓 (𝒈 (𝒏, 𝒎)) 𝒂𝒏𝒅 ((, 𝒎)).

# Definitions

**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.

# 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*≤*n*⟹*f*(*m*)≤*f*(*n*) **(1)**

*m*≤*n*⟹*g*(*m*)≤*g*(*n*) **(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*)⋅*g*(*m*)≤*f*(*n*)⋅*g*(*n*)

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

# 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:**

• bαnc: the greatest integer less than or equal to αn.

• d(1−α)n=(1−α)n.

**Decompose n:**

N=bαnc+d(1−α)n+r

where ris the remainder from αn.

Express bαnc: with 0≤r<10.

**Substituting:**

bαnc=αn−r

bαnc+d(1−α)n=(αn−r)+(1−α)n

**Simplify:**

=αn+(1−α)n−r=n−r

**Final equality:**

n=bαnc+d(1−α)n+r

Thus, the equation bαnc+d(1−α)n=n holds true.

# 3.3-3 Use equation (3.14) or other means to show that (n+0(n))k = Θ(nk) for all real constant k. Conclude that (nk)= Θ (nk) and (nk)= Θ (nk).

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

𝑓(𝑛) ⬚

lim ( ) = 0

𝑛→∞ 𝑛

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

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

(n+o(n))k = nk (1 + 0(𝑛)) 𝑘

𝑛

**Step 3: Analyzing** (𝟏 + (𝒏)) 𝒌

𝒏

Now, let’s analyze the term (𝟏 + (𝒏)) 𝒌. 𝒔𝒊𝒏𝒄𝒆 (𝒏) → 0 𝑎𝑠 𝑛 →

n n

∞ 𝑤𝑒 𝑐𝑎𝑛 𝑢𝑠𝑒 𝑡ℎ𝑒 𝑙𝑖𝑚𝑖𝑡:

**Step 4: Concluding** (n+o(n))k=Θ(nk) Thus, for large n:

(n+o(n))k=nk⋅(1+o(1)), where o(1)→0 as n→∞n →∞. This shows that: (n+o(n))k=nk+o(nk),

indicating that (n+o(n))k =Θ(nk. **Step 6: Concluding the Results** 1.(n+o( ))k =Θ(nk)

1. nk=Θ(nk)

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

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

# For (lgn)!:

* + The logarithm function lgn grows very slowly compared to

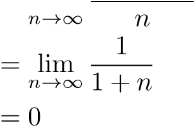
n. As n increases, lgn approaches infinity, and thus (lgn)! will also grow, but we need to analyze its growth in relation to polynomial functions.

* + 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∗(2*n*))

lim∗ = *n*lim→∞ lg∗(lg(2*n*)) *n*→∞ lg



(lg(*n*)) lg(1 + lg∗(*n*)) lg∗(*n*)

Lim

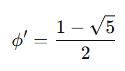
# 3.3-7 Show that the golden ratio ∅ and its conjugate ∅ y both satisfy the equation X2 =x+1.

To show that the golden ratio ϕ and its conjugate ψ satisfy the equation x2=x+1, we first need to define the golden ratio and its conjugate.

The golden ratio ϕ\phiϕ is defined as:

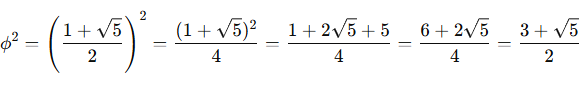
Screenshot (116).png

The conjugate of the golden ratio ϕ is given by:



Step 1: Verify for ϕ

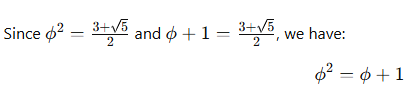
Calculate ϕ:



Substitute ϕ into the equation x+1x:

Screenshot (118.png

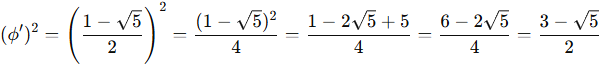
Compare ϕ2 with ϕ+1:



Thus, ϕ satisfies the equation x2=x+1.

**Step 2: Verify for ϕ′**

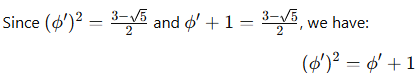
Calculate (ϕ′)2:



Substitute ϕ′ into the equation x+1:

Screenshot (120.png

Compare (ϕ′)2 with ϕ′+1 :



Thus, ϕ′ also satisfies the equation x2=x+1.

**Conclusion**

Both the golden ratio ϕ and its conjugate ϕ′ satisfy the equation:

x^2 = x + 1

**3.3-8 Prove by induction that the ith Fibonacci number satisûes the equation Fi D .�i  �yi /=p 5 ; where � is the golden ratio and �y is its conjugate.**

To prove that the i-th Fibonacci number Fi satisfies the equation

𝐹𝑖 = ⏀⬚\_𝜓

2√5

where ϕ=1+√5(the golden ratio) and

2

ψ=1\_√5 (𝑖𝑡𝑠 𝑐𝑜𝑛𝑗𝑢𝑔𝑎𝑡𝑒), 𝑤𝑒 𝑤𝑖𝑙𝑙 𝑢𝑠𝑒 𝑚𝑎𝑡ℎ𝑒𝑚𝑎𝑡𝑖𝑐𝑎𝑙 𝑖𝑛𝑑𝑢𝑐𝑡𝑖𝑜𝑛.

2

First, we check the base cases i=0 and i=1:

* For i=0:

F0=0

⏀1\_𝜓1 =⏀\_𝜓=1+√5 1\_√5 = √5 = 1

√5 √5 5 5 √5

Thus, the base case holds for i=1

𝐹𝑖 = ⏀n⬚\_𝜓𝑛

√5

and Fn−1= ⏀n\_1\_𝜓𝑛\_1

√5

𝐹𝑛+1=𝐹𝑛= + 𝐹𝑛\_1=

# 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

Screenshot (126.png

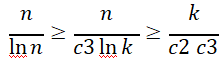
so that n

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



k = Θ .

ln 𝑛