USN										

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A++ GRADE BY NAAC

Internal Assessment Test III – December 2023

Sub:	Analysis & Design of Algorithms					Sub Code:	BCS401	Branch:	AIDS & CSE(AIDS)	
Date:	**5/8/2024**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**IV -A, B & C**			**OBE**

Answer any FIVE FULL Questions

			MARKS	CO	RBT
1	a	Apply backtracking to solve the following instance of the subset sum problem: A={1,3,4,5}and d=11 **Answer:** **The problem is a subset sum problem where you are tasked with finding subsets of the set whose sum is equal to , using the backtracking algorithm.** **Here's the step-by-step solution:** **Step 1: Problem Definition** **You are given:** **A set A = 1, 3, 4, 5** **A target sum d = 11** **The goal is to find all subsets of that sum to 11, using backtracking** **Step 2: Backtracking Algorithm** **1. Start with an empty subset and a current sum of 0.** **2. Add elements to the subset one by one, recursively checking if the current sum equals the target sum , exceeds it, or can be extended further.** **3. If the current sum exceeds , backtrack by removing the last added element.** **4. If the current sum equals , print/store the subset as a solution.** **5. Repeat until all possibilities are explored.**	8	6	L3

Step 3: Recursive Function for Backtracking

The recursive function works as follows:

Start with the first element of the set.

For each element, decide whether to include it in the subset.

Move to the next element and repeat the process.--

Step 4: Apply Backtracking

Step-by-step Execution:

1. Initial State: Start with an empty subset and sum = 0. Index = 0.

2. Include 1:

Subset: 1, Current Sum: 1.

Move to the next index (Index = 1).

3. Include 3:

Subset:1,3,4 , Current Sum: .4

Move to the next index (Index = 2).

4. Include 4:

Subset: 1,3,4, Current Sum: 8.

Move to the next index (Index = 3).

5. Include 5:

Subset: 1,3,4,5, Current Sum:13 .

Exceeds d =11 . Backtrack.

6. Backtrack

Remove 5.

Current Subset: 1,3,4, Current Sum: 8.

Skip 5. Backtrack.

7. Backtrack:

Remove 4.

Current Subset: 1,3, Current Sum: 4.

8. Include 5:

Subset: 1,3,5, Current Sum: 9.

Backtrack.

9. Backtrack:

Remove 3.

Current Subset: 1, Current Sum: 1.

10. Include 4:

Subset: 1,4, Current Sum: 5.

Move to the next index.

11. Include 5:

Subset: 1, 4,5, Current Sum: 11.

Solution Found: 1 ,4 ,5

12. Backtrack:

Remove 5.

Current Subset: 1,4 Current Sum: 5.

Backtrack.

13. Backtrack:

Remove 4.

Current Subset: 1, Current Sum: 1.

Backtrack.

14. Include 3:

Subset: 3,, Current Sum:3 .

Move to the next index.

15. Include 4:

Subset: 3,4, Current Sum: 7.

Move to the next index.

16. Include 5:

Subset: 3,4,5, Current Sum: 12.

Exceeds .d=11, Backtrack.

17. Backtrack:

Remove 5.

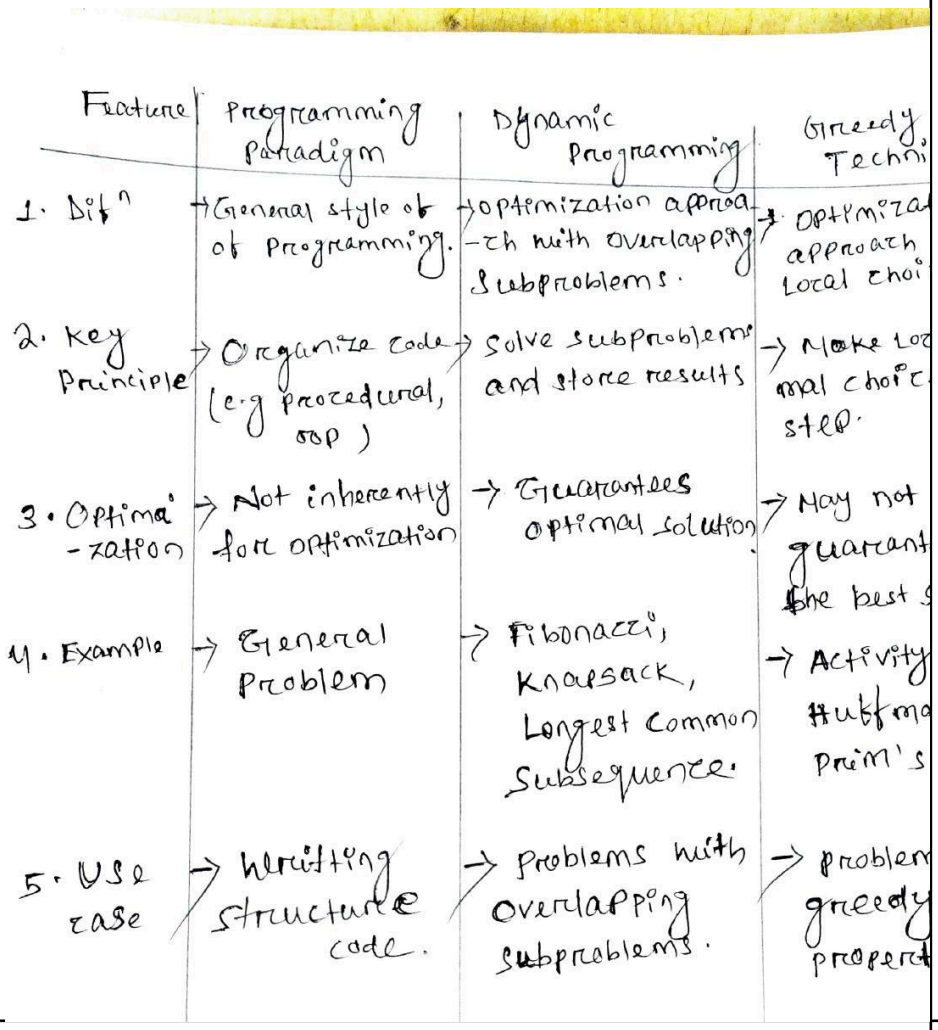
Current Subset:3,4 , Current Sum: 7.

Backtrack.

18. Include 5:

Subset: 3,5, Current Sum: .8

		Backtrack. **19. Backtrack:** **Remove 3.** **Current Subset: 4, Current Sum: 4.** **20. Include 5:** **Subset:4,5 , Current Sum: 9.** **Backtrack.** **21. Backtrack:** **Remove 4.** **Current Subset:5 , Current Sum: 5.** **Backtrack.** **Step 5: Final Solutions** **After exploring all possibilities, the subsets that sum to are: 1,4,5**			
	b	Give an example of an algorithm which is infinite in nature. **Answer:** An example of an infinite algorithm is the "infinite loop" algorithm, often used in situations where a program needs to run continuously until an external event stops it. For example: while True: print("This is an infinite loop.") Explanation: The while True condition always evaluates to True, so the loop never terminates on its own. The algorithm will continue printing "This is an infinite loop." until it is externally interrupted (e.g., by a user or system signal). Practical Use Cases:	2	5	L2

		Operating Systems: Infinite loops are used to keep event listeners or background processes running. Server Applications: Web servers continuously listen for incoming requests in an infinite loop. Real-Time Systems: Embedded systems or hardware controllers use infinite loops to monitor and respond to inputs. While the loop is infinite in principle, it often includes mechanisms to break or terminate the loop based on specific conditions, such as user input, error handling, or system shutdown.			
2	a	Compare programming paradigms, dynamic programming and greedy techniques with example. **Answer:**  	4	3	L3
	b	Explain P,NP,NP-complete problem with examples. P, NP, and NP-Complete Problems These terms are important concepts in computer science, particularly in computational complexity theory. Here's a brief explanation of each, along with examples: 1. **P (Polynomial Time)** Definition: The class of problems that can be solved by a deterministic Turing machine in polynomial time. In simpler terms, these are problems for which an efficient algorithm exists to find a solution.	6	5	L2

Characteristics:

Easy to solve and verify.

Examples of P problems include sorting an array, finding the shortest path in a graph (Dijkstra's Algorithm), and matrix multiplication.

Example:

Problem: Sorting a list of numbers.

Algorithm: Merge Sort or Quick Sort has a time complexity of O(n log n), which is polynomial.

2. **NP (Nondeterministic Polynomial Time)**

Definition: The class of problems for which a proposed solution can be verified in polynomial time by a deterministic Turing machine.

In other words, while finding the solution might be hard, verifying whether a given solution is correct is efficient.

Characteristics:

May not have efficient algorithms to find solutions.

Examples of NP problems include the Traveling Salesman Problem (TSP) and Subset Sum Problem.

Example:

Problem: Subset Sum Problem.

Description: Given a set of integers, is there a subset whose sum equals a given number?

Verification: If someone provides a subset, we can quickly verify if the sum matches the target.

3. **NP-Complete**

Definition: NP-complete problems are a subset of NP problems that are as hard as any problem in NP. If a polynomial-time solution is found for one NP-complete problem, it can be used to solve all NP problems in polynomial time.

Characteristics:

Solutions are hard to find, but easy to verify.

Known for their computational difficulty.

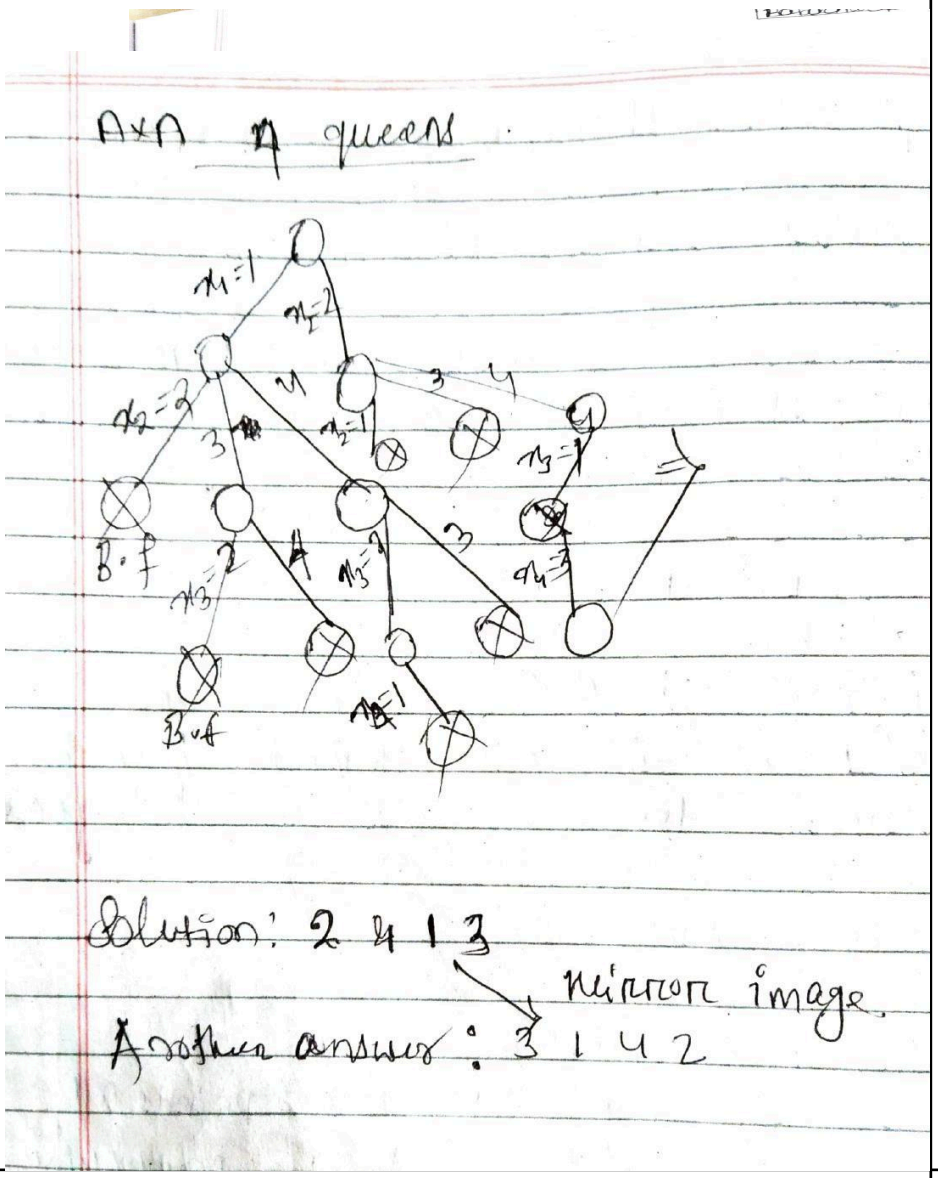
Examples include the SAT (Boolean Satisfiability) Problem and the Traveling Salesman Problem (optimization version).

Example:

Problem: Traveling Salesman Problem (Decision Version).

Description: Given a set of cities and distances, is there a route that visits each city exactly once and has a total distance less than or equal to a given value?

Verification: If someone provides a route, checking its distance is polynomial.

		Relationship Between P, NP, and NP-Complete P ⊆ NP: All problems in P are also in NP because if a problem can be solved quickly, it can also be verified quickly. NP-Complete ⊆ NP: NP-complete problems are the hardest problems in NP. It is unknown whether P = NP or P ≠ NP, and this is one of the biggest open questions in computer science. Diagram Representation P ⊆ NP ⊇ NP-Complete P problems are inside NP, and NP-Complete problems are the hardest subset of NP.			
3	a	What is Backtracking ?How is 4 queens problem solved using Backtracking. **Answer:**  	10	6	L3
4	a	Apply Horspool's algorithm and write its c function Text: JIM_SAW_ME_IN_BARBERSHOP Pattern : BARBER **Answer:** #include <stdio.h> #include <string.h>	10	4	L3

```
#include <stdlib.h>
#define MAX_CHAR 256 // Maximum number of characters in the ASCII set
// Function to create the shift table
void buildShiftTable(char *pattern, int table[], int patternLength) {
    for (int i = 0; i < MAX_CHAR; i++) {
        table[i] = patternLength; // Default shift for characters not in the pattern
    }
    for (int i = 0; i < patternLength - 1; i++) {
        table[(unsigned char)pattern[i]] = patternLength - 1 - i;
    }
}
// Horspool's algorithm function
int horspool(char *text, char *pattern) {
    int textLength = strlen(text);
    int patternLength = strlen(pattern);
    int shiftTable[MAX_CHAR];
    // Build the shift table
    buildShiftTable(pattern, shiftTable, patternLength);

    // Search for the pattern in the text
    int i = patternLength - 1; // Start matching from the end of the pattern
    while (i < textLength) {
        int k = 0;
        // Check for a match
        while (k < patternLength && pattern[patternLength - 1 - k] == text[i - k]) {
            k++;
        }
        if (k == patternLength) {
            return i - patternLength + 1; // Pattern found, return starting index
        }
        // Shift the pattern
        i += shiftTable[(unsigned char)text[i]];
    }
    return -1; // Pattern not found
}
int main() {
    char text[] = "JIM_SAW_ME_IN_BARBERSHOP";
```

		char pattern[] = "BARBER";
		int position = horspool(text, pattern);
		if (position != -1) {
		printf("Pattern found at position: %d\n", position);
		} else {
		printf("Pattern not found in the text.\n");
		}
		return 0;
		}			

5	a	Construct AVL Tree for the following: 21,26,30,9,4,14,28,18,15,10,2,3,7 along with explaining Rotations used to Balance the AVL Tree. teps to construct an AVL tree and balance it: An AVL tree is a self-balancing binary search tree (BST). After each insertion, we calculate the balance factor (difference in height of left and right subtrees) for each node. If the balance factor becomes -2 or 2, rotations are performed to restore balance. Given Elements: 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7 Insert Elements One by One and Balance: 1. Insert balancing needed. 2. Insert 26: Tree: 21 \ 26 No balancing needed. 3. Insert 30: Tree: 21 \ 26 \ 30 Balance Factor of 21 = -2 → Right-Right (RR) imbalance → Perform Left Rotation on 21. After rotation:	10	3	L3

26

/ \

21 30

4. Insert 9:

Tree:

26

/ \

21 30 / 9

- No balancing needed.

5. Insert 4:

Tree:

26

/ \

21 30

/

9

/

4

- Balance Factor of 21 = 2 → *Left-Left (LL)* imbalance → Perform *Right Rotation* on 21.

- After rotation:

26

/ \

9 30

/ \

4 21

6. Insert 14:

Tree:

26

/ \

9 30

/ \

4 21

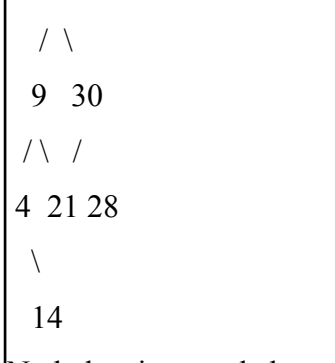
\

14

No balancing needed.

7. Insert 28:

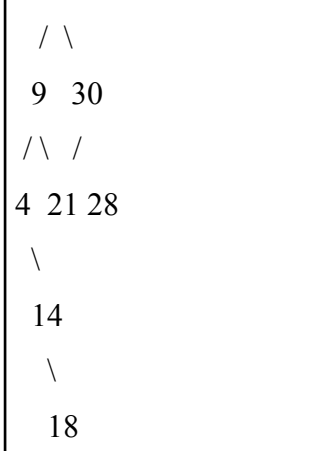
Tree:



No balancing needed.

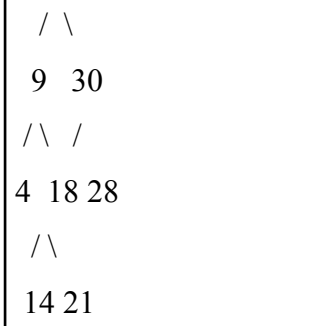
8. Insert 18:

Tree:



Balance Factor of 21 = 2 → Left-Right (LR) imbalance → Perform Left Rotation on 14, then Right Rotation on 21.

After rotation:



9. Insert 15:

Tree:

26

/ \

9 30

/ \ /

4 18 28

/ \

14 21

\

15

No balancing needed.

10. Insert 10:

Tree:

26

/ \

9 30

/ \ /

4 18 28

/ \

14 21

\

15

/

10

Balance Factor of 9 = -2 → Right-Left (RL) imbalance → Perform Right Rotation on 18, then Left Rotation on 9.

After rotation:

26

/ \

14 30

/ \ /

9 18 28

/ \

4 10 21 / 15

11. Insert 2:

Tree:

26

/ \

14 30

/ \ /

9 18 28

/ \

4 10 21 / 2 / 15

- No balancing needed.

12. Insert 3:

Tree:

26

/ \

14 30

/ \ /

9 18 28

/ \

4 10 21 /

2 3 / 15

- Balance Factor of 4 = 2 → *Left-Right (LR)* imbalance → Perform *Left Rotation* on 2, then *Right Rotation* on 4.

- After rotation:

26

/ \

14 30

/ \ / 9 18 28 / \

3 10 21 / / 2 15

13. Insert 7:

Tree:

26

/ \

14 30

/ \ /

9 18 28

/ \

3 10 21 / \ / 2 7 15

- No balancing needed.

		Final AVL Tree: 26 / \ 14 30 / \ / 9 18 28 /\ \ 3 10 21 / \ / 2 7 15 Rotations Used: 1. Left Rotation: After inserting 30 (RR imbalance) 2. Right Rotation: After inserting 4 (LL imbalance). 3. Left-Right Rotation: After inserting 18 (LR imbalance). 4. Right-Left Rotation: After inserting 10 (RL imbalance). 5. Left-Right Rotation: After inserting 3 (LR imbalance).			
6	a	Sort the given array using Counting sort: [45,2,19,10,33,2,2,1,23]. **Answer:** **Counting Sort works by counting the occurrences of each element in the input array and then using those counts to place elements in their correct positions. Here's how to sort the array [45, 2, 19, 10, 33, 2, 2, 1, 23] step-by-step:** **Steps:** **1. Find the range of the array:** **Identify the minimum and maximum values.** **Minimum = 1, Maximum = 45.** **2. Initialize the counting array:** **Create a counting array of size max - min + 1 to store the frequency of each element.** **3. Count the occurrences:** **Populate the counting array with the frequency of each element in the input array.** **4. Compute cumulative counts:** **Update the counting array to reflect the positions of elements.** **5. Place elements in the sorted array:** **Use the cumulative counts to place elements in their correct positions in the output array.** **6. Return the sorted array.** **#include <stdio.h>** **#include <stdlib.h>**	7	3	L3

```
void counting_sort(int arr[], int n) {
    // Find the minimum and maximum values in the array
    int min = arr[0], max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < min)
            min = arr[i];
        if (arr[i] > max)
            max = arr[i];
    }

    int range = max - min + 1;

    // Create and initialize the count array
    int *count = (int *)calloc(range, sizeof(int));

    // Count the occurrences of each element
    for (int i = 0; i < n; i++) {
        count[arr[i] - min]++;
    }

    // Compute cumulative counts
    for (int i = 1; i < range; i++) {
        count[i] += count[i - 1];
    }

    // Create a sorted array
    int *sorted_arr = (int *)malloc(n * sizeof(int));
    for (int i = n - 1; i >= 0; i--) {
        sorted_arr[count[arr[i] - min] - 1] = arr[i];
        count[arr[i] - min]--;
    }

    // Copy the sorted array back into the original array
    for (int i = 0; i < n; i++) {
        arr[i] = sorted_arr[i];
    }

    // Free dynamically allocated memory
    free(count);
    free(sorted_arr);
}

int main() {
    int arr[] = {45, 2, 19, 10, 33, 2, 2, 1, 23};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    counting_sort(arr, n);

    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
```

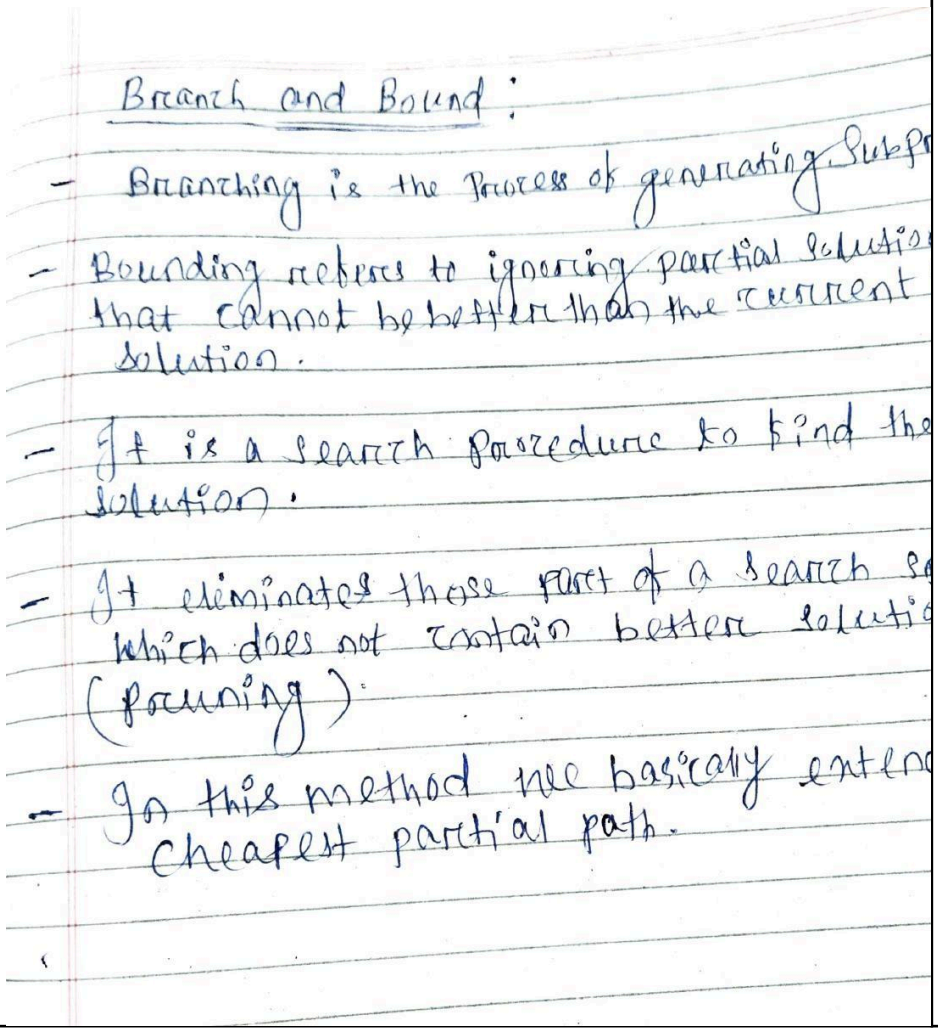
```
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

The sorted array is:

[1, 2, 2, 2, 10, 19, 23, 33, 45]

b	Write a Short note on Branch and Bound. **Answer:** 	3	6	L2

Branch and Bound:

- Branching is the process of generating Subp
- Bounding refers to ignoring partial solutio that cannot be better than the current solution.
- It is a search procedure to find the solution.
- It eliminates those part of a search s which does not contain better solutio (pruning).
- In this method we basically exten cheapest partial path.

CI **CCI** **HoD**