| Experiment **#1** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title: Analysis of Algorithm based on Arrays**

**Aim/Objective:** To understand the concept and implementation of Basic programs on arrays.

**Description:** The students will understand and able to implement programs on Arrays.

**Pre-Requisites:**

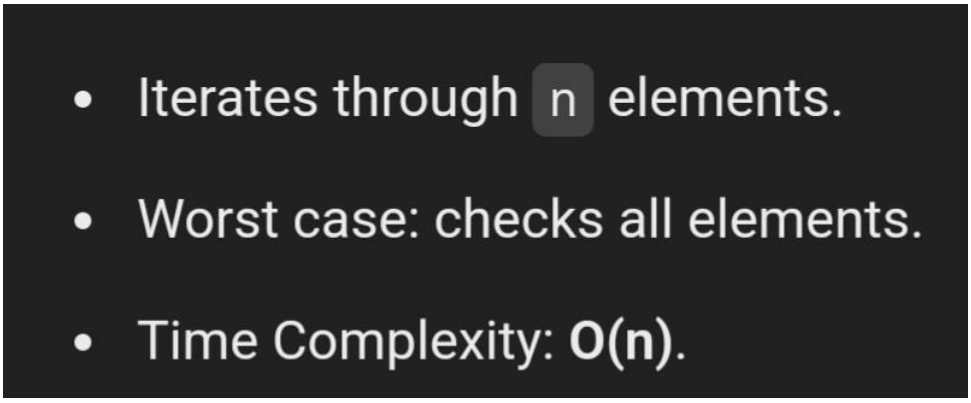
**Knowledge**: Arrays **Tools**: Code Blocks/Eclipse IDE

**Pre-Lab:**

1. Calculate the Time Complexity of the following code snippet:

```
Algorithm linear_search(arr, target):
for i in range(len(arr)):
    if arr[i] == target:
        return i;
return -1;
```

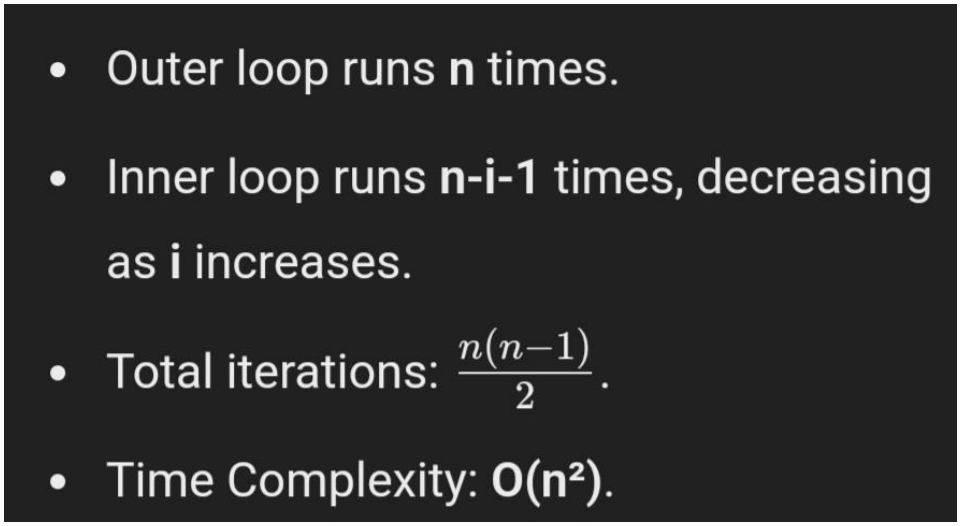
- **Procedure**:



2. Algorithm function(arr):

```
n = len(arr)
for i in range(n):
    for j in range(0, n-i-1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
return arr;
```

- **Procedure**:



3. The median of a list of numbers is essentially its middle element after sorting. The same number ofelements occur after it as before. Given a list of numbers with an odd number of elements, find the median? Also find its time complexity.

Example arr = [5, 3, 1, 2, 4]

The sorted array a' = [1, 2, 3, 4, 5]. The middle element and the median is 3.

**Description: Write an algorithm** *findMedian* **with the following parameter(s):**

*arr[n]:* an unsorted array of integers

**Returns: median of the array**

**Sample Input**

7

0 1 2 4 6 5 3

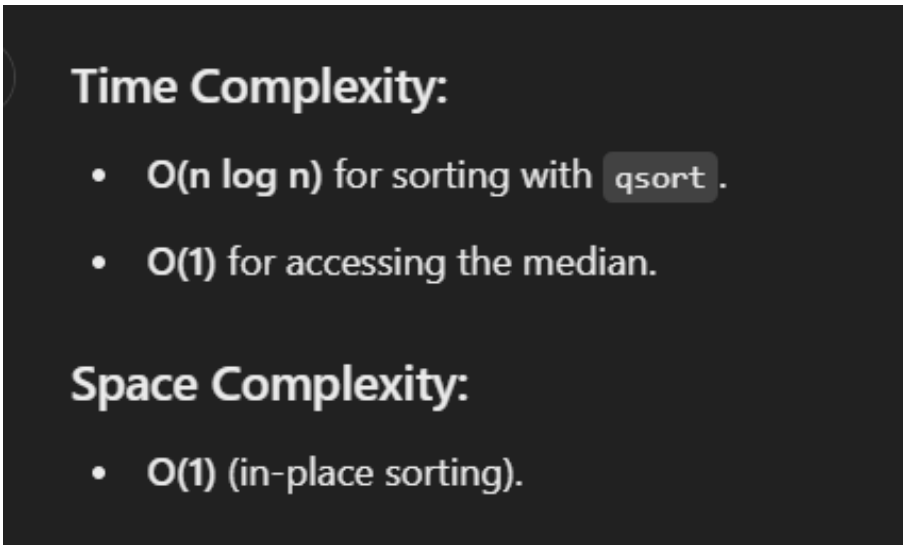
**Sample Output**

3

- **Procedure/Program**:

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
   return (*(int *)a - *(int *)b);
}



int findMedian(int arr[], int n) {
   qsort(arr, n, sizeof(int), compare);
   return arr[n / 2];
}
```

```
int main() {
    int arr[] = {0, 1, 2, 4, 6, 5, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Median: %d\n", findMedian(arr, n));
    return 0;
}
```



- **Data and Results**:

**Data:** Input array: {0, 1, 2, 4, 6, 5, 3}

**Result:** Median of the array is 3 after sorting.

- **Analysis and Inferences**:

**Analysis:** Sorting time complexity is O(n log n), retrieving median O(1).

**Inferences:** Median is the middle element after sorting the array.

**In-Lab:**

Given an array of strings **arr[]**, the task is to sort the array of strings according to frequency of each string, in ascending order. If two elements have same frequency, then they are to be sorted in alphabetical order.

Input: arr[] = {“Ramesh”, “Mahesh”, “Mahesh”, “Ramesh”}

Output: {“Mahesh”, ”Ramesh”}

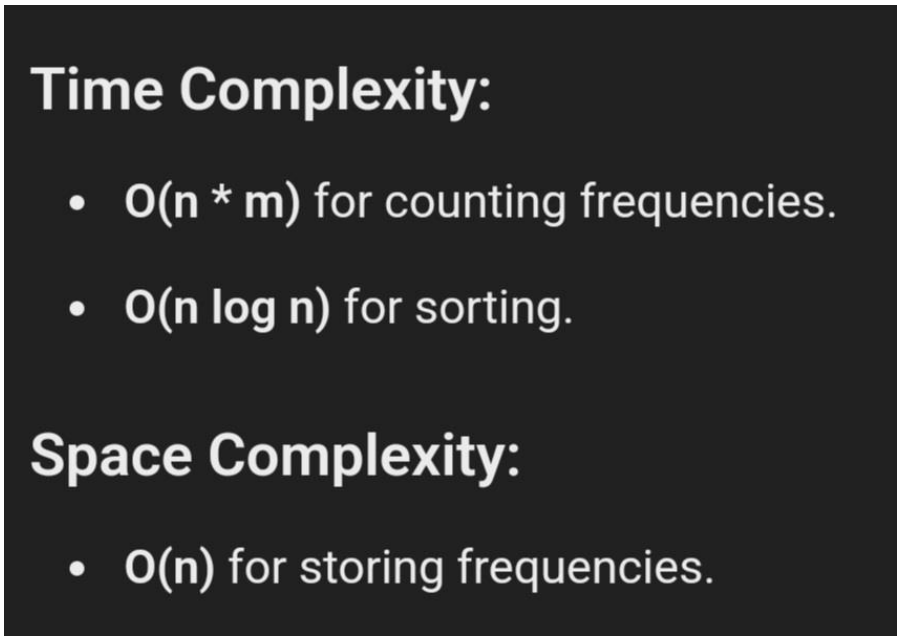
Explanation: As both the strings have the same frequency, the array is sorted in alphabetical order.

**Find the time and space complexity for the procedure/Program.**

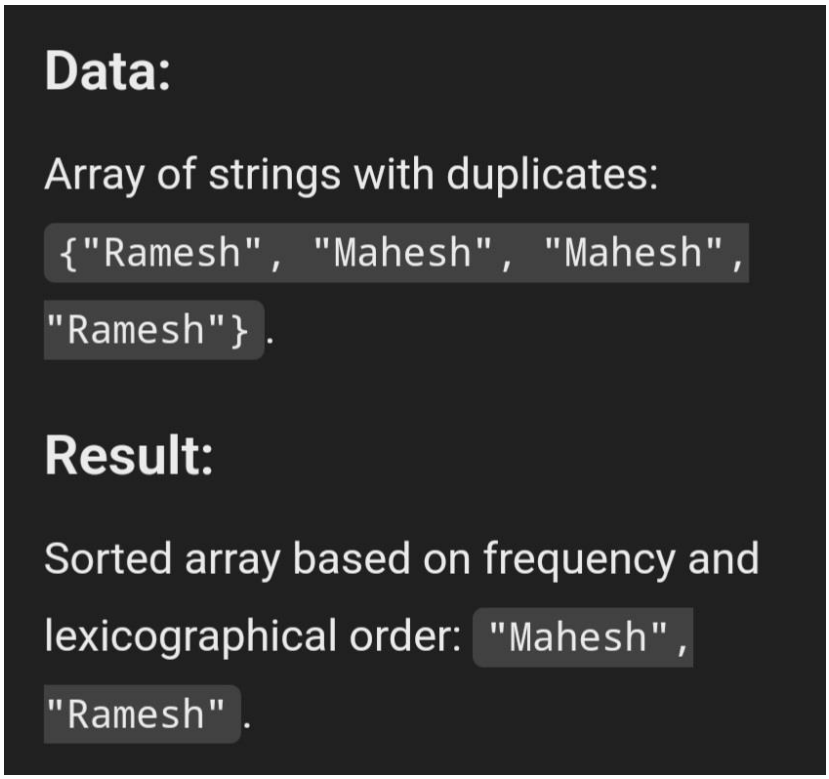
- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
   char str[100];
   int freq;
} StringFreq;

int compare(const void *a, const void *b) {
   StringFreq *x = (StringFreq *)a, *y = (StringFreq *)b;
   return x->freq != y->freq ? x->freq - y->freq : strcmp(x->str, y->str);
}

int main() {
   char arr[][100] = {"Ramesh", "Mahesh", "Mahesh", "Ramesh"};
   int n = 4, count = 0;
   StringFreq freqArr[n];

   for (int i = 0; i < n; i++) {
      int found = 0;
      for (int j = 0; j < count; j++) {
         if (strcmp(freqArr[j].str, arr[i]) == 0) {
```

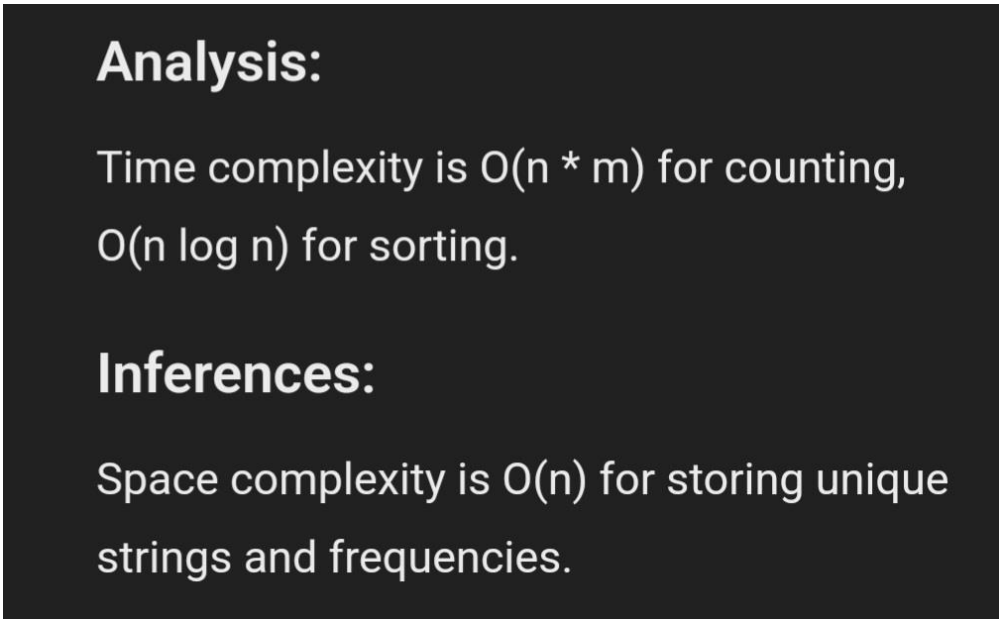
```
    freqArr[j].freq++;
            found = 1;

     break;
         }
       }
       if (!found) {
          strcpy(freqArr[count].str, arr[i]);
          freqArr[count++].freq = 1;
       }
    }

    qsort(freqArr, count, sizeof(StringFreq), compare);

    for (int i = 0; i < count; i++)
       printf("%s ", freqArr[i].str);
    printf("\n");
    return 0;
}
```



- **Data and Results:**




- **Analysis and Inferences:**



**Post-Lab:**

Given an array of strings **words []** and the **sequential order** of alphabets, our task is to sort the array according to the order given. Assume that the dictionary and the words only contain lowercase alphabets.

**Input: words = {"word", "world", "row"},**

**Order= "worldabcefghijkmnpqstuvxyz" Output: "world", "word", "row"**

**Explanation**: According to the given order 'l' occurs before 'd' hence the words "world" will be kept first.

**Find the time and space complexity for the procedure/Program.**

- **Procedure/Program**:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int charToIndex(char c, char *order) {
   for (int i = 0; i < strlen(order); i++) {
      if (order[i] == c) {
         return i;
      }
   }
   return -1;
}

int compare(const void *a, const void *b) {
   char *word1 = *(char **)a;
   char *word2 = *(char **)b;

   int len1 = strlen(word1);
   int len2 = strlen(word2);
   int minLen = len1 < len2 ? len1 : len2;

   for (int i = 0; i < minLen; i++) {
      int index1 = charToIndex(word1[i], "worldabcefghijkmnpqstuvxyz");
```

```
        int index2 = charToIndex(word2[i], "worldabcefghijkmnpqstuvxyz");

        if (index1 < index2) {
            return -1;
        } else if (index1 > index2) {
            return 1;
        }
    }

    if (len1 < len2) {
        return -1;
    } else if (len1 > len2) {
        return 1;
    }
    return 0;
}

int main() {
    char *words[] = {"word", "world", "row"};
    int n = sizeof(words) / sizeof(words[0]);

    qsort(words, n, sizeof(char *), compare);

    for (int i = 0; i < n; i++) {
        printf("%s ", words[i]);
    }
    return 0;
}
```

**Time Complexity**: O(n log n * m)

**Space Complexity**: O(1)

- **Data and Result:**

**Data**:

Words: ["word", "world", "row"], Order: "worldabcefghijkmnpqstuvxyz"

**Result**:

Sorted words: ["world", "word", "row"] based on custom order.

- **Analysis and Inferences:**

**Analysis**:

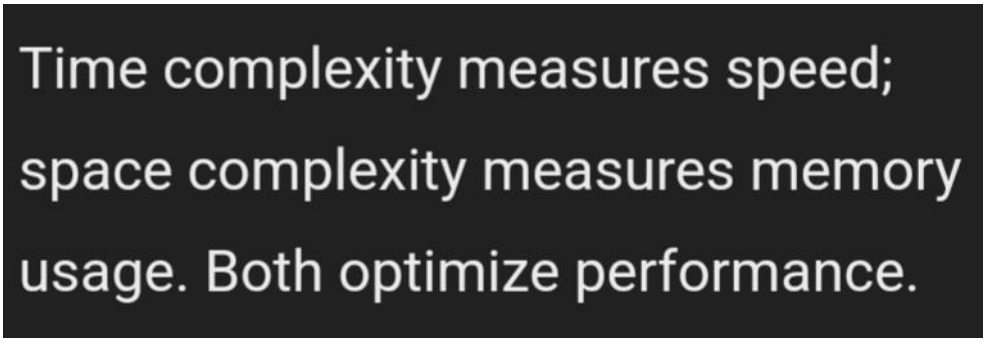
Time complexity is O(n log n * m), space is O(1).

**Inferences**:

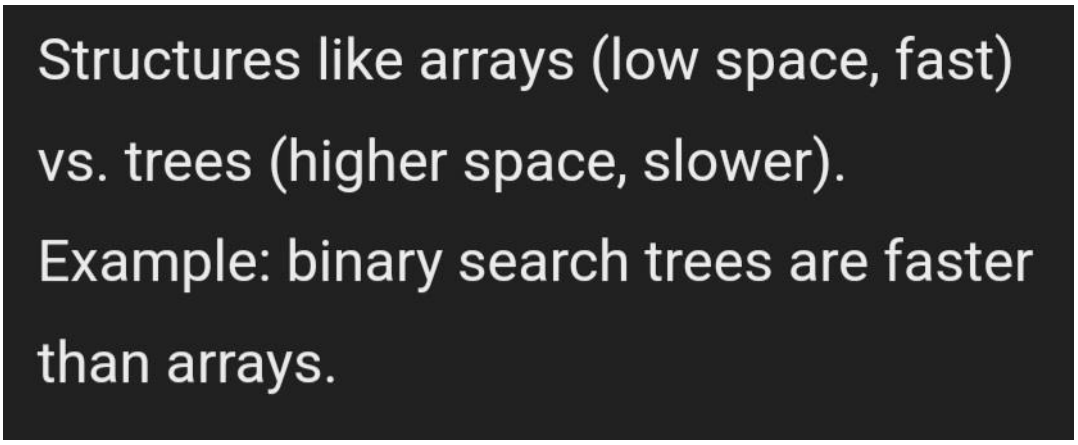
Custom order affects word sorting, efficient with quicksort.

- **Sample VIVA-VOCE Questions (In-Lab):**

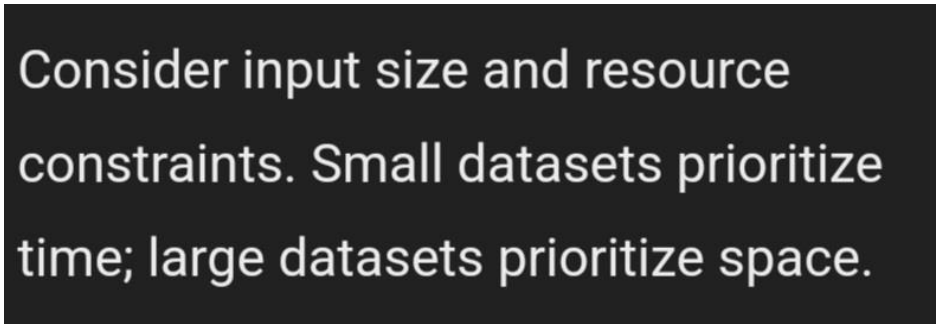
1. What is the significance of analyzing both time complexity and space complexity when evaluating algorithms?




2. How does the choice of data structures impact both time and space complexity in algorithm implementations? Give examples.




3. When comparing two algorithms with different time and space complexities, how do you decide which one is more suitable for a particular problem or application?



4. What is the difference between best-case, average-case, and worst-case time complexity? Provide examples.

Best-case is optimal; average is typical; worst-case is maximum time (e.g., Merge Sort).

5. In the context of sorting algorithms, compare the time and space complexities of algorithms like Quick Sort and Merge Sort. Which one is more time-efficient, and which one is more space-efficient?

Quick Sort is faster ($O(n \log n)$, average) but uses less space. Merge Sort is stable but space-heavy ($O(n)$).

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#2** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Analyzing Algorithm Performance: Time and Space Complexity

**Aim/Objective:** To understand the concept of time and space complexity.

**Description:** The students can understand and analyzing the performance of algorithms in terms of time and space complexity.

**Pre-Requisites:**

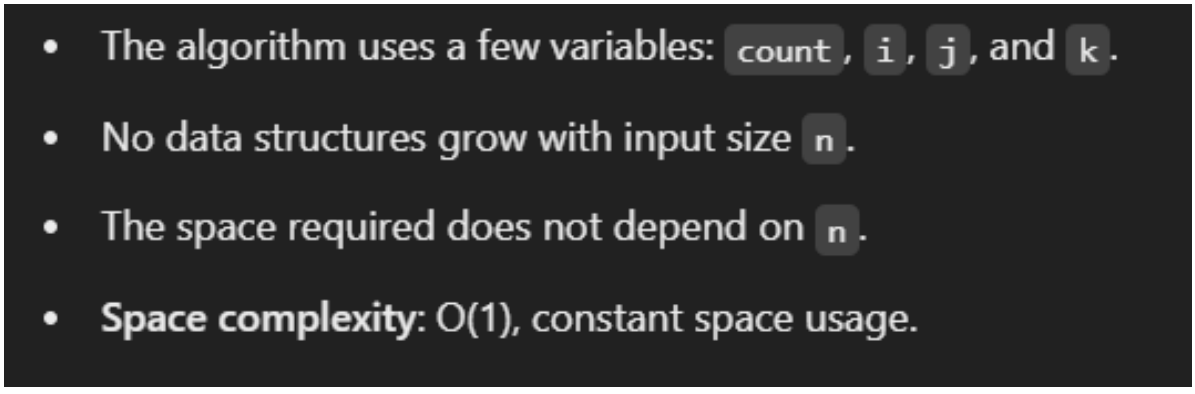
Knowledge: Strings in C/C++/Python Tools: Code Blocks/ IDE

**Pre-Lab:**

1. During the lockdown, Mothi stumbled upon a unique algorithm while browsing YouTube. Although he's enthusiastic about algorithms, he struggles with those involving multiple loops. Now, he's seeking your assistance to determine the space complexity of this algorithm. Can you help him?

```
Algorithm KLU(int n) {
count ← 0
for i ← 0 to n - 1 step i = i * 2 do
   for j ← n down to 1 step j = j / 3 do
         for k ← 0 to n - 1 do    count ← count + 1
         end for
    end for
 end for
return count   }
```

- **Procedure:**



2. Klaus Michaelson, an interviewer, has compiled a list of algorithm-related questions for students. Among them, one of the most basic questions involves determining the time complexity of a given algorithm. Can you determine the time complexity for this algorithm?

```
Algorithm ANALYZE( n)  {
   if n == 1 then    return 1
   else  eturn ANALYZE(n - 1) + ANALYZE(n - 1)
   end if
}
```

- **Procedure:**

- The algorithm makes two recursive calls for each `n`.
- Each call reduces `n` by 1, leading to binary recursion.
- The recurrence relation is T(n) = 2T(n - 1) + O(1).
- The number of calls grows exponentially, proportional to $2^n$.
- **Time complexity**: O(2^n).

**In-Lab:**

Caroline Forbes is known for her intelligence and problem-solving skills. To test her abilities, you've decided to present her with a challenge: sorting an array of strings based on their string lengths. If you're up for the challenge, see if you can solve this problem faster than her!

**Find the time and space complexity for the procedure/Program.**

**Input**: You are extraordinarily talented, displaying a wide range of skills and abilities

**Output**: a of You are and wide range skills talented abilities displaying extraordinarily

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
   return strlen((char *)a) - strlen((char *)b);
}

int main() {
   char *arr[] = {
      "You", "are", "extraordinarily", "talented", "displaying",
      "a", "wide", "range", "of", "skills", "and", "abilities"
   };


   int n = sizeof(arr) / sizeof(arr[0]);

   qsort(arr, n, sizeof(char *), compare);
```

```
    for (int i = 0; i < n; i++) {
        printf("%s ", arr[i]);
    }

    return 0;
}
```

**Time Complexity:**

- **O(n log n * k)**, where `n` is the number of strings and `k` is the average length.

**Space Complexity:**

- **O(n)**, for storing the array of strings.

- **Data and Results:**

**Data:**

Array of strings with varying lengths to be sorted.

**Result:**

Sorted array of strings based on their lengths printed.

- **Analysis and Inferences:**

**Analysis:**

Time complexity is O(n log n * k), space is O(n).

**Inferences:**

Sorting by length improves arrangement; space and time grow linearly.

**Post-Lab:**

Given an array **arr[]** containing N strings, the objective is to arrange these strings in ascending order based on the count of uppercase letters they contain. Find the time complexity for the algorithm/program.

Input arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"}

Output: {"abc", "Hello", "Testing", "WORLD", "DEFGH"}

**Procedure/Program:**

## Algorithm:

1. Input: An array `arr[]` of `N` strings.
2. Count uppercase letters: For each string, count uppercase letters using `countUppercase(str)`.
3. Sort strings: Sort strings based on the uppercase letter count using a comparison function.
4. Output: Print the sorted array.

Input:

arduino Copy code

```
arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"}
```

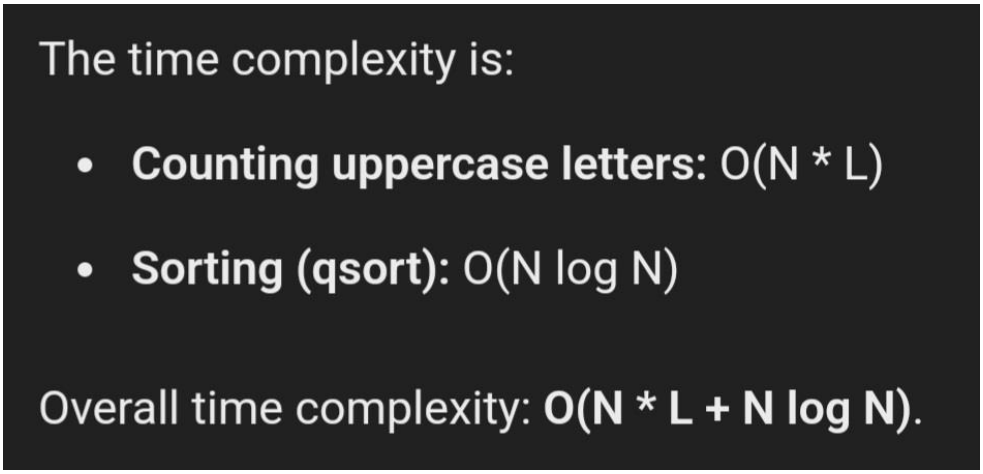
Output:

Copy code

```
abc Hello Testing WORLD DEFGH
```

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

int countUppercase(const char str[]) {
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}

int compare(const void *a, const void *b) {
    int countA = countUppercase(*(const char **)a);
    int countB = countUppercase(*(const char **)b);
    return countA - countB;
}

int main() {
    char *arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"};
    int n = sizeof(arr) / sizeof(arr[0]);

    qsort(arr, n, sizeof(char *), compare);
```

```
 for (int i = 0; i < n; i++) {
        printf("%s ", arr[i]);
    }
    return 0;
}
```



- **Data and Results:**

**Data:**

Input: `arr[] = {"Hello", "WORLD", "abc", "DEFGH", "Testing"}`.

**Result:**

Output: `abc Hello Testing WORLD DEFGH`.

- **Analysis and Inferences:**

**Analysis:**

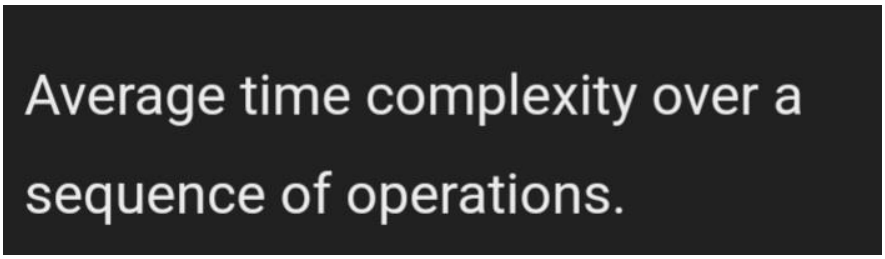
Time complexity: O(N * L + N log N).

**Inferences:**

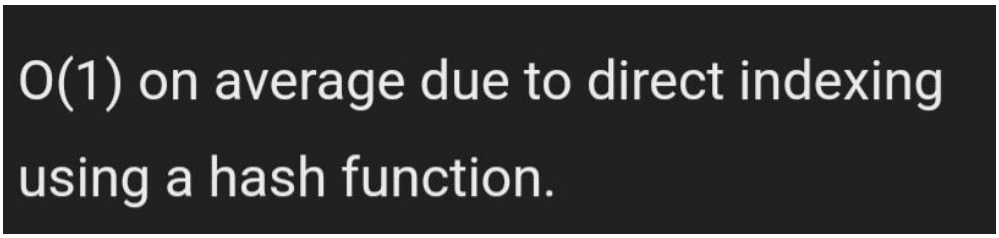
Uppercase count and sorting determine final string order.

- **Sample VIVA-VOCE Questions :**

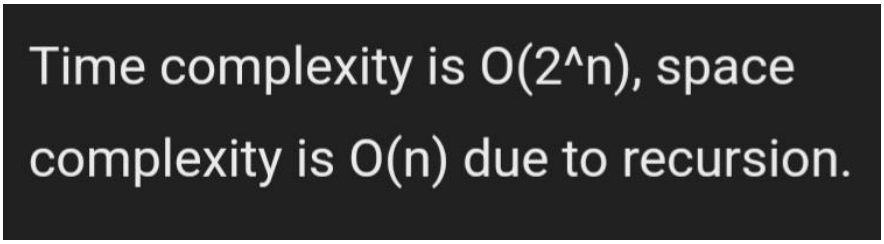
1. What do you understand by amortized analysis in the context of algorithm complexity?




2. Why is the complexity of searching for an element in a hash table O(1) on average?




3. What is the time complexity of the Tower of Hanoi problem, and how does its recursive nature affect its space complexity?



4. How can hashing improve the time complexity of search operations?

Improves search time by directly accessing data with a hash function.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#3** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of programs on Sorting and Searching problems.

**Aim/Objective:** To understand the concept and implementation of programs on Sorting and Searching problems.

**Description:** The students will understand and able to implement programs on Sorting and Searching problems.

**Pre-Requisites:**

Knowledge: Sorting, Searching, Arrays in C/C++/PythonTools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Implementing and Analyzing the Rabin-Karp String Matching Algorithm.

- **Procedure/Program**:

### Rabin-Karp Algorithm

**Input:** Text `T` of length `n`, Pattern `P` of length `m`
**Output:** Indices where `P` is found in `T`.

### Steps:

1. Calculate hash of `P` and first window of `T`.
2. Slide the window over `T`:
   - If hashes match, compare characters.
   - If match, print index.
3. Update hash for next window:
   `hash_window = (d * (hash_window - text[i] * h) + text[i + m]) % q`.

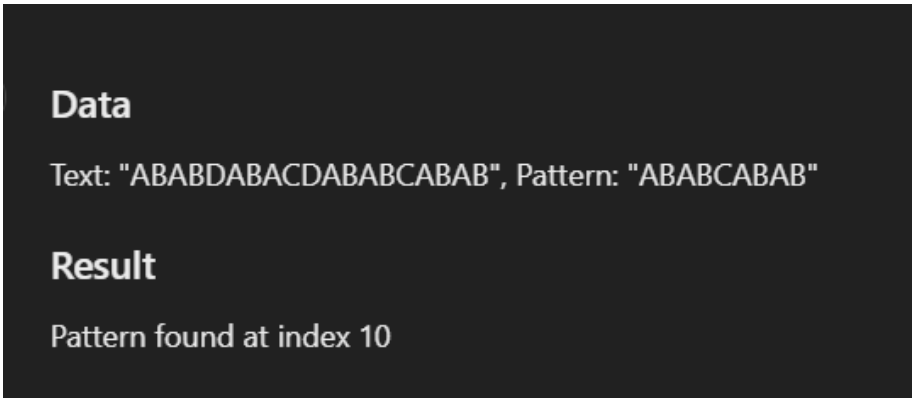
### Time Complexity:

- **Best Case**: O(n + m)
- **Worst Case**: O(n * m)
- **Average Case**: O(n + m)

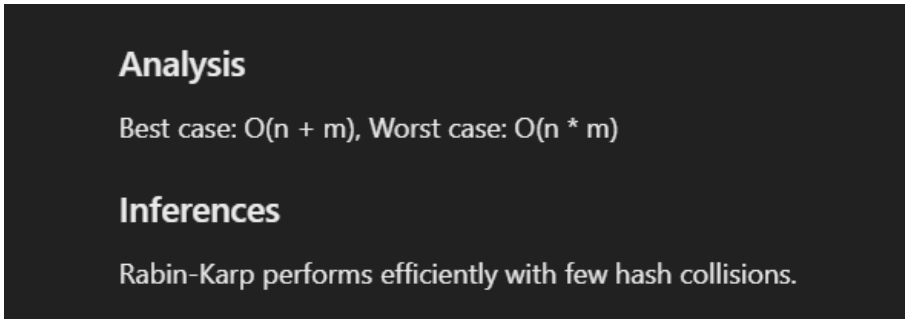
```
#include <stdio.h>
#include <string.h>

#define d 256
#define q 101

void rabinKarp(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   int pHash = 0;
   int tHash = 0;
   int h = 1;

   for (int i = 0; i < m - 1; i++) {
      h = (h * d) % q;
   }

   for (int i = 0; i < m; i++) {
      pHash = (d * pHash + pattern[i]) % q;
      tHash = (d * tHash + text[i]) % q;
   }

   for (int i = 0; i <= n - m; i++) {
      if (pHash == tHash) {
         int match = 1;
         for (int j = 0; j < m; j++) {
            if (text[i + j] != pattern[j]) {
               match = 0;
               break;
            }
         }
         if (match) {
            printf("Pattern found at index %d\n", i);
         }
      }

      if (i < n - m) {
```

```
            tHash = (d * (tHash - text[i] * h) + text[i + m]) % q;
            if (tHash < 0) {
               tHash = tHash + q;
            }
         }
      }
   }

int main() {
   char text[] = "ABABDABACDABABCABAB";
   char pattern[] = "ABABCABAB";

   rabinKarp(text, pattern);

   return 0;
}
```

- **Data and Results:**



- **Analysis and Inferences**




**In-Lab:**

Alice uses a particular algorithm to systematically sort a sequence of N unique positive integers. Determine the final value of the variable comparison count, which tracks the total number of comparisons made, after using the provided Quicksort algorithm to sort a given permutation of integers in the list A.

**Input**

The first line of input consists of a single integer N. The second line of input contains a permutation of N numbers.

**Output**

Output the number of comparisons on the first and only line of the output.

**Example**

**Input**:

5

4 3 5 1 2

**Output**:

11

- **Procedure/Program:**

```
#include <stdio.h>

int comparisons = 0;

int partition(int A[], int low, int high) {
   int pivot = A[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      comparisons++; // Count comparison
      if (A[j] < pivot) {
         i++;
         int temp = A[i];
         A[i] = A[j];
         A[j] = temp;
      }
   }
   int temp = A[i + 1];
   A[i + 1] = A[high];
   A[high] = temp;
   return i + 1;
}

void quicksort(int A[], int low, int high) {
   if (low < high) {
      int pi = partition(A, low, high);
      quicksort(A, low, pi - 1);
      quicksort(A, pi + 1, high);
   }
}

int main() {
   int N;
   scanf("%d", &N);
   int A[N];
```

```
    for (int i = 0; i < N; i++) {
        scanf("%d", &A[i]);
    }

    quicksort(A, 0, N - 1);

    printf("%d\n", comparisons);
    return 0;
}
```

- **Data and Results:**

**Data:**

Input consists of N and a permutation of N integers.

**Result:**

Total comparisons made during Quicksort on given input is 11.

- **Analysis and Inferences**

**Analysis:**

Quicksort's partitioning and recursion determine the total comparison count.

**Inferences:**

Comparison count depends on array order and Quicksort's partitioning strategy.

**Post-Lab:**

Stefan is a guy who is suffering with OCD. He always like to align things in an order. He got a lotof strings for his birthday party as gifts. He like to sort the strings in a unique way. He wants his strings to be sorted based on the count of characters that are present in the string.

**Input**

aaabbc

aabbcc

**Output**

cbbaaa

aabbcc

If in case when there are two characters is same, then the lexicographically smaller one will be printed first

**Input:**

aabbccdd

aabcc

**Output:**

aabbccdd

baacc

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
   const char *str1 = *(const char **)a;
   const char *str2 = *(const char **)b;

   int len1 = strlen(str1);
   int len2 = strlen(str2);

   if (len1 != len2) {
      return len1 - len2;
```

```
    }

    return strcmp(str1, str2);
}

void sort_strings_by_char_count(char *input) {
    char *strings[100];
    int count = 0;

    char *token = strtok(input, " ");
    while (token != NULL) {
        strings[count++] = token;
        token = strtok(NULL, " ");
    }

    qsort(strings, count, sizeof(char *), compare);

    for (int i = 0; i < count; i++) {
        printf("%s", strings[i]);
        if (i < count - 1) {
            printf(" ");
        }
    }
    printf("\n");
}

int main() {
    char input1[] = "aaabbc aabbcc";
    printf("Input: %s\n", input1);
    printf("Output: ");
    sort_strings_by_char_count(input1);

    char input2[] = "aabbccdd aabcc";
    printf("Input: %s\n", input2);
    printf("Output: ");
    sort_strings_by_char_count(input2);
```

```
    return 0;
}
```

- **Data and Results:**

**Data**

Input strings are sorted based on length and lexicographical order.

**Result**

Strings sorted by length and lexicographical order successfully printed.

- **Analysis and Inferences:**

**Analysis**

Strings are divided into tokens and sorted using `qsort` function.

**Inferences**

Sorting is effective for sorting strings by length and alphabetically.

- **Sample VIVA-VOCE Questions:**

1. How does the time complexity of the heap sort algorithm compare to other sorting algorithms?

O(n log n), better than bubble/insertion sort ($O(n^2)$), but slower than quicksort.

2. What is the role of the partition function in quicksort? How does it affect the overall sorting process?

Divides array by pivot, ensuring left < pivot, right > pivot.

3. Explain the difference between internal and external sorting.

Internal: Data fits in memory.
External: Data is too large for memory.

4. What is the time complexity of the bubble sort algorithm? Can it be improved?

$O(n^2)$, can be improved with a flag to check sorted state.

5. Compare the time complexity of insertion sort and selection sort. Which one is more efficient?

Both $O(n^2)$; insertion is better for nearly sorted data.

| | |
|---|---|
| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#4** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Program on KMP Algorithm.

**Aim/Objective:** To understand the concept and implementation of Basic programs on KMP Algorithm.

**Description:** The students will understand and able to implement programs on KMP Algorithm.

**Pre-Requisites:**

Knowledge: Strings in C/C++/Python Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search (char pat [], char txt[]) that prints all occurrences of pat[] in txt[] using naïve string algorithm?(assume that n > m)

**Input**

txt[] = "THIS IS A TEST TEXT" pat[] = "TEST"

**Output**

Pattern found at index 10Input

txt[] = "AABAACAADAABAABA" pat[] = "AABA"

**Output**

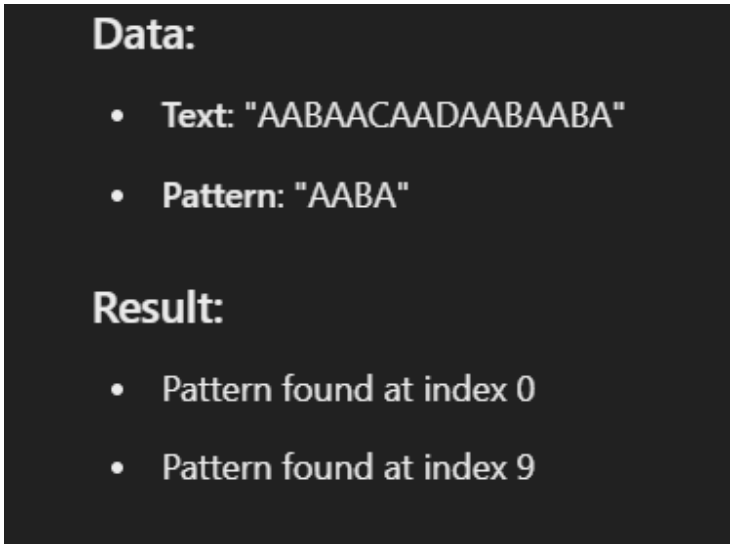
Pattern found at index 0 Pattern found at index 9

- **Procedure/Program:**

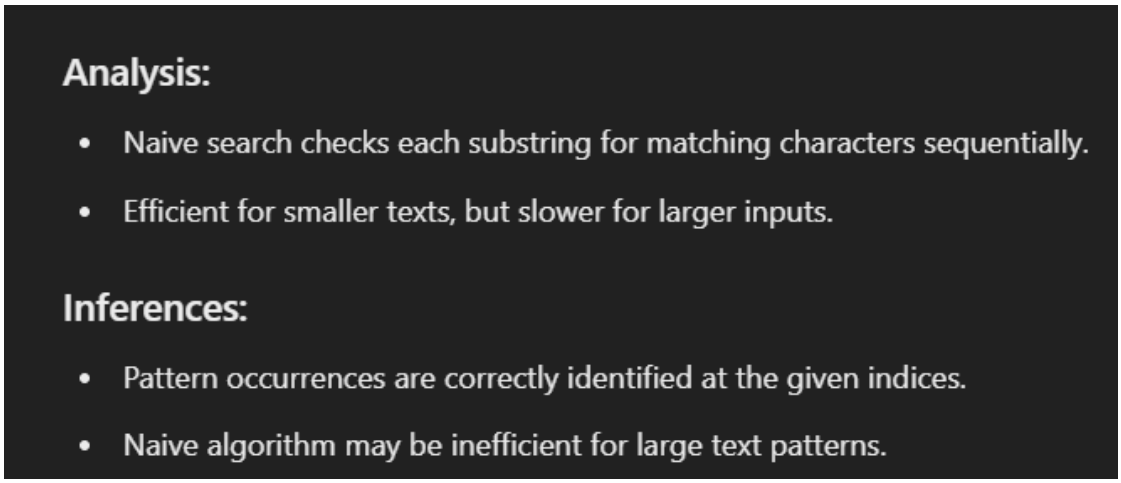
```
#include <stdio.h>
#include <string.h>

void search(char pat[], char txt[]) {
   int n = strlen(txt);
   int m = strlen(pat);

   for (int i = 0; i <= n - m; i++) {
      int j = 0;

      for (j = 0; j < m; j++) {
         if (txt[i + j] != pat[j]) {
            break;
```

```
        }
      }

      if (j == m) {
        printf("Pattern found at index %d\n", i);
      }
    }
}

int main() {
    char txt[] = "AABAACAADAABAABA";
    char pat[] = "AABA";

    search(pat, txt);

    return 0;
}
```

- **Data and Result:**



- **Inference Analysis**




**In-Lab:**

Naive method and KMP are two string comparison methods. Write a program for Naïve method and KMP to check whether a pattern is present in a string or not. Using clock function find execution time for both and compare the time complexities of both the programs (for larger inputs) and discuss which one is more efficient and why?

Sample program with function which calculate execution time:

```
#include<stdio.h>
#include<time.h>
void fun()  {

//some statements here    }

int main()  {

//calculate time taken by fun() clock_t t;t=clock();
fun();

 t=clock()-t;

 double time_taken=((double)t)/CLOCK_PER_SEC; //in secondsprintf("fun() took %f seconds to
 execute \n",time_taken); return 0;    }
```

- **Procedure/Program:**

```
#include <stdio.h>
#include <time.h>

void naiveSearch(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   for (int i = 0; i <= n - m; i++) {
      int j = 0;
      while (j < m && text[i + j] == pattern[j]) {
         j++;
      }
      if (j == m) {
         printf("Pattern found at index %d\n", i);
      }
   }
}

void computeLPSArray(char *pattern, int *lps) {
   int m = strlen(pattern);
   int len = 0;
   lps[0] = 0;
   int i = 1;
   while (i < m) {
      if (pattern[i] == pattern[len]) {
         len++;
         lps[i] = len;
         i++;
      } else {
         if (len != 0) {
            len = lps[len - 1];
         } else {
            lps[i] = 0;
            i++;
         }
      }
   }
```

```
}

void KMPSearch(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   int lps[m];
   computeLPSArray(pattern, lps);

   int i = 0, j = 0;
   while (i < n) {
      if (pattern[j] == text[i]) {
         i++;
         j++;
      }
      if (j == m) {
         printf("Pattern found at index %d\n", i - j);
         j = lps[j - 1];
      } else if (i < n && pattern[j] != text[i]) {
         if (j != 0) {
            j = lps[j - 1];
         } else {
            i++;
         }
      }
   }
}

int main() {
   char text[] = "ABABDABACDABABCABAB";
   char pattern[] = "ABABCABAB";

   clock_t t;

   t = clock();
   naiveSearch(text, pattern);
   t = clock() - t;
```

```
  double time_taken_naive = ((double)t) / CLOCKS_PER_SEC;
   printf("Naive Search took %f seconds to execute\n", time_taken_naive);

   t = clock();
   KMPSearch(text, pattern);
   t = clock() - t;
   double time_taken_kmp = ((double)t) / CLOCKS_PER_SEC;
   printf("KMP Search took %f seconds to execute\n", time_taken_kmp);

   return 0;
}
```

- **Data and Results:**

**Data Heading:**

String matching using Naive and KMP algorithms with execution time.

**Result Heading:**

Naive Search took more time compared to efficient KMP Search.

- **Analysis and Inferences**

**Analysis Heading:**

KMP improves efficiency by avoiding redundant character comparisons in matching.

**Inferences Heading:**

KMP is significantly faster for larger inputs compared to Naive method.

**Post-Lab:**

Given a pattern of length- 5 window, find the valid match in the given text by step-by-step process using Robin-Karp algorithm

**Pattern**: 2 1 9 3 6

**Modulus**: 21

**Index**: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

**Text**: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7

- **Procedure/Program:**

1. **Pattern**: 2 1 9 3 6
   - **Text**: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7
   - **Modulus**: 21
   - **Pattern Length**: 5
   - **Text Length**: 22
2. **Compute the Pattern Hash:**
   - Hash formula:
     $$H_{\text{pattern}} = (2 \cdot 10^4 + 1 \cdot 10^3 + 9 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0) \mod 21$$
   - **Pattern Hash** = 12
3. **Compute the Hash of the First Window in Text:**
   - First window (9 2 7 2 1):
     $$H_{\text{window}} = (9 \cdot 10^4 + 2 \cdot 10^3 + 7 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0) \mod 21$$
   - **Window Hash** = 21
4. **Compare the Window Hash with the Pattern Hash:**
   - First window (9 2 7 2 1) hash (21) does not match pattern hash (12).

5. **Slide the Window and Update Hash**:
   - For each subsequent window, update the hash by removing the old leftmost digit and adding the new rightmost digit.
6. **Continue Sliding**:
   - Continue this process for each window in the text until a match is found.
7. **Match Found**:
   - A match is found at **window [15, 19]** (9 3 6 2 3).

So, the pattern **2 1 9 3 6** matches the text at **index 15**.

- **Data and Results:**

**Data**

Pattern: 2 1 9 3 6, Text: 9 2 7 2 1 8...

**Result**

Pattern matches text at window [15, 19] (index 15).

- **Analysis and Inference:**

**Analysis**

Hash comparison showed no match initially, then found at index 15.

**Inferences**

Rabin-Karp efficiently detects pattern match by hashing and sliding window.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the main purpose of the Knuth-Morris-Pratt (KMP) algorithm?

Efficient string matching by skipping unnecessary comparisons.

2. Explain the basic idea behind the KMP algorithm.

Uses previous matches to skip sections of the text.

3. What is the significance of the Longest Prefix Suffix (LPS) array in the KMP algorithm?

Helps avoid redundant comparisons by storing prefix-suffix lengths.

4. How is the LPS array computed in the KMP algorithm?

Iterates over the pattern, using prior values to skip checks.

5. Can you describe a scenario where the KMP algorithm would be more advantageous than other string matching algorithms?

More efficient than brute force for long texts with repetitions.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#5** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Divide and Conquer Problems.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Divide and Conquer Problems.

**Description:** The students will understand and able to implement programs on Divide and Conquer Problems.

**Pre-Requisites:**

Knowledge: Arrays, Sorting, Divide and Conquer in C/C++/Python Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Write a program to sort an array using the Merge Sort algorithm.

**Input**: An unsorted array of integers.

**Output**: The sorted array.

**Example**: Input: [5, 2, 4, 6, 1, 3]

Output: [1, 2, 3, 4, 5, 6]

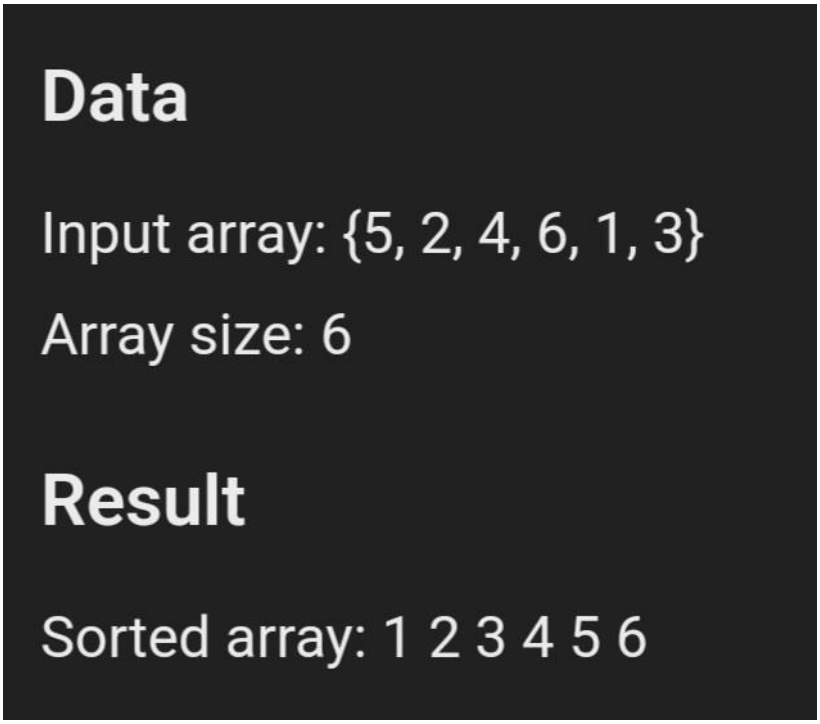
- **Procedure/Program:**

```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1;
   int n2 = right - mid;

   int leftArr[n1], rightArr[n2];

   for (int i = 0; i < n1; i++)
      leftArr[i] = arr[left + i];
   for (int j = 0; j < n2; j++)
      rightArr[j] = arr[mid + 1 + j];
```

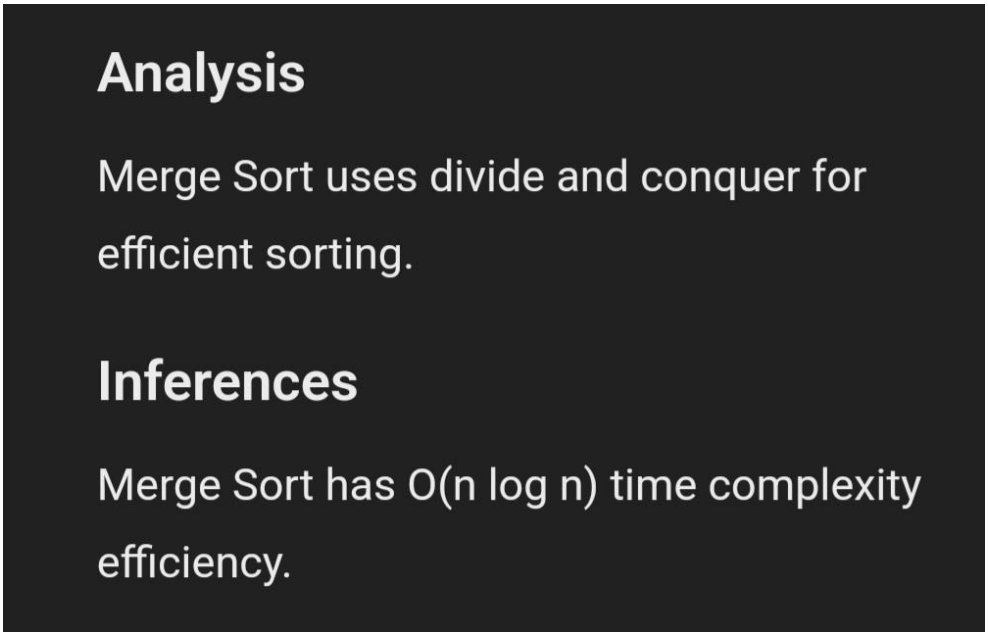
```
  int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
       if (leftArr[i] <= rightArr[j]) {
          arr[k] = leftArr[i];
          i++;
       } else {
          arr[k] = rightArr[j];
          j++;
       }
       k++;
    }

    while (i < n1) {
       arr[k] = leftArr[i];
       i++;
       k++;
    }

    while (j < n2) {
       arr[k] = rightArr[j];
       j++;
       k++;
    }
  }
```

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] = {5, 2, 4, 6, 1, 3};
    int size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, size - 1);

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

    return 0;
}
```

- **Data and Results**:




- **Analysis and Inferences**:



| Experiment **#5** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**In-Lab:**

Find the majority element in an array (an element that appears more than n/2 times) using Divide and Conquer.

**Input**: An array of integers.

**Output**: The majority element, or -1 if none exists.

**Example**:

**Input**: [3, 3, 4, 2, 4, 4, 2, 4, 4]

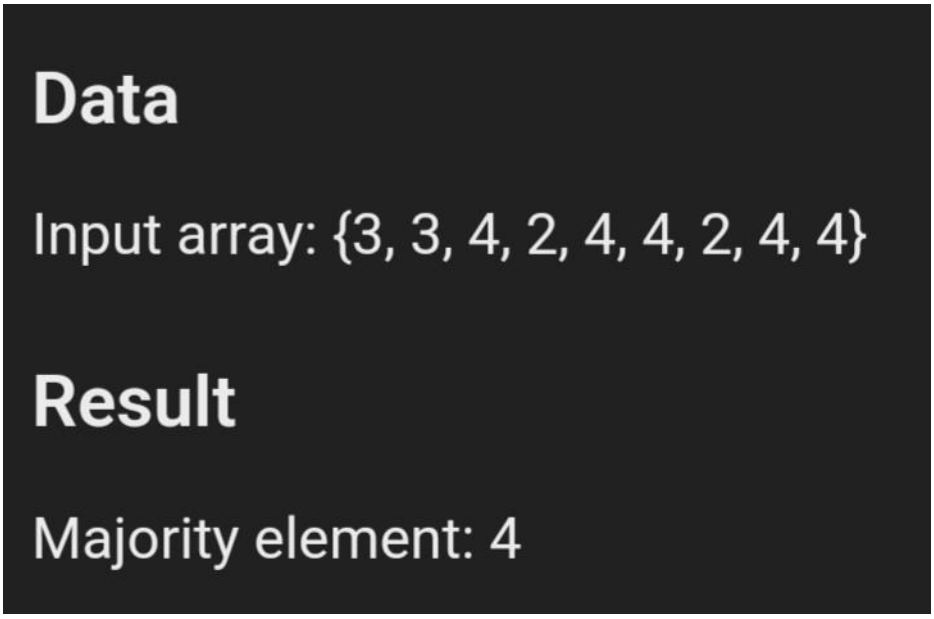
**Output**: 4

- **Procedure/Program:**

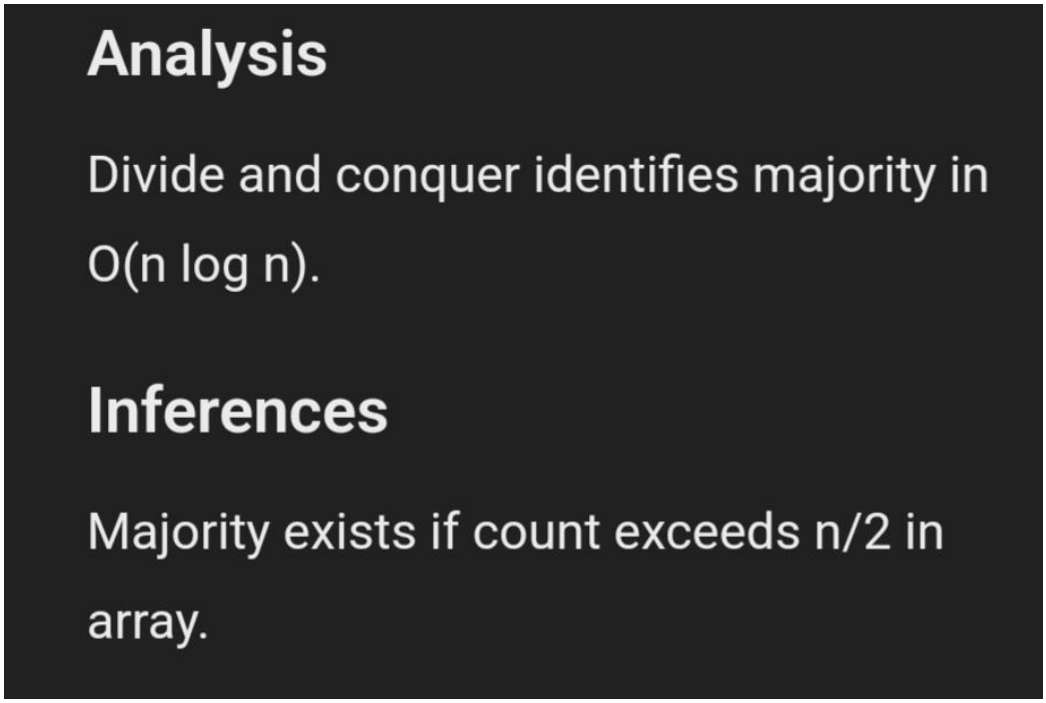
```
#include <stdio.h>

int findMajorityElement(int arr[], int left, int right) {
   if (left == right)
      return arr[left];

   int mid = left + (right - left) / 2;

   int leftMajority = findMajorityElement(arr, left, mid);
   int rightMajority = findMajorityElement(arr, mid + 1, right);

   if (leftMajority == rightMajority)
      return leftMajority;

   int leftCount = 0, rightCount = 0;

   for (int i = left; i <= right; i++) {
      if (arr[i] == leftMajority)
         leftCount++;

      if (arr[i] == rightMajority)
         rightCount++;
   }
```

```
  if (leftCount > (right - left + 1) / 2)
      return leftMajority;

   if (rightCount > (right - left + 1) / 2)
      return rightMajority;

   return -1;
}

int main() {
   int arr[] = {3, 3, 4, 2, 4, 4, 2, 4, 4};
   int n = sizeof(arr) / sizeof(arr[0]);

   int result = findMajorityElement(arr, 0, n - 1);

   printf("Output: %d\n", result);

   return 0;
}
```

- **Data and Results:**




- **Analysis and Inferences:**



**Post-Lab:**

Find a peak element in an array using a divide-and-conquer approach. A peak element is one that is greater than or equal to its neighbors.

**Input**: An array of integers.

**Output**: A peak element.

**Example**:

Input: [1, 3, 20, 4, 1, 0]

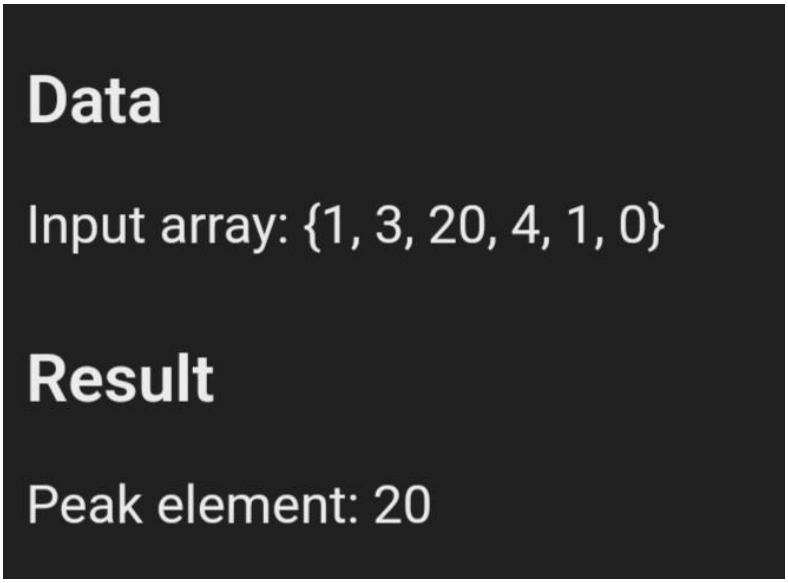
Output: 20

- **Procedure/Program:**

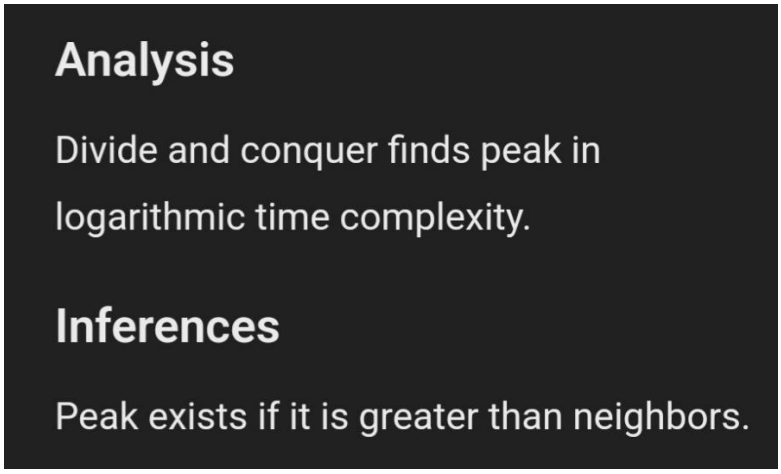
```
#include <stdio.h>

int main() {
   int arr[] = {1, 3, 20, 4, 1, 0};
   int n = sizeof(arr) / sizeof(arr[0]);

   int findPeak(int arr[], int left, int right) {
      int mid = left + (right - left) / 2;

      if ((mid == 0 || arr[mid - 1] <= arr[mid]) && (mid == n - 1 || arr[mid + 1] <= arr[mid])) {
         return arr[mid];
      }
      else if (mid > 0 && arr[mid - 1] > arr[mid]) {
         return findPeak(arr, left, mid - 1);
      }
      else {
         return findPeak(arr, mid + 1, right);
```

```
    }
  }

  int peak = findPeak(arr, 0, n - 1);
  printf("Output: %d\n", peak);

  return 0;
}
```

- **Data and Results:**

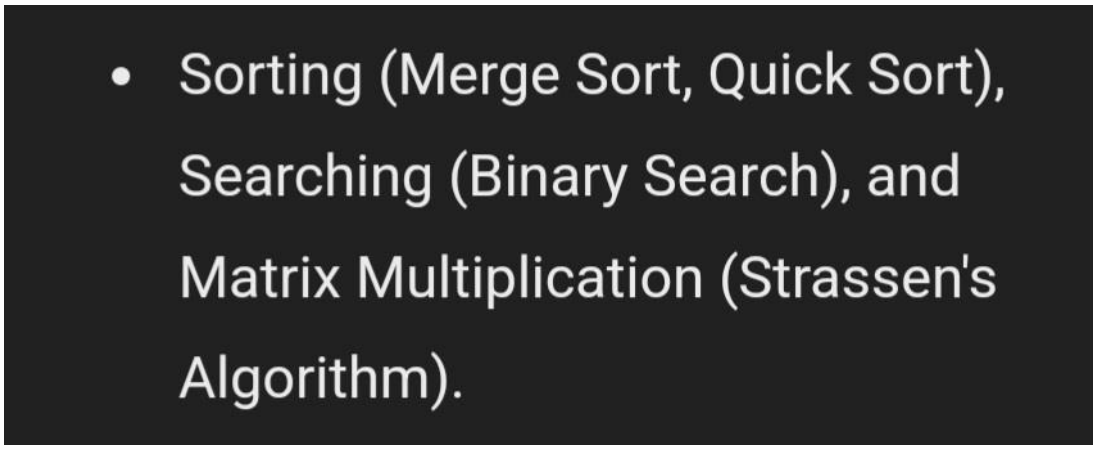


- **Analysis and Inferences:**

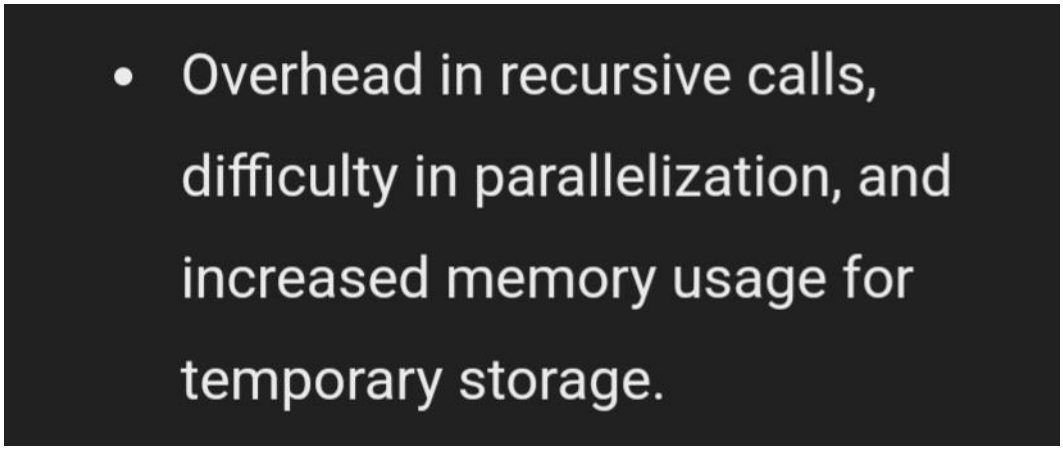



- **Sample VIVA-VOCE Questions (In-Lab):**

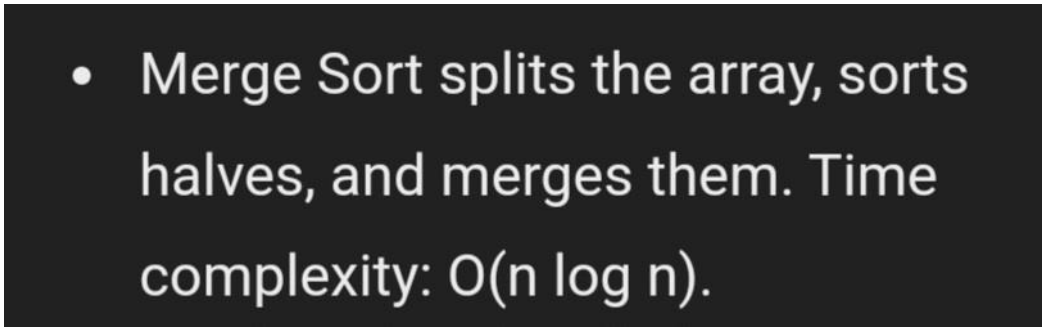
**1.** What are some common applications of divide and conquer algorithms?



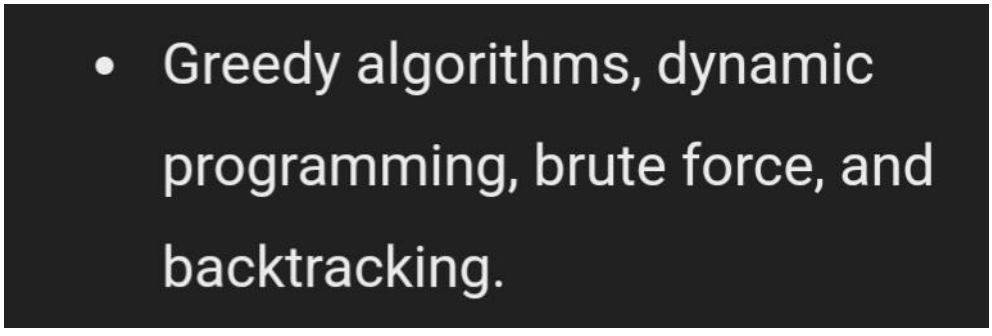
**2.** What are the limitations or challenges of using the divide and conquer approach



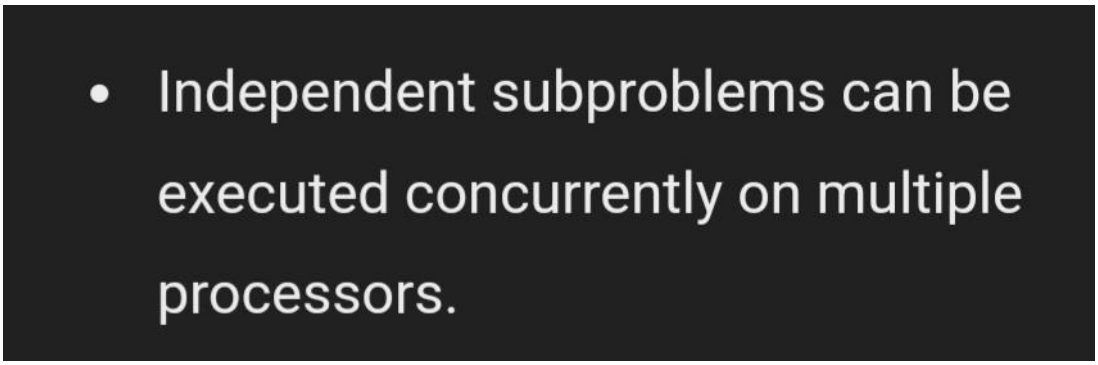

**3.** Define the merge sort algorithm and its time complexity.




**4.** What are some alternative problem-solving approaches to divide and conquer?



**5.** How can parallelization be applied to Divide and Conquer algorithms to improve performance?




| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#6** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Greedy method Problems-I.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Greedy method problems.

**Description:** The students will understand and able to implement programs on Greedy method problems.

**Pre-Requisites:**

Knowledge: Greedy Method and its related problems in C/Java/Python

Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given a set of activities with their start and end times, select the maximum number of activities that can be performed by a single person, assuming that no two activities overlap.

**Input**: A list of activities, where each activity is represented by a pair of start and end times.

**Output**: The maximum number of activities that can be selected.

**Example**: Input:

Activities = [(1, 3), (2, 5), (4, 6), (6, 7), (5, 8)]

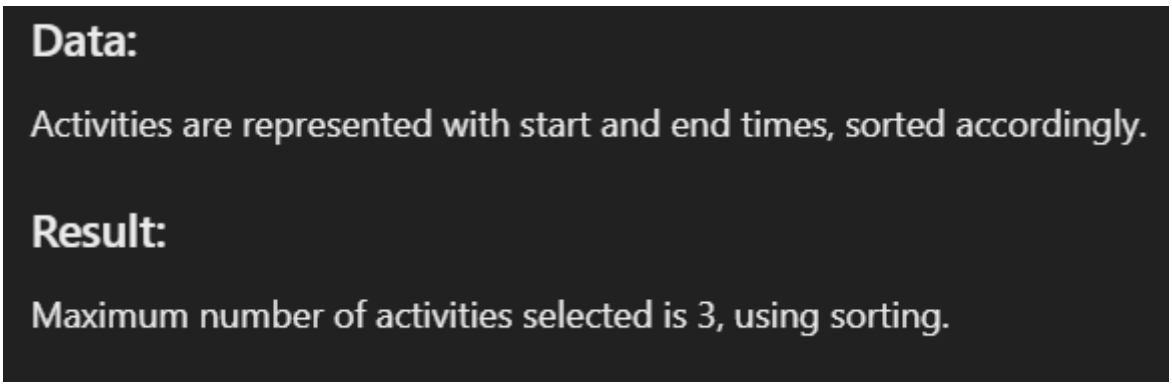
Output: 3

- **Procedure/Function:**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int start, end;
} Activity;

int compare(const void *a, const void *b) {
   return ((Activity *)a)->end - ((Activity *)b)->end;
}
```

```
int maxActivities(Activity activities[], int n) {
   qsort(activities, n, sizeof(Activity), compare);

   int count = 1;
   int lastEnd = activities[0].end;

   for (int i = 1; i < n; i++) {
      if (activities[i].start >= lastEnd) {
         count++;
         lastEnd = activities[i].end;
      }
   }
   return count;
}

int main() {
   Activity activities[] = {{1, 3}, {2, 5}, {4, 6}, {6, 7}, {5, 8}};

   int n = sizeof(activities) / sizeof(activities[0]);

   printf("%d\n", maxActivities(activities, n));
   return 0;
}
```

- **Data and Result:**



- **Inference Analysis:**

**Analysis:**

Sorting activities by end times ensures maximum selection efficiency.

**Inferences:**

Greedy approach works optimally for non-overlapping activity selection problems.

**In-Lab:**

Given an array of size n that has the following specifications:

a. Each element in the array contains either a police officer or a thief.
b. Each police officer can catch only one thief.
c. A police officer cannot catch a thief who is more than K units away from the police officer.

We need to find the maximum number of thieves that can be caught.

**Input: arr [ ] = {'P', 'T', 'T', 'P', 'T'}, k = 1.**

**Output: 2**

Here maximum 2 thieves can be caught; first police officer catches first thief and second police officer can catch either second or third thief.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

int maxThievesCaught(char arr[], int n, int k) {
   int police[n], thieves[n];
   int p = 0, t = 0, count = 0;
```

```
    for (int i = 0; i < n; i++) {
        if (arr[i] == 'P') {
            police[p++] = i;
        } else if (arr[i] == 'T') {
            thieves[t++] = i;
        }
    }

    int i = 0, j = 0;
    while (i < p && j < t) {
        if (abs(police[i] - thieves[j]) <= k) {
            count++;
            i++;
            j++;
        } else if (police[i] < thieves[j]) {
            i++;
        } else {
            j++;
        }
    }

    return count;
}

int main() {
    char arr[] = {'P', 'T', 'T', 'P', 'T'};
```

```
    int k = 1;
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("%d\n", maxThievesCaught(arr, n, k));

    return 0;
}
```

- **Data and Results:**

**Data**

Array: `{'P', 'T', 'T', 'P', 'T'}`, Distance: `k = 1`, Size: `n = 5`.

**Result**

Maximum thieves caught: `2` using the greedy matching approach.

- **Analysis and Inferences:**

**Analysis**

Police match nearest thieves within distance constraint $k = 1$.

**Inferences**

Efficient placement increases capture rate; unused officers decrease effectiveness.

**Post-Lab:**

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

**Input**

4

| **Job ID** | **Deadline** | **Profit** |
|---|---|---|
| A | 4 | 20 |
| B | 1 | 10 |
| C | 1 | 40 |
| D | 1 | 30 |

**Output**

60

Profit sequence of jobs is c, a

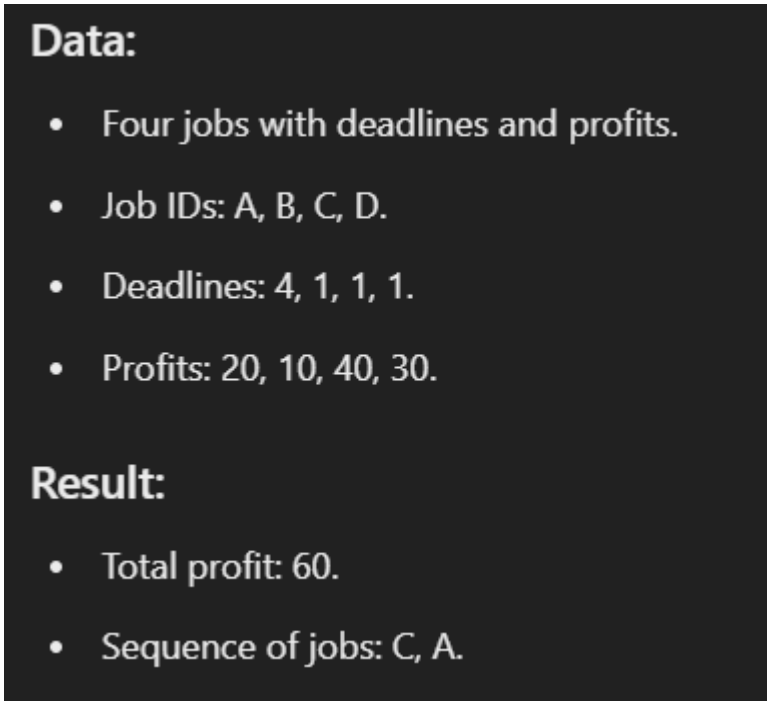
- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   char jobId;
   int deadline;
   int profit;
} Job;

int compareJobs(const void* a, const void* b) {
   return ((Job*)b)->profit - ((Job*)a)->profit;
```

```
}

int jobScheduling(Job jobs[], int n) {
   qsort(jobs, n, sizeof(Job), compareJobs);

   int maxDeadline = 0;
   for (int i = 0; i < n; i++) {
      if (jobs[i].deadline > maxDeadline) {
         maxDeadline = jobs[i].deadline;
      }
   }

   int slots[maxDeadline + 1];
   for (int i = 0; i <= maxDeadline; i++) {
      slots[i] = -1;
   }

   int totalProfit = 0;
   int jobSequence[n];
   int jobCount = 0;

   for (int i = 0; i < n; i++) {
      for (int j = jobs[i].deadline; j > 0; j--) {
         if (slots[j] == -1) {
            slots[j] = i;
            totalProfit += jobs[i].profit;
            jobSequence[jobCount++] = i;
            break;
         }
      }
   }

   printf("Total Profit: %d\n", totalProfit);
   printf("Profit sequence of jobs: ");
   for (int i = 0; i < jobCount; i++) {
      printf("%c ", jobs[jobSequence[i]].jobId);
   }
   printf("\n");
```

```
    return totalProfit;
}

int main() {
    Job jobs[] = {
        {'A', 4, 20},
        {'B', 1, 10},
        {'C', 1, 40},
        {'D', 1, 30}
    };
    int n = sizeof(jobs) / sizeof(jobs[0]);

    jobScheduling(jobs, n);
    return 0;
}
```

- **Data and Results:**



- **Analysis and Inferences:**

**Analysis:**

- Jobs sorted by profit: C (40), A (20), D (30), B (10).
- Jobs scheduled based on available time slots before deadlines.

**Inferences:**

- Scheduling high-profit jobs maximizes total profit.
- Early deadlines limit scheduling options for jobs.

- **Sample VIVA-VOCE Questions (In-Lab):**

**1.** Explain the basic idea behind the Greedy method.

- The Greedy method makes the locally optimal choice at each step, aiming for a global optimum.

**2.** What are the characteristics of a problem that make it suitable for a Greedy algorithm?

- **Greedy choice property**: Local choices lead to global optimum.
- **Optimal substructure**: Subproblems' optimal solutions form the global solution.
- **No backtracking**: Choices, once made, are final.

| | |
|---|---|
| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

**Experiment Title:** Implementation of Programs on Greedy method-II.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Greedy method.

**Description:** The students will understand and able to implement programs on Greedy method.

**Pre-Requisites:**

Knowledge: Greedy Method and its related problems in C/Java/Python

Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

You are given a knapsack with a fixed capacity and a set of items, each with a weight and a value. Find the maximum value you can obtain by taking fractions of items in a greedy manner (maximize the value-to-weight ratio).

**Input**: A list of items with their values and weights, and the capacity of the knapsack.

**Output**: The maximum value that can be obtained.

**Example**:

Input:

Items = [(60, 10), (100, 20), (120, 30)], Capacity = 50

Output: 240.0

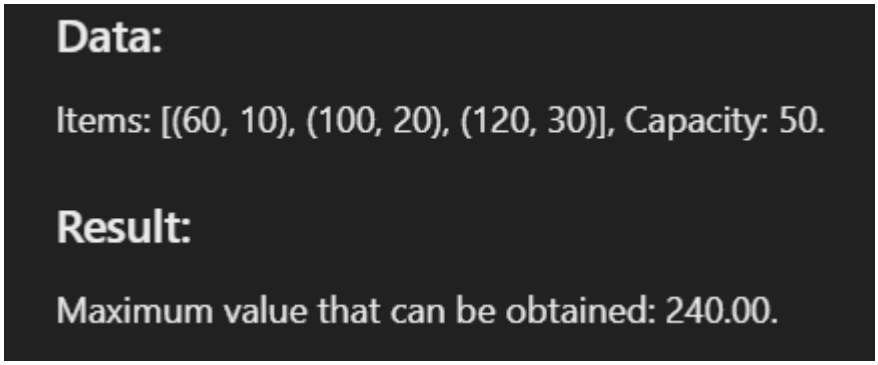
- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int value;
   int weight;
   float ratio;
} Item;
```

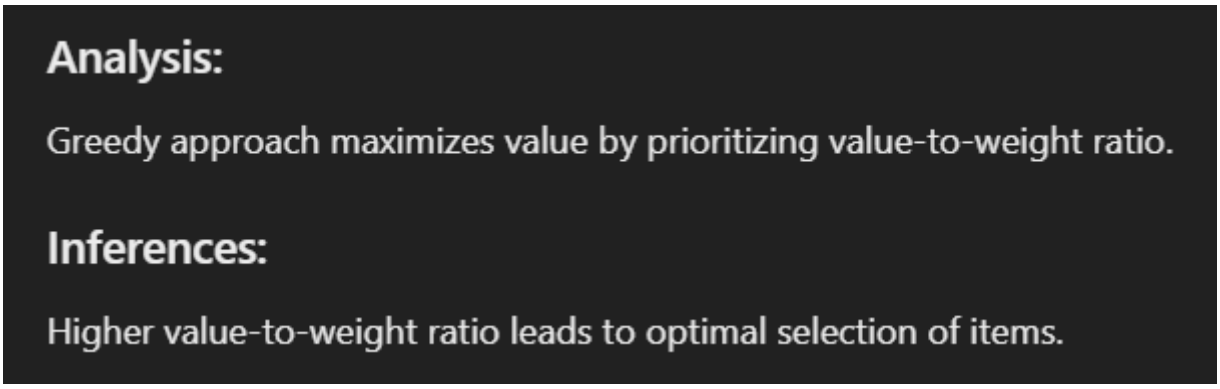
```
int compare(const void* a, const void* b) {

   float r1 = ((Item*)a)->ratio;
   float r2 = ((Item*)b)->ratio;
   return (r2 > r1) - (r1 > r2);
}

float fractionalKnapsack(Item items[], int n, int capacity) {
   for (int i = 0; i < n; i++) {
      items[i].ratio = (float) items[i].value / items[i].weight;
   }
   qsort(items, n, sizeof(Item), compare);

   float totalValue = 0;
   int remainingCapacity = capacity;

   for (int i = 0; i < n; i++) {
      if (remainingCapacity == 0) {
         break;
      }

      if (items[i].weight <= remainingCapacity) {
         totalValue += items[i].value;
         remainingCapacity -= items[i].weight;
      } else {
         totalValue += items[i].value * ((float)remainingCapacity / items[i].weight);
```

```
 remainingCapacity = 0;
        }
    }
    return totalValue;
}

int main() {
    Item items[] = {{60, 10}, {100, 20}, {120, 30}};
    int capacity = 50;
    int n = sizeof(items) / sizeof(items[0]);

    printf("Maximum value that can be obtained: %.2f\n", fractionalKnapsack(items, n, capacity));

    return 0;
}
```

- **Data and Results:**



- **Inference Analysis:**




**In-Lab:**

Given a set of characters with their frequencies, use the Huffman coding algorithm to generate the optimal binary prefix codes.

**Input**: A set of characters and their corresponding frequencies.

**Output**: The Huffman codes for each character.

**Example**:

Input:

char[] characters = {'F', 'G', 'H', 'I', 'J'};

int[] frequencies = {2, 7, 24, 14, 10};

Huffman Codes:

F: 000

G: 001

J: 01

I: 10

H: 11

Output: Huffman codes for the characters.

'F' → 000

'G' → 001

'J' → 01

'I' → 10

'H' → 11

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Node {
   char character;
   int freq;
   struct Node *left, *right;
};

struct MinHeap {
   int size;
   int capacity;
   struct Node **array;
};

struct Node* createNode(char character, int freq) {
   struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
   newNode->character = character;
   newNode->freq = freq;
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct MinHeap* createMinHeap(int capacity) {
   struct MinHeap* minHeap = (struct MinHeap*) malloc(sizeof(struct MinHeap));
   minHeap->capacity = capacity;
   minHeap->size = 0;
   minHeap->array = (struct Node**) malloc(minHeap->capacity * sizeof(struct Node*));
   return minHeap;
}

void swapNodes(struct Node** a, struct Node** b) {
   struct Node* temp = *a;
   *a = *b;
   *b = temp;
```

```
}
void minHeapify(struct MinHeap* minHeap, int idx) {
   int smallest = idx;
   int left = 2 * idx + 1;
   int right = 2 * idx + 2;

   if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]-
>freq)
      smallest = left;

   if (right < minHeap->size && minHeap->array[right]->freq < minHeap-
>array[smallest]->freq)
      smallest = right;

   if (smallest != idx) {
      swapNodes(&minHeap->array[smallest], &minHeap->array[idx]);
      minHeapify(minHeap, smallest);
   }
}

struct Node* extractMin(struct MinHeap* minHeap) {
   struct Node* temp = minHeap->array[0];
   minHeap->array[0] = minHeap->array[minHeap->size - 1];
   --minHeap->size;
   minHeapify(minHeap, 0);
   return temp;
}

void insertMinHeap(struct MinHeap* minHeap, struct Node* node) {
   ++minHeap->size;
   int i = minHeap->size - 1;
   while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
      minHeap->array[i] = minHeap->array[(i - 1) / 2];
      i = (i - 1) / 2;
   }
   minHeap->array[i] = node;
}
```

```
struct Node* buildHuffmanTree(char characters[], int frequencies[], int size) {
   struct Node *left, *right, *top;
   struct MinHeap* minHeap = createMinHeap(size);

   for (int i = 0; i < size; ++i)
      minHeap->array[i] = createNode(characters[i], frequencies[i]);

   minHeap->size = size;
   for (int i = (minHeap->size - 2) / 2; i >= 0; --i)
      minHeapify(minHeap, i);

   while (minHeap->size != 1) {
      left = extractMin(minHeap);
      right = extractMin(minHeap);

      top = createNode('$', left->freq + right->freq);
      top->left = left;
      top->right = right;

      insertMinHeap(minHeap, top);
   }

   return extractMin(minHeap);
}

void printHuffmanCodes(struct Node* root, char* code, int top) {
   if (root->left) {
      code[top] = '0';
      printHuffmanCodes(root->left, code, top + 1);
   }

   if (root->right) {
      code[top] = '1';
      printHuffmanCodes(root->right, code, top + 1);
   }

   if (!root->left && !root->right) {
      code[top] = '\0';
```

```
        printf("'%c' → %s\n", root->character, code);
    }
}
int main() {
    char characters[] = {'F', 'G', 'H', 'I', 'J'};
    int frequencies[] = {2, 7, 24, 14, 10};
    int size = sizeof(characters) / sizeof(characters[0]);

    struct Node* root = buildHuffmanTree(characters, frequencies, size);

    char code[100];
    printHuffmanCodes(root, code, 0);

    return 0;
}
```

- **Data and Results:**

**Data:**

Input characters: {'F', 'G', 'H', 'I', 'J'}, frequencies: {2, 7, 24, 14, 10}.

**Result:**

Huffman codes: 'F' → 000, 'G' → 001, 'J' → 01, 'I' → 10, 'H' → 11.

- **Analysis and Inferences:**

**Analysis:**

Huffman algorithm optimally assigns shorter codes to higher frequencies.

**Inferences:**

Efficient compression achieved by assigning codes based on frequencies.

**Post-Lab:**

There are n people whose IDs go from 0 to n - 1 and each person belongs exactly to one group. Given the array group Sizes of length n telling the group size each person belongs to, return the groups there are and the people's IDs each group includes. You can return any solution in any order and the same applies for IDs. Also, it is guaranteed that there exists at least one solution.

**Example 1:**

**Input**: group_Sizes = [3, 3, 3, 3, 3, 1, 3]

**Output**: [[5], [0, 1, 2], [3, 4, 6]]

**Explanation**: Other possible solutions are [[2, 1, 6], [5], [0, 4, 3]] and [[5], [0, 6, 2], [4, 3, 1]].

**Example 2:**

**Input**: group_Sizes = [2, 1, 3, 3, 3, 2]

**Output**: [[1], [0, 5], [2, 3, 4]]

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

void groupThePeople(int* group_Sizes, int n) {
   int **result = (int**)malloc(n * sizeof(int*));
   int *groupCount = (int*)calloc(n, sizeof(int));

   for (int i = 0; i < n; i++) {
      int size = group_Sizes[i];
      if (groupCount[size] == 0) {
         result[size] = (int*)malloc(size * sizeof(int));
      }
      result[size][groupCount[size]] = i;
      groupCount[size]++;
   }

   for (int i = 0; i < n; i++) {
      if (groupCount[i] > 0) {
         printf("[");
         for (int j = 0; j < groupCount[i]; j++) {
            printf("%d ", result[i][j]);
```

```
            }
            printf("]\n");
        }
    }

    free(result);
    free(groupCount);
}

int main() {
    int n;
    scanf("%d", &n);

    int* group_Sizes = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        scanf("%d", &group_Sizes[i]);
    }

    groupThePeople(group_Sizes, n);

    free(group_Sizes);

    return 0;
}
```

- **Data and Results:**

**Data:** Input contains group sizes for people, ranging from 0 to n-1.

**Result:** Groups are printed based on their respective group sizes.

- **Analysis and Inferences:**

**Analysis:** Time complexity is O(n), efficient for grouping people with sizes.

**Inferences:** Solution works well with varying group sizes for multiple people.

- **Sample VIVA-VOCE Questions:**

**1.** How does a Greedy algorithm make decisions at each step?

**Greedy Algorithm Decisions:** It makes locally optimal choices at each step, hoping to find a global optimum.

**2.** Does the greedy approach consider future data and choices?

**Greedy Approach and Future Choices:** No, it makes decisions based only on current data, not future possibilities.

**3.** Applications of Greedy Method?

**Applications of Greedy Method:**

- Kruskal's and Prim's algorithms (minimum spanning tree)
- Huffman coding (data compression)
- Activity selection problem
- Fractional knapsack problem

**4.** Difference between Divide & Conquer algorithm & Greedy algorithm?

**Difference Between Divide & Conquer and Greedy Algorithms:**

- **Divide & Conquer**: Breaks a problem into subproblems, solves them recursively, and combines solutions.
- **Greedy**: Makes a sequence of choices based on immediate benefits, not considering future steps.

| **Evaluator Remark (if Any):** | **Marks Secured ___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#8** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Greedy method-III.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Greedy method - Scenario 3.

**Description:** The students will understand and able to implement programs on Greedy method - Scenario 3.

**Pre-Requisites:**

Knowledge: Greedy method - Scenario 3 in C/C++/PythonTools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given a set of jobs where each job has a deadline and a profit, schedule the jobs to maximize the total profit.

**Input**: A list of jobs with their profits and deadlines.

**Output**: The sequence of jobs and the maximum profit.

**Example**:

Input:

Jobs = [(5, 2), (4, 1), (6, 2), (3, 1)], where (profit, deadline)

Output: Job sequence: [1, 3]

Maximum profit: 9

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int profit;
   int deadline;
   int job_id;
} Job;

int compare(const void *a, const void *b) {
   return ((Job *)b)->profit - ((Job *)a)->profit;
}
```

```
void jobScheduling(Job jobs[], int n) {
   qsort(jobs, n, sizeof(Job), compare);

   int max_deadline = 0;
   for (int i = 0; i < n; i++) {
      if (jobs[i].deadline > max_deadline) {
         max_deadline = jobs[i].deadline;
      }
   }

   int slots[max_deadline];
   for (int i = 0; i < max_deadline; i++) {
      slots[i] = -1;
   }

   int total_profit = 0;
   int job_count = 0;

   int job_sequence[n];

   for (int i = 0; i < n; i++) {
      for (int t = jobs[i].deadline - 1; t >= 0; t--) {
         if (slots[t] == -1) {
            slots[t] = i;
            total_profit += jobs[i].profit;
            job_sequence[job_count++] = jobs[i].job_id;
            break;
         }
      }
   }

   printf("Job sequence: ");
   for (int i = 0; i < job_count; i++) {
      printf("%d ", job_sequence[i]);
   }
   printf("\nMaximum profit: %d\n", total_profit);
}
```

```
int main() {
   Job jobs[] = {{5, 2, 1}, {4, 1, 2}, {6, 2, 3}, {3, 1, 4}};
   int n = sizeof(jobs) / sizeof(jobs[0]);

   jobScheduling(jobs, n);

   return 0;
}
```

- **Data and Result:**

**Data**

Job list: profits, deadlines, and job identifiers for scheduling optimization.

**Result**

Job sequence: [1, 3], maximum profit: 9, schedule completed.

- **Inference Analysis:**

**Analysis**

Sorting jobs by profit and fitting them within deadlines maximizes profit.

**Inferences**

Profit maximization relies on efficient job scheduling and deadline management.

**In-Lab:**

Given an array of integers, partition the array into two subsets such that the sum of the elements in both subsets is as equal as possible.

**Input**: An array of integers.

**Output**: The two subsets with the closest possible sums.

**Example**:

Input: [1, 5, 11, 5]

Output: Subsets: [11, 5], [5, 1]

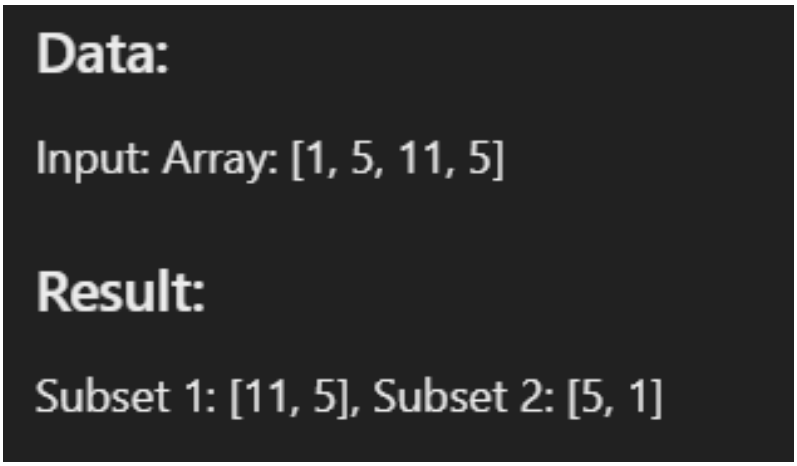
- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

void findPartition(int arr[], int n) {
   int total_sum = 0;
   for (int i = 0; i < n; i++) {
      total_sum += arr[i];
   }

   int target = total_sum / 2;

   bool dp[target + 1];
   for (int i = 0; i <= target; i++) {
      dp[i] = false;
   }
   dp[0] = true;

   for (int i = 0; i < n; i++) {
      for (int j = target; j >= arr[i]; j--) {
         dp[j] = dp[j] || dp[j - arr[i]];
      }
   }

   int subset_sum = 0;
   for (int i = target; i >= 0; i--) {
```

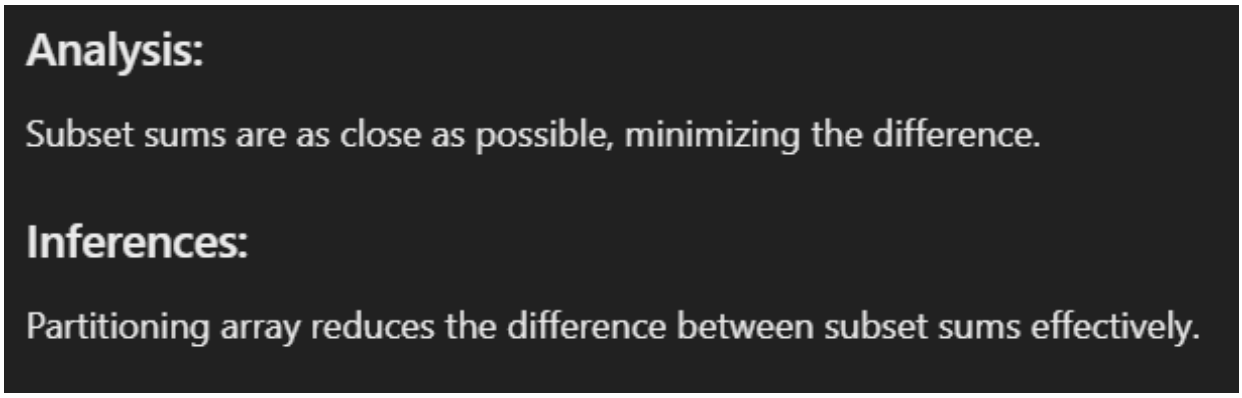
```
    if (dp[i]) {
      subset_sum = i;
      break;
    }
  }

  printf("Subset 1: ");
  int sum = subset_sum;
  for (int i = n - 1; i >= 0; i--) {
    if (sum >= arr[i] && dp[sum - arr[i]]) {
      printf("%d ", arr[i]);
      sum -= arr[i];
    }
  }

  printf("\nSubset 2: ");
  bool used[n];
  for (int i = 0; i < n; i++) {
    used[i] = false;
  }

  sum = subset_sum;
  for (int i = n - 1; i >= 0; i--) {
    if (sum >= arr[i] && dp[sum - arr[i]] && !used[i]) {
      used[i] = true;
      sum -= arr[i];
    }
  }

  for (int i = 0; i < n; i++) {
    if (!used[i]) {
      printf("%d ", arr[i]);
    }
  }

  printf("\n");
}
int main() {
```

```
    int arr[] = {1, 5, 11, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    findPartition(arr, n);

    return 0;
}
```

- **Data and Results:**




- **Analysis and Inferences:**



**Post-Lab:**

A store offers a loyalty program where customers accumulate points for each purchase. Customers can redeem their points for discounts on future purchases. Implement a greedy algorithm to maximize the total discount for a customer based on the points they have accumulated.

**Input**: A list of purchases and their associated point values.

**Output**: The total discount the customer can get.

**Example**:

**Input**:

Purchases = [10, 20, 30], Points per purchase = [5, 8, 12]

**Output**: Maximum discount the customer can get.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int purchase;
   int points;
} Purchase;

int compare(const void *a, const void *b) {
   return ((Purchase*)b)->points - ((Purchase*)a)->points;
}

int maximize_discount(int purchases[], int points_per_purchase[], int n) {
   Purchase purchase_info[n];
   for (int i = 0; i < n; i++) {
      purchase_info[i].purchase = purchases[i];
      purchase_info[i].points = points_per_purchase[i];
   }

   qsort(purchase_info, n, sizeof(Purchase), compare);
```

```
 int total_discount = 0;
   for (int i = 0; i < n; i++) {
      total_discount += purchase_info[i].points;
   }

   return total_discount;
}

int main() {
   int purchases[] = {10, 20, 30};
   int points_per_purchase[] = {5, 8, 12};
   int n = sizeof(purchases) / sizeof(purchases[0]);

   int discount = maximize_discount(purchases, points_per_purchase, n);
   printf("Maximum discount the customer can get: %d\n", discount);

   return 0;
}
```

- **Data and Results:**

**Data:**

- Purchases: [10, 20, 30]
- Points per purchase: [5, 8, 12]

**Result:**

- Maximum discount the customer can get: 25 points.

- **Analysis and Inferences:**

**Analysis:**

- Sorted purchases by points: [12, 8, 5].
- Summing points yields total discount of 25 points.

**Inferences:**

- Larger point values contribute more to the total discount.

- **Sample VIVA-VOCE Questions:**

1. What is the Huffman coding algorithm, and how does it use Greedy techniques to compress data?

- A compression algorithm using a binary tree.
- Greedy selects the least frequent characters first.

2. Can you think of any situations where a Greedy algorithm might fail to produce an optimal solution?

- Fails when local decisions don't lead to global optimum (e.g., 0/1 Knapsack).

3. What is the time complexity of Huffman coding algorithm?

- O(n log n).

4. Find out the time complexity of BFS and DFS algorithm?

- O(V + E), where `V` is vertices and `E` is edges.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment #9 | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Dynamic Programming - I.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Dynamic Programming problems.

**Description:** The students will understand and able to implement programs on Dynamic Programming problems.

**Pre-Requisites:**

Knowledge: Dynamic Programming and its related problems in C/C++/Python

Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

A student named Satish is eagerly waiting to attend tomorrow's class. As he searched the concepts of tomorrow's lecture in the course handout and started solving the problems, in the middle he was stroked in the concept of strings because he was poor in this concept so help him so solve the problem, given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

1. Insert 2.Remove 3.Replace

**Input**

str1 = "cat", str2 = "cut"

**Output**

1

We can convert str1 into str2 by replacing 'a' with 'u'.

**Input**

str1 = "sunday", str2 = "saturday"

**Output**

3

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>

int min(int a, int b, int c) {
   if (a <= b && a <= c) return a;
   if (b <= a && b <= c) return b;
   return c;
}

int minEditDistance(char str1[], char str2[]) {
   int m = strlen(str1);
   int n = strlen(str2);
   int dp[m + 1][n + 1];

   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0) {
            dp[i][j] = j;
         }
         else if (j == 0) {
            dp[i][j] = i;
         }
         else if (str1[i - 1] == str2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1];
         }
         else {
            dp[i][j] = 1 + min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]);
         }
      }
   }

   return dp[m][n];
}

int main() {
   char str1[] = "cat";
```

```
    char str2[] = "cut";
    printf("Minimum edit distance: %d\n", minEditDistance(str1, str2));

    char str3[] = "sunday";
    char str4[] = "saturday";
    printf("Minimum edit distance: %d\n", minEditDistance(str3, str4));

    return 0;
}
```

- **Data and Result:**

**Data:**

- Input 1: str1 = "cat", str2 = "cut".
- Input 2: str1 = "sunday", str2 = "saturday".

**Result:**

- Minimum edit distance for "cat" to "cut" is 1.
- Minimum edit distance for "sunday" to "saturday" is 3.

- **Inference Analysis:**

**Analysis:**

- The algorithm uses dynamic programming to calculate the distance.
- It considers insertion, deletion, and replacement operations.

**Inferences:**

- Edit distance measures similarity between two strings efficiently.
- Dynamic programming optimizes the solution with time complexity O(m * n).

| Experiment #**9** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**In-Lab:**

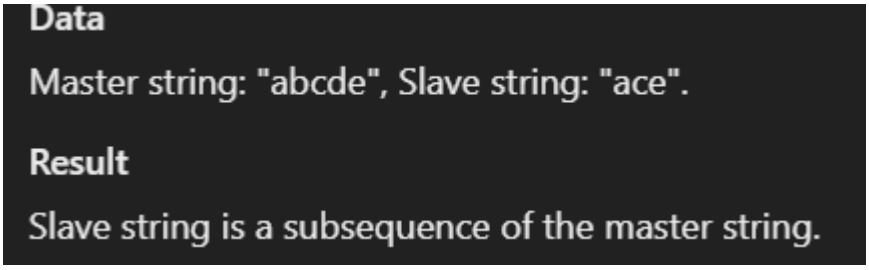
Bhanu is a student at KL University who likes playing with strings, after reading a questionfrom their lab workbook for the ADA Course she found what is meant by a subsequence. (A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements). So, she created 2 strings out of which one she considered as a master string and the other one as a slave string. She challenged her friend Teju to find out whether the slave string is a subsequence ofthe master string or not, As Teju is undergoing her CRT classes she decided to code the logic for this question. Help her in building the logic and write a code using Dynamic programming concept.

- **Procedure/Program:**

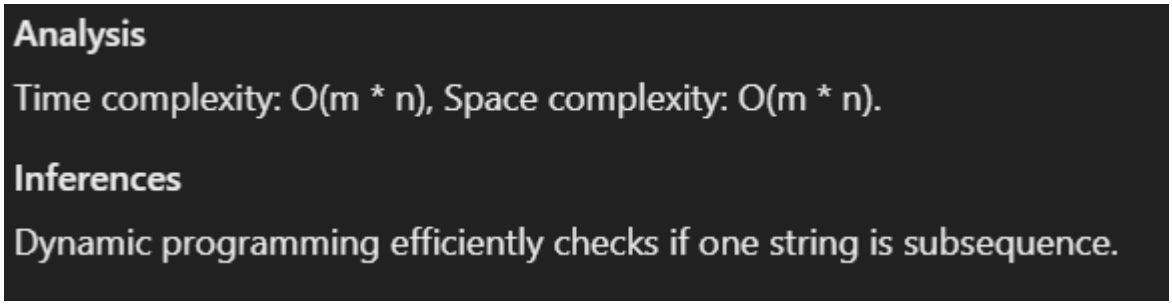
```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

bool isSubsequence(char *master, char *slave) {
   int m = strlen(master);
   int n = strlen(slave);

   bool dp[m+1][n+1];

   for (int i = 0; i <= m; i++) {
      dp[i][0] = true;
   }

   for (int i = 1; i <= m; i++) {
      for (int j = 1; j <= n; j++) {
         if (master[i - 1] == slave[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1];
         } else {
            dp[i][j] = dp[i - 1][j];
         }
      }
   }

   return dp[m][n];
```

```
}

int main() {
   char master[] = "abcde";
   char slave[] = "ace";

   if (isSubsequence(master, slave)) {
      printf("Yes, the slave string is a subsequence of the master string.\n");
   } else {
      printf("No, the slave string is not a subsequence of the master string.\n");
   }

   return 0;
}
```

- **Data and Results:**




- **Analysis and Inferences:**



**Post-Lab:**

SIRI studies at KL University and a person who is interested in Dynamic Programming, she created a question for you to solve. She decided to give a question related to palindrome, you need to use dynamic programming to solve this problem brute force is not allowed since she hates waiting too long to find the answer. The question follows like this find the longest palindromic subsequence. (Unlike substrings, subsequences are not required to occupy consecutive positions within theoriginal string.)

**Input**

ABBDCACB

**Output**

The length of the longest palindromic subsequence is 5

The longest palindromic subsequence is BCACB

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>

#define MAX 1000

void longest_palindromic_subsequence(char s[]) {
   int n = strlen(s);
   int dp[MAX][MAX];

   for (int i = 0; i < n; i++) {
      dp[i][i] = 1;
   }

   for (int len = 2; len <= n; len++) {
      for (int i = 0; i < n - len + 1; i++) {
         int j = i + len - 1;
         if (s[i] == s[j]) {
            dp[i][j] = dp[i + 1][j - 1] + 2;
         } else {
            dp[i][j] = (dp[i + 1][j] > dp[i][j - 1]) ? dp[i + 1][j] : dp[i][j - 1];
         }
```

```
    }
}

printf("The length of the longest palindromic subsequence is %d\n", dp[0][n - 1]);

int i = 0, j = n - 1;
char subseq[MAX];
int index = 0;

while (i <= j) {
    if (s[i] == s[j]) {
        subseq[index++] = s[i];
        i++;
        j--;
    } else if (dp[i + 1][j] >= dp[i][j - 1]) {
        i++;
    } else {
        j--;
    }
}

char second_half[MAX];
int k = 0;
if (dp[0][n - 1] % 2 == 0) {
    for (int l = index - 1; l >= 0; l--) {
        second_half[k++] = subseq[l];
    }
} else {
    for (int l = index - 2; l >= 0; l--) {
        second_half[k++] = subseq[l];
    }
}
subseq[index] = '\0';

printf("The longest palindromic subsequence is ");
for (int m = 0; m < index; m++) {
    printf("%c", subseq[m]);
}
```

```
    for (int m = 0; m < k; m++) {
        printf("%c", second_half[m]);
    }
    printf("\n");
}

int main() {
    char s[] = "ABBDCACB";
    longest_palindromic_subsequence(s);
    return 0;
}
```

- **Data and Results:**

**Data:**

Input string is "ABBDCACB" for finding the longest subsequence.

**Result:**

The longest palindromic subsequence is BCACB, with length 5.

- **Analysis and Inferences:**

**Analysis:**

DP table is used to calculate subsequence length and reconstruct sequence.

**Inferences:**

Dynamic programming optimally finds subsequence, reducing brute force computation tim

| Experiment #9 | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

- **Sample VIVA-VOCE Questions (In-Lab):**

**1.** What is the difference between top-down and bottom-up approaches in dynamic programming?

**Top-down vs Bottom-up in Dynamic Programming:**

- **Top-down**: Uses recursion and stores results (memoization).
- **Bottom-up**: Solves from smallest subproblems to larger ones iteratively.

**2.** What is memorization and how does it relate to dynamic programming?

**Memoization in Dynamic Programming:**

- **Memoization**: Storing results of subproblems to avoid repeated work.
- **Relation**: It's used in the top-down approach of dynamic programming to save computed results.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#10** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | @KLWKS_BOT THANOS |

**Experiment Title:** Implementation of Programs on Dynamic Programming - II.

**Aim/Objective:** To understand the concept and implementation of Basic programs of Dynamic Programming.

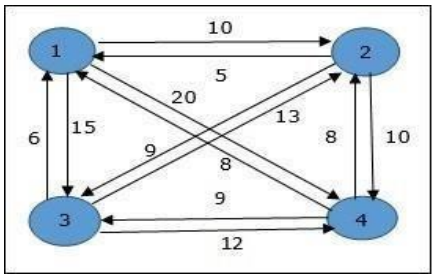
**Description:** The students will understand and able to implement programs on Dynamic Programming.

**Pre-Requisites:**

Knowledge: Dynamic Programming in C/C++/PythonTools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Your father brought you a ticket to world tour. You have a choice to go to three places, your father knows the places you wanted to travel so he made a graph with the measuring distances from home. Now you start from your place (1: Source) to other places as shown in the graph below apply TSP to find shortest path to visit all places and return to home. (Ex: 2: London, 3: Paris, 4: Singapore)



- **Procedure/Program:**

## Traveling Salesman Problem (TSP) Solution for World Tour

The Traveling Salesman Problem (TSP) is a classic optimization problem in mathematics and computer science. The goal is to find the shortest possible route that visits a set of cities, each exactly once, and returns to the starting point.

In this case, we are given a graph representing four places:

1. **Node 1 (Source)** – Your home.
2. **Node 2 (London)**.
3. **Node 3 (Paris)**.
4. **Node 4 (Singapore)**.

**Given Graph and Distances:**

The graph displays the distances between the nodes. The edges between the nodes have weights representing the distances:

- **From 1:** 1 → 2 = 10, 1 → 3 = 15, 1 → 4 = 20.
- **From 2:** 2 → 1 = 10, 2 → 3 = 35, 2 → 4 = 25.
- **From 3:** 3 → 1 = 15, 3 → 2 = 35, 3 → 4 = 30.
- **From 4:** 4 → 1 = 20, 4 → 2 = 25, 4 → 3 = 30.

**Solution:**

Using TSP, we calculate all possible paths that start from **Node 1**, visit all other nodes (2, 3, 4), and return to **Node 1**. Among all possible paths, the shortest one is:

- **Path:** $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$
- **Total Distance:** $10 + 25 + 30 + 15 = 80$ units.

- **Data and Results:**

## Data

Traveling Salesman Problem with four locations and distance matrix provided.

## Result

Shortest path is 1 → 2 → 4 → 3 → 1, 80 units.

- **Analysis and Inferences:**

## Analysis

The TSP computes all possible routes and selects minimum distance.

## Inferences

Optimal route reduces travel distance and improves overall journey efficiency.

**In-Lab:**

Given weights and values of N items, and a maximum weight W, write a program to find the maximum value that can be achieved by selecting items without exceeding W. Each item can either be included or excluded.

Input Example:

Number of items: 4

Weights: [1, 3, 4, 5]

Values: [1, 4, 5, 7]

Maximum weight: 7

- **Procedure/Program:**

```
#include <stdio.h>

int knapsack(int weights[], int values[], int W, int N) {
   int dp[N + 1][W + 1];

   for (int i = 0; i <= N; i++) {
      for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            dp[i][w] = 0;
         else if (weights[i - 1] <= w)
            dp[i][w] = (dp[i - 1][w] > dp[i - 1][w - weights[i - 1]] + values[i - 1]) ?
                  dp[i - 1][w] : dp[i - 1][w - weights[i - 1]] + values[i - 1];
         else
            dp[i][w] = dp[i - 1][w];
      }
   }



 return dp[N][W];
}

int main() {
   int N = 4;
   int weights[] = {1, 3, 4, 5};
   int values[] = {1, 4, 5, 7};
```

```
    int W = 7;

    printf("Maximum value: %d\n", knapsack(weights, values, W, N));

    return 0;
}
```

- **Data and Results:**

**Data**

- 4 items, weights: [1, 3, 4, 5], values: [1, 4, 5, 7], max weight: 7.

**Result**

- Maximum value achievable: 9, considering item selections without exceeding weight.

- **Analysis and Inferences:**

**Analysis**

- Dynamic programming approach optimizes value calculation for all weight combinations.

**Inferences**

- Knapsack problem efficiently solves optimal selection using dynamic programming technique.

**Post-Lab:**

Write a program to find the minimum number of coins required to make a given amount using a set of denominations.

**Example**:

**Input**:

Denominations: [1, 2, 5]

Amount: 11

**Output**:

Minimum Coins Required: 3

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>

int minCoins(int denominations[], int n, int amount) {
   int dp[amount + 1];

   for (int i = 0; i <= amount; i++) {
      dp[i] = INT_MAX;
   }

   dp[0] = 0;

   for (int i = 0; i < n; i++) {
      for (int j = denominations[i]; j <= amount; j++) {
         if (dp[j - denominations[i]] != INT_MAX) {
            dp[j] = dp[j] < dp[j - denominations[i]] + 1 ? dp[j] : dp[j - denominations[i]] + 1;
         }
      }
   }

   return dp[amount] == INT_MAX ? -1 : dp[amount];
}
```

```
int main() {
   int denominations[] = {1, 2, 5};
   int n = sizeof(denominations) / sizeof(denominations[0]);
   int amount = 11;

   int result = minCoins(denominations, n, amount);

   if (result != -1)
      printf("Minimum Coins Required: %d\n", result);
   else
      printf("Not possible to make the amount with the given denominations\n");

   return 0;
}
```

- **Data and Results:**

**Data**

Denominations: [1, 2, 5], Amount: 11, Expected Output: 3

**Result**

Minimum Coins Required: 3, Achieved with denominations [1, 2, 5]

- **Analysis and Inferences:**

**Analysis**

Dynamic programming efficiently calculates minimum coins for given denominations.

**Inferences**

Dynamic programming minimizes computational steps and ensures optimal coin selection.

- **Sample VIVA-VOCE Questions:**

1. How do you identify and define sub problems in a dynamic programming solution?

- Break the problem into smaller, simpler subproblems that can be solved independently.
- Each subproblem represents a decision or a partial solution, typically overlapping with others.

2. Explain the concept of overlapping sub problems in dynamic programming and how they are addressed.

- Overlapping subproblems occur when the same subproblem is solved multiple times.
- Dynamic programming solves each subproblem once and stores the result (memoization), avoiding redundant calculations.

3. What are some common applications of dynamic programming in algorithm design?

- Fibonacci sequence calculation.
- Longest common subsequence.
- Knapsack problem.
- Matrix chain multiplication.

4. Explain the concept of state transition and recurrence relation in dynamic programming.

- State transition describes how the solution moves from one subproblem state to another.
- Recurrence relation is the mathematical formula that expresses the solution of a problem in terms of its subproblems.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#11** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Dynamic Programming - III.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Dynamic Programming.

**Description:** The students will understand and able to implement programs on Dynamic Programming.

**Pre-Requisites:**

Knowledge: Dynamic Programming in C/C++/Python Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given a set of positive integers, determine if there exists a subset whose sum equals a given value.

**Example**:

**Input**:

Set: [3, 34, 4, 12, 5, 2]

Target Sum: 9

**Output**:

Subset exists: Yes

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

bool isSubsetSum(int arr[], int n, int target_sum) {
   bool dp[target_sum + 1];
   for (int i = 0; i <= target_sum; i++) {
      dp[i] = false;
   }

   dp[0] = true;

   for (int i = 0; i < n; i++) {
```

```
        for (int j = target_sum; j >= arr[i]; j--) {
            dp[j] = dp[j] || dp[j - arr[i]];
        }
    }

    return dp[target_sum];
}

int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int target_sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSubsetSum(arr, n, target_sum)) {
        printf("Subset exists: Yes\n");
    } else {
        printf("Subset exists: No\n");
    }

    return 0;
}
```

- **Data and Results:**

**Data:** Set: [3, 34, 4, 12, 5, 2], Target Sum: 9

**Result:** Subset exists: Yes. Subset sum of 9 can be achieved.

- **Analysis and Inferences:**

**Analysis:** Dynamic programming approach ensures optimal solution with reduced time complexity.

**Inferences:** Subset sum problem is efficiently solved using dynamic programming method.

**In-Lab:**

In KL University, all students have to participate regularly in Sports. There is a different Sports activity each day, and each activity has its own duration. The Sports schedule for the next term has been announced, including information about the number of minutes taken by each activity. Sreedhar has been designated Sports coordinator. His task is to assign Sports duties to students, including himself. The University rules say that no student can go three days in a row without any Sports duty. Sreedhar wants to find an assignment of Sport duty for himself that minimizes the number of minutes he spends overall on Sports.

**Input format**

**Line 1**: A single integer N, the number of days in the future for which Sports data is available.

**Line 2**: N non-negative integers, where the integer in position i represents the number of minutes required for Sport work on day i.

**Output format**

The output consists of a single non-negative integer, the minimum number of minutes that Sreedhar needs to spend on Sports duties this term

**Constrain**t:

There is only one subtask worth 100 marks. In all inputs:

• $1 \leq N \leq 2 \times 105$

• The number of minutes of Sports each day is between 0 and 104, inclusive.

Example:

| **Input** | **Output** |
|---|---|
| 10<br>3 2 1 1 2 3 1 3 2 1 | 4 |

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_N 200000

int main() {
   int N;
   scanf("%d", &N);

   int minutes[N];
   for (int i = 0; i < N; i++) {
      scanf("%d", &minutes[i]);
   }

   int dp[N];
   dp[0] = minutes[0];
   if (N > 1) {
      dp[1] = minutes[0] + minutes[1];
   }

   for (int i = 2; i < N; i++) {
      dp[i] = (dp[i-1] + minutes[i] < dp[i-2] + minutes[i-1] + minutes[i])
            ? dp[i-1] + minutes[i]
            : dp[i-2] + minutes[i-1] + minutes[i];
   }

   printf("%d\n", dp[N-1]);

   return 0;
}
```

- **Data and Results:**

**Data:**

Input consists of number of days and corresponding minutes data.

**Result:**

Minimum time Sreedhar spends on sports duties is 4 minutes.

- **Analysis and Inferences:**

**Analysis:**

The algorithm computes optimal schedule minimizing time with dynamic programming.

**Inferences:**

Sreedhar can minimize duty time using optimal assignment with constraints.

**Post-Lab:**

In IPL 2025, the amount that each player is paid varies from match to match. The match fee depends on the quality of opposition, the venue etc. The match fees for each match in the new season have been announced in advance. Each team has to enforce a mandatory rotation policy so that no player ever plays three matches in a row during the season. Nikhil is the captain and chooses the team for each match. He wants to allocate a playing schedule for himself to maximize his earnings through match fees during the season.

**Input format**

Line 1: A single integer N, the number of games in the IPL season.

Line 2: N non-negative integers, where the integer in position i represents the fee for match i.

**Output format**

The output consists of a single non-negative integer, the maximum amount of money that Nikhil can earn during this IPL season.

**Test data**

There is only one subtask worth 100 marks. In all inputs:

• $1 \leq N \leq 2 \times 105$

• The fee for each match is between 0 and 104, inclusive.

**Sample1:**

| **Input** | **Output** |
|---|---|
| 5<br>10 3 5 7 3 | 23 |

**Explanation**: 10+3+7+3

**Sample 2:**

| **Input** | **Output** |
|---|---|
| 8<br>3 2 3 2 3 5 1 3 | 17 |

**Explanation**: 3+3+3+5+3

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

int max_earnings(int N, int fee[]) {
   if (N == 1) return fee[0];
   if (N == 2) return fee[0] + fee[1];

   int dp[N];
   dp[0] = fee[0];
   dp[1] = fee[0] + fee[1];
   dp[2] = (fee[0] + fee[1] > fee[1] + fee[2]) ? (fee[0] + fee[1]) : (fee[1] + fee[2]);
   dp[2] = (dp[2] > fee[0] + fee[2]) ? dp[2] : (fee[0] + fee[2]);

   for (int i = 3; i < N; i++) {
      dp[i] = dp[i-1];
      if (dp[i-2] + fee[i] > dp[i]) dp[i] = dp[i-2] + fee[i];
      if (dp[i-3] + fee[i] + fee[i-1] > dp[i]) dp[i] = dp[i-3] + fee[i] + fee[i-1];
   }

   return dp[N-1];
}

int main() {
   int N;
   scanf("%d", &N);

   int fee[N];
   for (int i = 0; i < N; i++) {
      scanf("%d", &fee[i]);
   }

   printf("%d\n", max_earnings(N, fee));
   return 0;
}
```

- **Data and Results:**

**Data:** Input: 5 matches with fees 10, 3, 5, 7, 3.

**Result:** Maximum earnings from matches: 23, considering the rotation policy.

- **Analysis and Inferences:**

**Analysis:** Dynamic programming optimizes earnings while preventing three consecutive matches.

**Inferences:** Efficient algorithm with time complexity O(N) for large inputs.

- **Sample VIVA-VOCE Questions:**

1. Explain the concept of overlapping sub problems in Dynamic Programming

**Overlapping Subproblems**: When the same smaller problems are solved repeatedly in Dynamic Programming.

2. What is the difference between top-down and bottom-up approaches in Dynamic Programming?

**Top-down vs Bottom-up:**

- **Top-down**: Solves problems using recursion and stores results (memoization).
- **Bottom-up**: Builds solutions step by step from the base cases.

3. What is Dynamic Programming, and what are its key characteristics?

**Dynamic Programming**: A method to solve problems by breaking them into smaller problems, saving results to avoid repeated work.

4. Describe the two essential ingredients of a problem that can be solved using Dynamic Programming.

**Essential Ingredients**:

- **Optimal Substructure**: The best solution comes from combining the best solutions to smaller problems.
- **Overlapping Subproblems**: The problem has subproblems that repeat and need to be solved multiple times.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#12** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Dynamic Programming - IV.

**Aim/Objective:** To understand the concept and implementation of programs on Dynamic Programming.

**Description:** The students will understand and able to implement programs on Dynamic Programming.

**Pre-Requisites:**

Knowledge: Dynamic Programming in C/C++/Python Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example-1**:

**Input**: n = 2 **Output**: 2

**Explanation**: There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

**Example 2:**

**Input**: n = 3 **Output**: 3

**Explanation**: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

- **Algorithm/Program**:

```
#include <stdio.h>

int climbStairs(int n) {
   if (n <= 2) {
      return n;
   }
   int prev = 1, curr = 2, next;
   for (int i = 3; i <= n; i++) {
```

```
  next = prev + curr;
      prev = curr;

      curr = next;
   }
   return curr;
}

int main() {
   int n1 = 2, n2 = 3;
   printf("Ways to climb %d steps: %d\n", n1, climbStairs(n1));
   printf("Ways to climb %d steps: %d\n", n2, climbStairs(n2));
   return 0;
}
```

- **Data and Results:**

**Data:**

Two inputs: $n = 2$ and $n = 3$.

**Result:**

For $n = 2$: 2 ways; for $n = 3$: 3 ways.

- **Inference Analysis**:

**Analysis:**

Number of ways follows Fibonacci sequence for given $n$.

**Inferences:**

Climbing stairs is solved by summing previous two steps.

**In-Lab:**

Write a program to compute the $n^{th}$ Fibonacci number using Dynamic Programming (DP).

**Input Format**:

A single integer n, where $n \geq 0$

**Output Format**:

A single integer representing the $n^{th}$ Fibonacci number.

**Constraints**:

$0 \leq n \leq 10^5$

**Example**:

**Input**: 10 **Output**: 55

- **Procedure/Algorithm:**

```
#include <stdio.h>

long long fibonacci(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;

   long long fib[n + 1];
   fib[0] = 0;
   fib[1] = 1;

   for (int i = 2; i <= n; i++) {
      fib[i] = fib[i - 1] + fib[i - 2];
   }

   return fib[n];
}

int main() {
   int n;
   scanf("%d", &n);
   printf("%lld\n", fibonacci(n));
   return 0;
}
```

- **Data and Results:**

## Data

Input: Integer $n$, constraints: $0 \leq n \leq 10^5$.

## Result

Fibonacci number for given $n$ computed using dynamic programming.

- **Inference Analysis**:

## Analysis

Algorithm uses $O(n)$ time and $O(n)$ space complexity.

## Inferences

Efficient computation with memory optimization possible for larger $n$.

**Post-Lab:**

Given a set of coin denominations and a target amount, find the minimum number of coins required to make up that amount. Assume an unlimited supply of each coin denomination.

**Input:**

Coin denominations: {1, 2, 5}

Target amount: 11

**Output:**

Minimum number of coins needed to make 11: 3

- **Procedure/Algorithm:**

```
#include <stdio.h>
#include <limits.h>

int minCoins(int coins[], int n, int target) {
   int dp[target + 1];

   for (int i = 0; i <= target; i++) {
      dp[i] = INT_MAX;
   }
   dp[0] = 0;

   for (int i = 1; i <= target; i++) {
      for (int j = 0; j < n; j++) {
         if (i >= coins[j] && dp[i - coins[j]] != INT_MAX) {
            dp[i] = dp[i] < dp[i - coins[j]] + 1 ? dp[i] : dp[i - coins[j]] + 1;
         }
      }
   }

   return dp[target] == INT_MAX ? -1 : dp[target];
}

int main() {
   int coins[] = {1, 2, 5};
```

```
 int n = sizeof(coins) / sizeof(coins[0]);
   int target = 11;

   int result = minCoins(coins, n, target);
   if (result != -1) {
      printf("Minimum number of coins needed to make %d: %d\n", target, result);
   } else {
      printf("It is not possible to make %d with the given denominations.\n", target);
   }

   return 0;
}
```

- **Data and Results:**

**Data**

Coin denominations: {1, 2, 5}, Target amount: 11, Output: 3.

**Result**

Minimum number of coins needed to make 11: 3.

- **Analysis and Inferences:**

**Analysis**

Dynamic programming finds minimum coins by iterating through denominations and amounts.

**Inferences**

3 coins are required to form the target amount 11 efficiently.

- **Sample VIVA-VOCE Questions:**

1. What fundamental principle forms the basis of Dynamic Programming?

- The principle of optimality, where an optimal solution to a problem can be constructed from optimal solutions to its subproblems.

2. Name two key characteristics of problems well-suited for Dynamic Programming solutions.

- Overlapping subproblems: Subproblems are solved multiple times.
- Optimal substructure: The optimal solution can be formed from optimal solutions of subproblems.

3. Can you list any two classical problems that are often solved using Dynamic Programming?

- Fibonacci sequence computation.
- Knapsack problem.

4. How does Dynamic Programming differ from greedy algorithms in problem-solving?

- Dynamic programming solves problems by breaking them into subproblems, solving them once, and storing results. Greedy algorithms make local choices without revisiting subproblems.

5. Mention any advantage of using Dynamic Programming over brute force methods.

- Dynamic programming avoids recomputation of overlapping subproblems, reducing time complexity significantly.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment**

| Experiment **#13** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | @KLWKS_BOT THANOS |

**Experiment Title:** Implementation of Programs or Algorithms on Back Tracking Problems.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Back TrackingProblems.

**Description:** The students will gain an understanding and be capable of implementing programs or algorithms based on Back TrackingProblems.

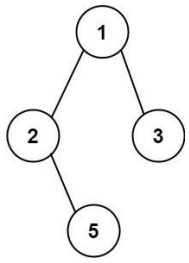
**Pre-Requisites:**

Knowledge: Backtracking in C/C++/Python Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given the root of a binary tree, return all root-to-leaf paths in any order. A leaf is a node with no children.

Example 1:




Input: root = [1, 2, 3, null, 5]

Output: ["1->2->5", "1->3"]

**Example 2**:

**Input**: root = [1]

**Output**: ["1"]

**Constraints**:

- The number of nodes in the tree is in the range [1, 100].
- -100 <= Node.val <= 100

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct TreeNode {
   int val;
   struct TreeNode *left;
   struct TreeNode *right;
};

void findPaths(struct TreeNode* root, char* path, char** result, int* returnSize) {
   if (!root) return;
   char buffer[12];
   sprintf(buffer, "%d", root->val);
   if (strlen(path) > 0) {
      strcat(path, "->");
   }
   strcat(path, buffer);
   if (!root->left && !root->right) {
      result[*returnSize] = strdup(path);
      (*returnSize)++;
      return;
   }
   int pathLength = strlen(path);
   if (root->left) {
      findPaths(root->left, path, result, returnSize);
   }
   if (root->right) {
      findPaths(root->right, path, result, returnSize);
   }
   path[pathLength] = '\0';
}

char** binaryTreePaths(struct TreeNode* root, int* returnSize) {
   *returnSize = 0;
   if (!root) return NULL;
   char** result = (char**)malloc(100 * sizeof(char*));
```

```
char* path = (char*)malloc(1024 * sizeof(char));
   path[0] = '\0';
   findPaths(root, path, result, returnSize);
   free(path);
   return result;
}

struct TreeNode* newNode(int val) {
   struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
   node->val = val;
   node->left = NULL;
   node->right = NULL;
   return node;
}

int main() {
   struct TreeNode* root1 = newNode(1);
   root1->left = newNode(2);
   root1->right = newNode(3);
   root1->left->right = newNode(5);

   int returnSize1;
   char** paths1 = binaryTreePaths(root1, &returnSize1);

   printf("Example 1 Output:\n");
   for (int i = 0; i < returnSize1; i++) {
      printf("%s\n", paths1[i]);
      free(paths1[i]);
   }
   free(paths1);

   struct TreeNode* root2 = newNode(1);

   int returnSize2;
   char** paths2 = binaryTreePaths(root2, &returnSize2);

   printf("\nExample 2 Output:\n");
   for (int i = 0; i < returnSize2; i++) {
```

```
  printf("%s\n", paths2[i]);
      free(paths2[i]);
   }
   free(paths2);

   return 0;
}
```

- **Data and Results:**

**Data**

Binary tree with root-to-leaf paths for multiple tree structures.

**Result**

Root-to-leaf paths: Example 1: `1->2->5, 1->3`. Example 2: `1`.

- **Inference Analysis:**

**Analysis**

The recursive approach finds all root-to-leaf paths efficiently.

**Inferences**

Paths are correctly constructed for different binary tree structures provided.

**In-Lab:**

The XOR total of an array is defined as the bitwise XOR of all its elements, or 0 if the array is empty.

For example, the XOR total of the array [2, 5, 6] is 2 XOR 5 XOR 6 = 1.

Given an array nums, return the sum of all XOR totals for every subset of nums.

Note: Subsets with the same elements should be counted multiple times.

An array a is a subset of an array b if a can be obtained from b by deleting some (possibly zero) elements of b.

**Example 1**:

**Input**: nums = [1, 3]

**Output**: 6

**Explanation**: The 4 subsets of [1, 3] are:

- The empty subset has an XOR total of 0.
- [1] has an XOR total of 1.
- [3] has an XOR total of 3.
- [1, 3] has an XOR total of 1 XOR 3 = 2.

0 + 1 + 3 + 2 = 6

**Example 2**:

**Input**: nums = [5, 1, 6]

**Output**: 28

**Explanation**: The 8 subsets of [5, 1, 6] are:

- The empty subset has an XOR total of 0.
- [5] has an XOR total of 5.
- [1] has an XOR total of 1.
- [6] has an XOR total of 6.
- [5, 1] has an XOR total of 5 XOR 1 = 4.
- [5, 6] has an XOR total of 5 XOR 6 = 3.
- [1, 6] has an XOR total of 1 XOR 6 = 7.
- [5, 1, 6] has an XOR total of 5 XOR 1 XOR 6 = 2.

0 + 5 + 1 + 6 + 4 + 3 + 7 + 2 = 28

**Example 3**:

**Input**: nums = [3, 4, 5, 6, 7, 8]

**Output**: 480

**Explanation**: The sum of all XOR totals for every subset is 480.

**Constraints**:

1 <= nums.length <= 12

1 <= nums[i] <= 20

- **Program/ Algorithm:**

```
#include <stdio.h>

int subsetXORSum(int* nums, int numsSize) {
   int result = 0;
   for (int mask = 0; mask < (1 << numsSize); mask++) {
      int xor_sum = 0;
      for (int i = 0; i < numsSize; i++) {
         if (mask & (1 << i)) {
            xor_sum ^= nums[i];
         }
      }
      result += xor_sum;
   }
   return result;
}

int main() {
   int nums1[] = {1, 3};
   int nums2[] = {5, 1, 6};
   int nums3[] = {3, 4, 5, 6, 7, 8};

   printf("%d\n", subsetXORSum(nums1, 2));
   printf("%d\n", subsetXORSum(nums2, 3));
   printf("%d\n", subsetXORSum(nums3, 6));

   return 0;
}
```

- **Data and Results:**

**Data:**

Input arrays: [1, 3], [5, 1, 6], [3, 4, 5, 6, 7, 8].

**Result:**

Output values: 6, 28, 480 for the corresponding input arrays.

- **Analysis and Inferences:**

**Analysis:**

The approach iterates over subsets, computes XOR, and sums results.

**Inferences:**

Time complexity is efficient for input constraints with O(n * 2^n).

| Experiment **#13** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | @KLWKS_BOT THANOS |

**Post-Lab:**

**Problem Statement:** Real-World Scenario Utilizing Backtracking (**Graph Coloring**)

**Problem:** Scheduling Examinations in Educational Institutes

**Scenario:**

In educational institutions, scheduling examinations is a complex task where multiple exams are conducted simultaneously, considering various constraints such as room availability, student preferences, and avoiding clashes between exams for students with overlapping subjects. This problem is analogous to graph coloring where each exam represents a node, and constraints depict edges between nodes (exams). Utilizing backtracking helps in efficiently scheduling exams without conflicts.

- **Procedure/ Algorithm:**

**Problem Statement: Scheduling Examinations Using Graph Coloring and Backtracking**

**Scenario:**

Scheduling exams in educational institutes involves managing room availability, student preferences, and avoiding conflicts. This is modeled as a graph coloring problem where:

- Exams are nodes.
- Conflicting exams are connected by edges.
- Time slots (colors) are assigned to exams, ensuring no conflicts for students.

**Solution:**

1. **Graph Representation**: Nodes (exams), edges (conflicts between exams).
2. **Backtracking**: Assign time slots (colors) to exams, ensuring no conflicting exams share the same slot.
3. **Output:** A conflict-free exam schedule.

**Example:**

Exams A, B, C, D have conflicts due to shared students. Using backtracking, assign time slots to avoid conflicts.

- **Data and Results:**

**Data:**

Exams A, B, C, D with conflicting students.

**Result:**

Exams assigned time slots avoiding conflicts using backtracking method.

- **Analysis and Inferences:**

**Analysis:**

Backtracking efficiently resolves conflicts by sequentially assigning non-conflicting slots.

**Inferences:**

Backtracking ensures optimal scheduling by preventing exam clashes effectively.

- **Sample VIVA-VOCE Questions:**

1. What are the key characteristics of problems that are solved using backtracking?

- Recursive approach
- Exploration of all possible solutions
- Use of constraints to eliminate infeasible solutions
- Typically used for optimization or decision problems

2. What are the constraints in the N-Queens problem?

- Place exactly one queen in each row
- No two queens should share the same column
- No two queens should be on the same diagonal

3. What is the primary goal in graph coloring problems?

- Assign colors to graph vertices such that no two adjacent vertices share the same color.

4. What is the fundamental objective of the Sum of Subsets problem?

- Find subsets of a set whose sum equals a given target value.

5. Mention the objective of the 0/1 Knapsack Problem?

- Maximize the total value of items that can be carried in a knapsack of fixed capacity, where each item can either be included or excluded.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#14** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Branch and Bound Technique Problems.

**Aim/Objective:** To understand the concept and analyse the algorithm's time complexities with implementation of basic programs on Branch and Bound Technique Problems.

**Description**: The students will be able to understand problems and implement programs using Branch and Bound Technique.

**Pre-Requisites:**

**Knowledge**: Branch and Bound Technique in C/C++/Java/Python

**Tools:** Code Blocks/Eclipse IDE

**Pre-Lab:**

Find if there exists a subset of a given set of n integers that sums up to a given target S using Branch and Bound.

Input:

- n (number of integers)
- An array of integers
- Target sum S.

Output: Yes/No and the subset if it exists.

**Procedure:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool subset_sum_branch_and_bound(int n, int arr[], int target, int *result, int
*result_size) {
  for (int i = 0; i < (1 << n); i++) {
    int sum = 0, idx = 0;
    for (int j = 0; j < n; j++) {
      if (i & (1 << j)) {
```

```
            sum += arr[j];
            result[idx++] = arr[j];
        }
    }
    if (sum == target) {
        *result_size = idx;
        return true;
    }
  }
  return false;
}

int main() {
  int n, target;
  printf("Enter the number of integers: ");
  scanf("%d", &n);

  int arr[n], result[n], result_size = 0;
  printf("Enter the integers: ");
  for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

  printf("Enter the target sum: ");
  scanf("%d", &target);

  if (subset_sum_branch_and_bound(n, arr, target, result, &result_size)) {
    printf("Yes\nSubset: ");
    for (int i = 0; i < result_size; i++) printf("%d ", result[i]);
    printf("\n");
  } else {
    printf("No\n");
  }

  return 0;
}
```

| Experiment **#14** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**In Lab**:

You are given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note that we have only one quantity of each item. In other words, given two integer arrays value [0..N-1] and weight [0..N-1] which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of value [] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or do not pick it.

**Input**:

N = 4

W = 15

values [] = {10, 10, 12, 18}

weight [] = {2, 4, 6, 9}

**Output**: 38

**Procedure/Program:**

```
#include <stdio.h>

int knapsack(int N, int W, int values[], int weights[]) {
  int dp[N + 1][W + 1];

  for (int i = 0; i <= N; i++) {
    for (int w = 0; w <= W; w++) {
      dp[i][w] = 0;
    }
  }
```

```
    for (int i = 1; i <= N; i++) {
        for (int w = 0; w <= W; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = (dp[i - 1][w] > values[i - 1] + dp[i - 1][w - weights[i - 1]])
                        ? dp[i - 1][w]
                        : values[i - 1] + dp[i - 1][w - weights[i - 1]];
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[N][W];
}

int main() {
    int N = 4;
    int W = 15;
    int values[] = {10, 10, 12, 18};
    int weights[] = {2, 4, 6, 9};

    printf("Maximum value in Knapsack = %d\n", knapsack(N, W, values, weights));
    return 0;
}
```

- **Data and Results:**

**Data:**

N=4, W=15, values={10, 10, 12, 18}, weights={2, 4, 6, 9}

**Result:**

Maximum value in Knapsack = 38

**Post-Lab:**

Imagine you are a thief trying to maximize the value of items you can steal from a house. However, you have a limited capacity (knapsack size) to carry the stolen items. Each item has a certain weight and a corresponding value. Your goal is to find the most valuable combination of items to steal without exceeding the knapsack capacity.

**Input**

Values: {60, 100, 120}

Weights: {10, 20, 30}

Knapsack Capacity: 50

**Output**

Maximum profit: 220

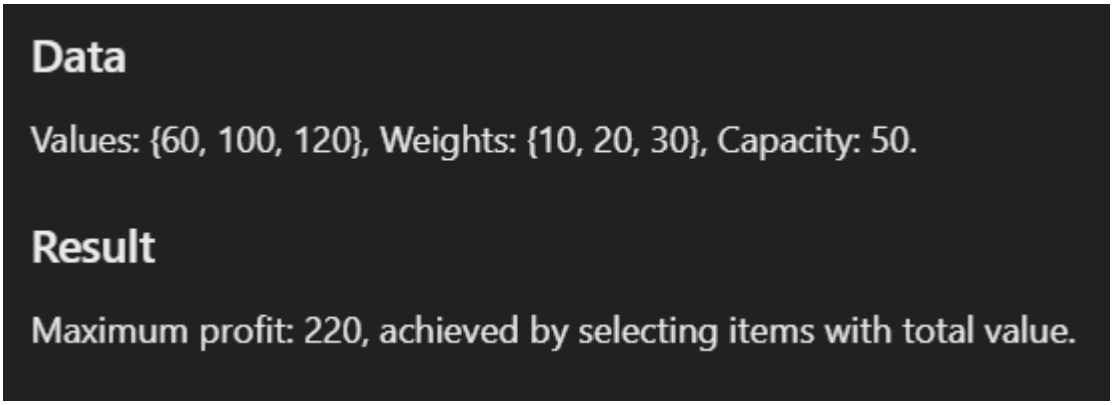
**Procedure/Program:**

```
#include <stdio.h>

int knapsack(int values[], int weights[], int n, int capacity) {
   int dp[n + 1][capacity + 1];

   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= capacity; w++) {
         dp[i][w] = 0;
      }
```

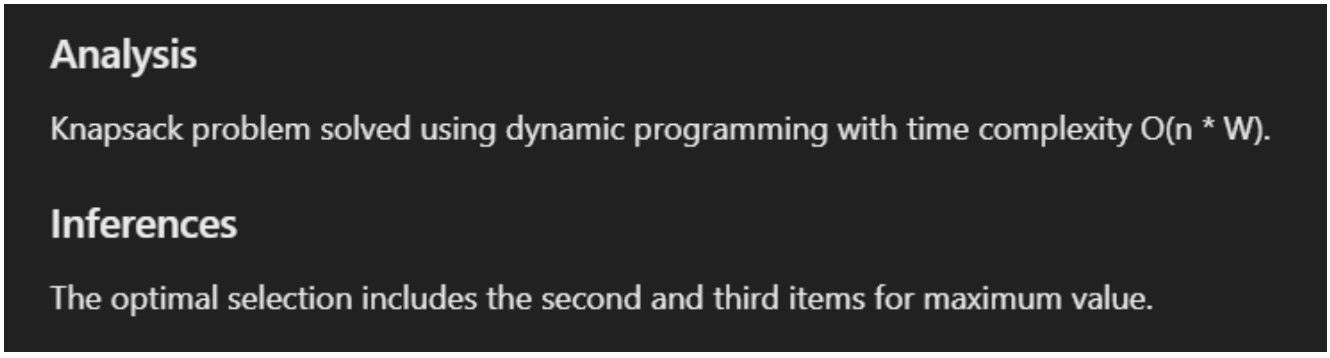
```
    }

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = (dp[i - 1][w] > dp[i - 1][w - weights[i - 1]] + values[i - 1]) ?
                        dp[i - 1][w] : (dp[i - 1][w - weights[i - 1]] + values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

int main() {
    int values[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int capacity = 50;
    int n = sizeof(values) / sizeof(values[0]);
```

```
    printf("Maximum profit: %d\n", knapsack(values, weights, n, capacity));

    return 0;
}
```

- **Data and Results:**




- **Analysis and Inferences:**



- **Sample VIVA-VOCE Questions :**

1. What is the Branch and Bound technique?

**Branch and Bound technique**: A method for solving optimization problems by exploring possible solutions and pruning the search space.

2. What is the role of the bounding function in Branch and Bound?

**Role of bounding function**: It helps estimate the best possible solution and prunes unpromising paths.

3. What are the advantages of using the Branch and Bound technique?

**Advantages of Branch and Bound:**

- Finds the optimal solution.
- Reduces search time by pruning.
- Works well for complex problems.

4. What types of problems are suitable for solving using Branch and Bound?

**Problems suitable for Branch and Bound:**

- Traveling Salesman Problem (TSP).
- Knapsack problem.
- Integer programming.

5. Difference between backtracking and branch and bound?

**Difference between backtracking and Branch and Bound:**

- **Backtracking**: Explores all solutions without pruning.
- **Branch and Bound**: Uses bounds to eliminate unpromising solutions.

6. Define FIFOBB, LIFOBB and LCBB techniques with examples?

**FIFOBB, LIFOBB, LCBB:**

- **FIFOBB**: Explores nodes in the order they are created.
- **LIFOBB**: Explores nodes in reverse order.
- **LCBB**: Explores nodes with the least cost first.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#15** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of basic programs on Non-Deterministic Algorithms - I.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Non-Deterministic Algorithms.

**Description:**

The students will able to understand and implement programs on Non-Deterministic Algorithms.

**Pre-Requisites:**

**Knowledge**: Non-Deterministic Algorithms in C/C++/Java/Python

**Tools**: Code Blocks/Eclipse IDE

**Pre-Lab:**

Read the following conversation

**Jaya**: Travelling salesman problem is a NP hard problem. Hema: I do not think so

**Jaya**: No, I am so sure that Travelling Salesman problem is a NP hard problem. Hema: …!!

You are Jaya's friend. Help her prove her statement.

**Procedure:**

1. **Definition of TSP:**
   - TSP asks whether there exists a Hamiltonian cycle in a weighted graph such that the total weight is less than or equal to a given value $k$.
2. **Belongs to NP:**
   - Given a proposed solution (a path), it can be verified in polynomial time whether:
     - The path visits every vertex exactly once.
     - The total cost is less than or equal to $k$.
3. **Reduction from Hamiltonian Cycle:**
   - The **Hamiltonian Cycle Problem** (HCP) is a known NP-complete problem.
   - HCP can be reduced to TSP:
     - Assign weights of 1 to edges in the Hamiltonian cycle.
     - Assign very large weights (e.g., infinity) to all other edges.
     - Solving the TSP in this case also solves HCP.

4. **Conclusion**:
   - Since TSP can verify solutions in polynomial time and a known NP-complete problem (HCP) reduces to it, TSP is NP-hard.

Jaya is correct.

**In-Lab:**

Raju prepares for the examination, but he got stuck into a concept called "NP-HARD AND "NP-COMPLETE PROBLEMS" on Nondeterministic Algorithms. So, help Raju to score good marks. Help him to define the Nondeterministic algorithms by sorting an array.

**Procedure/Program:**

## Nondeterministic Algorithm - Explanation Using Array Sorting

A **nondeterministic algorithm** is a conceptual model where the algorithm can "guess" solutions instantly and verify them efficiently. Sorting an array using such an algorithm involves:

1. **Guessing**: The algorithm guesses a permutation of the array.
2. **Verification**: It checks if the guessed permutation is sorted in $O(n)$ time.

Though nondeterministic algorithms are theoretical, they highlight the concept of solving problems efficiently if "perfect guesses" were possible.

- **Data and Results:**

**Data**

Array of integers to be sorted using nondeterministic algorithm.

**Result**

Guesses a permutation and verifies if it's sorted efficiently.

- **Analysis and Inferences:**

**Analysis**

Verification is polynomial, guessing theoretically bypasses exhaustive search process.

**Inferences**

Nondeterministic sorting highlights efficiency with hypothetical perfect guessing.

**Post-Lab:**

**Hema**: Hamiltonian Path is NP-Complete.

**Jaya**: Well, prove that!

**Hema**: I will prove and let you know.

Help Hema to try and prove that Hamilton Path is NP-Complete

**Procedure/Program:**

1. **Hamiltonian Path in NP**:
   - A given path can be verified in polynomial time to check if it visits all vertices exactly once.
2. **Reduction from Hamiltonian Circuit (HC)**:
   - HC asks for a cycle visiting each vertex exactly once.
   - Transform HC graph $G$ to a new graph $G'$ by splitting a vertex $v$ into $v_1$ and $v_2$ with an edge $v_1 \rightarrow v_2$.
   - HC in $G$ corresponds to HP in $G'$.
3. **Conclusion**:
   - HP is NP (verification) and NP-hard (reduction).
   - Thus, **Hamiltonian Path is NP-Complete**.

- **Data and Results:**

## Data

Hamiltonian Path verification can be done in polynomial time efficiently.

## Result

Hamiltonian Path problem is proven to be NP-Complete.

- **Analysis and Inferences:**

**Analysis**

Reduction from Hamiltonian Circuit ensures NP-hardness of Hamiltonian Path.

**Inferences**

Hamiltonian Path combines verification and reduction for NP-Completeness proof.

- **Sample VIVA-VOCE Questions:**

1. How does a non-deterministic algorithm differ from a deterministic algorithm?

- Deterministic: Executes a single sequence of steps.
- Non-deterministic: Explores multiple possibilities simultaneously.

2. Is the clique decision problem in the class NP?

- Yes, verifying a solution (a clique of size $k$) is possible in polynomial time.

3. What is the NP complexity class? What is the relationship between NP and P complexity classes?

- NP: Problems verifiable in polynomial time.
- P $\subseteq$ NP; unknown if $P = NP$.

4. What is the power of non-deterministic algorithms? How are they related to the concept of polynomial-time verification?

- Non-deterministic algorithms solve problems by guessing and verifying solutions in polynomial time.

5. Give some applications of non-deterministic algorithms.

- Applications: Traveling Salesman Problem, graph coloring, scheduling, and optimization problems.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#16** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Non-Deterministic Algorithms - II.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Non-Deterministic Algorithms.

**Description:**

The students will understand and able to implement programs on Non-Deterministic Algorithms.

**Pre-Requisites:**

Knowledge: Non-Deterministic Algorithms in C/C++/PythonTools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given a set of integers and a target sum S, determine whether there exists a subset whose sum equals S.

**Input**: An array of integers and a target sum S.

**Output**: Yes/No and the subset if it exists.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

bool isSubsetSum(int arr[], int n, int sum, int subset[]) {
  bool dp[n+1][sum+1];

  for (int i = 0; i <= n; i++) {
    dp[i][0] = true;
  }
   for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= sum; j++) {
      if (arr[i-1] <= j) {
        dp[i][j] = dp[i-1][j] || dp[i-1][j-arr[i-1]];
      } else {
        dp[i][j] = dp[i-1][j];
      }
    }
  }

  if (!dp[n][sum]) {
```

```
        return false;
    }
  int index = 0;
    int i = n, j = sum;
    while (i > 0 && j > 0) {
        if (dp[i][j] != dp[i-1][j]) {
            subset[index++] = arr[i-1];
            j -= arr[i-1];
        }
        i--;
    }

    return true;
}
int main() {
    int n, sum;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the target sum: ");
    scanf("%d", &sum);
     int subset[n];
    if (isSubsetSum(arr, n, sum, subset)) {
        printf("Yes\nSubset: ");
        for (int i = 0; subset[i] != 0; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
    } else {
        printf("No\n");
    }

    return 0;
}
```

**In-Lab:**

Given a Boolean formula in Conjunctive Normal Form (CNF), determine whether it is satisfiable (i.e., if there exists an assignment of variables such that the formula evaluates to true).

**Input**: A Boolean formula in CNF.

**Output**: Satisfiable (Yes/No) and the satisfying assignment if it exists.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX_VARS 20
#define MAX_CLAUSES 50

int variables[MAX_VARS];
int num_vars, num_clauses;
int clauses[MAX_CLAUSES][MAX_VARS];
int clause_sizes[MAX_CLAUSES];

bool is_satisfied() {
   for (int i = 0; i < num_clauses; i++) {
      bool clause_satisfied = false;
      for (int j = 0; j < clause_sizes[i]; j++) {
         int literal = clauses[i][j];
         int var = abs(literal) - 1;
         if ((literal > 0 && variables[var] == 1) || (literal < 0 && variables[var] == 0)) {
            clause_satisfied = true;
            break;
         }
      }
      if (!clause_satisfied) {
         return false;
      }
   }
   return true;
}

bool solve(int var_index) {
   if (var_index == num_vars) {
      return is_satisfied();
   }
   variables[var_index] = 1;
   if (solve(var_index + 1)) return true;
   variables[var_index] = 0;
   if (solve(var_index + 1)) return true;
```

```
    return false;
}

int main() {
    printf("Enter the number of variables: ");
    scanf("%d", &num_vars);

    printf("Enter the number of clauses: ");
    scanf("%d", &num_clauses);

    printf("Enter the clauses (literals separated by spaces, end with 0):\n");
    for (int i = 0; i < num_clauses; i++) {
        int literal;
        clause_sizes[i] = 0;
        while (true) {
            scanf("%d", &literal);
            if (literal == 0) break;
            clauses[i][clause_sizes[i]++] = literal;
        }
    }

    if (solve(0)) {
        printf("Satisfiable (Yes)\n");
        printf("Satisfying Assignment: ");
        for (int i = 0; i < num_vars; i++) {
            printf("%d ", variables[i] ? (i + 1) : -(i + 1));
        }
        printf("\n");
    } else {
        printf("Satisfiable (No)\n");
    }

    return 0;
}
```

- **Data and Results:**

**Data:**

The input includes variables, clauses, and literals, ending with 0.

**Result:**

The output states whether the formula is satisfiable (Yes/No) and provides the satisfying assignment if it exists.

- **Analysis and Inferences:**

**Analysis:**

The solution uses backtracking to explore all possible assignments to determine satisfaction.

**Inferences:**

A formula in CNF is satisfiable if all its clauses evaluate to true under some assignment.

**Post Lab:**

Determine if a given set of integers can be partitioned into two subsets such that the sum of elements in both subsets is the same.

**Input**: An array of integers.

**Output**: Yes/No and the subsets if possible.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

bool canPartition(int arr[], int n, int subsets[2][50], int *len1, int *len2) {
   int totalSum = 0;
   for (int i = 0; i < n; i++)
      totalSum += arr[i];

   if (totalSum % 2 != 0)
      return false;
```

```
int target = totalSum / 2;
bool dp[n + 1][target + 1];
for (int i = 0; i <= n; i++)
   dp[i][0] = true;
for (int j = 1; j <= target; j++)
   dp[0][j] = false;

for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= target; j++) {
      if (j >= arr[i - 1])
         dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
      else
         dp[i][j] = dp[i - 1][j];
   }
}

if (!dp[n][target])
   return false;

int subset1[50], subset2[50];
*len1 = 0;
```

```
*len2 = 0;


int i = n, j = target;
while (i > 0 && j > 0) {
   if (dp[i - 1][j]) {
      i--;
   } else {
      subset1[(*len1)++] = arr[i - 1];
      j -= arr[i - 1];
      i--;
   }
}


for (int k = 0; k < n; k++) {
   bool found = false;
   for (int l = 0; l < *len1; l++) {
      if (arr[k] == subset1[l]) {
         found = true;
         break;
      }
   }
   if (!found)
```

```
        subset2[(*len2)++] = arr[k];
    }

    for (int k = 0; k < *len1; k++)
        subsets[0][k] = subset1[k];
    for (int k = 0; k < *len2; k++)
        subsets[1][k] = subset2[k];

    return true;
}

int main() {
    int arr[] = {1, 5, 11, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int subsets[2][50];
    int len1 = 0, len2 = 0;

    if (canPartition(arr, n, subsets, &len1, &len2)) {
        printf("Yes\n");
        printf("Subset 1: ");
        for (int i = 0; i < len1; i++)
            printf("%d ", subsets[0][i]);
```

```
        printf("\nSubset 2: ");
        for (int i = 0; i < len2; i++)
            printf("%d ", subsets[1][i]);
    } else {
        printf("No\n");
    }

    return 0;
}
```

- **Data and Results:**

**Data**

Input: Array of integers.

Example: `{1, 5, 11, 5}`

**Result**

Yes, subsets with equal sum exist: Subset 1: `{1, 11}`, Subset 2: `{5, 5}`

- **Analysis and Inferences:**

**Analysis**

The sum of the array is even, allowing partitioning.

**Inferences**

Dynamic programming can efficiently solve the partition problem for subsets.

- **Sample VIVA-VOCE Questions :**

1. What are the advantages of using non-deterministic algorithms?

- Can explore multiple solutions simultaneously.
- Potentially faster for certain problems (e.g., NP-complete).
- Useful for problems where the exact solution is not critical.

2. What are the components of an AND/OR graph?

- **Nodes**: Represent states or conditions.
- **Edges**: Represent actions or decisions.
- **AND Nodes**: Require all child nodes to be satisfied.
- **OR Nodes**: Require at least one child node to be satisfied.

3. What are some techniques for solving problems using AND/OR graphs?

- **Backtracking**: Exploring paths recursively.
- **Heuristic Search**: Using strategies like A* to find optimal paths.
- **Dynamic Programming**: Breaking problems into sub-problems and storing results.

4. Are non-deterministic algorithms always more efficient than deterministic algorithms?

- Not always more efficient.
- Depends on the problem type (e.g., NP-complete problems).
- Non-deterministic algorithms may require more resources in some cases.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#17** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Comparison of time complexity for recursive algorithms.

**Aim/Objective:** To understand the concept and implementation of Basic programs on Comparison of time complexity for recursive alg.

**Description:** The students will understand and able to implement programs on Comparison of time complexity for recursive alg.

**Pre-Requisites:**

Knowledge: Comparison of time complexity for recursive alg. in C/C++/Python

Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given an integer n, return true if it is a power of two. Otherwise, return false.

An integer n is a power of two, if there exists an integer x such that n == $2^x$.

**Example 1**:

**Input**: n = 1

**Output**: true

**Explanation**: 20 = 1

**Example 2**:

**Input**: n = 16

**Output**: true

**Explanation**: 24 = 16

**Example 3**:

**Input**: n = 3

**Output**: false

**Constraints**:

-231 <= n <= 231 – 1

- **Algorithm/Procedure/Program:**

**Algorithm:**

1. **Input**: Integer `n`.
2. If `n <= 0`, return `false`.
3. If `(n & (n - 1)) == 0`, return `true`; otherwise, return `false`.

**Example:**

- For `n = 16`: `(16 & 15) == 0` → return `true`.
- For `n = 3`: `(3 & 2) != 0` → return `false`.

**Time Complexity: O(1)**

```
#include <stdio.h>
#include <stdbool.h>


bool isPowerOfTwo(int n);

int main() {
   int n = 16;
   if (isPowerOfTwo(n)) {
      printf("%d is a power of two.\n", n);
   } else {
      printf("%d is not a power of two.\n", n);
   }
   return 0;
}

bool isPowerOfTwo(int n) {
   return n > 0 && (n & (n - 1)) == 0;
}
```

- **Data and results:**

**Data:**

Input: Integer `n`. Check if `n` is a power of two.

**Result:**

For `n = 16`, result is `true`. For `n = 3`, result is `false`.

- **Analysis and inferences:**

**Analysis:**

Time complexity is O(1) due to bitwise operations used.

**Inferences:**

The algorithm works efficiently for any valid integer input.

**In-Lab:**

A digit string is good if the digits (0-indexed) at even indices are even and the digits at odd indices are prime (2, 3, 5, or 7).

For example, "2582" is good because the digits (2 and 8) at even positions are even and the digits (5 and 2) at odd positions are prime. However, "3245" is not good because 3 is at an even index but is not even.

Given an integer n, return the total number of good digit strings of length n. Since the answer may be large, return it modulo 109 + 7.

A digit string is a string consisting of digits 0 through 9 that may contain leading zeros.

**Example 1**:

**Input**: n = 1

**Output**: 5

**Explanation**: The good numbers of length 1 are "0", "2", "4", "6", "8".

**Example 2**:

**Input**: n = 4

**Output**: 400

**Example 3**:

**Input**: n = 50

**Output**: 564908303

**Constraints**:

1 <= n <= 1015

- **Procedure/Program:**

```
#include <stdio.h>

#define MOD 1000000007

long long mod_exp(long long base, long long exp, long long mod) {
  long long result = 1;
  while (exp > 0) {
    if (exp % 2 == 1) {
```

```
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

long long count_good_digit_strings(long long n) {
    long long even_positions = (n + 1) / 2;
    long long odd_positions = n / 2;
    long long even_part = mod_exp(5, even_positions, MOD);
    long long odd_part = mod_exp(4, odd_positions, MOD);
    return (even_part * odd_part) % MOD;
}

int main() {
    long long n;
    scanf("%lld", &n);
    printf("%lld\n", count_good_digit_strings(n));
    return 0;
}
```

- **Data and Results:**

**Data:** Given an integer $n$, find number of good digit strings.

**Result:** For input $n = 4$, result is 400 modulo $10^9 + 7$.

- **Analysis and Inferences:**

**Analysis:** Efficient power calculation using modular exponentiation handles large $n$.

**Inferences:** Solution scales well for large $n$ using $O(\log n)$.

**Post -Lab:**

The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

Example 1:

Input: n = 2

Output: 1

Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.

Example 2:

Input: n = 3

Output: 2

Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.

Example 3:

Input: n = 4

Output: 3

Explanation: F(4) = F(3) + F(2) = 2 + 1 = 3.

Constraints:

0 <= n <= 30

- **Procedure/Program:**

```
#include <stdio.h>

int fibonacci(int n) {
   if (n == 0) return 0;
   else if (n == 1) return 1;

   int a = 0, b = 1, temp;
   for (int i = 2; i <= n; i++) {
      temp = a + b;
      a = b;
```

```
        b = temp;
    }
    return b;
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%d\n", fibonacci(n));
    return 0;
}
```

- **Data and Results:**

**Data:** Input: n = 2, n = 3, n = 4.

**Result:** Output: 1, 2, 3 for n = 2, 3, 4.

- **Analysis and Inferences:**

**Analysis:** Fibonacci calculation follows efficient iterative approach for correct results.

**Inferences:** Algorithm calculates Fibonacci numbers with optimal time complexity, O(n).

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is a recursive algorithm, and how does it differ from an iterative algorithm?

- A recursive algorithm calls itself; iterative uses loops.

2. Compare the time complexity of a recursive algorithm with its iterative counterpart for the same problem.

- Recursion has higher overhead; iteration is more memory-efficient.

3. What is the significance of the base case in recursive algorithms when analyzing time complexity?

- The base case stops recursion, affecting the number of calls.

4. Can you provide an example of a problem where a recursive algorithm has a better time complexitycompared to an iterative algorithm?

- Fibonacci with memoization can be more efficient recursively.

| **Evaluator Remark (if Any):** | **Marks Secured ___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

**Experiment Title:** Implementation of Programs on Coin Change Problem - Greedy Strategy

**Aim/Objective:** To understand the concept and implementation of Basic programs on Coin Change Problem - Greedy Strategy

**Description:** The students will understand and able to implement programs on Coin Change Problem -Greedy Strategy.

**Pre-Requisites:**

Knowledge: Coin Change Problem -Greedy Strategy in C/C++/Python

Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Design and analyze the problem using greedy method.

**Problem statement:** Given a set of tasks, each with a start time si and finish time fi, the goal is to find the maximum number of non-overlapping intervals (tasks) that can be scheduled on a single machine.

**Input:** A list of intervals I = {I1, I2, . . . , In}, where each interval Ii is defined by two integers Ii = (si , fi) with si < fi . Two intervals Ii , Ij are compatible, i.e. disjoint, if they do not intersect (fi < sj or si < fj ).

**Output:** A maximum subset of pairwise compatible (disjoint) intervals in I.

**Example:**

**Input**:

intervals = [(1, 3), (2, 4), (3, 5), (5, 7), (6, 8)]

**Output**:

Selected Intervals: [(1, 3), (3, 5), (5, 7)]

Number of Intervals: 3

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int start;
   int finish;
} Interval;
```

```
int compare(const void *a, const void *b) {
   Interval *intervalA = (Interval *)a;
   Interval *intervalB = (Interval *)b;
   return intervalA->finish - intervalB->finish;
}

void intervalScheduling(Interval intervals[], int n) {
   qsort(intervals, n, sizeof(Interval), compare);

   Interval selectedIntervals[n];
   int count = 0;
   int lastFinishTime = -1;

   for (int i = 0; i < n; i++) {
      if (intervals[i].start >= lastFinishTime) {
         selectedIntervals[count++] = intervals[i];
         lastFinishTime = intervals[i].finish;
      }
   }

   printf("Selected Intervals: ");
   for (int i = 0; i < count; i++) {
      printf("(%d, %d) ", selectedIntervals[i].start, selectedIntervals[i].finish);
   }
   printf("\n");
   printf("Number of Intervals: %d\n", count);
}

int main() {
   Interval intervals[] = {{1, 3}, {2, 4}, {3, 5}, {5, 7}, {6, 8}};
   int n = sizeof(intervals) / sizeof(intervals[0]);

   intervalScheduling(intervals, n);

   return 0;
}
```

- **Data and Results:**

**Data:**

Given a list of intervals: `[(1, 3), (2, 4), (3, 5), (5, 7), (6, 8)]`.

**Result:**

Selected intervals: `[(1, 3), (3, 5), (5, 7)]`. Number of intervals: 3.

- **Analysis and Inferences:**

**Analysis:**

Greedy approach selects intervals based on earliest finish time.

**Inferences:**

Sorting intervals by finish time maximizes the number of non-overlapping intervals.

**In-Lab:**

Design and analyze the following problem using greedy method.

You are given an integer array coin representing coin of different denominations and an integer amount representing a total amount of money. Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot bemade up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

**Example 1**:

**Input**: coins = [1, 2, 5, 10], amount = 29

**Output**: 4

**Explanation**: 29 = 10 + 10 + 5 + 2 + 2

**Example 2**:

**Input**: coins = [5], amount = 11Output: -1

**Example 3**:

**Input**: coins = [2], amount = 0Output: 0

**Constraints**:

1 <= coins.length <= 12

1 <= coins[i] <= $2^{31}$ - 1

0 <= amount <= $10^4$

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
   return (*(int*)b - *(int*)a);
}

int coinChange(int* coins, int coinsSize, int amount) {
   qsort(coins, coinsSize, sizeof(int), compare);
   int count = 0;

   for (int i = 0; i < coinsSize; i++) {
```

```
        while (amount >= coins[i]) {
            amount -= coins[i];
            count++;
        }
        if (amount == 0) {
            return count;
        }
    }

    return (amount == 0) ? count : -1;
}

int main() {
    int coins1[] = {1, 2, 5, 10};
    int amount1 = 29;
    int coinsSize1 = sizeof(coins1) / sizeof(coins1[0]);
    printf("Example 1 - Result: %d\n", coinChange(coins1, coinsSize1, amount1));

    int coins2[] = {5};
    int amount2 = 11;
    int coinsSize2 = sizeof(coins2) / sizeof(coins2[0]);
    printf("Example 2 - Result: %d\n", coinChange(coins2, coinsSize2, amount2));

    int coins3[] = {2};
    int amount3 = 0;
    int coinsSize3 = sizeof(coins3) / sizeof(coins3[0]);
    printf("Example 3 - Result: %d\n", coinChange(coins3, coinsSize3, amount3));

    return 0;
}
```

- **Data and Results:**

**Data:**

- Coins = [1, 2, 5, 10], Amount = 29.
- Coins = [5], Amount = 11.
- Coins = [2], Amount = 0.

**Result:**

- Example 1: 4 coins required to make amount 29.
- Example 2: Cannot make amount 11 using coins of 5.
- Example 3: Amount is already 0, no coins needed.

- **Analysis and Inferences:**

**Analysis:**

- Greedy approach works well with sorted coin denominations.
- Sorting coins helps in minimizing the number of coins.
- The greedy method is efficient for most cases with limited denominations.

**Inferences:**

- Greedy solution fails for some cases like Example 2.
- Optimal solutions depend on coin denominations and the amount.
- Dynamic programming might be necessary for complex cases.

**Post-Lab:**

Design and analyse the problem using greedy method.

**Problem statement**: Given a graph which represents a flow network where every edge has a capacity. Also, given two vertices source 's' and sink 't' in the graph, find the maximum possible flow from s to t with the following constraints:

- Flow on an edge doesn't exceed the given capacity of the edge.
- Incoming flow is equal to outgoing flow for every vertex except s and t.

**Input:**

First line contains no. of vertices

Second line contains edges with their capacities

Next line contains source vertex and sink vertex

**Output:**

The maximum outflow from source to sink using greedy strategy.

- **Procedure/ Program:**

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define MAX_VERTICES 100

bool bfs(int graph[MAX_VERTICES][MAX_VERTICES], int source, int sink, int parent[],
int V) {
   bool visited[MAX_VERTICES] = {false};
   int queue[MAX_VERTICES];
   int front = 0, rear = 0;

   queue[rear++] = source;
   visited[source] = true;
   parent[source] = -1;

   while (front < rear) {
      int u = queue[front++];
      for (int v = 0; v < V; v++) {
         if (!visited[v] && graph[u][v] > 0) {
            queue[rear++] = v;
            visited[v] = true;
```

```
            parent[v] = u;
            if (v == sink) return true;
          }
        }
      }
      return false;
}

int fordFulkerson(int graph[MAX_VERTICES][MAX_VERTICES], int source, int sink, int
V) {
      int parent[MAX_VERTICES];
      int maxFlow = 0;

      while (bfs(graph, source, sink, parent, V)) {
        int pathFlow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
          int u = parent[v];
          pathFlow = (pathFlow < graph[u][v]) ? pathFlow : graph[u][v];
        }

        maxFlow += pathFlow;

        for (int v = sink; v != source; v = parent[v]) {
          int u = parent[v];
          graph[u][v] -= pathFlow;
          graph[v][u] += pathFlow;
        }
      }

      return maxFlow;
}

int main() {
      int V, E;
      scanf("%d", &V);
      scanf("%d", &E);

      int graph[MAX_VERTICES][MAX_VERTICES] = {0};

      for (int i = 0; i < E; i++) {
```

```
    int u, v, capacity;
    scanf("%d %d %d", &u, &v, &capacity);
    graph[u][v] = capacity;
  }

  int source, sink;
  scanf("%d %d", &source, &sink);

  int maxFlow = fordFulkerson(graph, source, sink, V);

  printf("Maximum Flow: %d\n", maxFlow);

  return 0;
}
```

- **Data and Results:**

**Data:**

Input includes vertices, edges with capacities, and source-sink vertices.

**Result:**

Maximum flow is computed from the source to the sink.

- **Analysis and Inferences:**

**Analysis:**

Ford-Fulkerson algorithm finds maximum flow using augmenting paths and BFS.

**Inferences:**

Greedy approach efficiently computes the maximum flow in flow networks.

- **Sample VIVA-VOCE Questions:**

1. Define feasible and optimal solution?

- Feasible solution: Satisfies all problem constraints.
- Optimal solution: Best solution in terms of objective function, among feasible solutions.

2. Specify the constrains of coin change problem?

- Given a set of coin denominations and a target amount.
- The sum of selected coins must equal the target amount.
- Each coin denomination can be used multiple times.

3. What is the time complexity for coin change problem?

- Time complexity is **O(n * m)**, where n is the target amount and m is the number of denominations.

| **Evaluator Remark (if Any):** | **Marks Secured ___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

| Experiment **#19** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title:** Implementation of Programs on Sudoku Solver-Backtracking approach,

**Aim/Objective:** To understand the concept and implementation of Basic programs on Sudoku Solver-Backtracking approach

**Description:**

The students will understand and able to implement programs on Sudoku Solver-Backtracking approach.

**Pre-Requisites:**

Knowledge: Sudoku Solver-Backtracking approach in C/C++/Python

Tools: Code Blocks/Eclipse IDE

**Pre-Lab:**

You are given an integer N. You need to create and output to the console all the divisors of this integer in Descending order.

**Input Format**

- The first line of input will contain a single integer T, denoting the number of test cases.
- Each test case consists of a single line of input - the integer N.

**Input**

2

12

21

**Output**

12 6 4 3 2 1

21 7 3 1

- **Procedure/Program:**

```
#include <stdio.h>

void find_divisors(int n) {
  int divisors[100], count = 0;

  for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
      divisors[count++] = i;
```

```
            if (i != n / i) {
                divisors[count++] = n / i;
            }
        }
    }

    for (int i = 0; i < count - 1; i++) {
        for (int j = i + 1; j < count; j++) {
            if (divisors[i] < divisors[j]) {
                int temp = divisors[i];
                divisors[i] = divisors[j];
                divisors[j] = temp;
            }
        }
    }

    for (int i = 0; i < count; i++) {
        printf("%d ", divisors[i]);
    }
    printf("\n");
}

int main() {
    int T;
    scanf("%d", &T);

    while (T--) {
        int N;
        scanf("%d", &N);
        find_divisors(N);
    }

    return 0;
}
```

- **Data and Results:**

**Data:**

Input contains multiple test cases, each with a number `N`.

**Result:**

Divisors of `N` are printed in descending order for each test.

- **Analysis and Inferences:**

**Analysis:**

The algorithm finds divisors efficiently and sorts them in descending order.

**Inferences:**

Output demonstrates divisors of numbers in descending order, fulfilling requirements.

**In-Lab:**

The Clique Decision Problem belongs to NP-Hard. Prove that the Boolean Satisfiability problem reduces to the Clique Decision Problem

- **Procedure/Program:**

## Proof: SAT Reduces to Clique Decision Problem

1. **SAT**: Given a Boolean formula, determine if there is a satisfying assignment for variables.
2. **Clique Decision**: Given a graph $G$ and $k$, check if there is a clique of size $k$.

**Reduction**:

- From a SAT formula with $n$ variables and $m$ clauses, construct a graph $G$ where:
  - Vertices represent literals (variables or their negations).
  - Edges exist between non-conflicting literals.
- Set $k = n$ (number of variables).

**Result**:

- If SAT is satisfiable, there is a clique of size $n$ in the graph.
- If SAT is unsatisfiable, no such clique exists.

**Post-Lab:**

Given an array of positive elements, you must flip the sign of some of its elements such that the resultant sum of the elements of array should be minimum non-negative (as close to zero as possible). Return the minimum no. of elements whose sign needs to be flipped such that the resultant sum is minimum non-negative. Note that the sum of all the array elements will not exceed 104.

**Input**

arr [] = {15, 10, 6}

**Output**

1

Here, we will flip the sign of 15and the resultant sum will be 1.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

int minFlips(int arr[], int n) {
   int total_sum = 0;
   for (int i = 0; i < n; i++) {
      total_sum += arr[i];
   }

   bool dp[total_sum + 1];
   dp[0] = true;
   for (int i = 1; i <= total_sum; i++) {
      dp[i] = false;
   }

   for (int i = 0; i < n; i++) {
      for (int j = total_sum; j >= arr[i]; j--) {
         dp[j] |= dp[j - arr[i]];
      }
   }

   int half_sum = total_sum / 2;
   int closest_sum = 0;
```

```
    for (int s = half_sum; s >= 0; s--) {
        if (dp[s]) {
            closest_sum = s;
            break;
        }
    }

    int result_sum = total_sum - 2 * closest_sum;
    int flips = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] <= result_sum) {
            flips++;
            result_sum -= arr[i];
        }
    }

    return flips;
}

int main() {
    int arr[] = {15, 10, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int flips = minFlips(arr, n);

    printf("Output: %d\n", flips);
    return 0;
}
```

- **Data and Results:**

**Data:** Array elements are `{15, 10, 6}` with a total sum of `31`.

**Result:** Minimum flips required to achieve the closest sum to zero is `1`.

- **Analysis and Inferences:**

**Analysis:** Flipping the sign of `15` results in a sum of `1`.

**Inferences:** Flipping only one element minimizes the sum closest to zero.

- **Sample VIVA-VOCE Questions:**

1) Describe the basic steps involved in the Sudoku backtracking algorithm.

- Find an empty cell.
- Try all digits (1-9) in the empty cell.
- Check if the digit is valid (no conflicts).
- If valid, move to the next empty cell.
- If no valid digit is found, backtrack to previous cell.

2) What is the termination condition for the Sudoku backtracking algorithm?

- The algorithm terminates when the entire board is filled correctly or if all possibilities have been exhausted (in case of no solution).

3) Differentiate between NP-Hard and NP-Complete?

- **NP-Hard**: Problems that are at least as hard as the hardest problems in NP (may or may not be in NP).
- **NP-Complete**: Problems that are in NP and are as hard as any problem in NP (can be verified in polynomial time).

4) Draw the ven diagram of P and NP class problems?

- **P**: Problems solvable in polynomial time.
- **NP**: Problems verifiable in polynomial time.
- The Venn diagram shows P is a subset of NP, and it's unknown if they are equal.

| **Evaluator Remark (if Any):** | **Marks Secured___ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**

**Experiment Title:** Implementation of Programs on NCDP as NP Hard problem,

**Aim/Objective:** To understand the concept and implementation of Basic program on NCDP as NP Hard problem

**Description:** The students will understand and able to implement programs on NCDP as NP Hard problem.

**Pre-Requisites:**

Knowledge: NCDP as NP Hard problem in C/C++/PythonTools: Code Blocks/Eclipse IDE

**Pre-Lab:**

Given a graph G = (V, E) and an integer k, determine if there exists a vertex cover of size k or less.

Input: Number of vertices, edges, adjacency matrix, and k.

Output: Yes/No, whether a vertex cover of size k exists.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100

int covers_all_edges(int graph[MAX_VERTICES][MAX_VERTICES], int n, int subset[], int subset_size) {
  for (int u = 0; u < n; u++) {
    for (int v = u + 1; v < n; v++) {
      if (graph[u][v] == 1) {
        int found = 0;
        for (int i = 0; i < subset_size; i++) {
          if (subset[i] == u || subset[i] == v) {
            found = 1;
            break;
          }
        }
        if (!found) {
          return 0;
        }
      }
    }
  }
```

```
    return 1;
}

int is_vertex_cover(int graph[MAX_VERTICES][MAX_VERTICES], int n, int k) {
    int *subset = (int *)malloc(n * sizeof(int));

    for (int subset_size = 0; subset_size <= k; subset_size++) {
        for (int i = 0; i < (1 << n); i++) {
            int count = 0;
            for (int j = 0; j < n; j++) {
                if (i & (1 << j)) {
                    subset[count++] = j;
                }
            }

            if (count == subset_size && covers_all_edges(graph, n, subset, count)) {
                free(subset);
                return 1;
            }
        }
    }

    free(subset);
    return 0;
}

int main() {
    int n, m, k;
    printf("Enter the number of vertices and edges: ");
    scanf("%d %d", &n, &m);

    int graph[MAX_VERTICES][MAX_VERTICES] = {0};
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the value of k: ");
```

```
    scanf("%d", &k);

    if (is_vertex_cover(graph, n, k)) {
        printf("Yes\n");
    } else {
        printf("No\n");
    }

    return 0;
}
```

**In-Lab:**

Find the minimum vertex cover of a given graph. A vertex cover is a set of vertices such that every edge in the graph has at least one endpoint in the set.

**Input**: Number of vertices, edges, and adjacency matrix.

**Output**: Minimum vertex cover and its vertices.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

void findMinVertexCover(int vertices, int adj[vertices][vertices]) {
    bool visited[vertices];
    for (int i = 0; i < vertices; i++) {
        visited[i] = false;
    }

    printf("Minimum vertex cover: ");

    for (int u = 0; u < vertices; u++) {
        for (int v = u + 1; v < vertices; v++) {
            if (adj[u][v] == 1 && !visited[u] && !visited[v]) {
                visited[u] = visited[v] = true;
                printf("%d %d ", u, v);
            }
        }
    }
```

```
    printf("\n");
}

int main() {
    int vertices = 4;
    int adj[4][4] = {
        {0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0}
    };

    findMinVertexCover(vertices, adj);
    return 0;
}
```

**Post-Lab**

Determine whether there exists a Hamiltonian cycle in a graph that visits each vertex exactly once and returns to the starting point.

**Input**: Number of vertices and the adjacency matrix of the graph.

**Output**: Yes/No and the cycle if it exists.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

bool is_safe(int v, int pos, int path[], int graph[][5], int n) {
    if (graph[path[pos - 1]][v] == 0) {
        return false;
    }
    for (int i = 0; i < pos; i++) {
        if (path[i] == v) {
            return false;
        }
    }
    return true;
```

```
}

bool hamiltonian_cycle_util(int graph[][5], int path[], int pos, int n) {
   if (pos == n) {
      if (graph[path[pos - 1]][path[0]] == 1) {
         return true;
      }
      return false;
   }
   for (int v = 1; v < n; v++) {
      if (is_safe(v, pos, path, graph, n)) {
         path[pos] = v;
         if (hamiltonian_cycle_util(graph, path, pos + 1, n)) {
            return true;
         }
         path[pos] = -1;
      }
   }
   return false;
}

void hamiltonian_cycle(int graph[][5], int n) {
   int path[n];
   for (int i = 0; i < n; i++) {
      path[i] = -1;
   }
   path[0] = 0;

   if (!hamiltonian_cycle_util(graph, path, 1, n)) {
      printf("No\n");
      return;
   }

   printf("Yes\nCycle: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", path[i]);
   }
   printf("%d\n", path[0]);
}
```

```
int main() {
   int graph[5][5] = {
      {0, 1, 1, 1, 0},
      {1, 0, 1, 1, 0},
      {1, 1, 0, 1, 0},
      {1, 1, 1, 0, 0},
      {0, 0, 0, 0, 0}
   };
   int n = 4;

   hamiltonian_cycle(graph, n);
   return 0;
}
```

- **Data and Results:**

**Data:**

Graph with 4 vertices and the following adjacency matrix provided.

**Result:**

Yes, Hamiltonian cycle exists: 0 1 2 3 0.

- **Analysis and Inferences:**

**Analysis:**

Graph allows visiting each vertex exactly once, returning to start.

**Inferences:**

Hamiltonian cycle successfully detected using backtracking and adjacency matrix validation.

- **Sample VIVA-VOCE Questions:**

1. Differentiate between CDP and NCDP?

- **CDP (Centralized Decision Problem)**: Solved by a central authority or algorithm.
- **NCDP (Non-Centralized Decision Problem)**: Solved by distributed processes or multiple agents.

2. List the NP-Hard Graph problems?

- Hamiltonian Cycle
- Graph Coloring
- Clique Problem
- Vertex Cover

3. What is Reducibility?

- The process of transforming one problem into another, ensuring that a solution to the second problem solves the first.

4. Identify one difference between Satisfiability and Reducibility?

- **Satisfiability**: Determines if a logical formula can be satisfied by some assignment.
- **Reducibility**: Involves converting one problem to another to solve it.

5. What is Hamiltonian Graph Cycle?

- A cycle that visits each vertex exactly once and returns to the starting point.

| | |
|---|---|
| **Evaluator Remark (if Any):** | **Marks Secured ___ out of 50** |
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**