DESIGN AND ANALYSIS OF ALGORITHMS

NAME : MALREDDY.P

REGISTER NO: 192372015

DAY-1 PROGRAMMES

1)Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward. Example 1: Input: words = ["abc","car","ada","racecar","cool"]

Output: "ada"

Explanation: The first string that is palindromic is "ada".

Note that "racecar" is also palindromic, but it is not the first.

Example 2: Input: words = ["notapalindrome","racecar"]

Output: "racecar"

Explanation: The first and only string that is palindromic is "racecar.

```
#include <stdio.h>
#include <string.h>
int isPalindrome(char *str) {
   int len = strlen(str);
   for (int i = 0; i < len / 2; i++) {
      if (str[i] != str[len - i - 1]) {
         return 0;
      }
   }
   return 1;
}
char* findPalindromicString(char **words, int wordsSize) {
   for (int i = 0; i < wordsSize; i++) {
      if (isPalindrome(words[i])) {
         return words[i];
      }
   }
   return "";
}
int main() {
```

```
    char *words1[] = {"abc", "car", "ada", "racecar", "cool"};
    char *words2[] = {"notapalindrome", "racecar"};
    char *result1 = findPalindromicString(words1, 5);
    char *result2 = findPalindromicString(words2, 2);
    printf("Output 1: %s\n", result1);
    printf("Output 2: %s\n", result2);
    return 0;
}
```

Sample output:

Output 1: ada

Output 2: racecar

2)You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1 Return [answer1,answer2].

Example 1: Input: nums1 = [2,3,2], nums2 = [1,2] Output: [2,1] :

Example 2: Input: nums1 = [4,3,2,3,1], nums2 = [2,2,5,2,3,6]

Output: [3,4]

Explanation: The elements at indices 1, 2, and 3 in nums1 exist in nums2 as well. So answer1 is 3.

The elements at indices 0, 1, 3, and 4 in nums2 exist in nums1. So answer2 is 4.

```
#include <stdio.h>
int existsInArray(int arr[], int size, int element) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == element) {
            return 1;
        }
    }
    return 0;
}
```

```
void calculateAnswers(int nums1[], int n, int nums2[], int m, int result[]) {
    int answer1 = 0, answer2 = 0;
    for (int i = 0; i < n; i++) {
        if (existsInArray(nums2, m, nums1[i])) {
            answer1++;
        }
    }
    for (int i = 0; i < m; i++) {
        if (existsInArray(nums1, n, nums2[i])) {
            answer2++;
        }
    }
    result[0] = answer1;
    result[1] = answer2;
}
int main() {
    int nums1_1[] = {2, 3, 2};
    int nums2_1[] = {1, 2};
    int n1 = sizeof(nums1_1) / sizeof(nums1_1[0]);
    int m1 = sizeof(nums2_1) / sizeof(nums2_1[0]);
    int result1[2];
    calculateAnswers(nums1_1, n1, nums2_1, m1, result1);
    printf("Output: [%d, %d]\n", result1[0], result1[1]);
    return 0;
}
```

Sample output:

Output: [2, 1]

3) You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j <

nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array. Example 1: Input: nums = [1,2,1]

Output: 15

Explanation: Six possible subarrays are: [1]: 1 distinct value [2]: 1 distinct value [1]: 1 distinct value [1,2]: 2 distinct values [2,1]: 2 distinct values [1,2,1]: 2 distinct values The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 + 22 + 22 + 22 = 15.

Example 2: Input: nums = [1,1]

Output: 3

Explanation: Three possible subarrays are: [1]: 1 distinct value [1]: 1 distinct value [1,1]: 1 distinct value The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 = 3.

```
#include <stdio.h>
#include <stdlib.h>
int countDistinct(int *arr, int start, int end) {
   int count = 0;
   int found[1001] = {0};
   for (int i = start; i <= end; i++) {
      if (found[arr[i]] == 0) {
         count++;
         found[arr[i]] = 1;
      }
   }
   return count;
}
int sumOfSquaresOfDistinctCounts(int *nums, int numsSize) {
   int sum = 0;
```

```
    for (int i = 0; i < numsSize; i++) {
        for (int j = i; j < numsSize; j++) {
            int distinctCount = countDistinct(nums, i, j);
            sum += distinctCount * distinctCount;
        }
    }
    return sum;
}
int main() {
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int *nums = (int *)malloc(n * sizeof(int));
    printf("Enter the elements of the array: \n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &nums[i]);
    }
    int result = sumOfSquaresOfDistinctCounts(nums, n);
    printf("The sum of the squares of distinct counts of all subarrays is: %d\n", result);
    free(nums);
    return 0;
}
```

Sample output:

Enter the number of elements in the array: 3

Enter the elements of the array:

1,2,1

The sum of the squares of distinct counts of all subarrays is: 12

4) Given a 0-indexed integer array nums of length n and an integer k, return the number of pairs (i, j) where 0 <= i < j < n, such that nums[i]

== nums[j] and (i * j) is divisible by k. Example 1: Input: nums = [3,1,2,2,2,1,3], k = 2

Output: 4

Explanation: There are 4 pairs that meet all the requirements: - nums[0] == nums[6], and 0 * 6 == 0, which is divisible by 2. - nums[2] == nums[3], and 2 * 3 == 6, which is divisible by 2. - nums[2] == nums[4], and 2 * 4 == 8, which is divisible by 2. - nums[3] == nums[4], and 3 * 4 == 12, which is divisible by 2.

Example 2: Input: nums = [1,2,3,4], k = 1

Output: 0

Explanation: Since no value in nums is repeated, there are no pairs (i,j) that meet all the requirements**.**

```
#include <stdio.h>
#include <stdlib.h>
int countPairs(int *nums, int n, int k) {
   int count = 0;
   for (int i = 0; i < n; i++) {
      for (int j = i + 1; j < n; j++) {
         if (nums[i] == nums[j] && (i * j) % k == 0) {
            count++;
         }
      }
   }
   return count;
}
int main() {
   int n, k;
   printf("Enter the number of elements in the array: ");
   if (scanf("%d", &n) != 1 || n <= 0) {
      printf("Invalid input. Number of elements must be a positive integer.\n");
      return 1;
```

```
    }
    int *nums = (int *)malloc(n * sizeof(int));
    if (nums == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        if (scanf("%d", &nums[i]) != 1) {
            printf("Invalid input. Please enter integers only.\n");
            free(nums);
            return 1;
        }
    }
    printf("Enter the value of k: ");
    if (scanf("%d", &k) != 1 || k <= 0) {
        printf("Invalid input. Value of k must be a positive integer.\n");
        free(nums);
        return 1;
    }
    int result = countPairs(nums, n, k);
    printf("The number of valid pairs is: %d\n", result);
    free(nums);
    return 0;
}
```

Sample output:

Enter the number of elements in the array: 7

Enter the elements of the array: 3 1 2 2 2 1 3

Enter the value of k: 2

The number of valid pairs is: 4

5) Write a program FOR THE BELOW TEST CASES with least time complexity Test Cases: - 1) Input: {1, 2, 3, 4, 5}

Expected Output: 5 2)

Input: {7, 7, 7, 7, 7}

Expected Output: 7 3)

Input: {-10, 2, 3, -4, 5}

Expected Output: 5

```
#include <stdio.h>
#include <stdlib.h>
int findMax(int arr[], int size) {
   int max = arr[0];
   for (int i = 1; i < size; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
   }
   return max;
}
int main() {
   int size;
   printf("Enter the number of elements: ");
   scanf("%d", &size);
   if (size <= 0) {
      printf("Invalid size. Exiting...\n");
      return 1;
   }
   int *arr = (int *)malloc(size * sizeof(int));
```

```
    if (arr == NULL) {
        printf("Memory allocation failed. Exiting...\n");
        return 1;
    }
    printf("Enter the elements:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Time complexity is: %d\n", findMax(arr, size));
    free(arr);

    return 0;
}
```

Sample output:

Enter the number of elements: 5

Enter the elements:

1 2 3 4 5

Time complexity is: 5

6) You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.

Test Cases 1.

Empty List 1. Input: []

2. Expected Output: None or an appropriate message indicating that the list is empty.

2. Single Element List 1.

Input: [5]

2. Expected Output: 5

3. All Elements are the Same

1. Input: [3, 3, 3, 3, 3]

2. Expected Output: 3

```
#include <stdio.h>
#include <stdlib.h>
void heapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
   if (left < n && arr[left] > arr[largest]) {
      largest = left;
   }
   if (right < n && arr[right] > arr[largest]) {
      largest = right;
   }
   if (largest != i) {
      int temp = arr[i];
      arr[i] = arr[largest];
      arr[largest] = temp;
      heapify(arr, n, largest);
   }
}
void heapSort(int arr[], int n) {
   for (int i = n / 2 - 1; i >= 0; i--) {
      heapify(arr, n, i);
   }
   for (int i = n - 1; i >= 0; i--) {
      int temp = arr[0];
      arr[0] = arr[i];
      arr[i] = temp;
      heapify(arr, i, 0);
   }
```

```
}
int findMax(int arr[], int size) {
    if (size == 0) {
        printf("The list is empty.\n");
        return -1;
    }
    return arr[size - 1];
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("The list is empty.\n");
        return 0;
    }
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    heapSort(arr, n);
    printf("Maximum element: %d\n", findMax(arr, n));
    return 0;
}
```

Sample output:

Enter the number of elements: 5

Enter 5 elements:

3 3 3 3 3

Maximum element: 3

7) Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?

Test Cases Some Duplicate Elements

- Input: [3, 7, 3, 5, 2, 5, 9, 2]
- Expected Output: [3, 7, 5, 2, 9] (Order may vary based on the algorithm used) Negative and Positive Numbers
- Input: [-1, 2, -1, 3, 2, -2]
- Expected Output: [-1, 2, 3, -2] (Order may vary) List with Large Numbers
- Input: [1000000, 999999, 1000000]
- Expected Output: [1000000, 999999]

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 1000
typedef struct Node {
   int value;
   struct Node* next;
} Node;
Node* hashTable[TABLE_SIZE];
unsigned int hash(int value) {
```

```
    return abs(value) % TABLE_SIZE;
}
void insert(int value) {
    unsigned int index = hash(value);
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->value = value;
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
int exists(int value) {
    unsigned int index = hash(value);
    Node* current = hashTable[index];
    while (current != NULL) {
        if (current->value == value) {
            return 1;
        }
        current = current->next;
    }
    return 0;
}
void freeHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        Node* current = hashTable[i];
        while (current != NULL) {
            Node* temp = current;
```

```
            current = current->next;
            free(temp);
        }
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("The list is empty.\n");
        return 0;
    }
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }
    for (int i = 0; i < n; i++) {
        if (!exists(arr[i])) {
            insert(arr[i]);
        }
```

```
    }
    printf("Unique elements:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        Node* current = hashTable[i];
        while (current != NULL) {
            printf("%d ", current->value);
            current = current->next;
        }
    }
    printf("\n");
    free(arr);
    freeHashTable();
    return 0;
}
```

Sample output:

Enter the number of elements: 6

Enter 6 elements:

-1 2 -1 3 2 -2

Unique elements:

-1 -2 2 3

8) Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code.

```
#include <stdio.h>
#include <stdlib.h>
void bubbleSort(int arr[], int n) {
    int swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = 0;
```

```
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("The list is empty.\n");
        return 0;
    }
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
```

```
    }
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    bubbleSort(arr, n);
    printf("Sorted array:\n");
    printArray(arr, n);
    free(arr);
    return 0;
}
```

Sample output:

Enter the number of elements: 5

Enter 5 elements:

1 4 5 3 6

Sorted array:

1 3 4 5 6

9) Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.

Test Case: Example X={ 3,4,6,-9,10,8,9,30} KEY=10

Output: Element 10 is found at position 5

Example X={ 3,4,6,-9,10,8,9,30} KEY=100

Output : Element 100 is not found

```
#include <stdio.h>
int binarySearch(int arr[], int size, int key) {
    int left = 0;
```

```
    int right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
int main() {
    int arr[] = {-9, 3, 4, 6, 8, 9, 10, 30};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key1 = 10;
    int key2 = 100;
    int result = binarySearch(arr, size, key1);
    if (result != -1)
        printf("Element %d is found at position %d\n", key1, result + 1);
    else
        printf("Element %d is not found\n", key1);
    return 0;
}
```

Sample output:

Element 10 is found at position 7

10) Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in

functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

```
#include <stdio.h>
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
   for (int j = low; j <= high - 1; j++) {
      if (arr[j] <= pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
void printArray(int arr[], int size) {
```

```
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int arr[] = {3, 4, 6, -9, 10, 8, 9, 30};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

Sample output:

Original array: 3 4 6 -9 10 8 9 30

Sorted array: -9 3 4 6 8 9 10 30