DSA ASSIGNMENT-2

G.ADITHYA PRASAD

22BCE1861

1. . Consider an array L contains ‘n’ positive integers. Sort the array L in increasing order based on the length of the positive numbers, where the length of the number represents a total number of digits of the number. For example, the length of 1231 is 4. While sorting, if more than one number is in same length, those numbers are to be sorted in decreasing order within them. Write a suitable pseudo code for the same and find the time complexity.

CODE:

```
void sort_by_length(int L[], int n) {
 // Create a dictionary to store the length of each number as
the key and the number itself as the value.
 int length_dict[n];
 for (int i = 0; i < n; i++) {
 int length = floor(log10(L[i]) + 1);
 length_dict[length] = L[i];
 }
 // Sort the dictionary by the length of the numbers.
 qsort(length_dict, n, sizeof(int), compare_lengths);
 // Create a list to store the sorted numbers.
 int sorted_list[n];
 int sorted_list_index = 0;
 for (int i = 0; i < n; i++) {
 int length = length_dict[i];
 sorted_list[sorted_list_index++] = length_dict[length];
 }
 // Copy the sorted list back to the original array.
 for (int i = 0; i < n; i++) {
 L[i] = sorted_list[i];
 }
}
// Compares two lengths and returns -1 if the first length is
less than the second length, 0 if the two lengths are equal, and
1 if the first length is greater than the second length.
int compare_lengths(const void *a, const void *b) {
 int length_a = *(int *)a;
 int length_b = *(int *)b;
 if (length_a < length_b) {
 return -1;
 } else if (length_a > length_b) {
 return 1;
 } else {
 return 0
 }
```

The time complexity of this algorithm is O(n log n), where n is the length of the input array. This is because the algorithm sorts the dictionary of lengths in logarithmic time, and then sorts each list of numbers in decreasing order in linear time.

2. Consider an array A[0,…,n-1] contains 'n' integers. A peak of an array A is defined as A[i-1] < A[i] and A[i] > A[i+1]. i.e A peak A[i] is greater than to its neighbors in an array A. Assume that first and last integers having only one neighbor (A[0] is peak, if A[0]>A[1] and A[n-1] is peak, if A[n-1] >A[n-2]). Write a pseudo code to find and print the peaks and your code should give time complexity as O(log n). Example: A=[10,6,4,3,12,19,18] Output: 10,19

   CODE:

```
#include <stdio.h>
// Function to find a peak in the array A using binary
search
int findPeak(int A[], int left, int right, int n) {
 // Check if the array has only one element (base case)
 if (left == right) {
```

```
 return A[left];
 }
 // Find the middle element of the array
 int mid = left + (right - left) / 2;
 // Check if the middle element is a peak
 if (mid > 0 && A[mid] > A[mid - 1] && mid < n - 1 &&
A[mid] > A[mid + 1]) {
 return A[mid];
 }
 // If the middle element is not a peak and the element
to the
left is greater, search in the left half
 if (mid > 0 && A[mid] < A[mid - 1]) {
 return findPeak(A, left, mid - 1, n);
 }
 // If the middle element is not a peak and the element
to the
right is greater, search in the right half
 return findPeak(A, mid + 1, right, n);
}
int main() {
 int A[] = {10, 6, 4, 3, 12, 19, 18};
 int n = sizeof(A) / sizeof(A[0]);
 // Find and print the peak
 int peak = findPeak(A, 0, n - 1, n);
 printf("%d\n", peak);
 return 0;
}
```

This code uses a binary search algorithm to find a peak in the array A. The time complexity of this algorithm is O(log n), where n is the number of elements in the array. It repeatedly divides the search range in half until it finds a peak element.

3. Consider a stack contains ‘n’ integers. Write a pseudo code to move all negative integers to the bottom of stack and move all positive integers to the top of stack without changing the order of the elements in the stack. Your pseudo code should use stack data structure only and find it’s time complexity.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
// Structure for the stack
struct Stack {
 int* array;
 int top;
 int capacity;
};
// Function to create a new stack with the given capacity
struct Stack* createStack(int capacity) {
 struct Stack* stack = (struct Stack*)malloc(sizeof(struct
Stack));
 stack->capacity = capacity;
 stack->top = -1;
 stack->array = (int*)malloc(stack->capacity * sizeof(int));
 return stack;
}
```

```
// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
 return stack->top == -1;
}
// Function to push an element onto the stack
void push(struct Stack* stack, int item) {
 stack->array[++stack->top] = item;
}
// Function to pop an element from the stack
int pop(struct Stack* stack) {
 if (!isEmpty(stack)) {
 return stack->array[stack->top--];
 }
 return -1; // Stack is empty
}
// Function to move negative integers to the bottom of the
stack
void moveNegativesToBottom(struct Stack* stack, int n) {
 struct Stack* tempStack = createStack(n);

 while (!isEmpty(stack)) {
```

```
 int item = pop(stack);
 if (item < 0) {
 push(tempStack, item);
 }
 }
 while (!isEmpty(tempStack)) {
 int item = pop(tempStack);
 push(stack, item);
 }
 free(tempStack);
}
int main() {
 struct Stack* stack = createStack(10); // Create a stack
with
capacity 10
 // Push elements onto the stack
 push(stack, 5);
 push(stack, -2);
 push(stack, 7);
 push(stack, -1);
 push(stack, 3);
 // Move negative integers to the bottom of the stack
 moveNegativesToBottom(stack, 5);
 // Print the stack
 while (!isEmpty(stack)) {
 printf("%d ", pop(stack));
 }
 free(stack->array);
 free(stack);
 return 0;
}
```

The time complexity of this algorithm is O(n), where n is the number of elements in the stack. We iterate through the stack once to move the negative integers to the bottom

4. Let Q be a Queue data structure contains ‘n’ integers. And also consider another integer ‘x’. Write a pseudo code using queue data structure (only) to determine whether there exist two elements in Q whose difference is exactly equal to ‘x’ or not. If yes, then display those numbers, otherwise print ‘not found’.

Code:

```
#include <stdio.h>
#include <stdlib.h>
// Structure for a node in the queue
struct QueueNode {
 int data;
 struct QueueNode* next;
};
// Structure for the queue
```

```
struct Queue {
 struct QueueNode* front;
 struct QueueNode* rear;
};
// Function to create an empty queue
struct Queue* createQueue() {
 struct Queue* queue = (struct
Queue*)malloc(sizeof(struct
Queue));
 queue->front = queue->rear = NULL;
 return queue;
}
// Function to enqueue an element
void enqueue(struct Queue* queue, int item) {
 struct QueueNode* newNode = (struct
QueueNode*)malloc(sizeof(struct QueueNode));
 newNode->data = item;
 newNode->next = NULL;

 if (queue->rear == NULL) {
 queue->front = queue->rear = newNode;
 return;
 }

 queue->rear->next = newNode;
 queue->rear = newNode;
}
// Function to dequeue an element
int dequeue(struct Queue* queue) {
 if (queue->front == NULL) {
 return -1; // Queue is empty
 }

 struct QueueNode* temp = queue->front;
 int item = temp->data;
 queue->front = temp->next;

 if (queue->front == NULL) {
 queue->rear = NULL;
 }

 free(temp);
 return item;
}
// Function to check if there exist two elements in the
queue
whose difference is 'x'
int hasPairWithDifferenceX(struct Queue* queue, int x) {
 struct Queue* tempQueue = createQueue();
 int found = 0;
 while (queue->front != NULL) {
 int current = dequeue(queue);
 // Check if there is another element in the tempQueue
whose difference with 'x' is equal to the current element
 struct QueueNode* tempNode = tempQueue->front;
 while (tempNode != NULL) {
 if (abs(tempNode->data - current) == x) {
 printf("Pair found: %d, %d\n", tempNode->data,
current);
 found = 1;
 break;
 }
 tempNode = tempNode->next;
 }
 enqueue(tempQueue, current);
 }
 free(tempQueue);
 return found;
}
int main() {
 struct Queue* queue = createQueue();
 int x = 5;
 // Enqueue elements into the queue
 enqueue(queue, 10);
 enqueue(queue, 15);
 enqueue(queue, 2);
 enqueue(queue, 7);

 if (!hasPairWithDifferenceX(queue, x)) {
 printf("Not found\n");
 }
 return 0;
}
```
The time complexity of this algorithm is O(n^2), where n is the number of elements in the queue.