A)#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 1000000

// Define a structure for stack
struct Stack
{
char arr[MAX_SIZE];
int top;
};

// Function to create an empty stack
struct Stack *createStack()
{
struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
stack->top = -1;
return stack;
}

// Function to push an element onto the stack
void push(struct Stack *stack, char c)
{
stack->arr[++stack->top] = c;
}

// Function to pop an element from the stack
char pop(struct Stack *stack)
{
if (stack->top == -1)
return '\0'; // Stack is empty
return stack->arr[stack->top--];
}

// Function to check if the given string is a valid parentheses string
char *isValidParentheses(char *s)
{
int n = strlen(s);
struct Stack *stack = createStack();

for (int i = 0; i < n; i++)
{
if (s[i] == '(' || s[i] == '[' || s[i] == '{')
{
push(stack, s[i]);
}
else
{
if (stack->top == -1)
return "NO"; // Closing bracket with no corresponding opening bracket
char c = pop(stack);
if ((s[i] == ')' && c != '(') || (s[i] == ']' && c != '[') || (s[i] == '}' && c != '{'))
return "NO"; // Mismatched closing and opening brackets
}
}

if (stack->top != -1)
return "NO"; // Unmatched opening brackets left in the stack

return "YES"; // All brackets matched
}

int main()
{
char s[MAX_SIZE];
fgets(s, MAX_SIZE, stdin);
s[strcspn(s, "\n")] = '\0'; // Removing trailing newline character from fgets

printf("%s\n", isValidParentheses(s));

return 0;
}

C)#include <stdio.h>
#define SIZE 100000
unsigned int MOD = 1000000007;

int front = -1;
int rear = -1;
int arr[SIZE];
void enqueue(int x)
{
if (front < SIZE - 1)
{
if (front == -1)
{
front++;
}
rear++;
arr[rear] = x;
}
}
void pop()
{
if ((front <= rear) && (front != -1))
{
front++;
}
}
int dequeue()
{
if (front != -1)
{
return arr[front];
}
else
{
return -1;
}
}
int isEmpty()
{
if ((front > rear) || (front == -1))
{
return 1;
}
else
{
return 0;
}
}
int min()
{
int min = arr[front];
for (int i = front + 1; i <= rear; i++)
{
if (arr[i] < min)
{
min = arr[i];
}
}
return min;
}

void clear_queue()
{
front = -1;
rear = -1;
}

int main()
{
int n;
scanf("%d", &n);
int sum = 0;
int arr1[n];
for (int i = 0; i < n; i++)
{
scanf("%d", &arr1[i]);
}

for (int w = 1; w <= n; w++)
{
for (int i = 0; i < w; i++)
{
enqueue(arr1[i]);
}
sum = (sum + min()) % MOD;
for (int i = w; i < n; i++)
{
enqueue(arr1[i]);
pop();
sum = (sum + min()) % MOD;
}
clear_queue();
}
printf("%d\n", sum);

return 0;
}

B)#include <stdio.h>
#include <stdlib.h>

struct node
{
int data;
struct node *next;
};

struct Queue
{
struct node *front;
struct node *rear;
int max;
};

int dequeue(struct Queue *queue)
{
if (queue->front == NULL)
{
printf("Queue is empty\n");
return -1;
}

struct node *temp = queue->front;
int val = temp->data;
queue->front = queue->front->next;

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

if (val == queue->max)
{
queue->max = -1;
struct node *curr = queue->front;
while (curr != NULL)
{
if (curr->data > queue->max)
{
queue->max = curr->data;
}
curr = curr->next;
}
}

free(temp);
return val;
}

void enqueue(struct Queue *queue, int val)
{
struct node *new_node = (struct node *)malloc(sizeof(struct node));
new_node->data = val;
new_node->next = NULL;

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

if (val > queue->max)
{
queue->max = val;
}
}

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

struct Queue *queue = (struct Queue *)malloc(sizeof(struct Queue));
queue->front = NULL;
queue->rear = NULL;
queue->max = -1;

int head_pointer = 0;
int tail_pointer = 0;

// Setting up the initial window
for (int i = 0; i < k; i++)
{
enqueue(queue, arr[i]);
}
tail_pointer = k;

printf("%d ", queue->max);

while (tail_pointer < n)
{
int element = dequeue(queue);
enqueue(queue, arr[tail_pointer]);
tail_pointer++;
printf("%d ", queue->max);
head_pointer++;
}

free(queue);
return 0;
}

D) #include <stdio.h>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

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

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

int stack[n]; // stack to store indices of histogram bars
int top = -1; // initialize stack top

int max_area = 0;
int i = 0;
while (i < n)
{
if (top == -1 || heights[stack[top]] <= heights[i])
stack[++top] = i++; // push current index to stack
else
{
int tp = stack[top--]; //
store the top index and pop from stack
int area_with_top = heights[tp] * (top == -1 ? i : i - stack[top] - 1); // calculate
area with the popped bar as the smallest bar
max_area = MAX(max_area, area_with_top); //
update maximum area if needed
}
}

while (top != -1)
{
int tp = stack[top--]; //
store the top index and pop from stack
int area_with_top = heights[tp] * (top == -1 ? i : i - stack[top] - 1); // calculate
area with the popped bar as the smallest bar
max_area = MAX(max_area, area_with_top); //
update maximum area if needed
}

printf("%d\n", max_area);

return 0;
}

E)#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int pop(char Q[], int base)
{
int offset = 1;
while (Q[base + offset] == '0')
{
offset++;
}
return base + offset;
}

long long str_to_int(char *str)
{
char *endptr;
long long val = strtoll(str, &endptr, 10);
if (*endptr != '\0')
{
printf("Conversion failed. Invalid character '%c' found.\n", *endptr);
}

return val;
}

int main()
{
int n, k;
scanf("%d %d", &n, &k);
char Q[n];
scanf("%s", Q);
int base = 0;
for (int i = 0; i < k - 1; i++)
{
base = pop(Q, base);
}
char price_string[n - base];
int j = 0;
for (int i = base + 1; i < n; i++)
{
price_string[j] = Q[i];
j++;
}
price_string[j] = '\0';
long long new_price = str_to_int(price_string);
long long old_price = str_to_int(Q);
long long profit = old_price - new_price;
printf("%lld\n", profit);
return 0;
}

F)#include <stdio.h>
#include <stdlib.h>

// Structure to represent elements in the stack
typedef struct
{
int index;
int value;
} StackElement;

// Structure to represent the stack
typedef struct
{
int top;
StackElement *array;
} Stack;

// Function to initialize a stack
Stack *createStack(int size)
{
Stack *stack = (Stack *)malloc(sizeof(Stack));
stack->top = -1;
stack->array = (StackElement *)malloc(size * sizeof(StackElement));
return stack;
}

// Function to check if the stack is empty
int isEmpty(Stack *stack)
{
return stack->top == -1;
}

// Function to push an element onto the stack
void push(Stack *stack, int index, int value)
{
stack->top++;
stack->array[stack->top].index = index;
stack->array[stack->top].value = value;
}

// Function to pop an element from the stack
StackElement pop(Stack *stack)
{
return stack->array[stack->top--];
}

// Function to find the next greater elements
void nextGreaterElements(int *weights, int n, int *result)
{
Stack *stack = createStack(n);
for (int i = 0; i < n; i++)
{
while (!isEmpty(stack) && weights[i] > stack->array[stack->top].value)
{
StackElement element = pop(stack);
result[element.index] = i - element.index;
}
push(stack, i, weights[i]);
}
while (!isEmpty(stack))
{
StackElement element = pop(stack);
result[element.index] = 0;
}
free(stack->array);
free(stack);
}

// Function to find the next smaller elements
void nextSmallerElements(int *weights, int n, int *result)
{
Stack *stack = createStack(n);
for (int i = 0; i < n; i++)
{
while (!isEmpty(stack) && weights[i] < stack->array[stack->top].value)
{
StackElement element = pop(stack);
result[element.index] = i - element.index;
}
push(stack, i, weights[i]);
}
while (!isEmpty(stack))
{
StackElement element = pop(stack);
result[element.index] = 0;
}
free(stack->array);
free(stack);
}

int main()
{
int n;
scanf("%d", &n);
int *weights = (int *)malloc(n * sizeof(int));
for (int i = 0; i < n; i++)
{
scanf("%d", &weights[i]);
}

int *nextGreater = (int *)calloc(n, sizeof(int));
int *nextSmaller = (int *)calloc(n, sizeof(int));

```
                nextGreaterElements(weights, n, nextGreater);
                nextSmallerElements(weights, n, nextSmaller);

                // Output the results
                for (int i = 0; i < n; i++)
                {
                printf("%d ", nextGreater[i]);
                }
                printf("\n");
                for (int i = 0; i < n; i++)
                {
                printf("%d ", nextSmaller[i]);
                }
                printf("\n");

                return 0;
}


G)#include <stdio.h>
#include <stdlib.h>

int count_subarrays(int n, int k, int *arr)
{
                int result = 0;
                int *window = (int *)malloc(n * sizeof(int));
                int left = 0, right = 0;
                int min_val = arr[0], max_val = arr[0];

                while (right < n)
                {
                // Update min_val and max_val
                min_val = (arr[right] < min_val) ? arr[right] : min_val;
                max_val = (arr[right] > max_val) ? arr[right] : max_val;

                // Remove elements from the left side of the window that violate the
condition
                while (max_val - min_val > k)
                {
                if (arr[left] == min_val)
                {
                min_val = max_val;
                for (int i = left + 1; i <= right; i++)
                {
                min_val = (arr[i] < min_val) ? arr[i] : min_val;
                }
                }
                if (arr[left] == max_val)
                {
                max_val = min_val;
                for (int i = left + 1; i <= right; i++)
                {
                max_val = (arr[i] > max_val) ? arr[i] : max_val;
                }
                }
                left++;
                }

                // Count the number of subarrays that satisfy the condition
                result += right - left + 1;
                right++;
                }

                free(window);
                return result;
}

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

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

                int result = count_subarrays(n, k, arr);
                printf("%d\n", result);

                free(arr);
                return 0;
}


I)#include <stdio.h>

// Swap two integers
void swap(int *a, int *b)
{
                int temp = *a;
                *a = *b;
                *b = temp;
}

// Partition function used in QuickSort
int partition(int arr[], int low, int high)
{
                int pivot = arr[high];
                int i = low - 1;

                for (int j = low; j < high; j++)
                {
                if (arr[j] <= pivot)
                {
                i++;
                swap(&arr[i], &arr[j]);
                }
                }

                swap(&arr[i + 1], &arr[high]);
                return i + 1;
}

// Modified version of QuickSort to find kth smallest element
int select(int arr[], int low, int high, int k)
{
                if (low == high)
                return arr[low];

                int pivotIndex = partition(arr, low, high);
                int pivotRank = pivotIndex - low + 1;

                if (k == pivotRank)
                return arr[pivotIndex];
                else if (k < pivotRank)
                return select(arr, low, pivotIndex - 1, k);
                else
                return select(arr, pivotIndex + 1, high, k - pivotRank);
}

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

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

                int kthLargest = select(arr, 0, n - 1, n - k + 1);
                printf("%d\n", kthLargest);

                return 0;
}

J)#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Swap two integers
void swap(int *a, int *b)
{
                int temp = *a;
                *a = *b;
                *b = temp;
}

// Partition function used in Quickselect
int partition(int arr[], int low, int high)
{
                int pivot = arr[high];
                int i = low - 1;

                for (int j = low; j < high; j++)
                {
                if (arr[j] <= pivot)
                {
                i++;
                swap(&arr[i], &arr[j]);
                }
                }

                swap(&arr[i + 1], &arr[high]);
                return i + 1;
}

// Quickselect algorithm
int quickselect(int arr[], int low, int high, int k)
{
                if (low == high)
                return arr[low];

                int pivotIndex = partition(arr, low, high);
                int pivotRank = pivotIndex - low + 1;

                if (k == pivotRank)
                return arr[pivotIndex];
                else if (k < pivotRank)
                return quickselect(arr, low, pivotIndex - 1, k);
                else
                return quickselect(arr, pivotIndex + 1, high, k - pivotRank);
}

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

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

                srand(time(NULL)); // Seed the random number generator

                int kthLargest = quickselect(arr, 0, n - 1, n - k + 1);
                printf("%d\n", kthLargest);

                return 0;
}

H)#include <stdio.h>
#include <stdlib.h>

typedef struct
{
                int *arr;
                int capacity;
                int size;
} MinHeap;

typedef struct
{
                int *arr;
                int capacity;
                int size;
                double sum;
} AvgHeap;

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

AvgHeap *createAvgHeap(int capacity)
{
                AvgHeap *avgHeap = (AvgHeap *)malloc(sizeof(AvgHeap));
                avgHeap->arr = (int *)malloc(capacity * sizeof(int));
                avgHeap->capacity = capacity;
                avgHeap->size = 0;
                avgHeap->sum = 0;
                return avgHeap;
}

void swap(int *a, int *b)
{
                int temp = *a;
                *a = *b;
                *b = temp;
}

void heapifyMin(MinHeap *minHeap, int i)
{
                int smallest = i;
                int left = 2 * i + 1;
                int right = 2 * i + 2;

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

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

                if (smallest != i)
                {
                swap(&minHeap->arr[i], &minHeap->arr[smallest]);
                heapifyMin(minHeap, smallest);
                }
}

void heapifyAvg(AvgHeap *avgHeap, int i)
{
                int smallest = i;
                int left = 2 * i + 1;
                int right = 2 * i + 2;

                if (left < avgHeap->size && avgHeap->arr[left] < avgHeap->arr[smallest])
                smallest = left;

                if (right < avgHeap->size && avgHeap->arr[right] < avgHeap->arr[smallest])
                smallest = right;

                if (smallest != i)
                {
                swap(&avgHeap->arr[i], &avgHeap->arr[smallest]);
                heapifyAvg(avgHeap, smallest);
                }
}

void pushMin(MinHeap *minHeap, int val)
{
                if (minHeap->size == minHeap->capacity)
                {
                printf("Overflow: Cannot push %d to min heap\n", val);
                return;
                }

                minHeap->arr[minHeap->size] = val;
                int i = minHeap->size;
                minHeap->size++;

                while (i && minHeap->arr[i] < minHeap->arr[(i - 1) / 2])
                {
                swap(&minHeap->arr[i], &minHeap->arr[(i - 1) / 2]);
                i = (i - 1) / 2;
                }
}

void pushAvg(AvgHeap *avgHeap, int val)
{
                if (avgHeap->size == avgHeap->capacity)
                {
                printf("Overflow: Cannot push %d to avg heap\n", val);
                return;
                }

                avgHeap->arr[avgHeap->size] = val;
                int i = avgHeap->size;
                avgHeap->size++;
                avgHeap->sum += val;
                while (i && avgHeap->arr[i] < avgHeap->arr[(i - 1) / 2])
                {
                swap(&avgHeap->arr[i], &avgHeap->arr[(i - 1) / 2]);
                i = (i - 1) / 2;
                }
}

int popMin(MinHeap *minHeap)
{
                if (minHeap->size == 0)
                {
                printf("Underflow: Cannot pop from min heap\n");
                return -1;
                }

                int root = minHeap->arr[0];
                minHeap->arr[0] = minHeap->arr[minHeap->size - 1];
                minHeap->size--;
                heapifyMin(minHeap, 0);

                return root;
}

int popAvg(AvgHeap *avgHeap)
{
                if (avgHeap->size == 0)
                {
                printf("Underflow: Cannot pop from avg heap\n");
                return -1;
                }

                int root = avgHeap->arr[0];
                avgHeap->arr[0] = avgHeap->arr[avgHeap->size - 1];
                avgHeap->sum -= root;
                avgHeap->size--;
                heapifyAvg(avgHeap, 0);

                return root;
}

int getMin(MinHeap *minHeap)
{
                if (minHeap->size == 0)
                {
                printf("Min heap is empty\n");
                return -1;
                }
                return minHeap->arr[0];
}

int getAvg(AvgHeap *avgHeap)
{
                if (avgHeap->size == 0)
                {
                printf("Avg heap is empty\n");
                return -1;
                }
                return (int)(avgHeap->sum / avgHeap->size);
}

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

                MinHeap *minHeap = createMinHeap(q);
                AvgHeap *avgHeap = createAvgHeap(q);

                for (int i = 0; i < q; i++)
                {
                int op;
                scanf("%d", &op);

                if (op == 1)
                {
                int val;
                scanf("%d", &val);
                pushMin(minHeap, val);
                pushAvg(avgHeap, val);
                }
                else if (op == 2)
                {
                popMin(minHeap);
                popAvg(avgHeap);
                }
                else if (op == 3)
                {
                int minVal = getMin(minHeap);
                int avgVal = getAvg(avgHeap);
                printf("%d %d\n", minVal, avgVal);
                }
                }

                free(minHeap->arr);
                free(minHeap);
                free(avgHeap->arr);
                free(avgHeap);

                return 0;
}
```