**CS2420 2021 Fall Midterm 2**

1 **(10 points)**. Given a binary array (i.e., contains only 0 and 1), find the maximum number of consecutive 1s in this array. For example:

Input: [1,1,0,1,1,1]

Output: 3

The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

int max\_consecutive\_ones(vector<int>& nums) {

// TODO

}

Now, analyze the time complexity of your algorithm using Big-O notation.

2 **(10 points)**. Given a limited range array of size N where the array contains elements between 1 to n-1 with one duplicate number, write a program to find the duplicate number. For example

Input: {1, 2, 4, 3, 4}

Output: 4

Input: {1, 2, 1, 3, 4, 6, 5}

Output: 1

int duplicate\_number(std::vector<int>& array) {

}

Analyze the time complexity using the Big-O notation.

3 **(10 points)**. In the early lecture, we have wrote a program of using “two-level for-loop” to find the maximum subarray sum (i.e., contiguous array sum) over an array of integer numbers. For example:



The solution of using two-level for-loop is essentially “brute-force” since we enumerate all possible length of the subarray. The number of required iterations is approximately *N2* where *N* is the size of the array, and it is very slow when the array is large. Your job is to solve this problem again ***using only one level of for-loop*** such that the number of iterations you need is no larger than *N*.

int max\_subarray\_sum(const std::vector<int>& array) {

// TODO

}

int main() {

std::vector<int> array = {-2, -3, 4, -1, -2, 1, 5, -3};

std::cout << max\_subarray\_sum(array) << ‘\n’;

}

Analyze the time complexity of your algorithm using big-O notation.

4 **(10 points)**. We have talked about Merge Sort in the lecture. Merge Sort is one of the most powerful sorting algorithms using divide and conquer. It iteratively partitions the array by half and combines two subproblems into a single sorted array. For example:



Your job is to complete the following merge\_sort function that takes a reference to an input array of integer numbers and sort them. You are free to declare other functions you need.

void merge\_sort(std::vector<int>& array) {

// TODO

}

int main() {

std::vector<int> array = {20, 90, 40, 30, 80, 70, 50};

merge\_sort(array);

for(auto i : array) { // sorted numbers

std::cout << i << ‘\n’;

}

}

5 **(10 points)**. Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in exp.

**Input**: exp = “[()]{}{[()()]()}”   
**Output**: Balanced

**Input**: exp = “[(])”   
**Output**: Not Balanced

bool check\_expression(const std::string& exp) {

// TODO

}

int main() {

std::string exp1 = “()()”;

if(check\_expression(exp1)) {

std::cout << “Balanced\n”;

}

else {

std::cout << “Not Balanced\n”;

}

}

Analyze the runtime complexity of your algorithm using Big-O notation.

6 **(10 points)** Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

* void push(int x) Pushes element x to the top of the stack.
* int pop() Removes the element on the top of the stack and returns it.
* int top() Returns the element on the top of the stack.
* boolean empty() Returns true if the stack is empty, false otherwise.

**Notes:**

* You must use **only** standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.
* Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

class MyStack {

public:

MyStack() {

}

void push(int x) {

}

int pop() {

}

int top() {

}

bool empty() {

}

};

Analyze the time complexity of each method in the class using Big-O notation.

7 (**10 points**) There are n people standing in a queue, and they numbered from 0 to n - 1 in **left to right** order. You are given an array heights of **distinct** integers where heights[i] represents the height of the ith person.

A person can **see** another person to their right in the queue if everybody in between is **shorter** than both of them. More formally, the ith person can see the jth person if i < j and min(heights[i], heights[j]) > max(heights[i+1], heights[i+2], ..., heights[j-1]).

Return *an array*answer*of length*n*where*answer[i]*is the****number of people****the*ith*person can****see****to their right in the queue*.

**Example 1:**



**Input:** heights = [10,6,8,5,11,9]

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

**Explanation:**

Person 0 can see person 1, 2, and 4.

Person 1 can see person 2.

Person 2 can see person 3 and 4.

Person 3 can see person 4.

Person 4 can see person 5.

Person 5 can see no one since nobody is to the right of them.

**Example 2:**

**Input:** heights = [5,1,2,3,10]

**Output:** [4,1,1,1,0]

**Constraints:**

* n == heights.length
* 1 <= n <= 105
* 1 <= heights[i] <= 105
* All the values of heights are **unique**.

Finish the function below to solve this question:

vector<int> canSeePersonsCount(vector<int>& heights) {

}

Analyze the time complexity of your algorithm using Big-O notation.

8 (**10 points**). Given an integer array nums and an integer k, return *the* kth *largest element in the array*. Note that it is the kth largest element in the sorted order, not the kth distinct element.

**Example 1:**

**Input:** nums = [3,2,1,5,6,4], k = 2

**Output:** 5

**Example 2:**

**Input:** nums = [3,2,3,1,2,4,5,5,6], k = 4

**Output:** 4

**Constraints:**

* 1 <= k <= nums.length <= 104
* -104 <= nums[i] <= 104

Finish the function below to solve this question:

int findKthLargest(vector<int>& nums, int k) {

}

9 (**10 points**) You are given an array of integers stones where stones[i] is the weight of the ith stone. We are playing a game with the stones. On each turn, we choose the **heaviest two stones** and smash them together. Suppose the heaviest two stones have weights x and y with x <= y. The result of this smash is:

* If x == y, both stones are destroyed, and
* If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end of the game, there is **at most one** stone left.

Return *the smallest possible weight of the left stone*. If there are no stones left, return 0.

**Example 1:**

**Input:** stones = [2,7,4,1,8,1]

**Output:** 1

**Explanation:**

We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then,

we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then,

we combine 2 and 1 to get 1 so the array converts to [1,1,1] then,

we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of the last stone.

**Example 2:**

**Input:** stones = [1]

**Output:** 1

**Constraints:**

* 1 <= stones.length <= 30
* 1 <= stones[i] <= 1000

Finish the function below to solve the problem:

int lastStoneWeight(vector<int>& stones) {

}

Analyze the time complexity of your algorithm using the big-O notation.

10 (**10 points**) Given an array nums with n objects colored red, white, or blue, sort them in-placeso that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

**Example 1:**

**Input:** nums = [2,0,2,1,1,0]

**Output:** [0,0,1,1,2,2]

**Example 2:**

**Input:** nums = [2,0,1]

**Output:** [0,1,2]

**Example 3:**

**Input:** nums = [0]

**Output:** [0]

**Example 4:**

**Input:** nums = [1]

**Output:** [1]

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* nums[i] is 0, 1, or 2.

Finish the function below to solve this problem.

void sortColors(vector<int>& nums) { }

Analyze the time complexity of your algorithm using the big-O notation