SI@UCF Computer Science Camp
Intro to Competitive Programming in C++
Quiz #1 Solutions
Date: 6/14/2025

1) (10 pts) Write a segment of code in C++ which reads in a positive integer, n, from the user (no prompt necessary), and then prints out the values 2^0, 2^1, 2^2, …, 2^n to the screen. Your code MAY NOT use any intermediate values that are floating point numbers. You may assume that the user enters a number in between 0 and 30, inclusive.

```
int n;                          // 1 pt
cin >> n;                       // 1 pt
int res = 1;                    // 1 pt
for (int i=0; i<=n; i++) {      // 2 pts
    cout << res << endl;        // 2 pts
    res = 2*res;                // 3 pts
}
```

2) (5 pts) Show the output of the following C++ program.

```
using namespace std;
#include <bits/stdc++.h>

int main() {
    vector<int> arr = {3,2,12,6,8,5};
    sort(arr.rbegin(), arr.rend());
    for (int i=0; i<arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

12 8 6 5 3 2

Grading: 5 pts correct, 2 pts if sorted ascending, 0 pts otherwise

3) (5 pts) Explain why the following boolean expression might equal false. Assume that the math library is imported.

```
pow(cbrt(123),3) == 123
```

Inherent in any function that returns a floating point number (float, double) is error because real numbers are typically not stored exactly in the computer, but just to some number of significant digits. Thus, the cube root call returns a number that's probably slightly different than the actual cube root of 123, so the value being raised to the 3rd power by the pow function is "a bit off". In addition, the cubing process itself might be a "bit off", so both function calls introduce the possibility of error, meaning that the final evaluation of the expression on the left might be a little bit more or less than 123 instead of exactly equal to it. **(Grading: grader discretion)**

4) (25 pts) A common task in programming is taking in a long string and separating it into several strings according to a splitting character (often a space). For example, given the string "Hello how are you" and the tokenizer character " ", the result of tokenizing the original string would be the list of strings "Hello", "how", "are", and "you", It's natural to store these resultant strings in a vector of strings. Complete the function below, so that it takes in a string and a tokenizing character stored in a string, and returns a vector storing each of the tokens formed after the input string is tokenized. (None of the strings in the vector returned should contain the tokenizing character. These characters should be deleted.) To make the task a bit easier, you may assume that the tokenizing character won't appear two consecutive times and that it won't be either the first or last character of the input string. It is possible that the tokenizing character (called split below) doesn't appear in the input string (original) at all.

```
vector<string> tokenize(string original, string split) {

    int prev = 0;
    int idx = original.find(split, 0);
    vector<string> res;

    while (idx != -1) {
        int len = idx - prev;
        res.push_back(original.substr(prev, len));
        prev = idx+split.size();
        idx = original.find(split, prev);

    }

    res.push_back(original.substr(prev, original.size()-prev));
    return res;


}
```

Grading: **Give partial as needed.**
Basic set up in the beginning before loop: 5 pts
Looping until split isn't found: 3 pts
Using find call to figure out where appropriate substring is: 5 pts
Adding substring to vector: 3 pts
Updating indexes: 4 pts
Adding last token: 4 pts
Returning: 1 pt

5) (15 pts) In the problem downtime, which was solved in lecture, the input given was a list of requests for jobs to be completed. Each of the jobs took 1000 ms and each server could simultaneously work on at most k jobs. The problem asked to compute the minimum number of servers necessary so that each job could immediately be given to a server at the time of the request. One solution to this problem was to simulate the process. (Go through the jobs in order, and pick a server that had fewer than k active jobs to start working on it. Mark that server, etc.) It was mentioned that this solution would be tedious and unnecessary and that there was a cleaner, easier solution. In your own words, (a) describe the key observation and that easier solution, (b) explain the key idea behind an efficient implementation of the solution, and (c) a very similar solution to the problem got a Time Limited Exceeded verdict in class, explain what inefficiency was added to the correct solution to obtain this outcome.

(a) The key idea was that if there were n jobs running simultaneously, and each server could only do k jobs in parallel, then at least ceiling(n/k) servers would be needed. Thus, it's good enough to find the maximum number of jobs that are all running at the same time. This is equivalent to finding the maximum number of jobs that have start times within 999 ms of each other, since the first of these jobs isn't complete 999 ms later while all of the future ones will also have to be running. So, the easier solution is to sort the input list and have a sliding window of values where the difference between the first and last value in the window is less than 1000. Then, take the maximum size of all such windows.

(b) The key ideas is to have two "pointers": one to the left side of the window and one to the right. The left side pointer moves 1 spot at a time. So if the list starts 10, 100, 101, 200, …, then the left pointer starts at 10 and the right pointer crawls ahead to the first number on the list greater than 1009. These two end points represent all the jobs that are running simultaneously. Now, when we move our left pointer from 10 to 100, there's no need to move the right pointer back…just increment the right pointer from where it left off since we know that it could never go backwards…

(c) In the TLE version of the solution, we reset the right pointer, j, to equal the left pointer i, after finding each window. This means that if the window sizes were really big, then j would increment many times for each increment of i. Imagine if all the job were at time 0. This would be the worst case of this inefficient algorithm.

Grading: 5 pts for each part, grader discretion

6) (10 pts) A solution to a problem called the following function roughly 3,000,000 times:

```
int binarysearch(vector<int> arr);
```

The vector passed to the function was of size 20,000 and the number of operations the function executed on each run was relatively small (less than 100). The solution got a time limit exceeded verdict. It turns out that a very simple fix existed to change the verdict to correct. What is that fix and why does it speed up the program so much?

The fix is this: `int binarysearch(vector<int>& arr);` This passes the vector by reference and every time the function is called a full copy of the vector (20000 steps) isn't made. Now, the overall run time of the program is closer to 100 times the number of binary search calls, instead of 20000 times the number of binary search calls. Since there's no longer the overhead of making a full copy of the vector every time the function is called.

Grading: 5 pts for any mention of pass by reference, 5 pts for the explanation of what it is.

7) (25 pts) What follows is the prompt for a Kattis problem that can be solved with a string. Complete the solution for it on the space provided.

Coffee Cup Combo

Jonna is a university student who attends n lectures every day. Since most lectures are way too simple for an algorithmic expert such as Jonna, she can only stay awake during a lecture if she is drinking coffee. During a single lecture she needs to drink exactly one cup of coffee to stay awake. Some of the lecture halls have coffee machines, so Jonna can always make sure to get coffee there. Furthermore, when Jonna leaves a lecture hall, she can bring at most two coffee cups with her to the following lectures (one cup in each hand). But note that she cannot bring more than two coffee cups with her at any given time.

Given which of Jonna's lectures have coffee machines, compute the maximum number of lectures during which Jonna can stay awake.

Input

The first line contains the integer, **n** ($1 \leq \mathbf{n} \leq 10^5$), the number of lectures Jonna attends.

The second line contains a string **s** of length **n**. The i^{th} letter is 1 if there is a coffee machine in the i^{th} lecture hall, and otherwise it is 0.

Output

Print one integer, the maximum number of lectures during which Jonna can stay awake.

Sample Input #1	Sample Output #1
`10` `0100010100`	`8`

Sample Input #1	Sample Output #1
`10` `1100000000`	`4`

Note: In the first sample, she can't get coffee in the first lecture or fifth lecture. In the second sample, she only has coffee for the first four lectures and has none for lectures 5 through 10, inclusive.

```
using namespace std;

#include <bits/stdc++.h>

int main() {

     // Get input.
     int n, res = 0;
     string s;
     cin >> n >> s;

     int res = 0, last = -3;              // 5 pts

     for (int i=0; i<n; i++) {            // 5 pts
          if (s[i] == '1') last = i;      // 7 pts
          if (i-2 <= last) res++;         // 8 pts
     }

     // Ta da!
     cout << res << endl;

     return 0;
}
```

8) (5 pts) June 14th (the day this exam was made) was World Blood Donor Day. What do Blood Donors donate?

<u>Blood</u> (Grading: give to all)

<u>String Class Documentation</u>

```
// Returns the substring of this string starting at index pos
// that is of length len.
string substr (size_t pos, size_t len);

// Returns the index in this string, starting at position pos,
// where the substring is equal to str.
size_t find (const string& str, size_t pos);
```

<u>Vector Class Documentation</u>

```
// Adds val to the end of this vector.
void push_back (const value_type& val);
```