Intro

The review session will cover the following topics:

1. C++ Template
2. Components of an Algorithmic Problem
3. Complexity Analysis
4. Basic Maths

C++ Template:

For a thorough C++ intro you may check the slides available in the program repository.
But as a swift reminder here is the basic template you need to run a C++ program:

Algorithm 1: C++ Template

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

int main() {

}
```

Components of an Algorithmic Problem:

An algorithmic problem can be defined as follows: given some data as input (can be anything, integers, words, characters, ...etc), you need to use the data to perform some task, this can be doing some checks, or computing the value of something or any other type of task. Then you need to print the result required (confirmation or computer value, ..etc), and this output will be checked if its correct or not. Your solution gets accepted if your program outputs a correct value for set of test cases.

A test case is a certain input example that can be given to your program, and has a correct output associated with it, which is the expected value to be produced from your program.

All problems you would see on a contest must have main components. These components are:

- Time Limit
- Memory Limit

- Problem Story
- Constraints
- Input Format
- Output Format

Time Limit:

Any solution you submit to a problem must produce an answer under some time range. Most problems have time limits of 1 or 2 seconds, and in some rare cases it can go up to 4 seconds. Considering the time limit can give you a slight freedom of choosing your solution, so it is a good practice to pay attention to it.

Memory Limit:

When defining variables, data structures of containers, you are technically reserving memory of the machine you are using to create these containers. Of course you cannot use an infinite amount of memory to do so, so you fill find an upper limit of memory you can use for solving a certain problem, the limits are usually $256 - 512$ Mega Bytes.

Problem Story:

This is the main body of the problem. It explains the problems and definitions of terms used in the problem, and explains the task you need to achieve or compute. Understanding this part is the most crucial to solving the problem properly

Constraints:

The constraints define restrictions on the values of the input, for example, if a problem says that an input n satisfies $1 \leq n \leq 10^5$, then it is guaranteed that no test case will contain a value of n that is outside the provided range. Most of the time these is no clear section for constraints, but you may find it usually embedded in the *Problem Story* or *Input Format* parts. It is crucial to spot the constraints because they play an intrinsic role for guiding you towards the solution. For example, a low constraint on the size of the input implies that you may use a slow or naive algorithm to solve it. On the other hand, a high constraint indicates that a slow algorithm will not work, and you will need to find a fast algorithm to solve the problem. More on this point in the *Complexity Analysis* section.

Input Format:

Each problem has some values that must be inputted to your program to be processed and to find the solution. This section explains what values you shall expect in take as input, and how they are ordered, so that you can place the values into the correct storage while taking the input into the program.

Output Format:

As you might have guessed, after you take the input and apply some algorithm to solve the described problem, you need to output some values to check the correctness of your solution. You usually know what you are expected to print in the *Problem Story* part, but there might be a certain format you need to follow for your output. For example, if you need to print n values, the checker program might either expect all these values to be printed in the same line, or each value to be printed in a separate line. It makes a difference so one should stick to the format described.

Complexity Analysis:

Definition:

Complexity analysis is a way of finding how fast an algorithm can perform as a function of the input size. By finding the time complexity, we can **estimate** the efficiency of the algorithm and whether it will pass the times limit of a problem or not.

Time complexity has the form $O(f(n))$, where $f(n)$ is a function of the input size.

Calculation Rules:

There are several indicators in the code that can guide us to find the complexity.

Loops:

Using loops to iterate over the input elements to do some operation on each element would cost us $O(n)$, because we are iterating n times, and in each iteration we are doing small constant number of operations (not dependent on n).

Algorithm 2: Single Loop

```
for(int i = 0 ; i < n ; ++i){
    // do some simple operation
}
```

Another example, using two nested loops inside each other would produce $O(n^2)$ algorithm, because the outer loop iterates n times, and for each iteration of those n, we are doing another n iterations, leading to $O(n \times n) = O(n^2)$

Algorithm 3: Two Nested Loops

```
for(int i = 0 ; i < n ; ++i){
    for(int j = 0 ; j < n ; ++j){
        // do a simple operation
    }
}
```

Observe how we analyze each of the example below:

The complexity of this loop is still $O(n)$ even though we are iterating $3n$ times. Because complexity is only an estimate, not an exact measure of number of operations, and while estimating the complexity we usually drop any constants multiplied by n

Algorithm 4: Loop Example 3

```
for(int i = 0 ; i <= 3 * n ; ++i){
    // code
}
```

In this loop we are iterating $\frac{n}{2}$ times, which is still $O(n)$, because as in the previous example, we drop the constants.

Algorithm 5: Loop Example 4

```
for(int i = 0 ; i < n ; i += 2){
    // code
}
```

Let's try to count how many iteration are happening in this code snippet: The outer loop is iterating n times. In the i^{th} iteration, we are iterating i times, so the number of operations is the sum of all $i's$ from 1 to n, i.e. $1 + 2 + ... + n$. By using the arithmetic series sum law, we can see that the number of operations is $\frac{n(n+1)}{2}$, which can be rewritten as $\frac{n^2}{2} + \frac{n}{2}$. For this function, we say that the complexity is $O(n^2)$, because first we drop the constants to get $n^2 + n$, but we also drop n, because when we have the sum of several functions, we only take the function of higher magnitude and drop the rest. Hence $O(n^2)$

Algorithm 6: Loop Example 5

```
for(int i = 1 ; i <= n ; ++i){
    for(int j = 1 ; j <= i ; ++j){
        // code
    }
}
```

From the previous examples we can conclude two rules for estimating complexities:

1. Drop constants
2. When the number of operations is the sum of several functions, take the function with higher magnitude and drop the rest.

Some Frequently encountered complexities

- $O(1)$: Constant-Time, usually a simple arithmetic operation or formula.
- $O(log_2 n)$: logarithmic, remember that $log_2 n$ is the number of times we need to divide a number by 2 to reach 1
- $O(sqrtn)$: slower than $O(log_2 n)$, but faster than $O(n)$
- $O(n)$: simple for loop
- $O(nlogn)$: the cost of the fastest comparison-based sorting algorithm
- $O(n^2)$: 2 nested loops
- $O(n^3)$: 3 nested loops
- $O(2^n)$: That's a very big function, the cost of iterating through the subsets of a set of size n.
- $O(n!)$: monstrous in size.

Why finding complexity is important:

In a contest, your solution should produce an output under a certain time limit usually 1 or 2 seconds. A typical computer is capable of performing around $10^7 - 10^8$ operations per second, hence the complexity of you algorithm can give you an idea whether your solution would pass the time limit or not. This is why it is extremely important to check the constraint of a problem before attempting to solve it so that Here is a table showing for each complexity, the upper bound of size of input which allows it to pass the time limit:

Input Size	Maximum Possible Complexity
$n <= 10$	$O(n!)$
$n <= 20$	$O(2^n)$
$n <= 500$	$O(n^3)$
$n <= 500$	$O(n^2)$
$n <= 10^6$	$O(nlogn)$ or $O(n()$
$n > 10^6$	$O(logn)$ or $O(1)$

Example Problem: Maximum Subarray Sum

Problem Definition:

Given an array of n numbers, the task is to compute the maximum sub-array sum, i.e. The largest possible sum of a sequence of consecutive values in the array. **NOTE**: there can be positive and negative numbers. Also, we assume that the empty subarray is allowed, that is, the minimum sum is at least zero.

We will not impose any constraints now, but we will think of easy solutions and try to improve those.

Example: [-1,2,4,-3,5,2,-5,2]

Algorithm 1:

Go over all possible sub-arrays, and compute the sum of each sub-array, and take the maximum value. A sub-array is defined by its left and right bounds. Hence we can make an outer loop to try every possible left bound, and for each left bound, we can use another loop to go over all possible right bounds. After fixing the left and right bounds, we can use a third for loop inside to go over the elements from left bound to right bound and compute their sum. The code would look like this:

Algorithm 7: Maximum Subarray Sum 1

```
int best = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int sum = 0;
        for (int k = a; k <= b; k++) {
            sum += array[k];
        }
        best = max(best,sum);
    }
}
cout << best << "\n";
```

The complexity of this code is $O(n^3)$, because we have 3 loops nested inside each other, and each loop approximately runs n times. Another way to reason about it would be that there are $\frac{n(n-1)}{2}$ ways to choose a left bound and upper bound, which is $O(n^2)$, and computing the sum from left to right will consider all possible lengths in the range $[1, n]$, so $O(n)$ on average. Hence overall we have $O(n^3)$. This algorithm works for any array of size $n \leq 500$

Algorithm 2:

In order to improve the previous algorithm, we need to find a way to reduce the number of loops inside each other. One thing we could do is instead of having a separate loop for computing the sum, we could compute the sum while we are moving from left bound to reach the right bound (in the second loop). The code would be reduced to:

Algorithm 8: Maximum Subarray Sum 1

```
int best = 0;
for (int a = 0; a < n; a++) {
    int sum = 0;
    for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best,sum);
    }
}
cout << best << "\n";
```

As you can see from the code, we only have 2 loops nested, hence we have an $O(n^2)$ solution. This algorithm works for input of size $n \leq 5000$

Algorithm 3:

Surprisingly, there exists a way to solve this problem in $O(n)$!. It is a known algorithm called **Kadane's Algorithm**. It's idea depends on an observation that is not quite trivial to spot. The observation is as follows: Every element can either be a part of an ongoing sub-array or a beginning of a new one. It is always optimal to add the current number to an ongoing sub-array as long as it does not make the sum negative. If the adding an element to a sub-array makes the sum of the sub-array negative, then we can start a new sub-array with the next element and keep accumulating its sum until we either reach the end of the sub-array, or start a new sub-array because of reaching a negative sum. We will not provide a formal proof, many can be found online, but intuitively speaking, when reaching a negative sub-array sum, imagine we continue adding elements to it, it should make sense that cutting off this sub-array which is located in the prefix (beginning) and has negative sum, will only increase the sum of the rest of the sub-array (assuming we continue adding elements beyond the point where we got a negative sum). You may find the code below.

Algorithm 9: Maximum Subarray Sum 3 (Kadane's Algorithm)

```
int best_sum = 0;
int cur_sum = 0;  // current sum
for(int i = 0 ; i < n ; ++i){
    cur += array[i];
    best = max(best,cur);
    // in this part we are implicitly starting a new sub-array
    // because cur will be set to zero if it was negative.
    cur = max(cur,0);
}
```

Basic Maths:

Two of the most important mathematical algorithms are primality testing and GCD. They are usually involved not as an independent problem but you need to use them to solve a bigger problem.

Primality Testing:

Problem is defined as follows: Given an integer n as input, check whether n is a prime number or not. Remember that a prime number is a number that is only divisible on itself and 1.

The brute force algorithm is to check whether any number in the range $[2, n-1]$ divides the number n. A check can be done using the modulo operation: if $i\%n = 0$, then i divides n. So this costs $O(n)$.

A faster way would be to check only numbers in the range $[2, \sqrt{n}]$, because if you check any number $i \in [2, \sqrt{n}]$, then you automatically check $\frac{n}{i} \in [\sqrt{n}+1, n]$. Hence the complexity of this algorithm is $O(\sqrt{n})$.

Algorithm 10: Primality Test

```
bool Is_Prime(int n){
    for(int i = 2 ; i * i <= n ; ++i){
        if(n % i == 0) return false;
    }
    return true;
}
```

GCD:

GCD stands for Greatest Common Divisor. The GCD of two integers a and b is defined as the greatest number that divides both a and b. More formally, $GCD(a, b) = c$ is the maximum c such that $a\%c = 0$ and $b\%c = 0$.

The fastest way to compute GCD is called the *Euclidean Algorithm*, which depends on the fact that:

$$GCD(a, b) = GCD(b, a\%b)$$

. It's implementation is just that simple:

Algorithm 11: Euclidean Algorithm

```
int GCD(int x,int y){
    if(y == 0) return x
    return GCD(y,x % y);
}
```

The complexity of this algorithm is $O(log(max(a, b)))$. We will not provide a formal proof for this, but for convenience, here is a reasoning that would make it feel intuitive. Observe that for each call of $GCD(a, b)$, the maximum of a and b, is being reduced to at least half of it. Without loss of generality, assume that $a > b$. There are two mutually exclusive cases for the relationship between a and b: either $b > \frac{a}{2}$, or $b \leq \frac{a}{2}$.

When $b > \frac{a}{2}$, the value $a\%b$ cannot be greater than $\frac{a}{2}$, because in this case $a\%b = a - b$. We know that:

$$b > \frac{a}{2}$$
$$-b \leq -\frac{a}{2}$$
$$a - b \leq \frac{a}{2}$$
$$a\%b \leq \frac{a}{2}$$

Hence you can see that it is at most half of the original value.

The other case can be found in an analogous method and is left as an exercise for the reader. (The implicit message being that I'm too lazy to write it :P, but seriously though if someone felt that it should be added please contact me :D).

References:

- Competitive Programmer's Handbook by Antti Laaksonen