Competitive Programming and Mathematics Society

Intro to Competitive Programming

And Rocking Coding Interviews

Kyle and Freddie

Attendance

Table of contents

Welcome!

- We will run programming workshops every three weeks throughout Term 1 (weeks 1, 4, 7).
- Each workshop will last for approximately two hours.

Welcome!

- We will run programming workshops every three weeks throughout Term 1 (weeks 1, 4, 7).
- Each workshop will last for approximately two hours.
- This workshop in particular is more suited towards beginners and those who have never heard of competitive programming before (workshops will cater to a variety of audiences throughout the year).

Welcome!

- We will run programming workshops every three weeks throughout Term 1 (weeks 1, 4, 7).
- Each workshop will last for approximately two hours.
- This workshop in particular is more suited towards beginners and those who have never heard of competitive programming before (workshops will cater to a variety of audiences throughout the year).
- Please feel free to ask questions at any time.
- Slides will be uploaded to unswcpmsoc.com
- Pizza at the end!

What Is Competitive Programming?

Competitive Programming

Competitive programming is an activity where participants solve algorithmic problems within a fixed timeframe, aiming for efficient solutions.

What Is Competitive Programming?

Competitive Programming

Competitive programming is an activity where participants solve algorithmic problems within a fixed timeframe, aiming for efficient solutions.

In most competitive programming problems, you will be provided with a problem statement which contains

- Flavour Text (Problem Description)
- Constraints
- Input and Output Format

Let’s Read a Problem

Did you know that 2024 is the year of the dragon? In fact, any year which is 8 more than a multiple of 12 is the year of the dragon. Given a positive integer, determine whether it is the year of the dragon.

- **Input:** N $(1 \leq N \leq 100\,000)$
- **Output:** "YES" if N is the year of the dragon, and "NO" otherwise.

Let’s Read a Problem

Did you know that 2024 is the year of the dragon? In fact, any year which is 8 more than a multiple of 12 is the year of the dragon. Given a positive integer, determine whether it is the year of the dragon.

- **Input:** N $(1 \leq N \leq 100\,000)$
- **Output:** "YES" if N is the year of the dragon, and "NO" otherwise.
- **Solution:** We output "YES" if $12 \mid (2024 - 8)$, and "NO if $12 \nmid (2024 - 8)$,

The Importance of Time Complexity

As competitive programmers, we don't only care about whether our program can give us a 'correct' answer, but we also care equally about how 'fast' our program runs.

- The **time complexity** of an algorithm estimates how much time the algorithm will use for some input. The idea is to represent the efficiency as a function whose input is the size of the input.

The Importance of Time Complexity

As competitive programmers, we don’t only care about whether our program can give us a ’correct’ answer, but we also care equally about how ’fast’ our program runs.

- The **time complexity** of an algorithm estimates how much time the algorithm will use for some input. The idea is to represent the efficiency as a function whose input is the size of the input.
- Complexity is an upper bound for the number of steps an algorithm requires as a function of the input size.

The Importance of Time Complexity

As competitive programmers, we don't only care about whether our program can give us a 'correct' answer, but we also care equally about how 'fast' our program runs.

- The **time complexity** of an algorithm estimates how much time the algorithm will use for some input. The idea is to represent the efficiency as a function whose input is the size of the input.
- Complexity is an upper bound for the number of steps an algorithm requires as a function of the input size.
- We can denote time complexity with big O notation. For example, if we add up all elements in an N elements array, that would take $O(N)$ steps.

The Importance of Time Complexity

As competitive programmers, we don't only care about whether our program can give us a 'correct' answer, but we also care equally about how 'fast' our program runs.

- The **time complexity** of an algorithm estimates how much time the algorithm will use for some input. The idea is to represent the efficiency as a function whose input is the size of the input.
- Complexity is an upper bound for the number of steps an algorithm requires as a function of the input size.
- We can denote time complexity with big O notation. For example, if we add up all elements in an N elements array, that would take $O(N)$ steps.
- We usually care about the time complexity in the worst case.

How many multiples of 5?

We are given a list of N numbers, how many numbers are divisible by 5?

What should we do?

How many multiples of 5?

We are given a list of N numbers, how many numbers are divisible by 5?

What should we do?

Answer:

- Go through the N element array and check if each element is divisible by 5.

What is our input size?

How many multiples of 5?

We are given a list of N numbers, how many numbers are divisible by 5?

What should we do?

Answer:

- Go through the N element array and check if each element is divisible by 5.

What is our input size?

- N, as we have an N element array.

How many multiples of 5?

We are given a list of N numbers, how many numbers are divisible by 5?

What should we do?

Answer:

- Go through the N element array and check if each element is divisible by 5.

What is our input size?

- N, as we have an N element array.

So what is the time complexity?

How many multiples of 5?

We are given a list of N numbers, how many numbers are divisible by 5?

What should we do?

Answer:

- Go through the N element array and check if each element is divisible by 5.

What is our input size?

- N, as we have an N element array.

So what is the time complexity?

- $O(N)$, as we simply apply the modulus operation to each element.

How many multiples of 5? Continued

We are given an array A of N numbers, how many pairs of different elements exist such that their sum is divisible by 5?

What should we do?

How many multiples of 5? Continued

We are given an array A of N numbers, how many pairs of different elements exist such that their sum is divisible by 5?

What should we do?

Answer:

- Iterate through all pairs in the set $\{(x, y) : x \in A, y \in A\}$ and check if $5 \mid (x + y)$

What is our input size?

How many multiples of 5? Continued

We are given an array A of N numbers, how many pairs of different elements exist such that their sum is divisible by 5?

What should we do?

Answer:

- Iterate through all pairs in the set $\{(x, y) : x \in A, y \in A\}$ and check if $5 \mid (x + y)$

What is our input size?

- N, as we have an N element array.

How many multiples of 5? Continued

We are given an array A of N numbers, how many pairs of different elements exist such that their sum is divisible by 5?

What should we do?

Answer:

- Iterate through all pairs in the set $\{(x, y) : x \in A, y \in A\}$ and check if $5 \mid (x + y)$

What is our input size?

- N, as we have an N element array.

So what is the time complexity?

How many multiples of 5? Continued

We are given an array A of N numbers, how many pairs of different elements exist such that their sum is divisible by 5?

What should we do?

Answer:

- Iterate through all pairs in the set $\{(x, y) : x \in A, y \in A\}$ and check if $5 \mid (x + y)$

What is our input size?

- N, as we have an N element array.

So what is the time complexity?

- $O(N^2)$, as we are now iterating through all pairs, and since there are $\binom{N}{2}$ pairs, the time complexity is $O(\frac{N\cdot(N-1)}{2})$. Since we only care about the dominating term, this is simply equivalent to $O(N^2)$

Let's Play a Game

I am thinking of an integer A, where $1 \leq A \leq 1000$. Try to guess my number in Q tries.

- When $Q = 1000$

Let’s Play a Game

I am thinking of an integer A, where $1 \leq A \leq 1000$. Try to guess my number in Q tries.

- When $Q = 1000$
- When $Q = 11$

Let's Play a Game

I am thinking of an integer A, where $1 \leq A \leq 1000$. Try to guess my number in Q tries.

- When $Q = 1000$
- When $Q = 11$

Solution:

- The best strategy is to pick the midpoint every time
- The number of remaining options halves each time
- We only need to guess at most $\log_2 Q$ times.

So what is the time complexity?

Let's Play a Game

I am thinking of an integer A, where $1 \le A \le 1000$. Try to guess my number in Q tries.

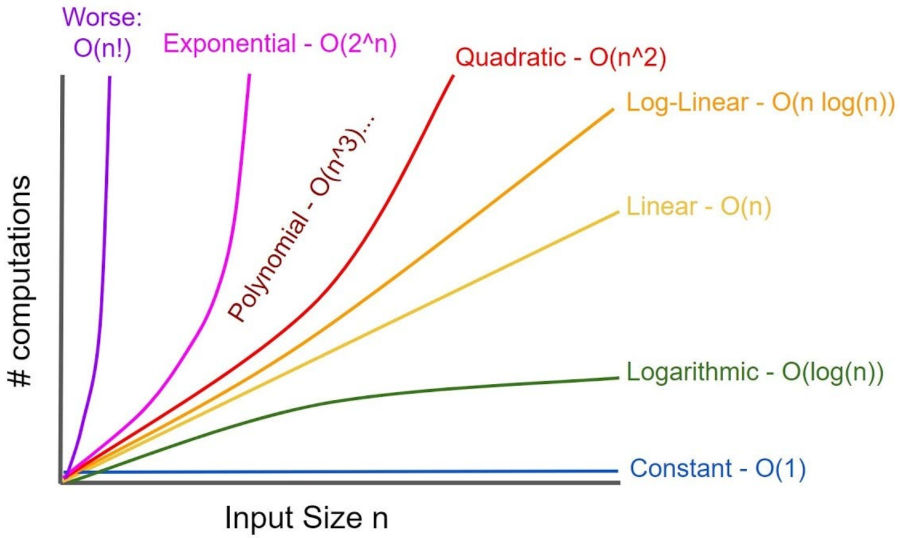
- When $Q = 1000$
- When $Q = 11$

Solution:

- The best strategy is to pick the midpoint every time
- The number of remaining options halves each time
- We only need to guess at most $\log_2 Q$ times.

So what is the time complexity?

- $O(\log_2 N)$, as every try we make, we are reducing our sample space by half. This is what we call a logarithmic complexity.


Worse: O(n!)
Exponential - O(2^n)
Quadratic - O(n^2)
Polynomial - O(n^3)...
Log-Linear - O(n log(n))
Linear - O(n)
Logarithmic - O(log(n))
Constant - O(1)
# computations
Input Size n

Relevance to Coding Interviews

- Gets you super comfortable creating efficient solutions under time pressure

Relevance to Coding Interviews

- Gets you super comfortable creating efficient solutions under time pressure

Steps for Acing Coding Interviews:

Relevance to Coding Interviews

- Gets you super comfortable creating efficient solutions under time pressure

Steps for Acing Coding Interviews:

1. Understand your task. Ask clarifying questions. Eg.
    - 'Can we assume the input is valid'?
    - 'Do we need to consider this extreme case [...]'?

Relevance to Coding Interviews

- Gets you super comfortable creating efficient solutions under time pressure

Steps for Acing Coding Interviews:

1. Understand your task. Ask clarifying questions. Eg.
   - 'Can we assume the input is valid'?
   - 'Do we need to consider this extreme case [...]'?
2. Only begin to code once you fully appreciate the question
   - Explain the purpose of your code at a higher-level (ie. the big picture, and **NOT** word for word)
   - Refine your solution to be more efficient as you go
   - *Consistently* keep your code easy to read and well-commented

Relevance to Coding Interviews

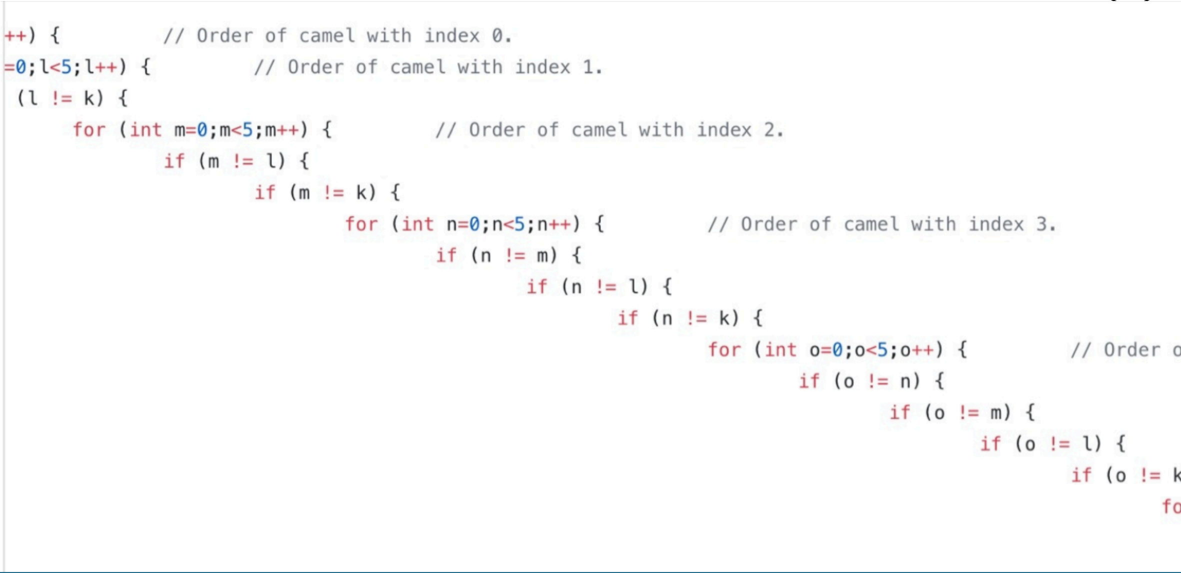
- Gets you super comfortable creating efficient solutions under time pressure

Steps for Acing Coding Interviews:

1. Understand your task. Ask clarifying questions. Eg.
   - 'Can we assume the input is valid'?
   - 'Do we need to consider this extreme case [...]'?
2. Only begin to code once you fully appreciate the question
   - Explain the purpose of your code at a higher-level (ie. the big picture, and **NOT** word for word)
   - Refine your solution to be more efficient as you go
   - *Consistently* keep your code easy to read and well-commented
3. Clean up syntax and readability
   - Ensure clear variable names, no overdeep nesting, and clear commenting to explain more sophisticated logic

However...

However...



```
++) {          // Order of camel with index 0.
=0;l<5;l++) {           // Order of camel with index 1.
 (l != k) {
     for (int m=0;m<5;m++) {          // Order of camel with index 2.
            if (m != l) {
                   if (m != k) {
                          for (int n=0;n<5;n++) {          // Order of camel with index 3.
                                 if (n != m) {
                                        if (n != l) {
                                               if (n != k) {
                                                      for (int o=0;o<5;o++) {          // Order o
                                                             if (o != n) {
                                                                    if (o != m) {
                                                                           if (o != l) {
                                                                                  if (o != k
                                                                                         fo
```

Mock Interview Time

Mock Interview Time

Take a look at the power of time complexity here: Say we are given an array of size n and we're tasked with creating another array, whose ith element must equal the average of all the elements up to index i in the first array.

Say our original array $= [1, 2, 3, 4, 5]$. Then our new array should be: [1, 1.5, 2, 2.5, 3.75].

Array Averages

```
void ComputeAverages(int *original, double *newArray, int numElements) {
    for (int i = 0; i < numElements; i++) {
        double current_average = 0;

        for (int j = 0; j <= i; j++) {
            current_average = current_average + original[j];
        }

        current_average = current_average / (i + 1);
        newArray[i] = current_average
    }
}
```

Array Averages

```
void ComputeAverages(int *original, double *newArray, int numElements) {
    for (int i = 0; i < numElements; i++) {
        double current_average = 0;

        for (int j = 0; j <= i; j++) {
            current_average = current_average + original[j];
        }

        current_average = current_average / (i + 1);
        newArray[i] = current_average
    }
}
```

Time complexity = $1 + 2 + 3 + ... + n = \frac{n}{2} \cdot (n + 1) = O(n^2)$

We Can Do Better!

We Can Do Better!

```
void ComputeAverages(int *original, double *newArray, int numElements) {
    int running_total = 0;

    for (int i = 0; i < numElements; i++) {
        running_total += original[i];

        double current_average = running_total / (i + 1);
        newArray[i] = current_average
    }
}
```

We Can Do Better!

```
void ComputeAverages(int *original, double *newArray, int numElements) {
    int running_total = 0;

    for (int i = 0; i < numElements; i++) {
        running_total += original[i];

        double current_average = running_total / (i + 1);
        newArray[i] = current_average
    }
}
```

Time complexity = $O(n)$, as we avoid the nested loop

It’s Your Turn!

It’s Your Turn!

https://leetcode.com/problems/search-insert-position/

Maximum Contiguous Subarray Sum

Given an array of integers of length N, your task is to find the contiguous subarray (sequence of elements within an array that are adjacent to each other) that has the largest sum and print that sum.

If our given array is $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$, then the output should be 6 which corresponds to the sum of the contiguous subarray $[4, -1, 2, 1]$.

Maximum Contiguous Subarray Sum

Given an array of integers of length N, your task is to find the contiguous subarray (sequence of elements within an array that are adjacent to each other) that has the largest sum and print that sum.

If our given array is $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$, then the output should be 6 which corresponds to the sum of the contiguous subarray $[4, -1, 2, 1]$.

- Have a go at thinking about an $O(N^3)$ solution!

Maximum Contiguous Subarray Sum

Observation:

- There are only $\binom{N}{2} + N$ valid contiguous subarrays.

Maximum Contiguous Subarray Sum

Observation:

- There are only $\binom{N}{2} + N$ valid contiguous subarrays.
- Each subarray will take in the worst case, $O(N)$ operations to calculate its sum.

Maximum Contiguous Subarray Sum

Observation:

- There are only $\binom{N}{2} + N$ valid contiguous subarrays.
- Each subarray will take in the worst case, $O(N)$ operations to calculate its sum.
- Thus, the total time complexity would be $O(N^3)$. As there are roughly $O(N^2)$ valid contiguous subarrays and every subarray takes $O(N)$ to summate, yielding a time complexity of $O(N^3)$.

Maximum Contiguous Subarray Sum

```
// N = number of elements in array, arr = initialised array
int ans = -1e9;

for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
        // iterate through all valid intervals [i, j]

        int sum = 0;
        for (int k = i; k <= j; k++) {
            sum += arr[k];
        }
        ans = max(ans, sum);
    }
}

cout << ans;
```

Time complexity = $O(N^3)$, as we go through all $O(N^2)$ subarrays and then sum each one.

Maximum Contiguous Subarray Sum

How can we make our solution faster?

Maximum Contiguous Subarray Sum

How can we make our solution faster?

- Instead of calculating the sum of the subarray with another nested loop, we can instead calculate the sum as we iterate through j.

Maximum Contiguous Subarray Sum

```
// N = number of elements in array, arr = initialised array
int ans = -1e9;

for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = i; j < N; j++) {
        // iterate through all valid intervals [i, j]
        sum += arr[j];
        ans = max(ans, sum);
    }
}

cout << ans;
```

Time complexity = $O(N^2)$. The complexity of the inner j for loop is $O(N)$, and we run the j for loop a total of N times. So the total complexity if $O(N^2)$

Maximum Contiguous Subarray Sum

We can do even **better**!

Maximum Contiguous Subarray Sum

We can do even **better**!

- Hint: We can reuse our previous result, if we are currently at position i, we can make use of the maximum contiguous subarray sum ending at position $i - 1$.

Maximum Contiguous Subarray Sum

We can do even **better**!

- Hint: We can reuse our previous result, if we are currently at position i, we can make use of the maximum contiguous subarray sum ending at position $i-1$.
- If the maximum contiguous subarray sum ending at position $i-1$ is positive, we can simply add on our current number to form an even longer contiguous subarray.

Maximum Contiguous Subarray Sum

We can do even **better**!

- Hint: We can reuse our previous result, if we are currently at position i, we can make use of the maximum contiguous subarray sum ending at position $i - 1$.
- If the maximum contiguous subarray sum ending at position $i - 1$ is positive, we can simply add on our current number to form an even longer contiguous subarray.
- If the maximum contiguous subarray sum ending at position $i - 1$ is negative, we just start a new contiguous subarray at position i and reset the sum variable.

Maximum Contiguous Subarray Sum

```
// N = number of elements in array, arr = initialised array
int max_ending_here = arr[0];
int ans = arr[0];

for (int i = 1; i < n; i++) {
    max_ending_here = max(max_ending_here + arr[i], arr[i]);
    ans = max(ans, max_ending_here);
}

cout << ans;
```

Time complexity = $O(N)$ as it's one simple for loop!

Longest Increasing Subsequence

You are given an N element array of integers. Your task is to find the length of the longest increasing subsequence (LIS) within the array.

An increasing subsequence is a sequence of numbers in the array where each number is greater than the previous number. However, the numbers in the subsequence do not have to appear consecutively in the array.

For example, given the array [10, 9, 2, 5, 3, 7, 101, 18], the longest increasing subsequence is [2, 3, 7, 101] with a length of 4.

Longest Increasing Subsequence

You are given an N element array of integers. Your task is to find the length of the longest increasing subsequence (LIS) within the array.

An increasing subsequence is a sequence of numbers in the array where each number is greater than the previous number. However, the numbers in the subsequence do not have to appear consecutively in the array.

For example, given the array [10, 9, 2, 5, 3, 7, 101, 18], the longest increasing subsequence is [2, 3, 7, 101] with a length of 4.

- Have a go at thinking about an $O(N^2)$ solution.

Longest Increasing Subsequence

Observation:

- Hint: Very similar idea to the $O(N)$ solution to Maximum Contiguous Subarray Sum.

Longest Increasing Subsequence

Observation:

- Hint: Very similar idea to the $O(N)$ solution to Maximum Contiguous Subarray Sum.
- The idea we used before the reduce time complexity was to re-use previous results. Let ans$[i]$ store the length of the LIS ending at index i in the array. Can you now see a $O(N^2)$ solution?

Longest Increasing Subsequence

Observation:

- Hint: Very similar idea to the $O(N)$ solution to Maximum Contiguous Subarray Sum.
- The idea we used before the reduce time complexity was to re-use previous results. Let ans[i] store the length of the LIS ending at index i in the array. Can you now see a $O(N^2)$ solution?
- We go through the array once, and if we are currently at index i in the array, we check all index j such that $arr[j] \leq arr[i]$. If $j \leq i$, we know we can extend the LIS ending at j such that it now ends at i.

Longest Increasing Subsequence

Observation:

- Hint: Very similar idea to the $O(N)$ solution to Maximum Contiguous Subarray Sum.
- The idea we used before the reduce time complexity was to re-use previous results. Let ans[i] store the length of the LIS ending at index i in the array. Can you now see a $O(N^2)$ solution?
- We go through the array once, and if we are currently at index i in the array, we check all index j such that $arr[j] \leq arr[i]$. If $j \leq i$, we know we can extend the LIS ending at j such that it now ends at i.
- More formally, if $arr[j] \leq arr[i]$, we can update $ans[i]$ such that $ans[i] = max(ans[i], ans[j] + 1)$.

Longest Increasing Subsequence

```
// initialise the answer array
for (int i = 0; i < N; i++) answer_arr[i] = 1;

for (int i = 1; i < N; i++) {
    for (int j = 0; j < i; j++) {
        if (arr[j] < arr[i]) {
            answer_arr[i] = max(answer_arr[i], answer_arr[j] + 1);
        }
    }
}

int final_ans = 0;
for (int i = 0; i < N; i++) {
    final_ans = max(final_ans, answer_arr[i]);
}
cout << final_ans;
```

Time complexity = $O(N^2)$ as we have two nested loops.

Our Parting Words

Our Parting Words

- Focus on 2 main things when practising for interviews:
  - Efficient solutions
  - Readable and maintainable code

Our Parting Words

- Focus on 2 main things when practising for interviews:
    - Efficient solutions
    - Readable and maintainable code
- Go out there on Leetcode, CodeForces, there's so much out there for you to explore!

Our Parting Words

- Focus on 2 main things when practising for interviews:
  - Efficient solutions
  - Readable and maintainable code
- Go out there on Leetcode, CodeForces, there’s so much out there for you to explore!
- Join our subcommittee!