COMPUTER SCIENCE CLUB

Intermediate Stream Textbook

By: Gurjas Dhillon
Intermediate Teacher at Vincent Massey Secondary School
Windsor, Ontario
August 28th, 2023

Contents

Chapter 1

Getting Started with C++

1.1 Installing C++

Before we are actually able to write any C++ code, we need to first install the C++ compiler and optionally a text editor/integrated development environment (IDE). Visual Studio Code stands out as the most popular IDE for programmers, but CLion is a lot more light-weight in terms of getting straight into the code after installing the IDE. Installation vary depending on your operating system.

MacOS

Installing the C++ compiler on MacOS is very straightforward. All you do is put command **xcode-select –install** into the terminal and go through the installer. You can use the command **clang – version** to verify everything was installed properly. The only drawback of clang++ is that it does not include the `<bits/stdc++.h>` file. We'll talk about this more in the next section.

Windows

Follow this article to install the G++ compiler: `https://www.freecodecamp.org/news/how-to-install-c-and-cpp-compiler-on-windows/`

IDE Installation

Visual Studio Code: `https://code.visualstudio.com/Download`
CLion Student Account: `https://www.jetbrains.com/shop/eform/students`
CLion: `https://www.jetbrains.com/clion/download/`

1.2 Introduction to C++

Note 1.2.1. For the first few sessions, we'll be using Repl.it

In order to get C++ running, you need to understand the boilerplate code.

Listing 1.1: Basic C++ Syntax

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

int main() {
    // print "Hello, World!"
    cout << "Hello, World!" << endl;
    return 0;
}
```

The **bits/stdc++.h** is a feature of the G++ compiler. It is a header file that includes most of the standard C++ libraries into one file. This is especially useful for competitive programming since time is a huge factor when writing a contest, and this one line of code simplifies our code ten-folds. Sadly, this is only a feature for the G++ compiler. So, anyone who uses MacOS will have to write all the libraries directly. Not to worry though! While we are going through all the data structures and the ins and outs of the language, I will also include all the libraries required to use their full power.

The **using** line allows us to use all the functions and classes in the C++ standard library directly in our code. Without it, we would have had to write **std::cout**.

int main() is a C++ function (more on that later). This is where all the code will be executed.

// print "Hello, World!" is an inline comment (indicated by the "//"). It is used to personal notes when programs get very complex and to explain your code when given to someone else. Multi line comments can be used by **/* comments placed here */**.

The **cout** (pronounced see-out) line is outputting information to the terminal. **endl** (pronounced end line) is equivalent to the new line character.

Note 1.2.2. Do not forget your semi-colons! Everything in C++ ends with them;

Practice Problem

Hello, World!
DMOJ Link: https://dmoj.ca/problem/helloworld

1.3 Basic Syntax

1.3.1 Variables

In the world of programming, variables serve as containers for holding and manipulating data. Think of them as labeled boxes that store information, allowing us to perform calculations, make decisions, and create dynamic applications. In C++, variables are essential building blocks that enable us to work with different kinds of data, ranging from numbers and text to more complex structures. Unlike Python, you need indicate the data type when declaring the variable. This is the case for all mid-level programming languages. To create a variable, you need to first declare the data type, give the variable a name, and finally assign it a value. E.g. `int x = 5;`

One big property of C++ variables are their **mutability**. In C++, variables are dynamic in the manner that they can be re-assigned after initial declaration. C++ variables are mutable by default, but it is possible to change this my adding the keyword **const** (constant). E.g. `const int MAXSZ = 1e6 + 2;`

Listing 1.2: Variables

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

int x = 2;
int y;
int main() {
    cout << "The value of x is : "<< x << endl; /* Outputs: 2 */
    x = 5;
    cout << "The value of x is : "<< x << endl; /* Outputs: 5 */
    y = x + 2;
    cout << "The value of y is : "<< y << endl; /* Outputs: 7 */
}
```

Note 1.3.1. 1e6 + 2 is equivalent to $10^6 + 2$ in the math world while 4e6 + 2 is equivalent to $4 \cdot 10^6 + 2$

Principle 1.3.2. To re-assign a variable, you don't need to include

Principle 1.3.3. It is convention to name a constant in ALL CAPS!

1.3.2 Data Types

Data types are fundamental constructs that decide what a variable can hold. Once a variable is given a data type, it is impossible to change the type. Each data type has it's own memory size, and provides a range of operations that can be performed on the associated values. The most basic data types are known as **Primitive Data Types**. These types include but are

not limited to integers (int), decimals (float - floating point numbers), individual characters (char), and boolean values (bool: true or false). Then, there are **Derived Data Types** where you use primitive types to manage more complex **data structures** and organize collections of data. The last kind are called **User-defined Data Types** where you create your own data type using **classes** and **structs**. We will be mainly focusing on Primitive and Derived in this course.

Data Type	Range	Memory Size (bytes)	Description
`bool`	`true` or `false`	1	Boolean
`char`	-128 to 127	1	Character
`short`	-32,768 to 32,767	2	Short Integer
`int`	-2^{31} to $2^{31}-1$	4	Integer
`long long`	-2^{63} to $2^{63}-1$	8	Long Long Integer
`float`	6 decimal digits	4	Single-Precision Floating num
`double`	15 decimal digits	8	Double-Precision Floating num
`long double`	18 decimal digits	12	Extended Precision Floating num

Table 1.1: C++ Primitive Data Types, Ranges, and Memory Sizes

Principle 1.3.4. Auto - auto is a keyword that automatically infers the type from the value. `auto x = "hello";`

Principle 1.3.5. A char is a single character that must be wrapped in **single quotes**. `char x = 'a';`

Listing 1.3: Types

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

int main() {
    int thisIsInt = 5;
    cout << "This is an integer: " << thisIsInt << endl;

    long long thisIsLongLong = 9999999999LL;
    cout << "This is a long long: " << thisIsLongLong << endl;

    char thisIsChar = '@';
    cout << "This is a char: " << thisIsChar << endl;

    string thisIsString = "Vincent Massey";
    cout << "This is a string: " << thisIsString << endl;

    bool thisIsTrue = true;
    bool thisIsFalse = false;
    cout << "These are both bools: " << thisIsTrue << " and " <<
        thisIsFalse << endl;
```

```
}
```

Note 1.3.6. A long long always ends with the LL suffix to distinguish between int and long long.

Note 1.3.7. A string is not a primitive data type! It is a class apart of the C++ Standard Library. It is created by combining a series of chars, thus be called a **derived data type**. Import the `<string>` header file to use it.

1.3.3 Operators

Operators are symbols that perform operations on variables and values. They are the fundamental building blocks for all programming languages. They are used to calculation, comparisons, and assignments. There are six main operators: **Arithmetic Operators**, **Logical Operators**, **Comparison Operators**, **Assignment Operators**, **Increment Operators**, and finally, **Ternary Operators**.

Arithmetic Operators

Just like the name suggests, arithmetic operators perform arithmetic operations on variables and values.

Operator	Operation
`+`	addition
`-`	subtraction
`/`	division
`*`	multiplication
`%`	modulo (remainder after division)

Table 1.2: C++ Primitive Data Types, Ranges, and Memory Sizes

Note 1.3.8. Unlike python, there is no operator for exponent. You have to use the **pow()** function from `<cmath>`. E.g. `pow(2, 3)` $= 2^3$

Listing 1.4: Arithmetic Operators

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

int x = 7, y = 3;

int main() {
    cout << "x + y = " << x + y << endl; /* Output: 10 */
    cout << "x - y = " << x - y << endl; /* Output: 4 */
    cout << "x * y = " << x * y << endl; /* Output: 21 */
```

```
    cout << "x / y = " << x / y << endl; /* Output: 2 */
    cout << "x % y = " << x % y << endl; /* Output: 1 */
    return 0;
}
```

Principle 1.3.9. When you divide to integers in C++, the result will also return an integer by removing everything after the decimal. You can change this behaviour by type casting the numerator to a floating point number.

Exercise 1.3.10. Try playing around with the division operator! Understand the uniqueness of C++ and master this skill.

Comparison Operators

A comparison operator is used to check relationships between variables by returning boolean values. E.g. `x > y` is used to check if x is greater than y.

Operator	Meaning	Example
`>`	`Greater than`	`4 > 5` returns false
`<`	`Less than`	`4 < 5` returns true
`>=`	`Greater than or equal`	`4 >= 5` returns false
`<=`	`Less than or equal`	`5 <= 5` returns true
`==`	`is equal to`	`4 == 5` returns false
`!=`	`not equal to`	`4 != 5` returns true

Table 1.3: Comparison Operators

Assignment Operators

An assignment operator is used to assign values or change the values by a constant to a variable. The most simple operator for this category is `x = 5`.

Operator	Example	Equivalent
`=`	`x = 5`	`x = 5`
`+=`	`x += 5`	`x = x + 5`
`-=`	`x -= 5`	`x = x - 5`
`*=`	`x *= 5`	`x = x * 5`
`/=`	`x /= 5`	`x = x / 5`
`%=`	`x %= 5`	`x = x % 5`

Table 1.4: Assignment Operators

Logical Operators

Logical Operators are used to check if expressions are true or false.

Operator	Example	Meaning
&& (AND)	expression1 && expression2	true if both expressions are true
|| (OR)	expression1 || expression2	true if at least one expression is true
! (NOT)	!expression1	true if expression is false

Table 1.5: Logical Operators

Principle 1.3.11. The logical NOT operator flips the boolean value!

1.3.4 Input/Output

Input

In C++, cout is for outputting data, while cin is used to input data. If the input is separated with spaces or newline characters, you would get input by...

Listing 1.5: Input

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

string name;
int age;

int main(){
    cin >> name >> age;
    /*
    Input can be taken in two ways:
    Gurjas 16
    -----------------
    Gurjas
    16

    */
    return 0;
}
```

Output

You can then output their age in 1 year by...

Listing 1.6: Output

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

string name;
int age;
```

```
6
7  int main(){
8      cin >> name >> age;
9
10     cout << "Hey, " << name << "!" << endl;
11     cout << "You'll be " << age + 1 << " in one year!" << endl;
12     /*
13     Input: Gurjas 16
14     Output: Hey, Gurjas!
15             You'll be 17 in one year!
16     */
17     return 0;
18 }
```

Fast I/O

Somtimes, you may get TLE (Time Limit Exceeded) because standard I/O is too slow.

Listing 1.7: Fast I/O (New Template)

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      ios::sync_with_stdio(0); cin.tie(0);
6
7      /* Code here */
8      return 0;
9  }
```

Note 1.3.12. `<iostream>` - Standard input/output stream library

1.3.5 Problems - Operators and Input

A plus B!
DMOJ Link: https://dmoj.ca/problem/acc6p1

Squares
DMOJ Link: https://dmoj.ca/problem/ccc04j1

Next in line
DMOJ Link: https://dmoj.ca/problem/ccc13j1

Cupcakes
DMOJ Link: https://dmoj.ca/problem/ccc22j1

1.3.6 Conditionals

If statements

Operators are variables are cool, but not very versatile and pretty bland. Only if we could've executed code if some condition was true.... Not to worry, though! Since conditionals are here to save the day.

Comparison and logical operators are very powerful, but they are rarely ever used without conditionals. If statements, just like the name suggests, executes some part of the code if some condition is meant. Let's say we wanted to check if a number was odd, how would the code look for such a program?

Listing 1.8: Odd

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

int N;

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    cin >> N;

    if (N % 2 == 1) {
        cout << N << " is an odd number!" << endl;
    }

    return 0;
}
```

The **if** keyword indicates the start of a conditional statement. Then, in brackets, you have the condition in which you check if a number is odd. Finally, you place the code you want to execute in a pair of curly braces.

This is amazing! But, what if we want to also print something when N is even? Of course, we can just add another if statement, but we programmers are lazy. Rather then that, we can use something called an **else statement**. If the initial condition is **not** met, you execute this second piece of code.

Listing 1.9: Even/Odd

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

int N;

```

```
int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    cin >> N;

    if (N % 2 == 1) {
        cout << N << " is an odd number!" << endl;
    } else {
        cout << N << " is an even number!" << endl;
    }

    return 0;
}
```

The final scenario that we have to consider is when we have more than 2 factors to consider. In this program, there are only two cases to consider: odd or even. What if we want to check whether a number is positive, negative, or zero? In this case, we cannot just use if/else statements. We have to use something called **else if**.

Listing 1.10: Postive

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

int N;

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    cin >> N;

    if (N > 0) {
        cout << N << " is a postive integer." << endl;
    } else if (N < 0) {
        cout << N << " is a negative integer." << endl;
    } else {
        cout << N << " is 0!" << endl; /* 0 is 0 lol*/
    }

    return 0;
}
```

Ternary Operator

It's common knowledge that programmers hate to write more code than needed. If you only have two conditions for your conditional, instead of writing an if/else statement, you can use the full capability of ternary operators. E.g. `(condition ? expression1 : expression2 )` If the condition is true, expression1 will be returned, otherwise expression2 will be.

Let's write the same parity code, but with ternary operators!

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

```
3
4  int N;
5
6  int main() {
7      ios::sync_with_stdio(0); cin.tie(0);
8      cin >> N;
9
10     string result = (N % 2 == 1 ? "odd" : "even");
11     cout << cout << N << " is an " << result << " number!" << endl;
12
13     return 0;
14 }
```

1.3.7 Problems - Conditionals

Triangles
DMOJ Link: https://dmoj.ca/problem/ccc14j1

Boiling Water
DMOJ Link: https://dmoj.ca/problem/ccc21j1

Quadrants
DMOJ Link: https://dmoj.ca/problem/ccc17j1

Dog Treats
DMOJ Link: https://dmoj.ca/problem/ccc20j1

1.3.8 Loops

A lot of times in programming, we want to do some task multiple times. Of course, we can just copy paste our previous code multiply times, but this is not sustainable when you want the task done 10...100...100000x. You would be spamming ctrl-v until the contest time is over :skull:. This is where loops come in to save the day...literally!

For loops

Just like conditionals, for loops are the building blocks of any programming language. It's especially useful when you know the exact number of iterations you want to perform. A for loop is distinguished with the **for** keyword. The basic syntax of a for loop is...

Listing 1.11: For loop syntax

```
for (initialization ; condition ; increment/decrement) {
    // code put here
}
```

Let's say you want to find the average of the numbers from 1-100. How would you do that...?

Listing 1.12: Average from 1-100

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

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    double avg = 0;
    for (int i = 1; i <= 100; i++) {
        avg += i;
    }
    avg /= 100;

    cout << "The average of the numbers from 1-100 is " << avg <<
        endl;
}
```

While loops

In this C++ structure, a block of code will be executed again and again, until a specified condition becomes false.

Listing 1.13: While loop syntax

```
while (condition) {
    // code put here
}
```

The condition will be checked at the start of each iteration. If the condition returns false, the block of code will not be executed.

Let's say we want to create a safe! The user will keep inputting a password until he guesses the right password. There should be some prize when the safe is opened.

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

int passWord = 1234; // don't make this your password irl

int main() {
    ios::sync_with_stdio(0); cin.tie();

    int guess; cin >> guess;

```

```
    while (guess != passWord) {
        cout << "Wrong password. Try again!" << endl;
        cin >> password;
    }

    cout << "Here's your prize: $5" << endl;
}
```

It's also very common to do `while (true)`. This is useful when the conditions become more complex. Obviously, you can't just keep it just like that, since it will never stop executing the code. In this case, you can use the **break** keyword to break out of a loop in some sort of conditional.

Principle 1.3.13. When you don't include a break statement in a while true loop, you will run into something called a **infinite loop** in which case the program will crash. Always make sure there is a case where the program breaks out the while loop.

1.3.9 Problems - Loops

Silent Auction
DMOJ Link: https://dmoj.ca/problem/ccc21j2

Max flow
DMOJ Link: https://dmoj.ca/problem/acmtryouts0a

Epidemiology
DMOJ Link: https://dmoj.ca/problem/ccc20j2

Recruits
DMOJ Link: https://dmoj.ca/problem/vmss7wc15c2p1

1.3.10 Functions

Intro to functions

In ***Mathematics***, a function is a structure that takes in one or more arguments and returns a value after putting those arguments in an expression. Functions are also very common in programming but behave a bit differently. A function is a block of code that is only run

when called. They are very useful when you re-use the same code multiple times and for code organization. We have already seen one function; the `int main()` function.

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

void helloWorld() {
    cout << "Hello, World!" << endl;
}

int main() {
    helloWorld();
    helloWorld();
    helloWorld();
    return 0;
}
/*
Output:
Hello, World!
Hello, World!
Hello, World!
*/
```

Note 1.3.14. If a function is created after the main function, you will run into an error. All user-defined functions must be put above the main function.

Separating Function Declaration

Although, it is possible to separate the declaration and definition of the function.

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

void helloWorld();

int main() {
    helloWorld();
    return 0;
}

void helloWorld() {
    cout << "Hello, World!" << endl;
}

/*
Output:
Hello, World!
*/
```

Functions with parameters

You may be wondering, why would you use a function rather than a for loop? It's because functions can be made dynamic by passing it data as parameters. Parameters act like variables inside the function.

Listing 1.14: Function with parameters

```
void functionName(type parameterOne, type parameterTwo, type
    parameterThree){
    // code put here
}
```

Here's an example of a function using parameters

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

void greetings(string name);

int main() {
    string name; cin >> name;
    greetings(name);
    return 0;
}

void greetings(string name) {
    cout << "Hello, " << name << endl;
}
```

Functions with optional Parameter

You can also provide a default parameter if no argument is passed.

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

void greetings(string name = "Joe") {
    cout << "Hello, " << name << endl;
}

int main() {
    greetings("Jeff");
    greetings("Bob");
    greetings();
    return 0;
}
```

Functions with return

So far, we've defined functions using the `void` keyword. This indicated that our function won't have any `return value`. A return value allows you to send values back to the caller

of the function. It signifies the end of the function, and any code that precedes it **will not be executed**. Instead of using the `void` keyword, we will be using the type of the return value just like variables. Here's an example of a sum function:

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

int sum(int a, int b) {
    int result = a + b;
    return result;
}

int main() {
    int a = 5; int b = 2;
    int result = sum(a, b);
    cout << result << endl;
}
```

If anything was placed after the return statement, that code will be simply ignored.

Note 1.3.15. Changing the value of variables are known to be `local changes`. They won't affect the value of the actual variables passed. Playing around with this is left to the reader as an exercise.

Function with references

If you ever need to make the changes in a function `global`, they can be done by passing the parameter as a reference `&`.

Listing 1.15: Function with references

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

void myFunc(int& a) {
    a += 1;
}

int main() {
    int a = 5;
    myFunc(a);
    cout << a << endl; // a is now equivalent to 6
}
```

1.3.11 Arrays

Let's say you wanted to store the grades of 4 students. Just like we've seen in the past, the obvious answer is just to create four new variables to store the grades. But let's say our data set changes to 10...100...2000 students. You don't want to create that many variables.

You would much rather have another way where you can stay much more organized. This is where arrays come in. When declaring an array, we have to include the type of the variable it holds, a name for the array, and a size for the number of elements it stores using the syntax: `type name[size];`.

For example, let's say we want our array to store the numbers from 1-50. We'll call this array nums and give it a size of 50.

Listing 1.16: Array Example

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

int nums[50];

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    for (int i = 0; i < 50; i++) {
        nums[i] = i + 1;
        // example: i = 0 -> nums[i] = i + 1;
    }
}
```

Note 1.3.16. Arrays are accessed using square brackets such as `nums[index]`. Note that index's start counting from 0.

Principle 1.3.17. Be careful of accidentally accessing an index out of an array's range. This will cause erratic behaviour without the programming crashing.

Principle 1.3.18. It's always best to initialize arrays outside the main functions due to unexpected/weird behaviour.

You can alternatively initialize the array with the declaration.

Listing 1.17: Array initialization

```
char vowels[] = {'a', 'e', 'i', 'o', 'u'}; // size is automatically
    determined (only for direct initialization.
```

1.3.12 Summary

C++ is a complicated language and we threw quite a lot at you guys in the past few sections. But with practice, all this will become common knowledge. There's are many slick features and tricks that can only be found with practice. So, try the practice problems and read the editorials if you ever get stuck. Here are a few links to the official C++ documentation where you can find excessive articles on all the ins and outs of the language.

- `https://en.cppreference.com/w/`
- `http://www.cplusplus.com/reference/`

1.3.13 Homework Problems

Multiple Choice
DMOJ Link: https://dmoj.ca/problem/ccc11s2

1.4 C++ built-in data structures

1.4.1 Vector

1.4.2 Set

1.4.3 Map

1.5 Data Structures

1.5.1 Prefix Sum Arrays

Let's start with an exercise: Given a sequence and it's cumulative sum,

$$\begin{aligned} &\mathbf{1+2+5+3+9+10+56+8} \\ &\mathbf{=94} \end{aligned}$$

What is the sum after **16** is added?

$$\begin{aligned} &\mathbf{1+2+5+3+9+10+56+8+16} \\ &\mathbf{=?} \end{aligned}$$

Simple, right? All you do is add **16** to **94** and that's the new sum. You could have alternatively recalculated the entire sum, but of course that's very inefficient.

Now, let's try mapping this to the world of computer science:

Given a list of numbers, calculate the sum from the start of the list to position i. There will be Q different i's provided.

Index	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
a	5	1	3	4	6	3

For example:

N = 6 Q = 3
List: 5 1 3 4 6 3
Q1: 2
Q2: 4
Q3: 5

Naive Implementation

For each query, write a for loop and keep track of the cumulative sum. How can we make this solution faster?

Hint 1:

Use the same idea from the exercise to precompute the sums for each position.

Hint 1:

Imagine the numbers are placed on a number line. Prefix sum tells you the sum of all the numbers to the left of any position in that line, including the number at that position.

Index	Col 1	Col 2	Col 3	Col 4	Col 5
a	5	1	3	4	6
pref	a[1]	a[1]+a[2]	a[1] + a[2] + a[3]	a[1] + a[2] + a[3] + a[4]	a[1] + a[2] + a[3] + a[4] + a[5]
a	a[1]	pref[1] + a[2]	pref[2] + a[3]	pref[3] + a[4]	pref[4] + a[5]

Instructions:

- Create a list of numbers of the length N+1 (0-N inclusive).
- Loop through all the elements in the initial array. At each step, add the current number to the last number in your "prefix sum" list psa[i] = a[i] + psa[i - 1].
- After you're done, your "prefix sum" list is like a dictionary. If you want to know the sum of numbers from the start up to any position in your original line, just look at that position in the "prefix sum" list. It's the sum you're looking for!

Code:

Listing 1.18: Prefix Sum

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

int main() {
   ios::sync_with_stdio(0); cin.tie(0);
   int N; int Q;
   cin >> N >> Q;
   int arr[N], psa[N + 1];

   psa[0] = 0;

   for (int i = 0; i < N; ++i){
      cin >> arr[i];
      psa[i + 1] = psa[i] + arr[i];
```

```
    }
}
```

Now, let's try generalizing this algorithm. So far, the left position had to be at index 0 for this process to work. How can we alter the algorithm such that it can work **without** fixing the left position? For example, how can we find the sum from position 3 to 4?

Index	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**
a	5	1	3	4	6
pref	a[1]	a[1]+a[2]	a[1] + a[2] + a[3]	a[1] + a[2] + a[3] + a[4]	a[1] + a[2] + a[3] + a[4] + a[5]
a	a[1]	pref[1] + a[2]	pref[2] + a[3]	pref[3] + a[4]	pref[4] + a[5]

We would like to manipulate these values to get **a[3] + a[4]**. Clearly, some part of our answer lies in pref[4] = a[1] + a[2] + a[3] + a[4]. Ideally, we would like to remove a[1] + a[2]. We can accomplish that..... by subtracting out pref[2]=a[1]+a[2]! This would leave us with a[3] + a[4] which was the result we were looking for.

$$\textbf{arr[L]} + \textbf{arr[L + 1]} + ... + \textbf{arr[R - 1]} + \textbf{arr[R]} = \textbf{pref[R]} - \textbf{pref[L - 1]}$$

Practice Problems:

Deforestation
DMOJ Link: https://dmoj.ca/problem/dmopc14c2p4

Silver - GG
DMOJ Link: https://dmoj.ca/problem/vmss7wc16c2p2

Avocado Trees!
DMOJ Link: https://dmoj.ca/problem/avocadotrees

Marathon
DMOJ Link: https://dmoj.ca/problem/gfssoc2j4

1.5.2 Binary Search

Suppose you left your bike at the mall and it got stolen. You know it was stolen at some point on September 15, but you don't know the exact time. Luckily, the police have camera footage of the area, so you can check when the bike was stolen. How can you find the time of robbery?

Solution 1: Linear Search

You watch the footage entirely until the moment your bike gets stolen. At worst, you spend 24 hours watching. Can you do better?

Observation 1:

If your bike is gone, it won't be there after that point. For example, if your bike is not there at 9:00, it won't be there at 9:01 or 9:10 or 10:00. Therefore, as the day goes on, the number of bikes is non-increasing.

Binary search?

You can binary search on monotonic values. Monotonic just means it's only going up or only going down. For example, the stolen bike example is monotonic since the number of bikes stolen only goes up. To binary search, keep track of a range of values you have to check. Then, check the middle and use information to determine if you should check higher or lower than that.

Why was solution 2 much faster than solution 1? Because of its time complexity! Time complexity is a measure of the number of operations your program does. Big O is the worst case time complexity which is what we use in competitive programming because the test cases go to the worst case.

Time Complexity	**Word Name**	**Estimated operations with $N = 10^6$**
$O(1)$	Constant	1
$O(\log_2 N)$	Logarithmic	20
$O(N)$	Linear	1,000,000
$O(Nlog_2N)$	Linearithmic	20,000,000
$O(N^2)$	Quadratic	1,000,000,000,000
$O(N^3)$	Cubic	1,000,000,000,000,000,000
$O(2^N)$	Exponential	$2^{1,000,000}$
$O(N!)$	Factorial	1000000!

Assuming N is the number of minutes, solution 1 (watching the whole video) was at worst case O(N), linear. This is the case if the bike was stolen at 11:59 PM. Solution 2 was O(lg(N)). This is because each check, you cut the possible minutes in half.

Pseudocode:

General Method:

Listing 1.19: Linear Search

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x found at index i
    }
}
```

The time complexity of this approach is $O(n)$, because in the worst case, it is necessary to check all elements of the array. If the order of the elements is arbitrary, this is also the best possible approach, because there is no additional information available where in the array we should search for the element x.

However, if the array is sorted, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in the array guides the search. The following **binary search** algorithm efficiently searches for an element in a sorted array in $O(log_2 N)$ time.

Binary Search: The usual way to implement binary search resembles looking for a word in a dictionary. The search maintains an active region in the array, which initially contains all array elements. Then, a number of steps is performed, each of which halves the size of the region.

At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element. The above idea can be implemented as follows:

Listing 1.20: Binary Search

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

In this implementation, the active region is a...b, and initially the region is $0...n-1$. The algorithm halves teh size of the region at each step, so the time complexity is $O(log_2 n)$

Built-in Functions:

The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:

- `lower_bound` - returns a pointer to the first array element whose value is at least x.
- `upper_bound` - returns a pointer to the first array element whose value is larger than x.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element. For example, the following code finds out whether an array contains an element with value x:

Listing 1.21: Binary Search using Built-in function

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

Practice Problems

Uneven Sand
DMOJ Link: https://dmoj.ca/problem/seed2

Array Anger
DMOJ Link: https://dmoj.ca/problem/macs2p3

Zeros
DMOJ Link: https://dmoj.ca/problem/dmopc16c2p4