AN INTRODUCTION TO THE USA COMPUTING OLYMPIAD

C++ Edition

Darren Yao

June 5th, 2020

Foreword

This book was written as a comprehensive and up-to-date training resource for the USA Computing Olympiad. The goal was to create an "Art of Problem Solving" of sorts for the USACO: a one-stop-shop guide to prepare competitive programmers for the Bronze and Silver divisions of the USACO contests.

My primary motivation for writing this book was the struggle to find the right resources when I first started doing USACO contests. When I eventually reached the Platinum division, new competitors often asked me for help in structuring their competitive programming practice. Since I always found myself explaining that the USACO lacked comprehensive training resources, I decided to write this book.

I would like to thank a number of people for their contributions to this book. In particular, Michael Cao for writing sections 10.6 and 10.7 and helping with content revisions, Jason Chen for writing section 14.2 and extensive help with both content and LaTeX formatting, Stephanie Wu for section 4.1 and revisions, and Aaryan Prakash, Rishab Parthasarathy, and Kevin Wang for their valuable and constructive feedback on early draft versions of the book.

Furthermore, I'd like to thank Jason, Stephanie, and Rishab again for helping translate code into C++ for this edition of the book, on very short notice.

I'd also like to thank the USACO discord community for supporting me through my competitive programming journey; it was because of them that my competitive programming successes, and this book, are possible.

Contents

Part I

Basic Techniques

Chapter 1

The Beginning

1.1 Competitive Programming

Welcome to the world of competitive programming! If you've had some basic programming experience with C++ (perhaps at the level of an introductory course), and are interested in competitive programming, then this book is for you. (If your primary language is Java, we also have a Java edition of this book; please refer to that instead). If you currently do not know how to code, there are numerous resources available online to help you learn.

This book aims to guide you through your competitive programming journey by providing a framework in which to learn the important contest topics. From competitive programming, not only do you improve at programming, but you improve your problem-solving skills which will help you in other areas. If at any point you have questions, feedback, or notice any mistakes, please contact me at `darren.yao@gmail.com`. Best of luck, and enjoy the ride!

The goal of competitive programming is to write code to solve given problems quickly. These problems are not open problems; they are problems that are designed to be solved in the short timeframe of a contest, and have already been solved by the problem writer and testers. In general, each problem in competitive programming is solved by a two-step process: coming up with the algorithm, which involves problem solving skills and intuition, and implementing the algorithm, which requires programming skills to translate the algorithm into working code. The degree of mathematics knowledge varies from contest to contest, but generally the level of mathematics required is relatively elementary, and we will review important topics in this book.

A contest generally lasts for several hours, and consists of a set of problems. For each problem, when you complete your code code, you submit it to a grader, which checks the answers calculated by the your program against a set of predetermined test cases. For each problem, you are given a time limit and a memory limit that your program must satisfy. Grading varies between contests; sometimes there is partial credit for passing some cases, while other times grading is all-or-nothing. For those of you with experience in software development, note that competitive programming is quite different, as the goal is to write programs that compute the correct answer, run quickly, and can be implemented quickly. Note that nowhere was maintainability of code mentioned. This means that you should throw away everything you know about traditional code writing; you don't need to bother

documenting your code, because it only needs to be readable to you, during the contest.

1.2 Contests and Resources

The USA Computing Olympiad is a national programming competition that occurs four times a year, with December, January, February, and US Open contests. The regular contests are four hours long, and the US Open is five hours long. Each contest contains three problems. Solutions are evaluated and scored against a set of predetermined test cases that are not visible to the student. Scoring is out of 1000 points, with each problem being weighted equally. There are four divisions of contests: Bronze, Silver, Gold, and Platinum. After each contest, students who meet the contest-dependent cutoff for promotion will compete in the next division for future contests.

While this book is primarily focused on the USACO, CodeForces is another contest programming platform that many students use for practice. CodeForces holds 2-hour contests very frequently, which are more focused on fast solving compared to USACO. However, we do think CodeForces is a valuable training platform, so many exercises and problems will come from there. We encourage you to create a CodeForces account and solve the provided problems there. CodeForces submissions are all-or-nothing; unlike USACO, there is no partial credit and you only receive credit for a problem if you pass *all* of the test cases.

We will also include some exercises from Antti Laaksonen's website CSES. It contains a selection of standard problems that you can use to learn and practice well-known algorithms and techniques. You should note that CSES's grader is very slow, so don't worry if you encounter a Time Limit Exceeded verdict; as long as you pass the majority of test cases within the time limit, you can consider the problem solved, and move on.

1.3 Competitive Programming Practice

Reaching a high level in competitive programming requires dedication and motivation. For many people, their practice is inefficient because they do problems that are too easy, too hard, or simply of the wrong type. This book aims to correct that by providing comprehensive problem sets for each topic covered on the USA Computing Olympiad, as well as an extensive selection of problems across all topics in the final chapter.

In the lower divisions, most problems use relatively elementary algorithms; the main challenge is deciding which algorithm to use, and implementing it correctly. In a contest, you should spend the bulk of your time thinking about the problem and coming up with the algorithm, rather than typing code. Thus, you should practice your implementation skills, so that during the contest, you can implement the algorithm quickly and correctly, without resorting to debugging.

1.4 About This Book

This book aims to prepare students for the Bronze and Silver division of the USACO, with the goal of qualifying for Gold. We will do this by covering all the necessary algorithms, data

structures, and skills to pass the Bronze and Silver contests. Many examples and practice problems have been provided; these are the most important part of studying competitive programming, so make sure you pay careful attention to the examples and solve the practice problems, which usually come from previous USACO contests. This book is intended for those who have some programming experience – Basic knowledge of C++ at the level of an introductory class is expected. This book begins with some necessary background knowledge, which is then followed by lessons on common topics that appear on the Bronze and Silver divisions of USACO, and then examples. At the end of each chapter will be a set of problems from USACO, CodeForces, and CSES, where you can practice what you've learned in the chapter.

You improve at competitive programming by solving problems, so we strongly recommend that you solve most or all of the problems in each section before moving on. Some of the problems will be easy, and some of them will be hard. We strongly recommend that you do not read the official USACO or CodeForces editorials unless you have thought about the problem for a while and used the given hints. If you are truly stuck, you should proceed about reading the official solutions as follows: read one step at a time. Once you've gotten un-stuck, finish the problem, and implement it so you remember how to do it.

The primary purpose of this book is to compile all of the topics needed for a beginner in one book, and provide all the resources needed, to make the process of studying for contests easier.

A brief outline of the book is as follows:

- Chapter 2 covers how to handle input/output in C++, the different data types used to store variables, and finally, binary representations of integers and bit operations on integers.
- Chapter 3 introduces time and space complexity and how to structure algorithm designs around these constraints.
- Chapter 4 covers data structures in the C++ standard library, such as stacks, sets, maps, and the various types of queues.
- Chapter 5 covers simulation problems where the solution is simply implementing the problem description.
- Chapter 6 discusses the complete search or brute force paradigm of problem-solving: checking all possible solutions.
- Chapter 7 presents some miscellaneous Bronze division topics
- Chapter 8 discusses solving problems by sorting and use of comparators.
- Chapter 9 introduces greedy algorithms and some common problems that can be solved using greedy algorithms
- Chapter 10 deals with graph theory: graph representations, traversal algorithms, and other related topics.

- Chapter 11 introduces prefix sums.
- Chapter 12 covers applications of binary search.
- Chapter 13 focuses on some number theory concepts that often appear on contests.
- Chapter 14 covers the remaining silver topics.
- Chapter 15 is a collection of practice problems to review all the topics covered in this book, and provide further preparation for contests.

Chapter 2

Elementary Techniques

2.1 Input and Output

In CodeForces and CSES, input and output are standard, meaning that using the library <iostream> suffices.

However, in USACO, input is read from a file called `problemname.in`, and printing output to a file called `problemname.out`. Note that you'll have to rename the `.in` and `.out` files. You will need the <cstdio> or the <fstream> library. Essentially, replace every instance of the word *template* in the word below with the input/output file name, which should be given in the problem.

Below, we have included C++ templates for input and output.

For USACO:

If <cstdio> is used:

```
#include <cstdio>

using namespace std;

int main() {
    freopen("template.in", "r", stdin);
    freopen("template.out", "w", stdout);
}
```

If <fstream> is used (note that if you use <fstream>, you must replace `cin` and `cout` with `fin` and `fout`):

```
#include <fstream>

using namespace std;

int main() {
    ifstream fin("template.in");
```

```
    ofstream fout("template.out");
}
```

For CodeForces, CSES, and other contests that use standard input and output, simply use the standard input / output from `<iostream>`.

When using C++, arrays and other data structures should be declared globally if at all possible. This avoids the common issue of initialization to garbage values. If you declare an array or vector locally, you may need to initialize the values to zero.

2.2 Data Types

There are several main data types that are used in contests: 32-bit and 64-bit integers, floating point numbers, booleans, characters, and strings.

The 32-bit integer supports values between $-2\,147\,483\,648$ and $2\,147\,483\,647$, which is roughly equal to $\pm\, 2\times 10^9$. If the input, output, or *any intermediate values used in calculations* exceed the range of a 32-bit integer, then a 64-bit integer must be used. The range of the 64-bit integer is between $-9\,223\,372\,036\,854\,775\,808$ and $9\,223\,372\,036\,854\,775\,807$ which is roughly equal to $\pm\, 9\times 10^{18}$. Contest problems are usually set such that the 64-bit integer is sufficient. If it's not, the problem will ask for the answer modulo m, instead of the answer itself, where m is a prime. In this case, make sure to use 64-bit integers, and take the remainder of x modulo m after every step using `x %= m;`.

Floating point numbers are used to store decimal values. It is important to know that floating point numbers are not exact, because the binary architecture of computers can only store decimals to a certain precision. Hence, we should always expect that floating point numbers are slightly off. Contest problems will accommodate this by either asking for the greatest integer less than 10^k times the value, or will mark as correct any output that is within a certain ϵ of the judge's answer.

Boolean variables have two possible states: true and false. We'll usually use booleans to mark whether a certain process is done, and arrays of booleans to mark which components of an algorithm have finished.

Character variables represent a single Unicode character. They are returned when you access the character at a certain index within a string. Characters are represented using the ASCII standard, which assigns each character to a corresponding integer; this allows us to do arithmetic with them, for example, `cout << ('f' - 'a');` will print `5`.

Strings are stored as an array of characters. You can easily access the character at a certain index and take substrings of the string. String problems on USACO are generally very easy and don't involve any special data structures.

Chapter 3

Time/Space Complexity and Algorithm Analysis

In programming contests, there is a strict limit on program runtime. This means that in order to pass, your program needs to finish running within a certain timeframe. For USACO, this limit is 2 seconds for C++ submissions. A conservative estimate for the number of operations the grading server can handle per second is 10^8 (but is really closer to $5 \cdot 10^8$ given good constant factors).

3.1 Big O Notation and Complexity Calculations

We want a method of how many operations it takes to run each algorithm, in terms of the input size n. Fortunately, this can be done relatively easily using Big O notation, which expresses worst-case complexity as a function of n, as n gets arbitrarily large. Complexity is an upper bound for the number of steps an algorithm requires, as a function of the input size. In Big O notation, we denote the complexity of a function as $O(f(n))$, where $f(n)$ is a function without constant factors or lower-order terms. We'll see some examples of how this works, as follows.

The following code is $O(1)$, because it executes a constant number of operations.

```
int a = 5;
int b = 7;
int c = 4;
int d = a + b + c + 153;
```

Input and output operations are also assumed to be $O(1)$.

In the following examples, we assume that the code inside the loops is $O(1)$.

The time complexity of loops is the number of iterations that the loop runs. For example, the following code examples are both $O(n)$.

```
for(int i = 1; i <= n; i++){
    // constant time code here
```

```
}
```

```
int i = 0;
while(i < n){
    // constant time node here
    i++;
}
```

Because we ignore constant factors and lower order terms, the following examples are also $O(n)$:

```
for(int i = 1; i <= 5*n + 17; i++){
    // constant time code here
}
```

```
for(int i = 1; i <= n + 457737; i++){
    // constant time code here
}
```

We can find the time complexity of multiple loops by multiplying together the time complexities of each loop. This example is $O(nm)$, because the outer loop runs $O(n)$ iterations and the inner loop $O(m)$.

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

In this example, the outer loop runs $O(n)$ iterations, and the inner loop runs anywhere between 1 and n iterations (which is a maximum of n). Since Big O notation calculates worst-case time complexity, we must take the factor of n from the inner loop. Thus, this code is $O(n^2)$.

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

If an algorithm contains multiple blocks, then its time complexity is the worst time complexity out of any block. For example, the following code is $O(n^2)$.

```
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
        // constant time code here
    }
}
for(int i = 1; i <= n + 58834; i++){
    // more constant time code here
}
```

The following code is $O(n^2 + nm)$, because it consists of two blocks of complexity $O(n^2)$ and $O(nm)$, and neither of them is a lower order function with respect to the other.

```
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
        // constant time code here
    }
}
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
        // more constant time code here
    }
}
```

3.2 Common Complexities and Constraints

Complexity factors that come from some common algorithms and data structures are as follows:

- Mathematical formulas that just calculate an answer: $O(1)$
- Unordered set/map: $O(1)$ per operation
- Binary search: $O(\log n)$
- Ordered set/map or priority queue: $O(\log n)$ per operation
- Prime factorization of an integer, or checking primality or compositeness of an integer: $O(\sqrt{n})$
- Reading in n items of input: $O(n)$
- Iterating through an array or a list of n elements: $O(n)$
- Sorting: usually $O(n \log n)$ for default sorting algorithms (mergesort and quicksort used in `std::sort()`)

- Iterating through all subsets of size k of the input elements: $O(n^k)$. For example, iterating through all triplets is $O(n^3)$.
- Iterating through all subsets: $O(2^n)$
- Iterating through all permutations: $O(n!)$

Here are **conservative** upper bounds on the value of n for each time complexity. You can probably get away with more than this, but this should allow you to quickly check whether an algorithm is viable.

n	Possible complexities
$n \leq 10$	$O(n!)$, $O(n^7)$, $O(n^6)$
$n \leq 20$	$O(2^n \cdot n)$, $O(n^5)$
$n \leq 80$	$O(n^4)$
$n \leq 400$	$O(n^3)$
$n \leq 7500$	$O(n^2)$
$n \leq 7 \cdot 10^4$	$O(n\sqrt{n})$
$n \leq 5 \cdot 10^5$	$O(n \log n)$
$n \leq 5 \cdot 10^6$	$O(n)$
$n \leq 10^{12}$	$O(\sqrt{n} \log n)$, $O(\sqrt{n})$
$n \leq 10^{18}$	$O(\log^2 n)$, $O(\log n)$, $O(1)$

Chapter 4

Built-in Data Structures

A data structure determines how data is stored. (is it sorted? indexed? what operations does it support?) Each data structure supports some operations efficiently, while other operations are either inefficient or not supported at all. This chapter introduces the data structures in the C++ standard library that are frequently used in competitive programming.

The C++ standard library data structures are designed to store any type of data. We put the desired data type within the <> brackets when declaring the data structure, as follows:

```
vector<string> v;
```

This creates a `vector` structure that only stores objects of type `string`.

For our examples below, we will primarily use the `int` data type, but note that you can use any data type including `string` and user-defined structures.

Essentially every standard library data structure supports the `size()` method, which returns the number of elements in the data structure, and the `empty()` method, which returns `true` if the data structure is empty, and `false` otherwise.

Once again, in C++, arrays and other data structures should be declared globally if at all possible. This avoids the common issue of initialization to garbage values. If you declare an array or vector locally, you may need to initialize the values to zero.

4.1 Iterators

Before the data structures are introduced, you should understand an iterator. An iterator allows you to traverse a container by providing a pointer. For example, `vector.begin()` returns an iterator pointing to the first element of the vector. Apart from the standard way of traversing a vector (by treating it as an array), you can also use iterators:

```
for (vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it) {
    cout << *it; //prints the values in the vector using the pointer
}
```

However, a more generic way to do this is with a for-each loop and `auto` (C++11 and later versions) that automatically infers the type of an object:

```
for(auto element : v) {
    cout << element; //prints the values in the vector
}
```

4.2 Dynamic Arrays

You're probably already familiar with regular (static) arrays. Now, there are also dynamic arrays (`vector` in C++) that support all the functions that a normal array does, and can resize itself to accommodate more elements. In a dynamic array, we can also add and delete elements at the end in $O(1)$ time.

For example, the following code creates a dynamic array and adds the numbers 1 through 10 to it:

```
vector<int> v;
for(int i = 1; i <= 10; i++){
    v.push_back(i);
}
```

When declaring a dynamic array we can give it an initial size, so it doesn't resize itself as we add elements to it. The following code initializes a `vector` with initial size 30:

```
vector<int> v(30);
```

However, we need to be careful that we only add elements to the end of the `vector`; insertion and deletion in the middle of the `vector` is $O(n)$.

```
vector<int> v;
v.push_back(2); // [2]
v.push_back(3); // [2, 3]
v.push_back(7); // [2, 3, 7]
v.push_back(5); // [2, 3, 7, 5]
v[1] = 4; // sets element at index 1 to 4 -> [2, 4, 7, 5]
v.erase(1); // removes element at index 1 -> [2, 7, 5]
// this remove method is O(n); to be avoided
v.push_back(8); // [2, 7, 5, 8]
v.erase(v.size()-1); // [2, 7, 5]
// here, we remove the element from the end of the list; this is O(1).
v.push_back(4); // [2, 7, 5, 4]
v.push_back(4); // [2, 7, 5, 4, 4]
v.push_back(9); // [2, 7, 5, 4, 4, 9]
```

```
cout << v[2]; // 5
v.erase(v.begin(), v.begin()+3) // [4, 4, 9]
// this erases the first three elements; O(n)
```

To iterate through a static or dynamic array, we can use either the regular for loop or the for-each loop.

```
vector<int> v;
v.push_back(1); v.push_back(7); v.push_back(4); v.push_back(5); v.push_back(2);
int arr[] = {1, 7, 4, 5, 2};
for(int i = 0; i < v.size(); i++){
    cout << v[i];
}
for(int element : arr){
    cout << element;
}
```

In order to sort a static or dynamic array, use `sort(v.begin(), v.end())`. The default sort function sorts the array in ascending order.

In array-based contest problems, we'll use one-, two-, and three-dimensional static arrays most of the time. However, we can also have static arrays of dynamic arrays, dynamic arrays of static arrays, and so on. Usually, the choice between a static array and a dynamic array is just personal preference.

4.3 Stacks and the Various Types of Queues

Stacks

A stack is a Last In First Out (LIFO) data structure that supports three operations: `push`, which adds an element to the top of the stack, `pop`, which removes an element from the top of the stack, and `top`, which retrieves the element at the top without removing it, all in $O(1)$ time. Think of it like a real-world stack of papers.

```
stack<int> s;
s.push(1); // [1]
s.push(13); // [1, 13]
s.push(7); // [1, 13, 7]
cout << s.top(); // 7
s.pop(); // [1, 13]
cout << s.size(); // 2
```

Queues

A queue is a First In First Out (FIFO) data structure that supports three operations of `push`, insertion at the back of the queue, `pop`, deletion from the front of the queue, and `front`, which retrieves the element at the front without removing it, all in $O(1)$ time.

```
queue<int> q;
q.push(1); // [1]
q.push(3); // [3, 1]
q.push(4); // [4, 3, 1]
q.pop(); // [4, 3]
cout << q.front(); // 3
```

Deques

A deque (usually pronounced "deck") stands for double ended queue and is a combination of a stack and a queue, in that it supports $O(1)$ insertions and deletions from both the front and the back of the deque. The four methods for adding and removing are `push_back()`, `pop_back()`, `push_front()`, and `pop_front()`.

```
deque<int> d;
d.push_front(3); // [3]
d.push_front(4); // [4, 3]
d.push_back(7); // [4, 3, 7]
d.pop_front(); // [3, 7]
d.push_front(1); // [1, 3, 7]
d.pop_back(); // [1, 3]
```

Priority Queues

A priority queue supports the following operations: insertion of elements, deletion of the element considered highest priority, and retrieval of the highest priority element, all in $O(\log n)$ time according to the number of elements in the priority queue. Priority is based on a comparator function, but by default the lowest element is at the front of the priority queue. The priority queue is one of the most important data structures in competitive programming, so make sure you understand how and when to use it. By default, the C++ Priority Queue puts the highest element at the front of the queue.

```
priority_queue<int> pq;
pq.push(7); // [7]
pq.push(2); // [2, 7]
pq.push(1); // [1, 2, 7]
pq.push(5); // [1, 2, 5, 7]
cout << pq.top(); // 7
```

```
pq.pop(); // [1, 2, 5]
pq.pop(); // [1, 2]
pq.push(6); // [1, 2, 6]
```

4.4 Sets and Maps

A set is a collection of objects that contains no duplicates. There are two types of sets: unordered sets (`unordered_set` in C++), and ordered set (`set` in C++).

Unordered Sets

The unordered set works by hashing, which is assigning a unique code to every variable/object which allows insertions, deletions, and searches in $O(1)$ time, albeit with a high constant factor, as hashing requires a large constant number of operations. However, as the name implies, elements are not ordered in any meaningful way, so traversals of an unordered set will return elements in some arbitrary order. The operations on an unordered set are `insert`, which adds an element to the set if not already present, `erase`, which deletes an element if it exists, and `count`, which returns 1 if the set contains the element and 0 if it doesn't.

```
unordered_set<int> set;
set.insert(1); // [1]
set.insert(4); // [1, 4] in arbitrary order
set.insert(2); // [1, 4, 2] in arbitrary order
set.insert(1); // [1, 4, 2] in arbitrary order
// the add method did nothing because 1 was already in the set
cout << set.count(1); // 1
set.erase(1); // [2, 4] in arbitrary order
cout << set.count(5); // 0
set.remove(0); // [2, 4] in arbitrary order
// if the element to be removed does not exist, nothing happens

for(int element : set){
    cout << element;
}
// You can iterate through an unordered set, but it will do so in arbitrary
↪  order
```

Ordered Sets

The second type of set data structure is the ordered or sorted set. Insertions, deletions, and searches on the ordered set require $O(\log n)$ time, based on the number of elements in the set. As well as those supported by the unordered set, the ordered set also allows four

additional operations: `begin()`, which returns the lowest element in the set, `end()`, which returns the highest element in the set, `lower_bound(k)`, which returns the greatest element less than or equal to some element k, and `upper_bound(k)`, which returns the least element strictly greater than some element k.

```
set<int> set;
set.insert(1); // [1]
set.insert(14); // [1, 14]
set.insert(9); // [1, 9, 14]
set.insert(2); // [1, 2, 9, 14]
cout << set.upper_bound(7) << '\n'; // 9
cout << set.upper_bound(9) << '\n'; // 14
cout << set.lower_bound(5) << '\n'; // 2
cout << set.lower_bound(9) << '\n'; // 9
cout << set.begin() << '\n'; // 1
cout << set.end() << '\n'; // 14
set.erase(set.upper_bound(6)); // [1, 2, 14]
```

The primary limitation of the ordered set is that we can't efficiently access the k^{th} largest element in the set, or find the number of elements in the set greater than some arbitrary x. These operations can be handled using a data structure called an order statistic tree, but that is beyond the scope of this book.

Maps

A map is a set of ordered pairs, each containing a key and a value. In a map, all keys are required to be unique, but values can be repeated. Maps have three primary methods: one to add a specified key-value pairing, one to retrieve the value for a given key, and one to remove a key-value pairing from the map. Like sets, maps can be unordered (`unordered_map` in C++) or ordered (`map` in C++). In an unordered map, hashing is used to support $O(1)$ operations. In an ordered map, the entries are sorted in order of key. Operations are $O(\log n)$, but accessing or removing the next key higher or lower than some input `k` is also supported.

Unordered Maps

In an unordered map `m`, the `m[key] = value` operator assigns a value to a key and places the key and value pair into the map. The operator `m[key]` returns the value associated with the key. The `count(key)` method returns the number of times the key is in the map (which is either one or zero), and therefore checks whether a key exists in the map. Lastly, `erase(key)` and `erase(it)` removes the map entry associated with the specified key or iterator. All of these operations are $O(1)$, but again, due to the hashing, this has a high constant factor.

```
unordered_map<int, int> map;
map[1] = 5; // [(1, 5)]
map[3] = 14; // [(1, 5); (3, 14)]
```

```
map[2] = 7; // [(1, 5); (3, 14); (2, 7)]
map.erase(2); // [(1, 5); (3, 14)]
cout << map[1] << '\n'; // 5
cout << map.count(7) << '\n' ; // 0
cout << map.count(1) << '\n' ; // 1
```

Ordered Maps

The ordered map supports all of the operations that an unordered map supports, and additionally supports `lower_bound` and `upper_bound`, returning the iterator pointing to the lowest entry not less than the specified key, and the iterator pointing to the lowest entry strictly greater than the specified key respectively.

```
unordered_map<int, int> map;
map[3] = 5; // [(3, 5)]
map[11] = 4; // [(3, 5); (11, 4)]
map[10] = 491; // [(3, 5); (10, 491); (11, 4)]
cout << map.lower_bound(10) << '\n'; // iterator to (10, 491)
cout << map.upper_bound(10) << '\n'; // iterator to (11, 4)
map.erase(11); // [(3, 5); (11, 4)]
cout << map.upper_bound(10)) << '\n'; // map.end()
```

A note on unordered sets and maps: In USACO contests, they're generally fine, but in CodeForces contests, you should always use sorted sets and maps. This is because the built-in hashing algorithm is vulnerable to pathological data sets causing abnormally slow runtimes, in turn causing failures on some test cases.

Multisets

Lastly, there is the multiset, which is essentially a sorted set that allows multiple copies of the same element. In addition to all of the regular set operations, the multiset `count()` method returns the number of times an element is present in the multiset.

```
multiset<int> set;
set.insert(1); // [1]
set.insert(14); // [1, 14]
set.insert(9); // [1, 9, 14]
set.insert(2); // [1, 2, 9, 14]
set.insert(9); // [1, 2, 9, 9, 14]
set.insert(9); // [1, 2, 9, 9, 9, 14]
cout << set.count(4) << '\n'; // 0
cout << set.count(9) << '\n'; // 3
cout << set.count(14) << '\n'; // 1
```

The `begin()`, `end()`, `lower_bound()`, and `upper_bound()` operations work the same way they do in the normal sorted set.

4.5 Problems

Again, note that CSES's grader is very slow, so don't worry if you encounter a Time Limit Exceeded verdict; as long as you pass the majority of test cases within the time limit, you can consider the problem solved, and move on.

1. CSES Problem Set Task 1621: Distinct Numbers
   `https://cses.fi/problemset/task/1621`
2. CSES Problem Set Task 1084: Apartments
   `https://cses.fi/problemset/task/1084`
3. CSES Problem Set Task 1091: Concert Tickets
   `https://cses.fi/problemset/task/1091`
4. CSES Problem Set Task 1163: Traffic Lights
   `https://cses.fi/problemset/task/1163`
5. CSES Problem Set Task 1164: Room Allocation
   `https://cses.fi/problemset/task/1164`

Part II

Bronze

Chapter 5

Simulation

In many problems, we can simply simulate what we're told to do by the problem statement. Since there's no formal algorithm involved, the intent of the problem is to assess competence with one's programming language of choice and knowledge of built-in data structures. At least in USACO Bronze, when a problem statement says to find the end result of some process, or to find when something occurs, it's usually sufficient to simulate the process.

5.1 Example 1

Alice and Bob are standing on a 2D plane. Alice starts at the point $(0, 0)$, and Bob starts at the point (R, S) $(1 \leq R, S \leq 1000)$. Every second, Alice moves M units to the right, and N units up. Every second, Bob moves P units to the left, and Q units down. $(1 \leq M, N, P, Q \leq 10)$. Determine if Alice and Bob will ever meet (be at the same point at the same time), and if so, when.

INPUT FORMAT:
The first line of the input contains R and S.
The second line of the input contains M, N, P, and Q.

OUTPUT FORMAT:
Please output a single integer containing the number of seconds after the start at which Alice and Bob meet. If they never meet, please output -1.

Solution
We can simulate the process. After inputting the values of R, S, M, N, P, and Q, we can keep track of Alice's and Bob's x- and y-coordinates. To start, we initialize variables for their respective positions. Alice's coordinates are initially $(0, 0)$, and Bob's coordinates are (R, S) respectively. Every second, we increase Alice's x-coordinate by M and her y-coordinate by N, and decrease Bob's x-coordinate by P and his y-coordinate by Q.

Now, when do we stop? First, if Alice and Bob ever have the same coordinates, then we are done. Also, since Alice strictly moves up and to the right and Bob strictly moves down and to the left, if Alice's x- or y-coordinates are ever greater than Bob's, then it is impossible for them to meet. Example code will be displayed below (Here, as in other examples, input processing will be omitted):

```
int ax = 0; int ay = 0; // alice's x and y coordinates
int bx = r; int by = s; // bob's x and y coordinates
int t = 0; // keep track of the current time
while(ax < bx && ay < by){
    // every second, update alice's and bob's coordinates and the time
    ax += m; ay += n;
    bx -= p; by -= q;
    t++;
}
if(ax == bx && ay == by){ // if they are in the same location
    cout << t << endl; // they meet at time t
} else {
    cout << -1 << endl; // they never meet
}
```

5.2 Example 2

There are N buckets ($5 \leq N \leq 10^5$), each with a certain capacity C_i ($1 \leq C_i \leq 100$). One day, after a rainstorm, each bucket is filled with A_i units of water ($1 \leq A_i \leq C_i$). Charlie then performs the following process: he pours bucket 1 into bucket 2, then bucket 2 into bucket 3, and so on, up until pouring bucket $N-1$ into bucket N. When Charlie pours bucket B into bucket $B+1$, he pours as much as possible until bucket B is empty or bucket $B+1$ is full. Find out how much water is in each bucket once Charlie is done pouring.

INPUT FORMAT:
The first line of the input contains N.
The second line of the input contains the capacities of the buckets, $C_1, C_2, \ldots, C_n$.
The third line of the input contains the amount of water in each bucket $A_1, A_2, \ldots, A_n$.

OUTPUT FORMAT:
Please print one line of output, containing N space-separated integers: the final amount of water in each bucket once Charlie is done pouring.

Solution:
Once again, we can simulate the process of pouring one bucket into the next. The amount of milk poured from bucket B to bucket $B+1$ is the smaller of the amount of water in bucket B (after all previous operations have been completed) and the remaining space in bucket $B+1$, which is $C_{B+1} - A_{B+1}$. We can just handle all of these operations in order, using an array C to store the maximum capacities of each bucket, and an array A to store the current water level in each bucket, which we update during the process. Example code is below (note that arrays are zero-indexed, so the indices of our buckets go from 0 to $N-1$ rather than from 1 to N).

```
for(int i = 0; i < n-1; i++){
    int amt = min(A[i], C[i+1]-A[i+1]);
```

```
    // the amount of water to be poured is the lesser of
    // the amount of water in the current bucket and
    // the amount of additional water that the next bucket can hold
    A[i] -= amt; // remove the amount from the current bucket
    A[i+1] += amt; // add it to the next bucket
}

for(int i = 0; i < n; i++){
    cout << A[i] << " ";
    // print the amount of water in each bucket at the end
}
cout << endl;
```

5.3 Problems

1. USACO December 2018 Bronze Problem 1: Mixing Milk
   http://www.usaco.org/index.php?page=viewproblem2&cpid=855
2. USACO December 2017 Bronze Problem 3: Milk Measurement
   http://www.usaco.org/index.php?page=viewproblem2&cpid=761
3. USACO US Open 2017 Bronze Problem 1: The Lost Cow
   http://www.usaco.org/index.php?page=viewproblem2&cpid=735
4. USACO February 2017 Bronze Problem 3: Why Did the Cow Cross the Road III
   http://www.usaco.org/index.php?page=viewproblem2&cpid=713
5. USACO January 2016 Bronze Problem 3: Mowing the Field
   http://www.usaco.org/index.php?page=viewproblem2&cpid=593
6. USACO December 2017 Bronze Problem 2: The Bovine Shuffle
   http://usaco.org/index.php?page=viewproblem2&cpid=760
7. USACO February 2016 Bronze Problem 2: Circular Barn
   http://usaco.org/index.php?page=viewproblem2&cpid=616

Chapter 6

Complete Search

In many problems (especially in Bronze), it's sufficient to check all possible cases in the solution space, whether it be all elements, all pairs of elements, or all subsets, or all permutations. Unsurprisingly, this is called complete search (or brute force), because it completely searches the entire solution space.

6.1 Example 1

You are given N ($3 \leq N \leq 5000$) integer points on the coordinate plane. Find the square of the maximum Euclidean distance (aka length of the straight line) between any two of the points.

INPUT FORMAT:
The first line contains an integer N.
The second line contains N integers, the x-coordinates of the points: $x_1, x_2, \ldots, x_n$ ($-1000 \leq x_i \leq 1000$).
The third line contains N integers, the y-coordinates of the points: $y_1, y_2, \ldots, y_n$ ($-1000 \leq y_i \leq 1000$).

OUTPUT FORMAT:
Print one integer, the square of the maximum Euclidean distance between any two of the points.

Solution:
We can brute-force every pair of points and find the square of the distance between them, by squaring the formula for Euclidean distance: distance$^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$. Thus, we store the coordinates in vectors `X[]` and `Y[]`, such that `X[i]` and `Y[i]` are the x- and y-coordinates of the i_{th} point, respectively. Then, we iterate through all possible pairs of points, using a variable max to store the maximum square of distance between any pair seen so far, and if the square of the distance between a pair is greater than our current maximum,

we set our current maximum to it.

Algorithm: Finds the maximum Euclidean distance between any two of the given points

Function `maxDist`
  Input : points an array of n ordered pairs
  Output : the maximum Euclidean distance between any two of the points
  max $\leftarrow 0$
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow i+1$ **to** n **do**
      if `dist`(points$[i]$, points$[j]$)$^2 >$ max **then**
        max $\leftarrow$ `dist`(points$[i]$, points$[j]$)^2
      end
    end
  end
  return max

```
int high = 0; // storing the current maximum
for(int i = 0; i < n; i++){ // for each first point
    for(int j = i+1; j < n; j++){ // for each second point
        int dx = x[i] - x[j];
        int dy = y[i] - y[j];
        high = max(high, dx*dx + dy*dy);
        // if the square of the distance between the two points is greater than
        // our current maximum, then update the maximum
    }
}
cout << high << endl;
```

A couple notes: first, since we're iterating through all pairs of points, we start the j loop from $j = i + 1$ so that point i and point j are never the same point. Furthermore, it makes it so that each pair is only counted once. In this problem, it doesn't matter whether we double-count pairs or whether we allow i and j to be the same point, but in other problems where we're counting something rather than looking at the maximum, it's important to be careful that we don't overcount. Secondly, the problem asks for the square of the maximum Euclidean distance between any two points. Some students may be tempted to maintain the maximum distance in a variable, and then square it at the end when outputting. However, the problem here is that while the square of the distance between two integer points is always an integer, the distance itself isn't guaranteed to be an integer. Thus, we'll end up shoving a non-integer value into an integer variable, which truncates the decimal part. Using a floating point variable isn't likely to work either, due to precision errors (use of floating point decimals should generally be avoided when possible).

6.2 Generating Permutations

A **permutation** is a reordering of a list of elements. Some problems will ask for an ordering of elements that satisfies certain conditions. In these problems, if $N \leq 10$, we can probably iterate through all permutations and check each permutation for validity. For a list of N elements, there are $N!$ ways to permute them, and generally we'll need to read through each permutation once to check its validity, for a time complexity of $O(N \cdot N!)$.

This is done using an algorithm called Heap's Algorithm. In C++, this is already implemented for us in the `next_permutation()` function. To iterate through all permutations, we simply place this inside a while loop.

As an example, here are the permutations generated by Heap's Algorithm for $[1, 2, 3]$:

$$[1, 2, 3], [2, 1, 3], [3, 1, 2], [1, 3, 2], [2, 3, 1], [3, 2, 1]$$

Code for iterating over all permutations is as follows:

```
while(next_permutation(v.begin(), v.end())){
    check(v); // process or check the current permutation for validity
}
```

6.3 Problems

1. USACO February 2020 Bronze Problem 1: Triangles
   `http://usaco.org/index.php?page=viewproblem2&cpid=1011`
2. USACO January 2020 Bronze Problem 2: Photoshoot
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=988`
   (Hint: Figure out what exactly you're complete searching)
3. USACO December 2019 Bronze Problem 1: Cow Gymnastics
   `http://usaco.org/index.php?page=viewproblem2&cpid=963`
   (Hint: Brute force over all possible pairs)
4. USACO February 2016 Bronze Problem 1: Milk Pails
   `http://usaco.org/index.php?page=viewproblem2&cpid=615`
5. USACO January 2018 Bronze Problem 2: Lifeguards
   `http://usaco.org/index.php?page=viewproblem2&cpid=784`
   (Hint: Try removing each lifeguard one at a time).
6. USACO December 2019 Bronze Problem 2: Where Am I?
   `http://usaco.org/index.php?page=viewproblem2&cpid=964`
   (Hint: Brute force over all possible substrings)
7. (Permutations) USACO December 2019 Bronze Problem 3: Livestock Lineup
   `http://usaco.org/index.php?page=viewproblem2&cpid=965`

8. (Permutations) CSES Problem Set Task 1624: Chessboard and Queens
   `https://cses.fi/problemset/task/1624`

9. USACO US Open 2016 Bronze Problem 3: Field Reduction
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=641`
   (Hint: For this problem, you can't do a full complete search; you have to do a reduced search)

10. USACO December 2018 Bronze Problem 3: Back and Forth
    `http://www.usaco.org/index.php?page=viewproblem2&cpid=857`
    (This problem is relatively hard)

Chapter 7

Additional Bronze Topics

7.1 Square and Rectangle Geometry

The extent of "geometry" problems on USACO Bronze are usually quite simple and limited to intersections and unions of squares and rectangles. These usually only include two or three squares or rectangles, in which case you can simply draw out cases on paper, which should logically lead to a solution.

The problems given at the end should encompass all the techniques you need to know for geometry problems in the Bronze division.

7.2 Ad-hoc

Ad-hoc problems are problems that don't fall into any standard algorithmic category with well known solutions. They are usually unique problems intended to be solved with unconventional techniques. In ad-hoc problems, it's helpful to look at the constraints given in the problem and devise potential time complexities of solutions; this, combined with details in the problem statement itself, may give an outline of the solution.

Unfortunately, since ad-hoc problems don't have solutions consisting of well known algorithms, we can't systematically teach you how to do them. The best way of learning how to do ad-hoc is to practice. Of course, the problem solving intuition from math contests (if you did them) is quite helpful, but otherwise, you can develop this intutition from practicing ad-hoc problems.

While solving these problems, make sure to utilize what you've learned about the built-in data structures and algorithmic complexity analysis, from chapters 2, 3, and 4. Since ad-hoc problems comprise a significant portion of bronze problems, we've included a large selection of them below for your practice.

7.3 Problems

Square and Rectangle Geometry

1. USACO December 2017 Bronze Problem 1: Blocked Billboard
   http://usaco.org/index.php?page=viewproblem2&cpid=759
2. USACO December 2018 Bronze Problem 1: Blocked Billboard II
   http://usaco.org/index.php?page=viewproblem2&cpid=783
3. CodeForces Round 587 (Div. 3) Problem C: White Sheet
   https://codeforces.com/contest/1216/problem/C
4. USACO December 2016 Bronze Problem 1: Square Pasture
   http://usaco.org/index.php?page=viewproblem2&cpid=663

Ad-hoc problems

5. USACO January 2016 Bronze Problem 1: Promotion Counting
   http://usaco.org/index.php?page=viewproblem2&cpid=591
6. USACO January 2020 Bronze Problem 1: Word Processor
   http://usaco.org/index.php?page=viewproblem2&cpid=987
7. USACO US Open 2019 Bronze Problem 1: Bucket Brigade
   http://usaco.org/index.php?page=viewproblem2&cpid=939
8. USACO January 2018 Bronze Problem 3: Out of Place
   http://usaco.org/index.php?page=viewproblem2&cpid=785
9. USACO December 2016 Bronze Problem 2: Block Game
   http://usaco.org/index.php?page=viewproblem2&cpid=664
10. USACO February 2020 Bronze Problem 3: Swapity Swap
    http://usaco.org/index.php?page=viewproblem2&cpid=1013
    (This problem is quite hard for bronze.)
11. USACO February 2018 Bronze Problem 1: Teleportation
    http://usaco.org/index.php?page=viewproblem2&cpid=807
12. USACO February 2018 Bronze Problem 2: Hoofball
    http://usaco.org/index.php?page=viewproblem2&cpid=808
13. USACO US Open 2019 Bronze Problem 3: Cow Evolution
    http://usaco.org/index.php?page=viewproblem2&cpid=941
    (Warning: This problem is extremely difficult for bronze.)

Part III

Silver

Chapter 8

Sorting and Comparators

8.1 Comparators

C++ has a built-in function for sorting: `std::sort(first, last)` that sorts the elements in the range in ascending order. In particular, `sort(arr, arr + N)` sorts an entire array of size N, and `sort(v.begin(), v.end())` sorts a `vector v`. However, if we we want to sort elements in a self-defined order, then we'll need to use a custom comparator.

Normally, sorting functions rely on moving objects with a lower value in front of objects with a higher value if sorting in ascending order, and vice versa if in descending order. This is done through comparing two objects at a time. What a comparator does is compare two objects as follows, based on our comparison criteria:

- If object x is less than object y, return true
- If object x is greater than or equal to object y, return false

Essentially, the comparator determines whether object x belongs to the left of object y in a sorted ordering.

In addition to returning the correct answer, comparators should also satisfy the following conditions:

- The function must be consistent with respect to reversing the order of the arguments: if $x \neq y$ and `compare(x, y)` is positive, then `compare(y, x)` should be negative and vice versa
- The function must be transitive. If `compare(x, y)` is true and `compare(y, z)` is true, then `compare(x, z)` should also be true. If the first two compare functions both return false, the third must also return false.

A generic way of implementing a custom comparator is to define a function. For our example, we'll use a `struct` of a Person that contains a person's height and weight, and sort in ascending order by height. A `struct` is essentially a class, in the sense that it allows you to create a data structure:

```
struct Person {
    int height;
    int weight;
}
int main() {
    Person p;
    p.height = 60; // assigns 60 to the height of p
    p.weight = 100; // assigns 100 to the weight of p
}
```

Let's say we have an array `Person arr[N]`. To sort the array, we need to make custom comparator which will be a function, and then pass the function as a parameter into the build-in sort function:

```
bool cmp(Person a, Person b) {
    return a.height < b.height;
}

int main() {
    sort(arr, arr+N, cmp); // sorts the array in ascending order by height
}
```

If we instead wanted to sort in descending order, this is also very simple. Instead of the `cmp` function returning `return a.height < b.height;`, it should do `return a.height > b.height;`.

8.2 Sorting by Multiple Criteria

Now, suppose we wanted to sort a list of `Person` in ascending order, primarily by height and secondarily by weight. We can do this quite similarly to how we handled sorting by one criterion earlier. What the comparator function needs to do is to compare the weights if the heights are equal, and otherwise compare heights, as that's the primary sorting criterion.

```
bool cmp(Person a, Person b) {
    if(a.height == b.height) {
        return a.weight < b.weight;
    }
    return a.height < b.height;
}

int main() {
    sort(arr, arr+N, cmp); // sorts the array in ascending order by height and
    ↪  then weight if the heights are equal
}
```

Sorting with an arbitrary number of criteria is done similarly.

An alternative way of representing custom objects is with the data structure `pair<int, int>`. In the above example, instead of creating a `struct`, we can simply declare an array of pairs. The sort function automatically uses the first element of the pair for comparison and the second element as a secondary point of comparison:

```
pair<int, int> arr[N];

int main() {
    sort(arr, arr+N); // sorts the array in ascending order by height and weight
    ↪  as a secondary if height is equal
}
```

8.3 Problems

1. USACO US Open 2018 Silver Problem 2: Lemonade Line
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=835`
2. CodeForces Round 633 (Div. 2) Problem B: Sorted Adjacent Differences
   `https://codeforces.com/problemset/problem/1339/B`
3. CodeForces Round 579 (Div. 3) Problem E: Boxers
   `https://codeforces.com/problemset/problem/1203/E`
4. USACO January 2019 Silver Problem 3: Mountain View
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=896`
5. USACO US Open 2016 Silver Problem 1: Field Reduction
   `http://www.usaco.org/index.php?page=open16results`

Chapter 9

Greedy Algorithms

Greedy algorithms are algorithms that select the most optimal choice at each step, instead of looking at the solution space as a whole. This reduces the problem to a smaller problem at each step. However, as greedy algorithms never recheck previous steps, they sometimes lead to incorrect answers. Moreover, in a certain problem, there may be more than one possible greedy algorithm; usually only one of them is correct. This means that we must be extremely careful when using the greedy method. However, when they are correct, greedy algorithms are extremely efficient.

Greedy is not a single algorithm, but rather a way of thinking that is applied to problems. There's no one way to do greedy algorithms. Hence, we use a selection of well-known examples to help you understand the greedy paradigm.

Usually, when using a greedy algorithm, there is a heuristic or value function that determines which choice is considered most optimal.

9.1 Introductory Example: Studying Algorithms

Steph wants to improve her knowledge of algorithms over winter break. She has a total of X ($1 \leq X \leq 10^4$) minutes to dedicate to learning algorithms. There are N ($1 \leq N \leq 100$) algorithms, and each one of them requires a_i ($1 \leq a_i \leq 100$) minutes to learn. Find the maximum number of algorithms she can learn.

The solution is quite simple. The first observation we make is that Steph should prioritize learning algorithms from easiest to hardest; in other words, start with learning the algorithm that requires the least amount of time, and then choose further algorithms in increasing order of time required. Let's look at the following example:

$$X = 15, \qquad N = 4, \qquad a_i = \{4, 3, 8, 4, 7, 3\}$$

After sorting the array, we have $\{3, 3, 4, 4, 7, 8\}$. Within the maximum of 15 minutes, Steph can learn four algorithms in a total of $3 + 3 + 4 + 4 = 14$ minutes. The implementation of this algorithm is very simple. We sort the array, and then take as many elements as possible while the sum of times of algorithms chosen so far is less than X. Sorting the array takes $O(N \log N)$ time, and iterating through the array takes $O(N)$ time, for a total time complexity of $O(N \log N)$.

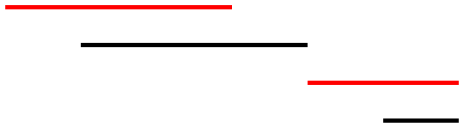
```
// read in the input, store the algorithms in a vector, algorithms
sort(algorithms.begin(), algorithms.end());
int count = 0; // number of minutes used so far
int i = 0;
while(count + algorithms[i] <= x){
    // while there is enough time, learn more algorithms
    count += algorithms[i];
    i++;
}
cout << i << endl; // print the ans
```

9.2 The Scheduling Problem

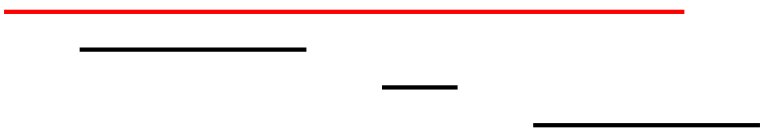
There are N events, each described by their starting and ending times. Jason would like to attend as many events as possible, but he can only attend one event at a time, and if he chooses to attend an event, he must attend the entire event. Traveling between events is instantaneous.

Bad Greedy: Earliest Starting Next Event

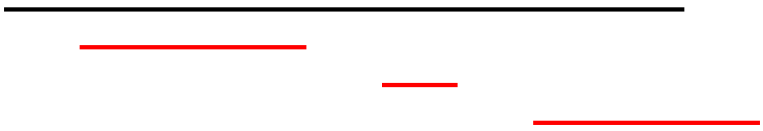
One possible ordering for a greedy algorithm would always select the next possible event that begins as soon as possible. Let's look at the following example, where the selected events are highlighted in red:



In this example, the greedy algorithm selects two events, which is optimal. However, this doesn't always work, as shown by the following counterexample:

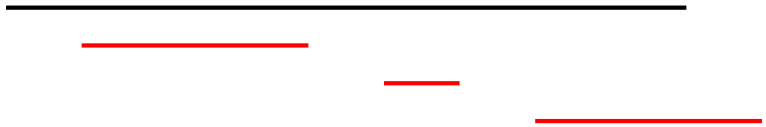


In this case, the greedy algorithm selects to attend only one event. However, the optimal solution would be the following:



Correct Greedy: Earliest Ending Next Event

Instead, we can select the event that ends as early as possible. This correctly selects the three events.



In fact, this algorithm always works. A brief explanation of correctness is as follows. If we have two events E_1 and E_2, with E_2 ending later than E_1, then it is always optimal to select E_1. This is because selecting E_1 gives us more choices for future events. If we can select an event to go after E_2, then that event can also go after E_1, because E_1 ends first. Thus, the set of events that can go after E_2 is a subset of the events that can go after E_1, making E_1 the optimal choice.

For the following code, let's say we have the array `events` of events, which each contain a start and an end point. We'll be using the C++ built in container pair to store each event. Note that since the standard sort in C++ sorts by first element, we will store each event as `pair<end, start>`.

```
// read in the input, store the events in pair<int, int>[] events.
sort(events, events + n); // sorts by first element (ending time)
int currentEventEnd = -1; // end of event currently attending
int ans = 0; // how many events were attended?
for(int i = 0; i < n; i++){ // process events in order of end time
    if(events[i].second >= currentEventEnd){ // if event can be attended
    // we know that this is the earliest ending event that we can attend
    // because of how the events are sorted
        currentEventEnd = events[i].first;
        ans++;
    }
}
cout << ans << endl;
```

9.3 When Greedy Fails

We'll provide a few common examples of when greedy fails, so that you can avoid falling into obvious traps and wasting time getting wrong answers in contest.

Coin Change

This problem gives several coin denominations, and asks for the minimum number of coins needed to make a certain value. Greedy algorithms can be used to solve this problem only in very specific cases (it can be proven that it works for the American as well as the Euro coin systems). However, it doesn't work in the general case. For example, let the coin

denominations be $\{1, 3, 4\}$, and say the value we want is 6. The optimal solution is $\{3, 3\}$, which requires only two coins, but the greedy method of taking the highest possible valued coin that fits in the remaining denomination gives the solution $\{4, 1, 1\}$, which is incorrect.

Knapsack

The knapsack problem gives a number of items, each having a weight and a value, and we want to choose a subset of these items. We are limited to a certain weight, and we want to maximize the value of the items that we take.

Let's take the following example, where we have a maximum capacity of 4:

Item	Weight	Value	Value Per Weight
A	3	18	6
B	2	10	5
C	2	10	5

If we use greedy based on highest value first, we choose item A and then we are done, as we don't have remaining weight to fit either of the other two. Using greedy based on value per weight again selects item A and then quits. However, the optimal solution is to select items B and C, as they combined have a higher value than item A alone. In fact, there is no working greedy solution. The solution to this problem uses dynamic programming, which is beyond the scope of this book.

9.4 Problems

1. USACO December 2015 Silver Problem 2: High Card Wins
   `http://usaco.org/index.php?page=viewproblem2&cpid=571`
2. USACO February 2018 Silver Problem 1: Rest Stops
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=810`
3. USACO February 2017 Silver Problem 1: Why Did The Cow Cross The Road
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=714`

Chapter 10

Graph Theory

Graph theory is one of the most important topics at the Silver level and above. Graphs can be used to represent many things, from images to wireless signals, but one of the simplest analogies is to a map. Consider a map with several cities and highways connecting the cities. Some of the problems relating to graphs are:

- If we have a map with some cities and roads, what's the shortest distance I have to travel to get from point A to point B?
- Consider a map of cities and roads. Is city A connected to city B? Consider a region to be a group of cities such that each city in the group can reach any other city in said group, but no other cities. How many regions are in this map, and which cities are in which region?

10.1 Graph Basics

Graphs are made up of **nodes** and **edges**, where nodes are connected by edges. Graphs can have either **weighted** edges, in which each edge has a certain length, or **unweighted**, in which case all edges have the same length. Edges are either **directed**, which means they can be traversed only in one direction, or **undirected**, which means that they can be traversed in both directions.

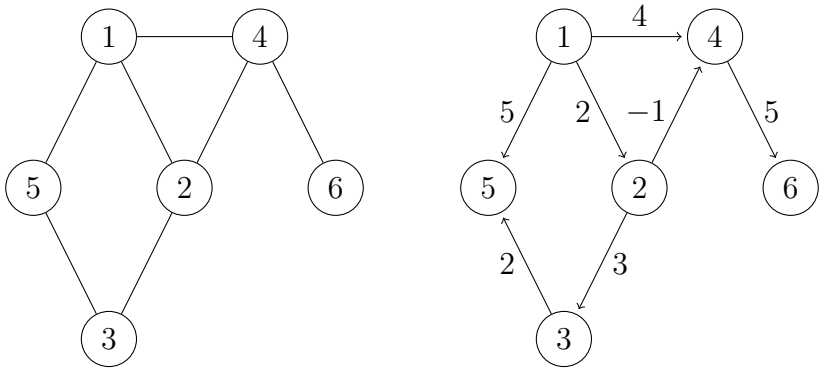


Figure 10.1: An undirected unweighted graph (left) and a directed weighted graph (right)

A **connected component** is a set of nodes within which any node can reach any other node. For example, in this graph, nodes 1, 2, and 3 are a connected component, nodes 4 and 5 are a connected component, and node 6 is its own component.

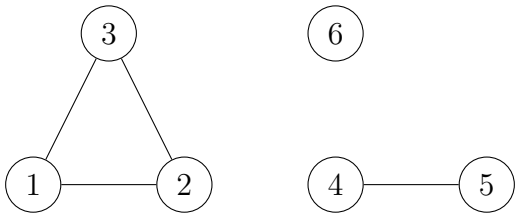


Figure 10.2: Connected components in a graph

10.2 Trees

A **tree** is a special type of graph satisfying two constraints: it is **acyclic**, meaning there are no cycles, and the number of edges is one less than the number of nodes. Trees satisfy the property that for any two nodes A and B, there is exactly one way to travel between A and B.

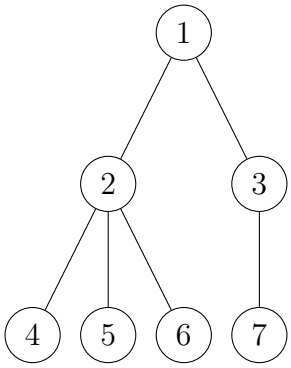


Figure 10.3: A tree graph

The **root** of a tree is the one vertex that is placed at the top, and is where we usually start our tree traversals from. Usually, problems don't tell us where the tree is rooted at, and it usually doesn't matter either; trees can be arbitrarily rooted (here, we'll use the convention of rooting at index 1).

Every node except the root node has a **parent**. The parent of a node s is defined as follows: On the path from the root to s, the node that is one closer to the root than s is the parent of s. Each non-root node has a unique parent.

Child nodes are the opposite. They lie one farther away from the root than their parent node. Unlike parent nodes, these are not unique. Each node can have arbitrarily many child nodes, and nodes can also have zero children. If a node s is the parent of a node t, then t is the child node of s.

A **leaf** node is a node that has no children. Leaf nodes can be identified quite easily because there is only one edge adjacent to them.

In our example tree above, node 1 is the root, nodes 2 and 3 are children of node 1, nodes 4, 5, and 6 are children of 2, and node 7 is child of 3. Nodes 4, 5, 6, and 7 are leaf nodes.

10.3 Graph Representations

Usually, in a graph with N edges and M edges, we'll number the nodes 0 through $N-1$. If the problem gives the nodes numbered 1 through N, simply decrease the endpoint node numbers of edges by 1 as you input them, in order to accommodate zero-indexing of arrays. However, in problem statements, input and output, the node labels will usually be 1 through N, so that's what we'll use in our examples.

Graphs will usually be given in an input format similar to the following: First, integers N and M denoting the number of nodes and edges, respectively. Then, M lines, each with integers a and b, representing edges; if the graph is undirected, then there is an edge between nodes a and b, and if the graph is directed, then there is an edge from a to b.

For example, the input below would be for the following graph (without the comments):

```
6 7 // 6 nodes, 7 edges
// the following lines represent edges.
1 2
1 4
1 5
2 3
2 4
3 5
4 6
```

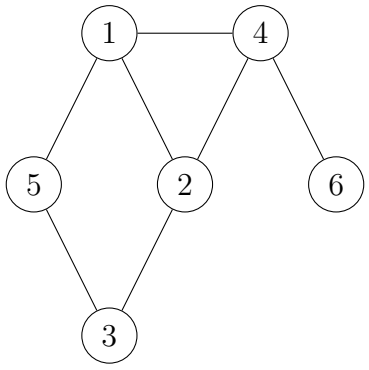


Figure 10.4: The graph corresponding to the above input

Graphs can be represented in three ways: Adjacency List, Adjacency Matrix, and Edge List. Regardless of how the graph is represented, it's important that it be stored globally and statically, because we need to be able to access it from outside the main method, and call the graph searching and traversal methods on it.

Adjacency List

The adjacency list is the most commonly used method of storing graphs. When we use DFS, BFS, Dijkstra's, or other single-source graph traversal algorithms, we'll want to use an adjacency list. In an adjacency list, we maintain a length N array of lists. Each list stores the neighbors of one node. In an undirected graph, if there is an edge between node a and node b, we add a to the list of `b`'s neighbors, and `b` to the list of `a`'s neighbors. In a directed graph, if there is an edge from node a to node `b`, we add `b` to the list of `a`'s neighbors, but not vice versa.

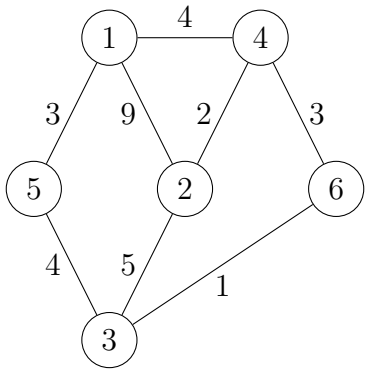


Figure 10.5: An example of a weighted undirected graph

Adjacency list representation of the graph in fig. 10.5:

`adj[0]`	(1, 9), (3, 4), (4, 3)
`adj[1]`	(0, 9), (2, 5), (3, 2)
`adj[2]`	(1, 5), (4, 4), (5, 1)
`adj[3]`	(0, 4), (1, 2), (5, 3)
`adj[4]`	(0, 3), (2, 4)
`adj[5]`	(2, 1), (3, 3)

Adjacency lists take up $O(N + M)$ space, and. In an adjacency list, we can find (and iterate through) the neighbors of a node easily. Hence, the adjacency list is the graph representation we should be using most of the time.

Often, we'll want to maintain a array `visited`, which is a boolean array representing whether each node has been visited. When we visit node `k` (0-indexed), we mark `visited[k]` true, so that we know not to return to it.

Code for setting up an adjacency list is as follows:

```
int n, m; // number of nodes and edges
vector<int>[MAXN] adj; // adjacency list where MAXN is max possible # of nodes
bool visited[MAXN] // visited array of size MAXN as well (use MAXN for global
↪  declaration)

int main(){
    cin >> n; // reads in number of nodes
```

```
    cin >> m; // reads in number of edges
    for(int i = 0; i < m; i++){ // reading in each of the m edges
        int a, b;
        cin >> a >> b;
        a--; b--; // we subtract 1 because our array is zero-indexed
        adj[a].push_back(b);
        adj[b].push_back(a); // omit this line if the graph is directed
    }
    return 0;
}
```

If we're dealing with a weighted graph, we'll declare an Edge class or struct that stores two variables: the second endpoint of the edge, and the weight of the edge, and we store an array of lists of edges rather than an array of lists of integers.

```
struct Edge
{
    int to, weight;
    Edge(int dest, int w):
    to(dest), weight(w)
    {
    }
};
```

Adjacency Matrix

Another way of representing graphs is the adjacency matrix, which is an N by N 2-dimensional array that stores for each pair of indices (a, b), stores whether there is an edge between a and b. Start by initializing every entry in the matrix to zero (this is done automatically in Java), and then for undirected graphs, for each edge between indices a and b, set `adj[a][b]` and `adj[b][a]` to 1 (if unweighted) or the edge weight (if weighted). If the graph is directed, for an edge from a to b, only set `adj[a][b]`.

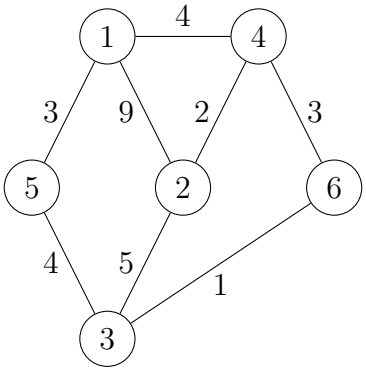


Figure 1.5 repeated for convenience

Adjacency matrix representation of the graph in fig. 1.5:

×	0	1	2	3	4	5
0	0	9	0	4	3	0
1	9	0	5	2	0	0
2	0	5	0	0	4	1
3	4	2	0	0	0	3
4	3	0	4	0	0	0
5	0	0	1	3	0	0

At the Silver level, we generally won't be using the adjacency matrix much, but it's helpful to know if it does come up. The primary use of the adjacency matrix is the Floyd-Warshall algorithm, which is beyond the scope of this book.

Code for setting up an adjacency matrix is as follows:

```
int n, m; // number of nodes and edges
int[MAXN][MAXN] adj; // adj matrix of size MAXN by MAXN in order to globally
↪  declare

int main(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){ // read in each of the m edges
        int a, b;
        cin >> a >> b;
        a--; b--; // we subtract 1 because our array is zero-indexed
        adj[a][b] = 1; // or set equal to w if graph is weighted
        adj[b][a] = 1; // or set equal to w if graph is weighted;
        // ignore above line if graph is directed
    }
    return 0;
}
```

Edge List

The last graph representation is the edge list. Usually, we use this in weighted undirected graphs when we want to sort the edges by weight (for DSU, for example). In the edge list, we simply store a single list of all the edges, in the form (a, b, w) where a and b are the nodes that the edge connects, and w is the edge weight. Note that in an edge list, we do NOT add each edge twice; there is only one place for us to add the edges, so we only do so once.

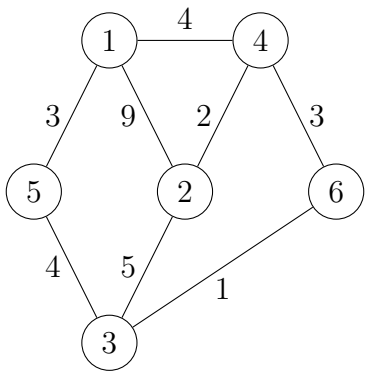


Figure 1.5 repeated for convenience

Edge list representation of the graph in fig. 1.5:

$$(0,1,9),(0,3,4),(0,4,3),(1,3,2),(3,5,3),(2,4,4),(2,1,5),(2,5,1)$$

We'll need an edge struct, such as the following:

```
struct Edge{
    int a, b, w;
    Edge(int start, int end, int weight):
    a(start), b(end), w(weight)
    {
    }
    bool operator<(const Edge & e){ // sort order is ascending, by weight
    // to sort in descending order, just negate the value of the compare
    ↪ function.
        return w < e.w;
    }
};
```

Code for the edge list is as follows, using the above edge class:

```
int n, m; // number of nodes and edges
vector<Edge> edges;

int main(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){ // for each of the m edges
        int a, b, w;
        cin >> a >> b >> w;
        a--; b--; // we subtract 1 because our array is zero-indexed
        edges.push_back(Edge(a, b, w)); // add the edge to the list
    }
    sort(edges.begin(), edges.end());
```

```
    return 0;
}
```

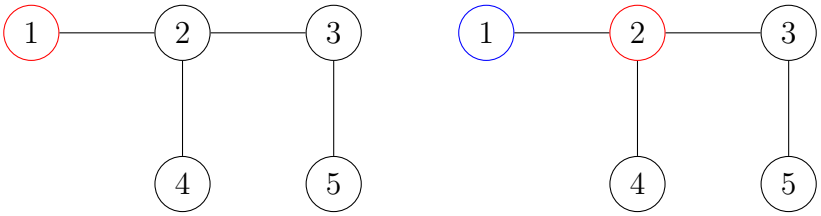
10.4 Graph Traversal Algorithms

Graph traversal is the process of visiting or checking each vertex in a graph. This is useful when we want to determine which vertices can be visited, whether there exists a path from one vertex to another, and so forth. There are two algorithms for graph traversal, namely depth-first search (DFS) and breadth-first search (BFS).

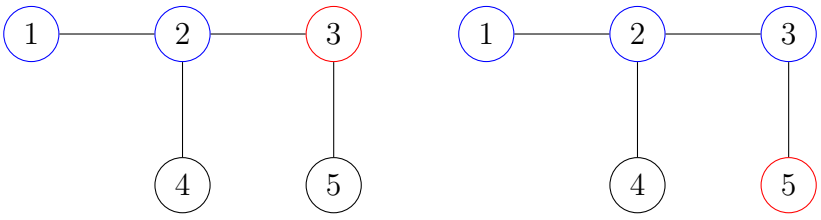
Depth-first search

Depth-first search continues down a single path as far as possible; once it has no more vertices to visit along that path, it backtracks until it finds more vertices to visit. Depth-first search will process all nodes that are reachable (connected by edges) to the starting node. Let's look at an example of how this works. Depth first-search can start at any node, but by convention we'll start the search at node 1. We'll use the following color scheme: blue for nodes we have already visited, red for nodes we are currently processing, and black for nodes that have not been visited yet.

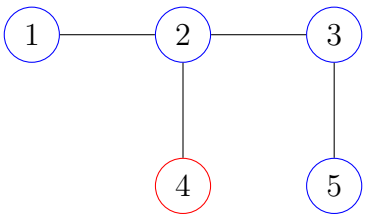
The DFS starts from node 1 and then proceeds to node 2, as it's the only neighbor of node 1:



Now, the DFS goes to node 3 and then 5, following a single path to the end until it has no more nodes to process:



Lastly, the DFS backtracks to visit node 4, which was skipped over previously.



Depth-first search is implemented recursively because it allows for much simpler and shorter code. The algorithm is as follows:

Algorithm: Recursive implementation for depth-first traversal of a graph

Function DFS
  Input : start, the 0-indexed number of the starting vertex
  visted(start) ← true
  foreach *vertex k adjacent to* start **do**
    if *visited(k) is false* **then**
      DFS (k)
    end
  end

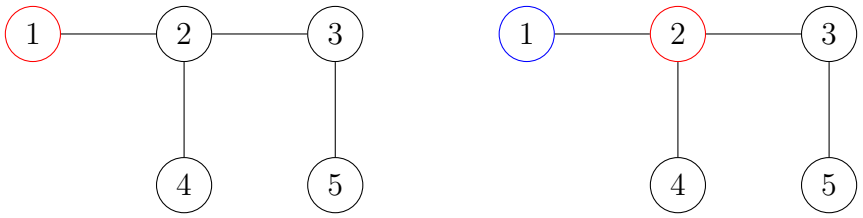
Code:

```
void dfs(int node){
    visited[node] = true;
    for(int next : adj[node]){
        if(!visited[next]){
            dfs(next);
        }
    }
}
```

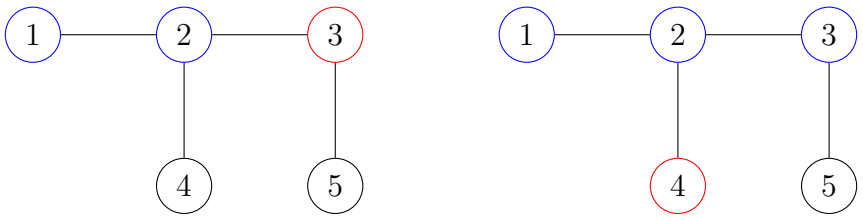
Breadth-first search

Breadth-first search visits nodes in order of distance away from the starting node; it first visits all nodes that are one edge away, then all nodes that are two edges away, and so on.

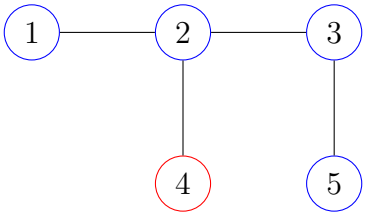
Let's use the same example graph that we used earlier: The BFS starts from node 1 and then proceeds to node 2, as it's the only neighbor of node 1:



Now, the BFS goes to node 3, and then node 4, because both of them are two edges away from node 1:



Lastly, the BFS visits node 5, which is farthest.



The breadth-first search algorithm cannot be implemented recursively, so it's significantly longer. Thus, when both BFS and DFS work, DFS is usually the better option.

BFS can be used for finding the distance away from a starting node for all nodes in an unweighted graph, as we show below:

The algorithm is as follows:

Algorithm: Breadth-first traversal of a graph

Function `BFS`
- **Input :** start, the 0-indexed number of the starting vertex
- **foreach** *vertex* v **do**
  - $\text{dist}[v] \leftarrow -1$
  - $\text{visited}[v] \leftarrow$ false
- **end**
- $\text{dist}[\text{start}] \leftarrow 0$
- Let q be a queue of integers
- Add k to q
- **while** q *is not empty* **do**
  - Pop the first element from q, call it v
  - **foreach** *neighbor* u *of* v **do**
    - **if** *node* u *has not yet been visited* **then**
      - $\text{dist}[u] \leftarrow \text{dist}[v] + 1$
      - Add u to q
    - **end**
  - **end**
- **end**

Once the BFS finishes, the array dist contains the distances from the start node to each node.

Example code is below. Note that the array `dist[]` is initially filled with -1's to denote that none of the nodes have been processed yet.

```
void bfs(int start){
    memset(dist, -1, sizeof dist) // fill distance array with -1's
    queue<int> q;
    dist[start] = 0;
    q.push(start);
    while(!q.empty()){
```

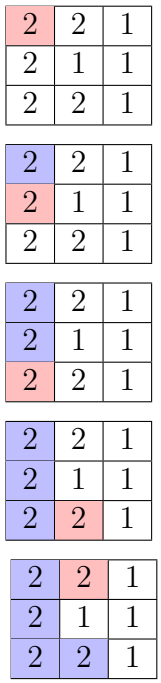
```
        int v = q.front();
        q.pop();
        for(int e : adj[v]){
            if(dist[e] == -1){
                dist[e] = dist[v] + 1;
                q.add(e);
            }
        }
    }
}
```

Iterative DFS

If you encounter stack overflows while using recursive DFS, you can write an iterative DFS, which is just BFS but with nodes stored on a stack rather than a queue.

10.5 Floodfill

Floodfill is an algorithm that identifies and labels the connected component that a particular cell belongs to, in a multi-dimensional array. Essentially, it's DFS, but on a grid, and we want to find the connected component of all the connected cells with the same number. For example, let's look at the following grid and see how floodfill works, starting from the top-left cell. The color scheme will be the same: red for the node currently being processed, blue for nodes already visited, and uncolored for nodes not yet visited.



2	2	1
2	1	1
2	2	1

As opposed to an explicit graph where the edges are given, a grid is an implicit graph. This means that the neighbors are just the nodes directly adjacent in the four cardinal directions.

Usually, grids given in problems will be N by M, so the first line of the input contains the numbers N and M. In this example, we will use an two-dimensional integer array to store the grid, but depending on the problem, a two-dimensional character array or a two-dimensional boolean array may be more appropriate. Then, there are N rows, each with M numbers containing the contents of each square in the grid. Example input might look like the following (varies between problems):

```
3 4
1 1 2 1
2 3 2 1
1 3 3 3
```

And we'll want to input the grid as follows:

```
int[MAXN][MAXM] grid;
int n, m;

int main(){
    cin >> n >> m;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            cin >> grid[i][j];
        }
    }
    return 0;
}
```

When doing floodfill, we will maintain an $N \times M$ array of `bool`s to keep track of which squares have been visited, and a global variable to maintain the size of the current component we are visiting. Make sure to store the grid, the visited array, dimensions, and the current size variable globally.

This means that we want to recursively call the search function from the squares above, below, and to the left and right of our current square. The algorithm to find the size of a connected component in a grid using floodfill is as follows (we'll also maintain a 2d visited

array):

Algorithm: Floodfill of a graph

```
Function main
    // Input/output, global vars, etc hidden
    for i ← 0 to n − 1 do
        for j ← 0 to m − 1 do
            if the square at (i, j) is not visited then
                currentSize ← 0
                floodfill(i, j, grid[i][j])
                Process the connected component
            end
        end
    end
Function floodfill
    Input  : r, c, color
    // row and column index of starting square, target color
    if r or c is out of bounds then
        return
    end
    if the cell at (r, c) is the wrong color then
        return
    end
    if the square at (r, c) has already been visited then
        return
    end
    visited[r][c] ← true
    currentSize ← currentSize + 1
    floodfill(r, c + 1, color)
    floodfill(r, c − 1, color)
    floodfill(r − 1, c, color)
    floodfill(r + 1, c, color)
```

The code below shows the global/static variables we need to maintain while doing floodfill, and the floodfill algorithm itself.

```
int[MAXN][MAXM] grid; // the grid itself
int n, m; // grid dimensions, rows and columns
bool[MAXN][MAXM] visited; // keeps track of which nodes have been visited
int currentSize = 0; // reset to 0 each time we start a new component

void floodfill(int r, int c, int color){
    if(r < 0 || r >= n || c < 0 || c >= m) return; // if outside grid
    if(grid[r][c] != color) return; // wrong color
    if(visited[r][c]) return; // already visited this square
```

```
    visited[r][c] = true; // mark current square as visited
    currentSize++; // increment the size for each square we visit
    // recursively call floodfill for neighboring squares
    floodfill(r, c+1, color);
    floodfill(r, c-1, color);
    floodfill(r-1, c, color);
    floodfill(r+1, c, color);
}

int main(){
    /**
    * input code and other problem-specific stuff here
    */
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            if(!visited[i][j]){
                currentSize = 0;
                floodfill(i, j, grid[i][j]);
                // start a floodfill if the square hasn't
                // already been visited, and then
                // store or otherwise use the component size for whatever
                // it's needed for
            }
        }
    }
    return 0;
}
```

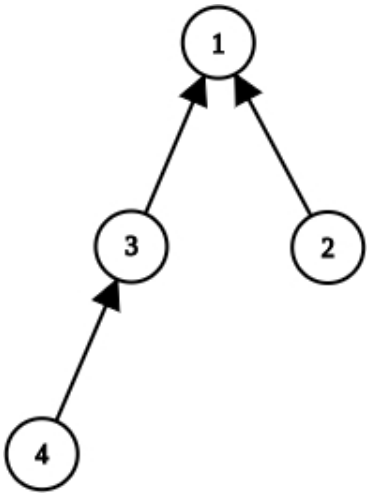
10.6 Disjoint-Set Data Structure

Let's say we want to construct a graph, one edge at a time. We also want to be able to add additional nodes, and query whether two nodes are connected. We can naively solve this problem by adding the edges and running a floodfill each time, before finally checking whether two nodes have the same color. This yields a time complexity of $O(nm)$ for a graph of n nodes and m edges.

However, we can do better than this using a data structure known as Disjoint-Set Union, or DSU for short. This data structure supports two operations:

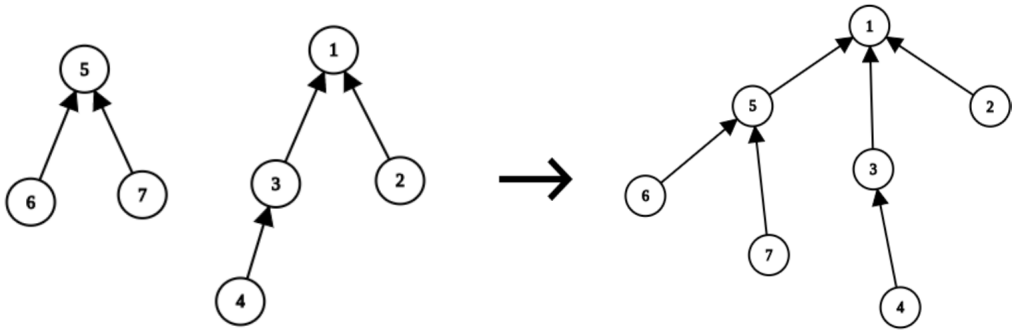
- Add an edge between two nodes.
- Check if two nodes are connected.

To achieve this, we store sets as trees, with the root of the tree representing the "parent" of the set. Initially, we store each node as its own set. Then, we combine their sets when we add an edge between two nodes. The image below illustrates this structure.



In this graph, 1 is the parent of the set containing 3, 2, and 4.

To implement this, let's store the parent of each node in the tree represented by that node's set. Then, to merge two sets, we set the parent of one tree's root to the other tree's root, like so:



The following methods demonstrate this idea:

```
int[MAXN] parent; //stores the root of each set

void initialize(int N){
    for(int i = 0; i < N; i++){
        parent[i] = i; //initially, the root of each set is the node itself
    }
}

int find(int x){ //finds the root of the set of x
    if(x == parent[x]){ //if x is the parent of itself, it is the root
        return x;
    }
    else{
        return find(parent[x]); //otherwise, recurse to the parent of x
    }
```

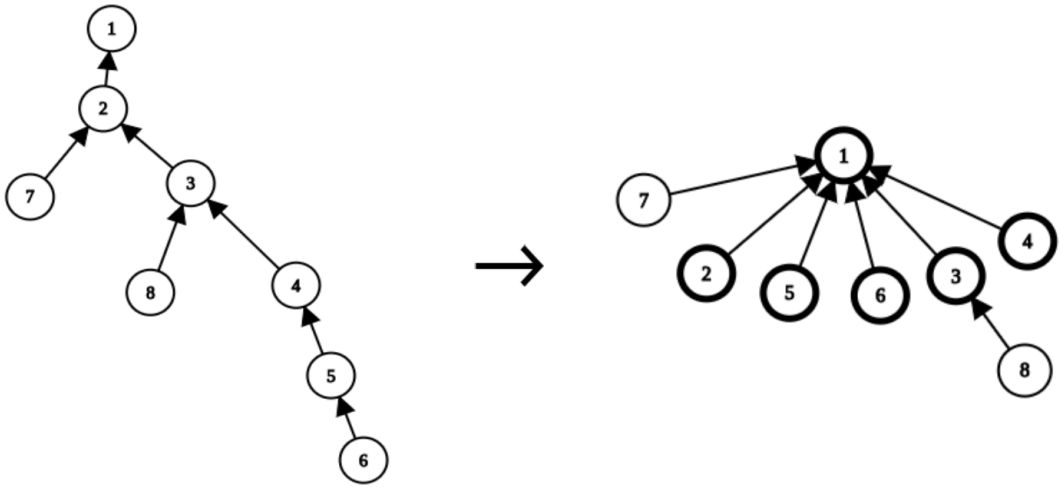
```
}

void union(int a, int b){ //merges the sets of a and b
    int c = find(a); //find the root of a
    int d = find(b); //find the root of b
    if(c != d){
        parent[d] = c; //merge the sets by setting the parent of d to c
    }
}
```

However, this naive implementation of a DSU isn't much better than simply running a floodfill. As the recursing up the tree of a set to find it's root can be time-consuming for trees with long chains, the runtime ultimately degrades to still being $O(nm)$ for n nodes and m edges.

Now that we understand the general idea of a DSU, we can improve the runtime of this implementation using an optimization known as path compression. The general idea is to reassign nodes in the tree as you are recursively calling the find method to prevent long chains from forming. Here is a rewritten find method representing this idea:

```
int find(int x){
    if(x == parent[x]){
        return x;
    }
    else{
        // we set the direct parent to the root of the set to reduce path length
        return parent[x] = find(parent[x]);
    }
}
```

The following image demonstrates how the tree with parent 1 is compressed after `find(6)` is called. All of the bolded nodes in the final tree were visited during the recursive operation, and now point to the root.



With this new optimization, the runtime reduces to $O(n \log n)$, far better than our naive algorithm. Further optimizations can reduce the runtime of DSU to nearly constant. However, those techniques and the proof of complexity for these optimizations are both unnecessary for and out of the scope of the USACO Silver division, so they will not be included in this book.

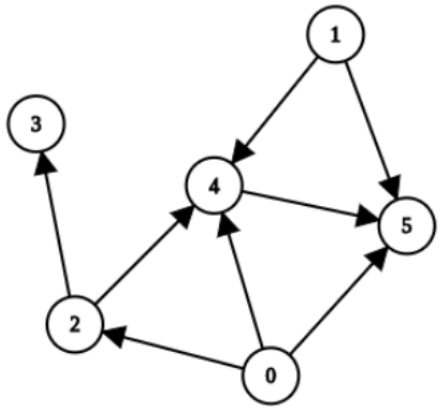
10.7 Other Types of Graphs

Alongside trees, several other types of specific graphs also exist in graph theory. While they are unlikely to show up in the silver division, they're nice to know for future divisions and competitive programming overall. The following sections serve as an introduction into these topics, not a comprehensive tutorial.

Directed Acyclic Graphs (DAG)

To review, a **directed** graph consists of edges that can only be traversed in one direction. Additionally, a **acyclic** graph defines a graph which does not contain cycles, meaning you are unable to traverse across one or more edges and return to the node you started on. Putting these definitions together, a **directed acyclic** graph, sometimes abbreviated as DAG, is a graph which has edges which can only be traversed in one direction and does not contain cycles.

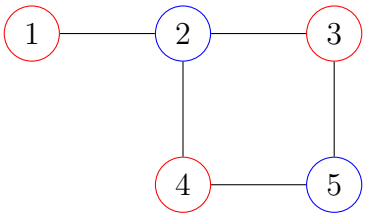
Here is an illustration of one such graph:



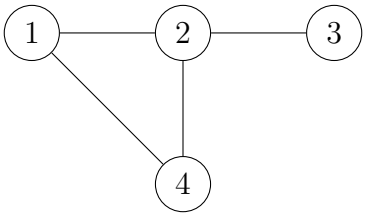
As there does not contain cycles in the graph, a useful property of directed acyclic graphs is that you can construct an ordering of nodes such that for any edge from node u to node v, node u appears in the ordering before node v. More formally, such an ordering is known as a **topological sorting** of a directed acyclic graph, which can be useful for algorithms like dynamic programming. However, as such algorithms are out of the scope of this book, topological sorting simply serves as a clean way of envisioning an ordering of nodes in the graph.

Bipartite Graphs

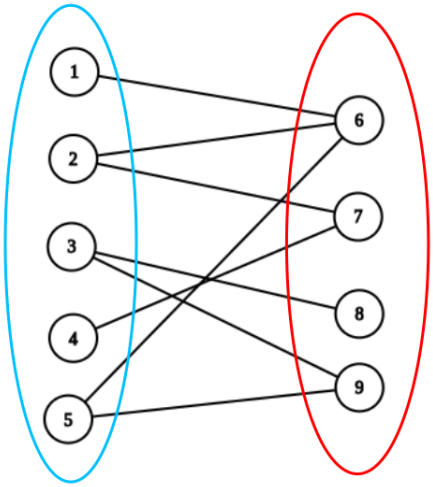
A **bipartite** graph is a graph such that each node can be colored in one of two colors, such that no two adjacent nodes have the same color. For example, the following graph is bipartite:



A graph is bipartite if and only if there are no cycles of odd length. For example, the following graph is not bipartite, because it contains a cycle of length 3.



The following image depicts how a bipartite graph splits vertices into two "groups" depending on their color.



In order to check whether a graph is bipartite, we use a modified breadth-first search.

Algorithm: Bipartiteness check

```
Function bipartite
    Input  : a graph
    Input  : whether the graph is bipartite or not
    Assign color 1 to the starting vertex
    // Use the following modified bfs
    foreach vertex v processed in bfs do
        d ← dist(start, v)
        if d is odd then
            Assign color 2 to vertex v
        else
            Assign color 1 to vertex v
        end
        foreach vertex w adjacent to v do
            if w and v are the same color then
                return false // not bipartite
            end
        end
    end
    return true // bipartite
```

10.8 Problems

DFS/BFS Problems

1. USACO January 2018 Silver Problem 3: MooTube
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=788`
2. USACO December 2016 Silver Problem 3: Moocast
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=668`
3. USACO US Open 2016 Silver Problem 3: Closing the Farm
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=644`

DSU Problems

Many of these problems do not require DSU. However, they become much easier to do if you understand it.

4. USACO US Open Silver Problem 3: The Moo Particle
   `http://usaco.org/index.php?page=viewproblem2&cpid=1040`
5. USACO January 2018 Silver Problem 3: MooTube
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=788`

6. USACO December 2019 December Problem 3: Milk Visits
   http://usaco.org/index.php?page=viewproblem2&cpid=968
7. USACO US Open 2016 Gold Problem 2: Closing the Farm
   http://www.usaco.org/index.php?page=viewproblem2&cpid=646
8. USACO January Contest 2020 Silver Problem 3: Wormhole Sort
   http://www.usaco.org/index.php?page=viewproblem2&cpid=992

Other Graph Problems

9. (Bipartite Graphs) USACO February 2019 Silver Problem 3: The Great Revegetation
   http://www.usaco.org/index.php?page=viewproblem2&cpid=920
10. CodeForces Round 595 (Div. 3) Problem B2: Books Exchange
    https://codeforces.com/problemset/problem/1249/B2

Chapter 11

Prefix Sums

11.1 Prefix Sums

Let's say we have an integer array `arr` with N elements, and we want to process Q queries to find the sum of the elements between two indices a and b, inclusive, with different values of a and b for every query. For the purposes of this chapter, we will assume that the original array is 1-indexed, meaning `arr[0] = 0` (which is a dummy index), and the actual array elements occupy indices 1 through N (this means that the array actually has length $N + 1$).

Let's use the following example 1-indexed array `arr`, with $N = 6$:

Index i	0	1	2	3	4	5	6
`arr[i]`	0	1	6	4	2	5	3

Naively, for every query, we can iterate through all entries from index a to index b to add them up. Since we have Q queries and each query requires a maximum of $O(N)$ operations to calculate the sum, our total time complexity is $O(NQ)$. For most problems of this nature, the constraints will be $N, Q \leq 10^5$, so NQ is on the order of 10^{10}. This is not acceptable; it will almost always exceed the time limit.

Instead, we can use prefix sums to process these array sum queries. We designate a prefix sum array `prefix`. First, because we're 1-indexing the array, set `prefix[0] = 0`, then for indices k such that $1 \leq k \leq n$, define the prefix sum array as follows:

$$\texttt{prefix[k]} = \sum_{i=1}^{k} \texttt{arr[i]}$$

Basically, what this means is that the element at index k of the prefix sum array stores the sum of all the elements in the original array from index 1 up to k. This can be calculated easily in $O(N)$ by the following formula:

$$\texttt{prefix[k]} = \texttt{prefix[k-1]} + \texttt{arr[k]}$$

For the example case, our prefix sum array looks like this:

Index i	0	1	2	3	4	5	6
`prefix[i]`	0	1	7	11	13	18	21

Now, when we want to query for the sum of the elements of `arr` between (1-indexed) indices a and b inclusive, we can use the following formula:

$$\sum_{i=a}^{b} \texttt{arr[i]} = \sum_{i=1}^{b} \texttt{arr[i]} - \sum_{i=1}^{a-1} \texttt{arr[i]}$$

Using our definition of the elements in the prefix sum array, we have

$$\sum_{i=a}^{b} \texttt{arr[i]} = \texttt{prefix[b]} - \texttt{prefix[a-1]}$$

Since we are only querying two elements in the prefix sum array, we can calculate subarray sums in $O(1)$ per query, which is much better than the $O(N)$ per query that we had before. Now, after an $O(N)$ preprocessing to calculate the prefix sum array, each of the Q queries takes $O(1)$ time. Thus, our total time complexity is $O(N+Q)$, which should now pass the time limit.

Let's do an example query and find the subarray sum between indices $a = 2$ and $b = 5$, inclusive, in the 1-indexed *arr*. From looking at the original array, we see that this is $\sum_{i=2}^{5} \texttt{arr[i]} = 6 + 4 + 2 + 5 = 17$.

Index i	0	1	2	3	4	5	6
`arr[i]`	0	1	6	4	2	5	3

Using prefix sums: Using prefix sums: $\texttt{prefix[5]} - \texttt{prefix[1]} = 18 - 1 = 17$.

Index i	0	1	2	3	4	5	6
`prefix[i]`	0	1	7	11	13	18	21

11.2 Two Dimensional Prefix Sums

Now, what if we wanted to process Q queries for the sum over a subrectangle of a N rows by M columns matrix in two dimensions? Let's assume both rows and columns are 1-indexed, and we use the following matrix as an example:

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

Naively, each sum query would then take $O(NM)$ time, for a total of $O(QNM)$. This is too slow.

Let's take the following example region, which we want to sum:

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

Manually summing all the cells, we have a submatrix sum of $7 + 11 + 9 + 6 + 1 + 3 = 37$.

The first logical optimization would be to do one-dimensional prefix sums of each row. Then, we'd have the following row-prefix sum matrix. The desired subarray sum of each row in our desired region is simply the green cell minus the red cell in that respective row. We do this for each row, to get $(28 - 1) + (14 - 4) = 37$.

0	0	0	0	0	0
0	1	6	12	23	31
0	1	8	19	28	32
0	4	10	11	14	16
0	7	12	16	18	21

Now, if we wanted to find a submatrix sum, we could break up the submatrix into a subarray for each row, and then add their sums, which would be calculated using the prefix sums method described earlier. Since the matrix has N rows, the time complexity of this is $O(QN)$. This is better, but still usually not fast enough.

To do better, we can do two-dimensional prefix sums. In our two dimensional prefix sum array, we have

$$\texttt{prefix[a][b]} = \sum_{i=1}^{a} \sum_{j=1}^{b} \texttt{arr[i][j]}$$

This can be calculated as follows for row index $1 \le i \le n$ and column index $1 \le j \le m$:

$$\texttt{prefix[i][j]} = \texttt{prefix[i-1][j]} + \texttt{prefix[i][j-1]} - \texttt{prefix[i-1][j-1]} + \texttt{arr[i][j]}$$

The submatrix sum between rows a and A and columns b and B, can thus be expressed as follows:

$$\sum_{i=a}^{A} \sum_{j=b}^{B} \texttt{arr[i][j]} = \texttt{prefix[A][B]} - \texttt{prefix[a-1][B]} - \texttt{prefix[A][b-1]} + \texttt{prefix[a-1][b-1]}$$

Summing the blue region from above using the 2d prefix sums method, we add the value of the green square, subtract the values of the red squares, and then add the value of the gray square. In this example, we have $65 - 23 - 6 + 1 = 37$, as expected.

0	0	0	0	0	0
0	1	6	12	23	31
0	2	14	31	51	63
0	6	24	42	65	79
0	13	36	58	83	100

Since no matter the size of the submatrix we are summing, we only need to access 4 values of the 2d prefix sum array, this runs in $O(1)$ per query after an $O(NM)$ preprocessing. This is fast enough.

11.3 Problems

1. USACO December 2015 Silver Problem 3: Breed Counting
   http://usaco.org/index.php?page=viewproblem2&cpid=572
2. USACO January 2016 Silver Problem 2: Subsequences Summing to Sevens
   http://usaco.org/index.php?page=viewproblem2&cpid=595
3. USACO December 2017 Silver Problem 1: My Cow Ate My Homework
   http://www.usaco.org/index.php?page=viewproblem2&cpid=762
4. USACO January 2017 Silver Problem 2: Hoof, Paper, Scissors
   http://www.usaco.org/index.php?page=viewproblem2&cpid=691
5. (2D Prefix Sums) USACO February 2019 Silver Problem 2: Painting the Barn
   http://www.usaco.org/index.php?page=viewproblem2&cpid=919

Chapter 12

Binary Search

12.1 Binary Search on the Answer

You're probably already familiar with the concept of binary searching for a number in a sorted array. However, binary search can be extended to binary searching on the answer itself. When we binary search on the answer, we start with a search space, where we know the answer lies in. Then, each iteration of the binary search cuts the search space in half, so the algorithm tests $O(\log N)$ values, which is efficient and much better than testing each possible value in the search space.

Similarly to how binary search on an array only works on a sorted array, binary search on the answer only works if the answer function is monotonic. Let's say we have a function `check(x)` that returns true if the answer of x is possible, and false otherwise. Usually, in such problems, we'll want to find the maximum or minimum value of x such that `check(x)` is true. What monotonicity means, is as follows:

If we want the maximum possible value of x, then

- If `check(x)` is true, then `check(y)` is true for all $y \leq x$.
- If `check(x)` is false, then `check(y)` is false for all $y \geq x$.

In other words, we want to reduce the search space to something of the following form, using a check function as we described above.

true true true true true false false false false

Then, we find the point at which true becomes false, using binary search.

Below, we present two algorithms for binary search. The first implementation may be more intuitive, because it's closer to the binary search most students learned, while the

second implementation is shorter.

Algorithm: Binary searching for the answer

```
Function binarySearch1
    left ← lower bound of search space
    right ← upper bound of search space
    ans ← −1
    while left ≤ right do
        mid ← (left + right)/2
        if check(mid) then
            left ← mid + 1
            ans ← mid
        else
            right ← mid − 1
    end
    return ans
```

Algorithm: Binary searching for the answer

```
Function binarySearch2
    pos ← 0
    max ← upper bound of search space
    for (a = max; a ≥ 1; a /= 2) do
        while check(pos + a) do
            pos ← pos + a
        end
    end
    return pos
```

If instead we're looking for the minimum x that satisfies some condition, then

- If `check(x)` is true, then `check(y)` is true for all $y \geq x$.
- If `check(x)` is false, then `check(y)` is false for all $y \leq x$.

The binary search function for this is very similar. Find the last value of x such that `check(x)` is false, then the answer is $x + 1$.

12.2 Example

Source: Codeforces Round 577 (Div. 2) Problem C
`https://codeforces.com/contest/1201/problem/C`

Given an array `arr` of n integers, where n is odd, we can perform the following operation on it k times: take any element of the array and increase it by 1. We want to make the median of the array as large as possible, after k operations.

Constraints: $1 \leq n \leq 2 \cdot 10^5, 1 \leq k \leq 10^9$ and n is odd.

The solution is as follows: we first sort the array in ascending order. Then, we binary search for the maximum possible median. We know that the number of operations required to raise the median to x increases monotonically as x increases, so we can use binary search. For a given median value x, the number of operations required to raise the median to x is

$$\sum_{i=(n+1)/2}^{n} \max(0, x - \texttt{arr[i]})$$

If this value is less than or equal to k, then x can be the median, so our check function returns true. Otherwise, x cannot be the median, so our check function returns false.

Solution code (using the second implementation of binary search):

```
int n;
long long k;
vector<long long> v;
int main() {
    cin >> n >> k;
    for(int i = 0; i < n; i++){
        int t;
        cin >> t;
        v.push_back(t);
    }
    sort(v.begin(), v.end());

    cout << search() << '\n';
}

// binary searches for the correct answer
long long search(){
    long long pos = 0; long long max = 2E9;
    for(long long a = max; a >= 1; a /= 2){
        while(check(pos+a)) pos += a;
    }
    return pos;
}

// checks whether the number of given operations is sufficient
// to raise the median of the array to x
bool check(long long x){
    long long operationsNeeded = 0;
    for(int i = (n-1)/2; i < n; i++){
        operationsNeeded += max(0, x-v[i]);
    }
    if(operationsNeeded <= k) return true;
    else return false;
}
```

12.3 Problems

1. USACO December 2018 Silver Problem 1: Convention
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=858`
2. USACO January 2016 Silver Problem 1: Angry Cows
   `http://usaco.org/index.php?page=viewproblem2&cpid=594`
3. USACO January 2017 Silver Problem 1: Cow Dance Show
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=690`
4. Educational Codeforces Round 60 Problem C: Magic Ship
   `https://codeforces.com/problemset/problem/1117/C` (Also uses prefix sums)
5. USACO January 2020 Silver Problem 2: Loan Repayment
   `http://www.usaco.org/index.php?page=viewproblem2&cpid=991`
   (Warning: extremely difficult for silver)

Chapter 13

Elementary Number Theory

13.1 Prime Factorization

A number a is called a **divisor** or a **factor** of a number b if b is divisible by a, which means that there exists some integer k such that $b = ka$. Conventionally, 1 and n are considered divisors of n. A number $n > 1$ is **prime** if its only divisors are 1 and n. Numbers greater than 1 that are not prime are **composite**.

Every number has a unique **prime factorization**: a way of decomposing it into a product of primes, as follows:

$$n = {p_1}^{a_1} {p_2}^{a_2} \cdots {p_k}^{a_k}$$

where the p_i are distinct primes and the a_i are positive integers.

Now, we will discuss how to find the prime factorization of an integer.

Algorithm: Finds the prime factorization of a number

```
Function factor
    Input  : n, the number to be factorized
    Output : v, a list of all the prime factors
    v ← empty list
    for i ← 2 to ⌊√n⌋ do
        while n is divisible by i do
            n ← n/i
            Add i to the list v
        end
    end
    return v;
```

This algorithm runs in $O(\sqrt{n})$ time, because the for loop checks divisibility for at most $\sqrt{n}$ values. Even though there is a while loop inside the for loop, dividing n by i quickly reduces the value of n, which means that the outer for loop runs less iterations, which actually speeds up the code.

Let's look at an example of how this algorithm works, for $n = 252$.

i	n	v
2	252	$\{\}$
2	126	$\{2\}$
2	63	$\{2, 2\}$
3	21	$\{2, 2, 3\}$
3	7	$\{2, 2, 3, 3\}$

At this point, the for loop terminates, because i is already 3 which is greater than $\lfloor\sqrt{7}\rfloor$. In the last step, we add 7 to the list of factors v, because it otherwise won't be added, for a final prime factorization of $\{2, 2, 3, 3, 7\}$.

13.2 GCD and LCM

The **greatest common divisor (GCD)** of two integers a and b is the largest integer that is a factor of both a and b. In order to find the GCD of two numbers, we use the Euclidean Algorithm, which is as follows:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

This algorithm is very easy to implement using a recursive function, as follows:

```
int gcd(int a, int b){
    if(b == 0) return a;
    return gcd(b, a % b);
}
```

Finding the GCD of two numbers can be done in $O(\log n)$ time, where $n = \min(a, b)$.

The **least common multiple (LCM)** of two integers a and b is the smallest integer divisible by both a and b.

The LCM can easily be calculated from the following property with the GCD:

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

If we want to take the GCD or LCM of more than two elements, we can do so two at a time, in any order. For example,

$$\gcd(a_1, a_2, a_3, a_4) = \gcd(a_1, \gcd(a_2, \gcd(a_3, a_4)))$$

13.3 Modular Arithmetic

In modular arithmetic, instead of working with integers themselves, we work with their remainders when divided by m. We call this taking modulo m. For example, if we take $m = 23$, then instead of working with $x = 247$, we use $x \bmod 23 = 17$. Usually, m will be a large prime, given in the problem; the two most common values are $10^9 + 7$, and $998\,244\,353$. Modular arithmetic is used to avoid dealing with numbers that overflow built-in data types, because we can take remainders, according to the following formulas:

$$\begin{aligned}(a + b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\ (a - b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\ (a \cdot b) \pmod m &= ((a \bmod m) \cdot (b \bmod m)) \bmod m \\ a^b \bmod m &= (a \bmod m)^b \bmod m\end{aligned}$$

Under a prime moduli, division does exist; however it's rarely used in problems and is beyond the scope of this book.

13.4 Problems

1. CodeForces VK Cup 2012 Wildcard Round 1
   `https://codeforces.com/problemset/problem/162/C`

Chapter 14

Additional Silver Topics

14.1 Two Pointers

The two pointers method iterates two pointers across an array, to track the start and end of an interval, or two values in a sorted array that we are currently checking. Both pointers are monotonic; meaning each pointer starts at one end of the array and only move in one direction.

2SUM Problem

Given an array of N elements ($1 \leq N \leq 10^5$), find two elements that sum to X, if they exist. We can solve this problem using two pointers; set one pointer at the beginning and one pointer at the end of the array. Then, we consider the sum of the numbers at the indices of the pointers. If the sum is too small, advance the left pointer towards the right, and if the sum is too large, advance the right pointer towards the left. Repeat until either the correct sum is found, or the pointers meet (in which case there is no solution).

Let's take the following example array, where $N = 6$ and $X = 15$

1	7	11	10	5	13

First, we sort the array:

1	5	7	10	11	13

We then place the left pointer at the start of the array, and the right pointer at the end of the array.

1	5	7	10	11	13

Then, run and repeat this process: If the sum of the pointer elements is less than X, move the left pointer one step to the right. If the sum is greater than X, move the right pointer one step to the left. The example is as follows. First, the sum $1 + 13 = 14$ is too small, so we move the left pointer one step to the right.

1	5	7	10	11	13

Now, $5 + 13 = 18$ overshoots the sum we want, so we move the right pointer one step to the left.

1	5	7	10	11	13

At this point we have $5 + 11 = 16$, still too big. We continue moving the right pointer to the left.

1	5	7	10	11	13

Now, we have the correct sum, and we are done.

Code is as follows:

```
int left = 0; int right = n-1;
while(left < right){
    if(arr[left] + arr[right] == x){
        break;
    } else if(arr[left] + arr[right] < x){
        left++;
    } else {
        right--;
    }
}
// if left >= right after the loop ends, no answer exists.
```

Subarray Sum

Given an array of N ($1 \leq N \leq 10^5$) positive elements, find a contiguous subarray that sums to X. We can do this in a similar manner to how we did the 2SUM problem: except this time we start both pointers at the left, and the pointers mark the beginning and end of the subarray we are currently checking. We advance the right pointer one step to the right if the total of the current subarray is too small, advance the left pointer one step to the right if the current total is too large, and we are done when we find the correct total. Let's look at the following example, for $N = 6$ and $X = 17$.

This subarray contains only the first element. We advance the right pointer towards the right, until our current sum reaches or exceeds the target sum of X.

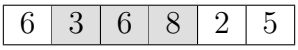
6	3	6	8	2	5

6	3	6	8	2	5

6	3	6	8	2	5

6	3	6	8	2	5

Now, our subarray contains a sum of 23, so we've gone too far. We now advance the left pointer towards the right, to eliminate the leftmost elements from the subarray we are considering.



Now, we have the subarray with the sum we want.

Code is as follows. Note that we can use a for loop for the right pointer, a while loop to advance the left pointer while the sum is too large, and a variable to maintain the sum of the current subarray.

```
int left = 0; int right = 0; int sum = arr[0];
for(right = 0; right < n; right++){
    sum += arr[right];
    while(sum > x && left < right){
        sum -= arr[left];
        left++;
    }
    if(sum == x){
        break;
    }
}
```

Maximum subarray sum

Another problem that isn't quite solved by two pointers, but is somewhat related, is the maximum subarray sum problem.

Given an array of N integers ($1 \leq N \leq 10^5$), which can be positive or negative, find the maximum sum of a contiguous subarray.

We can solve this problem using Kadane's algorithm, which works as follows: we iterate through the elements of the array, and for each index i, we maintain the maximum subarray sum of a subarray ending at i in the variable *current*, and the maximum subarray sum of a subarray ending at or before i, in the variable *best*.

Example code is below.

```
int best = 0, current = 0;
for(int i = 0; i < n; i++){
    current = max(0, current + arr[i]);
    best = max(best, current);
}
```

14.2 Line sweep

Line sweep is the technique of sorting a set of points or line segments and then processing them in order (this usually means from left to right). The name line sweep comes from the fact that we are sweeping an imaginary vertical line across the plane containing the points or segments.

To describe this technique, we'll be using the 2019 US Open problem, "Cow Steeplechase II".

`http://usaco.org/index.php?page=viewproblem2&cpid=943`

In this problem, we are given some line segments and asked to find one line segment and remove it such that the resulting segments form no intersections. It is guaranteed that this is always possible.

First of all, let's observe it is sufficient to find any two line segments that intersect. Once we have done this, the solution is guaranteed to be one of these two segments. Then, out of the two, the segment with multiple intersections is the answer (because removing any other segment decreases the number of intersections by at most 1, and only removing the segment with multiple intersections ensures there are no intersections).

If both segments have one intersection, that means the intersect with each other, so we should return the one with the smallest index (as per the problem statement). Now, the problem reduces to two parts: checking if two line segments intersect, and processing the line segments using a line sweep.

Checking If Two Segments Intersect

To check if two line segments intersect, we will use a fact from algebra: if we have the points $A = (x_a, y_a)$, $B = (x_b, y_b)$, and $C = (x_c, y_c)$, then the (signed) area of $\triangle ABC$, denoted $[ABC]$, is $(x_b - x_a)(y_c - y_a) - (x_c - x_a)(y_b - y_a)$. This can be derived from the cross product of the vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$.

The part that will help us is the fact that this area is signed, which means that $[ABC]$ is

positive if A, B, and C occur in counterclockwise order,

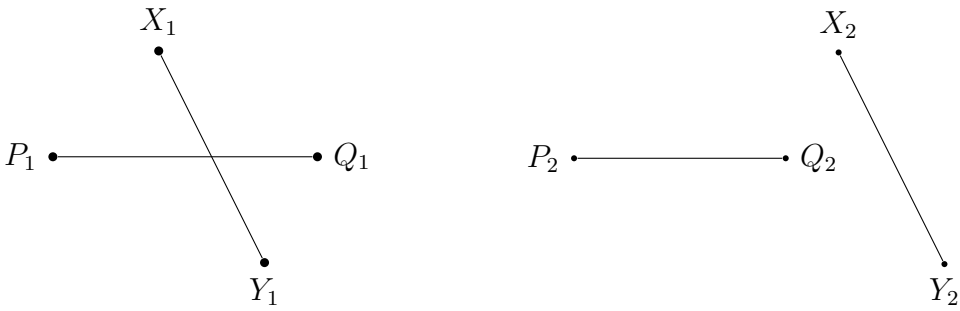
negative if A, B, and C occur in clockwise order, and

zero if A, B, and C are collinear.

Then, the key observation is that two segments $\overline{PQ}$ and $\overline{XY}$ intersect if the two conditions hold:

- $[XPQ]$ and $[YPQ]$ have different signs
- $[PXY]$ and $[QXY]$ have different signs

For example, in the figure below, $[X_1P_1Q_1]$ and $[Q_1X_1Y_1]$ are positive because their vertices occur in counterclockwise order, and $[Y_1P_1Q_1]$ and $[P_1X_1Y_1]$ are negative because their vertices occur in clockwise order. Therefore, we know that $\overline{X_1Y_1}$ and $\overline{P_1Q_1}$ intersect. Similarly, on the right, we know that $[P_2X_2Y_2]$ and $[Q_2X_2Y_2]$ have vertices both going in clockwise order, so their signed areas are the same, and therefore $\overline{P_2Q_2}$ and $\overline{X_2Y_2}$ don't intersect.



If the two conditions hold and some of the signs are zero, then this means that the segments intersect at their endpoints. If the problem does not count these as intersecting, then consider zero to have the same sign as both positive and negative.

However, there is a special case. If the signs of all four areas are zero, then all four points lie on a line. To check if they intersect in this case, we just check whether one point is between the others. In particular, we check if P or Q is on XY or if X is on PQ. We don't need to check if Y is on PQ because if the segments do intersect, we will have two instances of points on the other segments.

Here's a full implementation:

```
struct Point
{
    int x, y;
    Point (int xst, int yst):
    x(xst), y(yst)
    {
    }
};

int sign(Point A, Point B, Point C) {
    int area = (B.x-A.x) * (C.y-A.y) - (C.x-A.x) * (B.y-A.y);
    if (area > 0) return 1;
    if (area < 0) return -1;
    return 0;
}

bool between(Point P, Point X, Point Y) {
    return ((X.x <= P.x && P.x <= Y.x) || (Y.x <= P.x && P.x <= X.x))
        && ((X.y <= P.y && P.y <= Y.y) || (Y.y <= P.y && P.y <= X.y));
}

bool intersectQ(Point P, Point Q, Point X, Point Y) {
    int signs[4] = {sign(P, X, Y), sign(Q, X, Y), sign(X, P, Q), sign(Y, P, Q)};
    if (signs[0] == 0 && signs[1] == 0 && signs[2] == 0 && signs[3] == 0)
        return between(P, X, Y) || between(Q, X, Y) || between(X, P, Q);
    return signs[0] != signs[1] && signs[2] != sign[3];
}
```

Processing Line Segments

Let's break apart the N line segments into $2N$ events, one for each start and end point. We'll store whether some event is a start point or an end point, and which start points correspond to each end point.

Then, we process the endpoints in order of x coordinate from left to right, maintaining a set of currently processed segments, which is sorted by y. When we hit an endpoint, we either add or remove a segment from the set, depending on whether we start or end a segment. Every time we add a segment, we check it for intersection with the segment above it and the segment below it. In addition, every time we remove a segment, we check the segment above it and the segment below it for intersection. Once we find an intersection, we are done.

14.3 Bitwise Operations and Subsets

Binary Representations of Integers

In programming, numbers are stored as binary representations. This means that a number x is represented as

$$x = \sum_{i=0}^{n} a_i 2^i,$$

where the a_is are either 0 or 1 and $n = \lfloor \log_2 x \rfloor$. There is also a bit at the beginning designating whether the number is positive or negative, called the sign bit. In this section, we work only with positive numbers for the sake of simplicity, so we will omit the sign bit in binary representations.

For example:

$$17 = 2^4 + 2^0 = 10001_2$$

In a 32-bit integer, this is actually written as

$$00000000000000000000000000010001_2$$

but we'll leave out the sign bit and other leading zeros as they're not necessary for our purposes.

Each digit in the binary representation, which is either 0 or 1, is called a bit.

Bitwise Operations

There are several binary operations on binary numbers called bitwise operations. These operations are applied separately for each bit position. The common binary operations are shown in table 14.1:

Bit A	Bit B	A AND B	A OR B	A XOR B
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Table 14.1: The outputs of bitwise operations on two bits

The AND operation (&) returns 1 if and only if both bits are 1.

`19 & 26`

	1	0	0	1	1	=	19
&	1	1	0	1	0	=	26
=	1	0	0	1	0	=	18

The OR operation (|) returns 1 if either bit is 1.

`19 | 26`

	1	0	0	1	1	=	19
|	1	1	0	1	0	=	26
=	1	1	0	1	1	=	27

The XOR operation ($\wedge$) returns 1 if and only if exactly one of the bits is 1.

`19` $\wedge$ `26`

	1	0	0	1	1	=	19
&	1	1	0	1	0	=	26
=	0	1	0	0	1	=	9

Finally, the left shift operator `x << k` multiplies x by 2^k. In particular, `1 << k` is 2^k, so you can use this to check whether the kth bit of a number is 1 or 0. If `x & (1 << k)` is zero, then the kth bit of x is 0, otherwise it is 1. Watch for overflow and use the `long` data type if necessary.

$$1 << 5 = 1 \cdot 2^5 = 32$$
$$7 << 2 = 7 \cdot 2^2 = 28$$

Exercises

Calculate by converting the numbers to binary, applying the bit operations, and then converting back to decimal numbers:

(a) `19 & 34` Answer: 2

(b) `14 | 29` Answer: 31

(c) `10` $\wedge$ `19` Answer: 25

(d) `3 << 5` Answer: 96

Generating Subsets

Occasionally in a problem we'll want to iterate through every possible subset of a given set, either to find a subset that satisfies some condition, or to find the number of subsets that satisfy some condition. Also, some problems might ask you to find the number of partitions of a set into 2 groups that satisfy a certain condition. In this case, we will iterate through all possible subsets, and check each subset for validity (first adding the non-selected elements to the second subset if necessary).

In a set of N elements, there are 2^N possible subsets, because for each of the N elements, there are two choices: either in the subset, or not in the subset. Subset problems usually require a time complexity of $O(N \cdot 2^N)$, because each subset has an average of $O(N)$ elements.

Now, let's look at how we can generate the subsets. We can represent subsets as binary numbers from 0 to $2^N - 1$. If we zero-index our original array of elements, then the bit with value 2^k represents whether the element with index k is in the subset. If the 2^k bit is 1, then the element is included in the subset; if it's 0, then it's not included in the subset. Let's look at an example set of a, b, c.

number	binary	subset
0	000	{ }
1	001	{a}
2	010	{b}
3	011	{a, b}
4	100	{c}
5	101	{a, c}
6	110	{b, c}
7	111	{a, b, c}

Algorithm: The algorithm for generating all subsets of a given input array

Function `generateSubsets`
- **Input :** An array `arr`, and its length n
- **for** $i \leftarrow 0$ **to** $2^n - 1$ **do**
  - Declare list
  - **for** $j = 0$ *to n-1* **do**
    - **if** *the bit in the binary representation of* i *corresponding to* 2^j *is* 1 **then**
      - Add `arr`$[j]$ to the list
    - **end**
  - **end**
  - Process the list
- **end**

In the following code, our original set is represented by the array `arr[]` with length n.

```
int ans = 0;
for(int i = 0; i < (1<<n); i++){
```

```
// this loop iterates through the 2^n subsets, one by one.
// 1 << n is a shortcut for 2^n
    vector<int> v;
    // we create a new list for each subset and add
    // the elements to it
    for(int j = 0; j < n; j++){
        if((i & (1 << j)) > 0){
            // (1 << j) is the number where only the bit representing 2^j is 1.
            v.push_back(j); // if the respective bit of i is 1,
            // add that element to the list
        }
    }
    if(valid(list)){
        // code is not included here, but this method will vary depending on the
        // problem to check if a certain subset is valid
        // and increments the answer counter if so.
        ans++;
    }
}
```

14.4 Ad-hoc

The silver division also often has ad hoc problems. They primarily rely on non-standard algorithmic thinking and problem solving ability. You develop these skills by solving problems; thus, we don't have much content to teach you about ad hoc problems, but we provide a selection of problems at the end of the chapter for your practice.

14.5 Problems

Two Pointers

1. CSES Problem Set Task 1640: Sum of Two Values
   `https://cses.fi/problemset/task/1640`

2. CSES Problem Set Task 1643: Maximum Subarray Sum
   `https://cses.fi/problemset/task/1643`

Line Sweep

3. USACO US Open 2019 Silver Problem 2: Cow Steeplechase II
   `http://usaco.org/index.php?page=viewproblem2&cpid=943`

Subsets

4. (Subsets) CSES Problem Set Task 1623: Apple Division
   https://cses.fi/problemset/task/1623

Ad hoc problems

5. USACO February 2016 Silver Problem 1: Circular Barn
   http://usaco.org/index.php?page=viewproblem2&cpid=618
6. USACO US Open 2019 Silver Problem 1: Left Out
   http://www.usaco.org/index.php?page=viewproblem2&cpid=942
7. USACO February 2019 Silver Problem 1: Sleepy Cow Herding
   http://www.usaco.org/index.php?page=viewproblem2&cpid=918
8. USACO January 2017 Silver Problem 3: Secret Cow Code
   http://www.usaco.org/index.php?page=viewproblem2&cpid=692
9. USACO January 2020 Silver Problem 1: Berry Picking
   http://www.usaco.org/index.php?page=viewproblem2&cpid=990
10. USACO December 2019 Silver Problem 2: Meetings
    http://www.usaco.org/index.php?page=viewproblem2&cpid=967
    (Warning: extremely difficult)

Part IV

Problem Set

Chapter 15

Parting Shots

You improve at competitive programming primarily by doing problems, so we leave you with an extensive selection of CodeForces problems for your practice. This consists of five problem sets of ten problems each, increasing in difficulty. The problems mostly use topics covered in the book, but may require some ingenuity to find the solution. If you get stuck, you can search for the editorial. Best of luck!

Set 1

1. `https://codeforces.com/problemset/problem/1227/B`
2. `https://codeforces.com/problemset/problem/1196/B`
3. `https://codeforces.com/problemset/problem/1195/B`
4. `https://codeforces.com/problemset/problem/1294/B`
5. `https://codeforces.com/problemset/problem/1288/B`
6. `https://codeforces.com/problemset/problem/1293/A`
7. `https://codeforces.com/problemset/problem/1213/B`
8. `https://codeforces.com/problemset/problem/1207/B`
9. `https://codeforces.com/problemset/problem/1324/B`
10. `https://codeforces.com/problemset/problem/1327/A`

Set 2

1. `https://codeforces.com/problemset/problem/1182/B`
2. `https://codeforces.com/problemset/problem/1183/D`
3. `https://codeforces.com/problemset/problem/1183/C`
4. `https://codeforces.com/problemset/problem/1133/C`

5. https://codeforces.com/problemset/problem/1249/B2
6. https://codeforces.com/problemset/problem/1194/B
7. https://codeforces.com/problemset/problem/1271/C
8. https://codeforces.com/problemset/problem/1326/C
9. https://codeforces.com/problemset/problem/1294/C
10. https://codeforces.com/problemset/problem/1272/B

Set 3

1. https://codeforces.com/problemset/problem/1169/B
2. https://codeforces.com/problemset/problem/1102/D
3. https://codeforces.com/problemset/problem/978/F
4. https://codeforces.com/problemset/problem/1196/C
5. https://codeforces.com/problemset/problem/1154/D
6. https://codeforces.com/problemset/problem/1272/D
7. https://codeforces.com/problemset/problem/1304/C
8. https://codeforces.com/problemset/problem/1296/C
9. https://codeforces.com/contest/1263/problem/D
10. https://codeforces.com/contest/1339/problem/C

Set 4

1. https://codeforces.com/problemset/problem/1281/B
2. https://codeforces.com/problemset/problem/1196/D2
3. https://codeforces.com/problemset/problem/1165/D
4. https://codeforces.com/problemset/problem/1238/C
5. https://codeforces.com/problemset/problem/1234/D
6. https://codeforces.com/problemset/problem/1198/B
7. https://codeforces.com/problemset/problem/1198/A
8. https://codeforces.com/problemset/problem/1077/D
9. https://codeforces.com/problemset/problem/1303/C
10. https://codeforces.com/problemset/problem/1098/A

Set 5

1. https://codeforces.com/problemset/problem/1185/D
2. https://codeforces.com/problemset/problem/1195/D2
3. https://codeforces.com/problemset/problem/1154/E
4. https://codeforces.com/contest/1195/problem/C
5. https://codeforces.com/problemset/problem/1196/E
6. https://codeforces.com/problemset/problem/1328/D
7. https://codeforces.com/problemset/problem/1253/D
8. https://codeforces.com/problemset/problem/1157/E
9. https://codeforces.com/problemset/problem/1185/C2
10. https://codeforces.com/problemset/problem/1209/D