Hendrix Programming Team Reference

November 10, 2019

Contents

Chapter 1

Limits

As a rule of thumb, you should assume about 10^8 (= 100 million) operations per second. If you can think of a straightforward brute force solution to a problem, you should check whether it is likely to fit within the time limit; if so, go for it! Some problems are explicitly written to see if you will recognize this. If a brute force solution won't fit, the input size can help guide you to search for the right algorithm running time.

Example: suppose a problem requires you to find the length of a shortest path in a weighted graph.

- If the graph has $|V| = 400$ vertices, you should use Floyd-Warshall (§6.7, page 37): it is the easiest to code and takes $O(V^3)$ time which should be good enough.

- If the graph has $|V| = 4000$ vertices, especially if it doesn't have all possible edges, you can use Dijkstra's algorithm (§6.6, page 34), which is $O(E \log V)$.

- If the graph has $|V| = 10^5$ vertices, you should look for some special property of the graph which allows you to solve the problem in $O(V)$ or $O(V \log V)$ time—for example, perhaps the graph is a tree (§6.1, page 31), so you can run a BFS/DFS (§6.4, page 32) to find the unique path and then add up the weights. An input size of 10^5 is a common sign that you are expected to use an $O(n \lg n)$ or $O(n)$ algorithm—it's big enough to make $O(n^2)$ too slow but not so big that the time to do I/O makes a big difference.

n	Worst viable running time	Example
11	$O(n!)$	Generating all permutations (§9.7, page 55)
25	$O(2^n)$	Generating all subsets (§10, page 57)
100	$O(n^4)$	Some brute force algorithms
400	$O(n^3)$	Floyd-Warshall (§6.7, page 37)
10^4	$O(n^2)$	Testing all pairs
10^6	$O(n \lg n)$	BFS/DFS; sort+greedy

> 🐞 bing, transportationplanning, dancerecital, prozor, rectanglesurrounding, weakvertices

- $2^{10} = 1024 \approx 10^3$.

- One `int` is 32 bits = 4 bytes. So *e.g.* an array of 10^6 `int`s requires < 4 MB—no big deal since the typical memory limit is 1 GB. Don't be afraid to make arrays with millions of elements!

- `int` holds 32 bits; the largest `int` value is `Integer.MAX_VALUE` $= 2^{31} - 1$, a bit more than $2 \cdot 10^9$.

- `long` holds 64 bits; the largest `long` value is `Long.MAX_VALUE` $= 2^{63} - 1$, a bit more than $9 \cdot 10^{18}$. To write literal long values you can add an L suffix, as in `long x = 1234567890123L;`.

- If you need larger values, use BigInteger (§2.15, page 14) or just use Python (§3, page 19); see also Combinatorics (§9.7, page 55).

🔗 `different`

Chapter 2

Java Reference

2.1 Template

```java
1   // *Don't* include a package declaration!
2
3   import java.util.*;
4   import java.math.*;
5
6   public class ClassName {
7       public static void main(String[] args) {
8           Scanner in = new Scanner(System.in);
9
10          // Solution code here
11
12          System.out.println(answer);
13      }
14  }
```

2.2 Scanner

☕ Scanner is relatively slow but should usually be sufficient for most purposes. If the input or output is relatively large (> 1MB) and you suspect the time taken to read or write it may be a hindrance, you can use Fast I/O (§2.17, page 15).

> ☞ Be sure to read the warning in the comment below about calling `nextLine()` after `nextInt()` and the like!

```java
1   import java.util.*;
2
3   public class ScannerExample {
4       public static void main(String[] args) {
5           Scanner in = new Scanner(System.in);
6
7           // All these read a single token (ignore leading whitespace,
8           // then read up until but not including the next whitespace)
9           String s = in.next();
10          int    n = in.nextInt();
11          long   l = in.nextLong();
12          double d = in.nextDouble();
```

```
13
14          // WARNING!! A previous call to nextXXX will read up to a
15          // newline character but leave it unconsumed in the input, so
16          // the next call to nextLine() will just read that newline and
17          // return an empty string!
18          in.nextLine();   // throw away the empty line to get ready for the next
19
20          // Read a whole line up to the next newline character.
21          // Consumes the newline but does not include it in the
22          // returned String.
23          String line = in.nextLine();
24
25          // Read until end of input
26          while (in.hasNext()) {
27              line = in.nextLine();
28          }
29      }
30 }
```

2.3 Math

The standard ☕ Math class contains useful standard mathematical constants and operations. All are `static`, so they can be accessed by prefixing their names with `Math.`, *i.e.* `Math.cos`.

- Constants `E` and `PI` represent (floating-point approximations of) e and π.

- `abs` finds the absolute value.

- `min` and `max` find the min or max of two values. A common trick for saving a bit of typing is to use something like

  ```
  m = Math.max(m, val);
  ```

  if you need `m` to accumulate the maximum of a set of values.

- `round` rounds a floating-point number to the nearest integer. `ceil` and `floor` round up and down, respectively. Note that whereas `round` returns a `long` when given a `double`, for some reason `ceil` and `floor` both return `double`, so you may need to cast the results:

  ```
  double x = ...
  long n = (long)Math.floor(x);
  ```

- `exp(x)` computes e^x. `log(x)` computes $\ln x$.

- `sqrt` computes the square root. `hypot(x,y)` computes $\sqrt{x^2 + y^2}$.

- `pow(a, b)` computes a^b.

- `sin, cos, tan, acos, asin, atan` do what you would expect. Note also `atan2(y,x)` which returns an angle θ such that it converts rectangular coordinates (x, y) into polar coordinates (r, θ). This is almost like `atan(y/x)` except that it avoids division by zero and handles all four quadrants properly.

- `toDegrees` and `toRadians` convert angles from radians to degrees and degrees to radians, respectively.

2.4 String

> ☘ battlesimulation, bing, connectthedots, itsasecret, shiritori, suffixarrayreconstruction

The ☕ String type can be used in Java to represent sequences of characters. Some useful String methods include:

- concatenation (+)

- substring(i) yields the substring starting at index i up to the end of the string

- substring(i,j) yields the substring starting at i (inclusive) and ending *just before* j (same as Python slices).

- charAt(i) yields the char at index i.

- toCharArray() converts to a char[], which can be convenient if you need to do a lot of indexing ([i] instead of charAt(i)).

- split(String) splits a string into a String[] of pieces between occurrences of the splitting string.

- endsWith(String), startsWith(String), indexOf(String), and replace(...) can occasionally be useful.

Below is shown a solution to ☘ sumoftheothers , which uses split followed by Integer.parseInt to read the integers on each line (necessary in this case because the input does not specify how many integers will be on each line, although this is atypical).

```java
import java.util.Scanner;

public class sumOfTheOthers {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
            String[] s = in.nextLine().split(" ");
            int out = 0;
            for (String i : s) {
                out += Integer.parseInt(i);
            }
            System.out.println(out / 2);
        }
        in.close();
    }
}
```

Strings are immutable, which means in particular that concatenation has to allocate an entirely new String and copy both arguments. Hence repeatedly appending individual characters to the end of a String takes $O(n^2)$ time, since the entire string must be copied with each append operation. In this situation, either pre-allocate a sufficiently large char[], or use the ☕ StringBuilder class.

2.5   StringBuilder

> ☘ itsasecret, joinstrings

☕ StringBuilder is a *mutable* string class which supports efficient append and modification operations. If you need to build up a long string by incrementally appending text bit by bit, you should use StringBuilder instead of using String directly. StringBuilder also supports a reverse() method (unlike String).

As a simple example, the below code prints 0291817161514131211101987654321.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 20; i++) {
    sb.append("" + i);
}
System.out.println(sb.reverse());
```

2.6 Character

ummcode, softpasswords

[TODO: Useful Character class methods like isDigit, isAlphabetic, etc.]

2.7 Arrays

falcondive, freefood, traveltheskies

The basic syntax for creating a primitive array in Java is, for example,

```
int[] array = new int[500];
```

Some tips and tricks:

- Array indexing starts at 0; however, problems sometimes index things from $1 \ldots n$. In such a situation it is usually a good idea to simply create an array with one extra slot and leave index 0 unused. The alternative (fiddling with indices by subtracting and adding 1 in the right places) is quite error-prone.

- You can initialize an entire array to a given value using Arrays.fill(array, value).

- If you only want to initialize part of an array, use Arrays.fill(array, fromIndex, toIndex, value) to fill the array from fromIndex (inclusive) up to toIndex (exclusive).

- You can sort the contents of an array in-place using Arrays.sort; see Sorting (§2.16, page 15).

- You can use Arrays.binarySearch(array, key) to look for key within a sorted array. Read the documentation to make sure you understand how to interpret the return value. See also Binary search (§5.2, page 27).

- Other methods from the ☕ Arrays class may also occasionally be useful.

- To iterate over the items in an array, you can use foreach syntax:

```
for (int item : array) {
  // do something with i
}
```

2.8 ArrayList

☕ ArrayList represents a standard dynamically-extensible array, doubling the underlying storage when it runs out of space so that appending takes $O(1)$ amortized time. The add, get, set, size, and isEmpty methods are useful, in addition to the ability to iterate over the elements in order. Avoid methods such as contains, indexOf, remove, and the version of add that takes an index, all of which take linear time. (If you think you want any of these methods, it's probably a sign that you ought to be using a different data structure.)

If you need to store a list/array and you know in advance exactly how much storage space you will need, then prefer using a primitive array which has less overhead as well as more concise syntax. On the

other hand, if you want to be able to dynamically extend a list by appending new elements to the end, use
ArrayList. (If you want to be able to dynamically extend a list on *both* ends, use an ArrayDeque (§2.10,
page 11).)

```
ArrayList<Integer> lst = new ArrayList<>();
lst.add(3); lst.add(19); lst.add(6);
System.out.println(lst.get(2));     // prints 6
lst.set(1, 12);                     // changes 19 to 12
int sum = 0;
for (Integer i: lst) {              // iterate through all items
    sum += i;
}
System.out.println(sum);            // prints 3 + 12 + 6 = 21
```

2.9   Stack

> 🎱 backspace, delimitersoup, islands, pairingsocks, reservoir, restaurant, symmetricorder,
> throwns, zagrade

☕ Stack provides a generic stack implementation with $O(1)$ operations. Standard methods include
isEmpty, push, pop, peek, and size. The code below shows a sample solution to 🎱 backspace using ☕
Stack (and ☕ StringBuilder).

```
1  import java.util.*;
2
3  public class backspace {
4      public static void main(String[] args) {
5          Scanner in = new Scanner(System.in);
6          String line = in.next();
7          Stack<Character> editor = new Stack<>();
8          for(int i = 0; i < line.length(); i++) {
9              if(line.charAt(i) == '<')
10                 editor.pop();
11             else
12                 editor.push(line.charAt(i));
13         }
14         StringBuilder buildString = new StringBuilder();
15         while(!editor.isEmpty()) {
16             buildString.append(editor.pop());
17         }
18         System.out.println(buildString.reverse().toString());
19     }
20 }
```

Stacks are often used in implementing DFS (§6.4, page 32) as well as dealing with parentheses, or nesting
more generally (🎱 pairingsocks , 🎱 islands , 🎱 reservoir).

2.10   Queue/ArrayDeque

> 🎱 eenymeeny, brexit, coconut, ferryloading4, integerlists, shuffling

☕ Queue, unlike ☕ Stack, is not a class but an interface. There are several classes implementing the
Queue interface, but the best in the context of competitive programming is probably ☕ ArrayDeque, which

in fact implements a *double-ended queue* or *deque*, providing $O(1)$ amortized addition and removal from both ends.

The `add` and `remove` methods implement enqueueing and dequeueing. To access both ends, use `addFirst`, `addLast`, `removeFirst`, and `removeLast`, all of which run in $O(1)$ amortized time. (`add` is the same as `addLast` and `remove` is the same as `removeFirst`.)

Queues are very commonly used in implementing Breadth-First Search (§6.3, page 31) and in simulations of various sorts (for examples of the latter, see the selection of problems above).

As a simple example of the syntax for creating and using a queue, the below code puts the numbers 1 through 10 in a queue and then extracts them to print them out in the same order.

```
Queue<Integer> q = new ArrayDeque<>();
for (int i = 1; i <= 10; i++) {
  q.add(i);
}
while (!q.isEmpty()) {
  System.out.println(q.remove());
}o
```

2.11 Comparator

A ☕ `Comparator` is used to specify a custom ordering on some type, potentially different than its "natural" ordering. Typically a `Comparator` can be passed as an optional argument to things that require an ordering. For example, given an `ArrayList<Integer> arr`, one can use `Collections.sort(arr)` to sort it in increasing numeric order. If a different order is wanted, one can pass a `Comparator` as the second argument to `sort`, as in

```
Collections.sort(arr, Collections.reverseOrder());
```

to sort in descending order, or

```
Collections.sort(arr, (i,j) -> q[i] - q[j]);
```

to sort a list of *indices* by the corresponding value in an array `q`. A `Comparator` can also be used as an extra argument to the constructor when creating a data structure that depends on ordering, such as a `PriorityQueue`, `TreeSet`, or `TreeMap`.

[TODO: Constructing Comparators via lambda; constructing via things like `comparing`, `thenComparing`. `Collections.reverseOrder()`.]

2.12 PriorityQueue

bank, ferryloading3, guessthedatastructure, knigsoftheforest, vegetables

A ☕ `PriorityQueue` allows adding new elements (`add`) and removing the **minimum** element (`remove`), both in $O(\lg n)$ time. `peek` can also be used to get the minimum in $O(1)$ without removing it. Priority queues are commonly used in Dijkstra's algorithm (§6.6, page 34), event-based simulations (ferryloading3), and generally any situation where we need to do an "online sort", that is, we need to get items in order from smallest to biggest, but more items may continue to arrive/be generated as we go.

Methods you should generally *not* use with `PriorityQueue` include `remove(Object)` and `contains(Object)`, which take linear time.

The default constructor makes an empty min-PQ. If you want to use a different ordering, there is another constructor which takes a ☕ `Comparator`.

- For example, if you want a **max** priority queue, where `remove()` yields the largest element, write something like

```
PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
```

- If you want some other ordering, you can also use a lambda to construct a `Comparator` on the fly, for example:

```
PriorityQueue<Integer> pq = new PriorityQueue<>((a,b) -> dist[a] - dist[b]);
```

Code like the above is used in Dijkstra's algorithm where we want to compare vertices by their best recorded distance from the start vertex.

Traditional presentations of priority queues often have a *decrease key* operation which can decrease the priority of an item (or an *adjust key* operation which can arbitrarily change the priority) and reestablish the data structure invariants in $O(\lg n)$ time; this operation is used, for example, in implementing Dijkstra's algorithm efficiently (§6.6, page 34). However, the Java `PriorityQueue` class has no such method. One workaround is to simply call `remove` and then `add` so the item gets re-added with the new priority. However, `remove` takes linear time, so this is not ideal, although in many cases it is still good enough. For those (relatively rare) cases when an $O(\lg n)$ decrease key operation is truly essential, see Adjustable priority queue (§4.5, page 24).

2.13   Set

boatparts, bookingaroom, engineeringenglish, whatdoesthefoxsay, securedoors, bard, control

The Set interface represents a collection of items where each item occurs at most once. Operations supported by all `Set`s include `add`, `remove`, `contains`, `size`/`isEmpty`.

There are two main classes implementing `Set`:

- HashSet is implemented using a dynamically expanding hash table. It features $O(1)$ `add`, `remove`, and `contains`.

- TreeSet is implemented using a balanced binary tree (a red-black tree, in fact), and supports `add`, `remove`, and `contains` in guaranteed $O(\lg n)$ time. Of course this is slower than `HashSet`; however, `TreeSet` has several other advantages:

  - Since the elements are stored in order in the tree, iterating over a `TreeSet` is guaranteed to yield the items in order from smallest to biggest, whereas iterating over a `HashSet` yields the items in an arbitrary order. For example, if you want to remove duplicates from a set of items and then print them out in order, you might as well just throw them all into a `TreeSet` instead of putting them in a `HashSet` and then sorting (crowdcontrol). (And either one is probably going to be faster than sorting and *then* removing duplicates.)

  - If you need to put objects of a custom class into a set, it is sometimes easier to implement `Comparable` for your class and use a `TreeSet` than it is to override `hashCode` and use a `HashSet`. The $O(\lg n)$ difference is rarely, if ever, going to be the difference between AC and TLE, so you should use whichever approach will be easier to code.

  - `TreeSet` also supports the OrderedSet and NavigableSet interfaces, which provide additional methods like `first` and `last` (return the smallest or largest element in the set), `headSet` and `tailSet` (return the subset of all items less or greater than a specified element), and `floor`, `ceiling`, `lower`, and `higher` (find the first item in the set less/greater than a specified value). This last set of methods can be especially useful for some types of problems.

    closestsums, platforme, baloni, excellentengineers

There is also a LinkedHashSet class which in addition to providing all the same features as a HashSet, also remembers the order in which the items were added; iterating over the set is guaranteed to yield the items in this order. This is a bit more sophisticated than simply keeping an ArrayList and a HashSet side-by-side, in particular because a LinkedHashSet still supports $O(1)$ removal. We currently do not know of any example problems which can be solved most easily using a LinkedHashSet, but it never hurts to be prepared!

2.14 Map

awkwardparty, administrativeproblems, snowflakes, pizzahawaii, snowflakes

The Map interface represents a dictionary data structure associating keys to values. Supported operations include get(K), put(K,V), containsKey(K), remove(K), and size/isEmpty.

As with Set, there are two main classes implementing Map:

- HashMap is implemented using a hash table, and allows $O(1)$ get, put, remove, and containsKey.

- TreeMap is implemented using a balanced binary tree. Many of the same comments apply as for TreeSet:

    - All operations run in worst-case $O(\lg n)$ time.
    - Iterating over the keys in the map is guaranteed to return them in order from smallest to biggest.
    - TreeMap also implements the SortedMap interface (allowing one *e.g.* to access the first or last key or to get a submap of all the key/value pairs which lie in between certain keys) and NavigableMap interface (which lets you find the closest keys and values which are smaller/bigger than a given query key).

For the purposes of programming contests, TreeMap and HashMap are basically interchangeable. HashMap is faster in theory but a factor of $\lg n$ is not that much, and HashMap has its own overhead costs.

- To iterate over the keys of a map, use keySet:

```
for (K key : map.keySet()) {
  ...
}
```

- To iterate over the values, use values:

```
for (V val : map.values()) {
  ...
}
```

- You can also iterate over both at once:

```
for (Map.Entry<K,V> e : map.entrySet()) {
  ... e.getKey() ... e.getValue() ...
}
```

2.15 BigInteger

> ♻ basicremains

♨ `BigInteger` represents arbitrarily large integer values, though it's not actually needed all that often. You can import it as `java.math.BigInteger`. For combinatorics problems where the answer is going to be a big number, consider using Python instead.

- To turn an `int` or `long` into a `BigInteger`, use `BigInteger.valueOf`.

- `BigInteger` also provides the constants `BigInteger.ZERO`, `.ONE`, `.TWO`, and `.TEN`.

- To do arithmetic, use methods such as `add`, `subtract`, and `multiply`, which all return their result as a new `BigInteger`:

    ```
    BigInteger a = ...
    BigInteger b = ...
    BigInteger c = a.add(b.multiply(BigInteger.TEN));  // c = a + b*10
    ```

- `BigInteger` can occasionally be useful in scenarios other than representing large numbers, because of its useful utility methods such as `gcd` and its ability to convert between different bases.

2.16 Sorting

- You can sort an array using `Arrays.sort(array)`.

- You can sort a list-like collection (such as an `ArrayList`) using `Collections.sort(list)`.

- These `sort` methods also take an optional Comparator (§2.11, page 12).

You should rarely, if ever, have to code your own sorting algorithms; the exception is when you need to *enhance* a sorting algorithm to keep track of some extra information while sorting (for example, look up the divide-and-conquer algorithm for counting inversions).

It is relatively common that one needs to sort according to some key, but carry along additional information with the keys. The easiest way to do this is to make a small class to contain all the relevant information, then implement the `Comparable` interface.

```
class Person implements Comparable<Person> {
    int age; String name;
    public Person(int age, String name) { this.age = age; this.name = name; }
    int compareTo(Person p) { return this.age - p.age; }
}

ArrayList<Person> people = ...
Collections.sort(people);  // sort by age, carrying names along too
```

Occasionally, you need to sort an array of items, but also keep track of the original index of each item (♻ keepitcool). To do this, you can use a technique like the above, where each object stores an item as well as its index in the original list, and sorts according to the item. Then sorting the objects will sort the items, but each item carries along its original index.

2.17 Fast I/O

> ♻ avoidland, cd, minspantree, ozljeda

Typically ACM ICPC problems are designed so `Scanner` and `System.out.println` are fast enough to read and write the required input and output within the time limits. However, these are relatively slow since they are unbuffered (every single read and write happens immediately). Occasionally it can be useful to have faster I/O; indeed, a few problems on Kattis cannot be solved in Java without using this.

☞ Be sure to call `flush()` at the end of your program or else some output might be lost!

```java
/* Example usage:
 *
 * Kattio io = new Kattio(System.in, System.out);
 *
 * while (io.hasMoreTokens()) {
 *     int n = io.getInt();
 *     double d = io.getDouble();
 *     double ans = d*n;
 *
 *     io.println("Answer: " + ans);
 * }
 *
 * io.flush();   // DON'T FORGET THIS LINE!
 */

import java.util.*;
import java.io.*;

class Kattio extends PrintWriter {
    public Kattio(InputStream i) {
        super(new BufferedOutputStream(System.out));
        r = new BufferedReader(new InputStreamReader(i));
    }
    public Kattio(InputStream i, OutputStream o) {
        super(new BufferedOutputStream(o));
        r = new BufferedReader(new InputStreamReader(i));
    }

    public boolean hasMoreTokens() {
        return peekToken() != null;
    }

    public int getInt() {
        return Integer.parseInt(nextToken());
    }

    public double getDouble() {
        return Double.parseDouble(nextToken());
    }

    public long getLong() {
        return Long.parseLong(nextToken());
    }

    public String getWord() {
        return nextToken();
    }

    private BufferedReader r;
    private String line;
    private StringTokenizer st;
    private String token;
```

```
53
54      private String peekToken() {
55          if (token == null)
56              try {
57                  while (st == null || !st.hasMoreTokens()) {
58                      line = r.readLine();
59                      if (line == null) return null;
60                      st = new StringTokenizer(line);
61                  }
62                  token = st.nextToken();
63              } catch (IOException e) { }
64          return token;
65      }
66
67      private String nextToken() {
68          String ans = peekToken();
69          token = null;
70          return ans;
71      }
72  }
```

Chapter 3

Python Reference

Python's built-in support for arbitrary-size integers (using `BigInteger` in Java is a pain!) and built-in dictionaries with lightweight syntax make it attractive for certain kinds of problems.

3.1 Template

Below is a basic template showing how to read typical contest problem input in Python:

```python
import sys

if __name__ == '__main__':

    n = int(sys.stdin.readline())   # Read an int on a line by itself
    for _ in range(n):              # Do something n times

        # Read all the ints on a line into a list
        xs = map(int, sys.stdin.readline().split())

        # Read a known number of ints into variables
        p, q, r, y = map(int, sys.stdin.readline().split())
```

[TODO: Mention basic Python data structures such as set, deque, list methods]

Chapter 4

Data Structures

4.1 Pair

Java has a `Pair` class in `javafx.util.Pair`, but one can't necessarily assume that this will be available in a contest environment. The following simple class suffices to store pairs of values, which is especially useful in representing *e.g.* coordinates in a 2D grid. Note in a competitive programming context we don't bother adding getter and setter methods; the `a` and `b` fields are public so we can just access them directly.

```java
public class Pair<A,B> {
    A a; B b;
    public Pair(A a, B b) { this.a = a; this.b = b; }
    public int hashCode() { return 31 * (31 * 7 + a.hashCode()) + b.hashCode(); }
    public boolean equals(Object o) {
        Pair<A,B> p = (Pair<A,B>)o;
        return a.equals(p.a) && b.equals(p.b);
    }
    public String toString() { return "(" + a + "," + b + ")"; }
}
```

Below is a variant which implements the `Comparable` interface. `CPair` objects sort in lexicographic order: first by the first component, and then ties in the first component are broken by the second component.

```java
import java.util.Comparator;

public class CPair<A extends Comparable<A>, B extends Comparable<B>>
        implements Comparable<CPair<A,B>> {
    A a; B b;
    public CPair(A a, B b) { this.a = a; this.b = b; }
    public int hashCode() { return 31 * (31 * 7 + a.hashCode()) + b.hashCode(); }
    public boolean equals(Object o) {
        CPair<A,B> p = (CPair<A,B>)o;
        return a.equals(p.a) && b.equals(p.b);
    }
    public String toString() { return "(" + a + "," + b + ")"; }
    public int compareTo(CPair<A,B> p) {   // sort by a, then b
        if (!a.equals(p.a)) return a.compareTo(p.a);
        else                return b.compareTo(p.b);
    }
}
```

4.2 Bag/multiset

> 🪐 cookieselection, kattissquest

A *bag*, aka *multiset*, is a collection of elements where order does not matter (like a set) but multiplicity does matter, *i.e.* there can be duplicates and we need to keep track of how many duplicates there are of each item. Bags are not needed often but can occasionally be useful. It is not too hard to build a bag as a map from items to integer counts, but there are a few corner cases so it's worth copying a well-tested implementation instead of writing one from scratch.

The implementation below is based on a 🍵 TreeMap (§2.14, page 14), and hence supports operations like `first()` and `last()`. If desired one could easily change the `TreeMap` to a `HashMap` and remove the methods which are no longer supported, although the factor of $O(\lg n)$ is unlikely to make a practical difference.

```java
import java.util.*;

public class TreeBag<E extends Comparable<E>> implements Iterable<E> {
    private TreeMap<E, Integer> map;
    private int totalSize;
    public TreeBag() { map = new TreeMap<>(); }
    public void add(E e) {
        if (!map.containsKey(e)) map.put(e,0);
        map.put(e, map.get(e) + 1);
        totalSize++;
    }
    public void remove(E e) {
        if (map.containsKey(e)) {
            if (map.get(e) == 1) map.remove(e);
            else map.put(e, map.get(e) - 1);
            totalSize--;
        }
    }
    public int size()          { return totalSize; }
    public boolean isEmpty() { return map.isEmpty(); }
    public int count(E e)     { return map.containsKey(e) ? map.get(e) : 0; }
    public E first()           { return map.firstKey(); }
    public E last()            { return map.lastKey(); }
    public E pollFirst()      { E e = first(); remove(e); return e; }
    public E pollLast()       { E e = last(); remove(e); return e; }
    public E floor(E e)       { return map.floorKey(e); }
    public E ceiling(E e)     { return map.ceilingKey(e); }
    public E lower(E e)       { return map.lowerKey(e); }
    public E higher(E e)      { return map.higherKey(e); }
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            Iterator<E> it = map.keySet().iterator();
            E cur; int count = 0;
            public boolean hasNext() { return it.hasNext() || count > 0; }
            public E next() {
                if (count == 0) { cur = it.next(); count = map.get(cur); }
                count--; return cur;
            }
        };
    }
}
```

4.3 Union-find

> 🔗 `forestfires`, `kastenlauf`, `ladice`, `numbersetseasy`, `unionfind`, `virtualfriends`, `watersheds`, `wheresmyinternet`

A union-find structure can be used to keep track of a collection of disjoint sets, with the ability to quickly test whether two items are in the same set, and to quickly union two given sets into one. It is used in Kruskal's Minimum Spanning Tree algorithm (§6.8, page 38), and can also be useful on its own (see the above Kattis problems for examples). `find` and `union` both take essentially constant amortized time.

```
class UnionFind {
    private byte[] r; private int[] p;  // rank, parent

    // Make a new union-find structure with n items in singleton sets,
    // numbered 0 .. n-1 .
    public UnionFind(int n) {
        r = new byte[n]; p = new int[n];
        for (int i = 0; i < n; i++) {
            r[i] = 0; p[i] = i;
        }
    }

    // Return the root of the set containing v, with path compression. O(1).
    // Test whether u and v are in the same set with find(u) == find(v).
    public int find(int v) {
        return v == p[v] ? v : (p[v] = find(p[v]));
    }

    // Union the sets containing u and v. O(1).
    public void union(int u, int v) {
        int ru = find(u), rv = find(v);
        if (ru != rv) {
            if      (r[ru] > r[rv]) p[rv] = ru;
            else if (r[rv] > r[ru]) p[ru] = rv;
            else { p[ru] = rv; r[rv]++; }
        }
    }
}
```

The above code can easily be enhanced to keep track of the number of sets (initialize to `n`; subtract one every time `union` hits the `ru != rv` case), or to keep track of the actual size of each set instead of just the rank/height (keep a size for each index; initialize all to 1; add sizes appropriately when doing `union`).

4.4 Tries

> 🔗 `boggle`, `heritage`, `herkabe`, `phonelist`

The code below is a very simple implementation of a trie—there are many other methods that could be added, and it is not very efficient since it repeatedly uses the $O(n)$ `substring` operation as it recurses down the trie, but it is sufficient for some problems.
[TODO: More efficient/full-featured Trie class]

```
class Trie<K,V> {
    Map<K, V> children;
```

```
3          boolean mark;
4
5          public Trie() {
6              children = new HashMap<>(); mark = false;
7          }
8          public void add(String s) { addR(s); }
9          public void addR(String s) {
10             if (s.equals("")) mark = true;
11             else ensureChild(s.charAt(0)).addR(s.substring(1));
12         }
13         public Trie getChild(Character c) { return children.get(c); }
14         public Trie ensureChild(Character c) {
15             Trie t = getChild(c);
16             if (t == null) {
17                 t = new Trie();
18                 children.put(c, t);
19             }
20             return t;
21         }
22     }
```

Tries are intimately connected with MSD radix sort, which can be thought of as equivalent to building a trie and then traversing it in order. However, no implementation of radix sort actually builds an intermediate trie. Sometimes it can be helpful to think about a problem in terms of a trie, but never actually implement/materialize the trie at all (herkabe): just do a modified radix sort, first grouping strings by their first character, then recursing on each group, keeping track of needed auxiliary information (*e.g.* depth) along the way.

4.5   Adjustable priority queue

 flowerytrails

As discussed in PriorityQueue (§2.12, page 12), Java's `PriorityQueue` class has no way to efficiently alter the priority of an item already stored in the queue; simply removing and re-adding the item does the trick but takes $O(n)$ time. The efficiency of this operation really does make a difference in the asymptotic performance of Dijkstra's algorithm (§6.6, page 34), and occasionally it really needs to be $O(\lg n)$ in order to meet the time limits (*e.g.* flowerytrails). A suitable implementation of a priority queue with $O(\lg n)$ priority adjustment is shown below. The key idea is to keep a hash table on the side which can be used to quickly find the index of any item stored in the priority queue; of course, the hash table has to be kept suitably updated whenever items are shuffled in the heap. The `adjust(e)` method is used to inform the priority queue that the priority of item `e` has changed, so that the queue has an opportunity to move the item if necessary to reestablish the heap invariants.

```
1   import java.util.*;
2
3   public class AdjustablePQ<E extends Comparable<E>> {
4       protected ArrayList<E> elems;
5       protected HashMap<E, Integer> indices;
6       protected Comparator<E> cmp;
7       public AdjustablePQ() { this(Comparator.naturalOrder()); }
8       public AdjustablePQ(Comparator<E> cmp) {
9           elems = new ArrayList<>(); elems.add(null);
10          indices = new HashMap<>();
11          this.cmp = cmp;
```

```java
12        }
13        public int size() { return elems.size() - 1; }
14        public boolean isEmpty() { return size() == 0; }
15        public void add(E item) { set(elems.size(), item); reheapUp(last()); }
16        public E remove() {
17            E ret = elems.get(1);
18            set(1, elems.get(last())); elems.remove(last());
19            reheapDown(1);
20            return ret;
21        }
22        public E peek() { return elems.get(1); }
23        public void adjust(E item) { int i = indices.get(item); reheapUp(i); reheapDown(i); }
24
25        protected int last() { return elems.size() - 1; }
26        protected void set(int i, E item) {
27            if (i == elems.size()) elems.add(item);
28            else elems.set(i, item);
29            indices.put(item, i);
30        }
31        protected void swap(int i, int j) {
32            E tmp = elems.get(i); set(i, elems.get(j)); set(j, tmp);
33        }
34        protected void reheapUp(int i) {
35            if (i <= 1) return;
36            if (cmp.compare(elems.get(i), elems.get(i/2)) >= 0) return;
37            swap(i, i/2); reheapUp(i/2);
38        }
39        protected void reheapDown(int i) {
40            if (2*i > last()) return;
41            int small = 2*i;
42            if (2*i+1 <= last() && cmp.compare(elems.get(2*i), elems.get(2*i+1)) > 0)
43                small++;
44            if (cmp.compare(elems.get(i), elems.get(small)) > 0) {
45                swap(i, small); reheapDown(small);
46            }
47        }
48    }
```

4.6   Segment trees and Fenwick trees

See .

Chapter 5

Search

5.1 Complete search

> ♺ bing, classpicture, coloring, cycleseasy, dancerecital, lektira, freefood, gepetto, kastenlauf, mjehuric, paintings, prozor, rectanglesurrounding, reducedidnumbers, reseto, sheldon, shuffling, weakvertices, wheels, transportationplanning

See CP3 for a fuller discussion of complete search, aka brute force, and a list of relevant techniques (nested loops, recursive backtracking, *etc.*). Just remember that there's no need to code anything more sophisticated if a back-of-the-envelope analysis shows that a simple complete search will finish under the time limit. (Although some kinds of complete search can themselves be rather sophisticated. For example, see Bit Tricks (§10, page 57). Some of the above problems are much harder than others!)

Sometimes complete search isn't in and of itself the full solution to a problem, but the problem is set up so that a subpart can be done via complete search, to keep the solution complexity from getting out of hand and allowing you to focus your efforts on the more "interesting" part of the problem.

5.2 Binary search

If you need to do a traditional binary search—that is, finding the index where a given element occurs in a sorted array—you should just use the standard `Arrays.binarySearch` method. However, the underlying idea of binary search applies in many more contexts.

Binary search on a real interval

> ♺ bottles, cheese, speed, suspensionbridges, tetration

This is probably the most common form of binary search in competitive programming. Given a function f which is monotonic (*i.e.* always increasing, or always decreasing) on a given interval of the *real* line $[a, b]$, find $a \leq x \leq b$ such that $f(x)$ is equal to some target value. This can be accomplished by straightforward binary search: keep track of a current subinterval $[x_L, x_H]$; at each step, evaluate f at the midpoint $m = (x_L + x_H)/2$ of the interval, and update x_L or x_H to m depending on whether the value of f is too small or too big, respectively. Iterate until $x_H - x_L$ is within an appropriate tolerance (or simply iterate a fixed number of times—50 should be plenty), and return $(x_L + x_H)/2$. This is actually easier than traditional binary search since one doesn't have to worry about indexing, off-by-one errors, and the like.

The main trick is to realize when this technique is applicable. Sometimes the function f is plainly stated in the problem description, but sometimes the thing being searched for is more subtle. Whenever a problem asks for a floating-point number as the answer, it's worth considering whether you can binary search for it.

Binary search on an integer interval

> ♟   `guess`, `eko`, `freeweights`, `inversefactorial`, `reservoir`

Suppose we again have a monotonic function f and want to find a value n such that $f(n)$ is equal to some target value t—except that f is defined on the *integers* instead of the real numbers. We can again use binary search, but we have to be much more careful about potential off-by-one errors.

- Remember that to do binary search *in a sorted array* a, that is, find the index i such that $a[i]$ is equal to some target value, one can just use `Arrays.binarySearch`.

- In the basic version, we simply want to find n such that $f(n) = t$, or report that no such n exists. In this case it works well to use a half-open interval, that is, we maintain the invariant that possible values of n lie in the interval $[lo, hi)$, including lo but **excluding** hi. This has the advantage that the size of the remaining interval can be computed as simply $hi - lo$, and an appropriate condition for the loop is $hi - lo > 0$.

  The midpoint of $[lo, hi)$ can be computed as $mid = (lo + hi)/2$;[1] mid always lies within the interval, even if $hi - lo = 1$ (the rounding behavior of integer division plays a crucial role).

  If mid does not hold the target, one must then either update hi to mid, or lo to $mid + 1$ (not mid!) depending on whether the item at mid is larger or smaller than the target, respectively.

  [TODO: Example code]

- A slightly more sophisticated variant is where we need to find the largest n such that $f(n) \leq t$, or the smallest such that $f(n) \geq t$, or something similar. This requires even more care. In this situation it tends to be better to use a closed interval $[lo, hi]$, and using great care to update lo and hi appropriately (to $mid - 1$, mid, or $mid + 1$) depending on the desired properties of the value being searched for.

  In this scenario, when there can be duplicate values of $f(n)$, it's not possible to stop the search early, since any given n for which $f(n) = t$ may not be the optimal one. One must continue searching until the interval has reached size 1, and then return the sole remaining value.

  [TODO: Example code]

Unbounded binary search

Consider the following problem: given an increasing function f and a target value t, find the smallest positive value of x such that $f(x) \geq t$. (The domain of f can be either the reals or the integers.)

The idea is to start by finding an appropriate upper bound using repeated doubling: starting at $u = 1$, evaluate $f(u)$; if it is less than t, double u and repeat. Keep doubling u until finding the first such value of u (that is, the first power of two) such that $f(u) \geq t$. Then do a traditional binary search on the range $[1, u]$.

[TODO: Find some example Kattis problems that need an initial unbounded search?]

5.3 Ternary search

> ♟   `brocard`, `euclideantsp`, `infiniteslides`, `janitortroubles`, `dailydivision`

Ternary search can be used to find the minimum or maximum of a function which is concave or convex on a given interval (that is, the function only decreases until the minimum and then only increases, or vice versa). Binary search does not apply in this case, since just by looking at the value of the function at the midpoint of the interval, it is impossible to know whether we should recurse on the left or right half of the interval.

Suppose we are currently considering the interval $[L, R]$ and looking for the minimum of a function f on the interval. We compute the two points $1/3$ and $2/3$ of the way through the interval, namely $m_L = (2L + R)/3$ and $m_R = (L + 2R)/3$.

---

[1]Or $mid = lo + (hi - lo)/2$, if you are worried about $lo + hi$ overflowing, but this is unlikely to ever be an issue in a competitive programming context.

- If $m_L < m_R$, then we know the minimum can't be to the right of m_R (because then it would increase from m_L to m_R and then decrease—but we assume the function decreases until the minimum and then only increases after that). Hence, we can recurse on the interval $[L, m_R]$.

- If $m_L > m_R$, we can likewise recurse on $[m_L, R]$.

- If $m_L = m_R$, we can recurse on $[m_L, m_R]$ (though lumping this case in with either of the above two cases works fine and requires writing less code).

In any case, we decrease the size of the interval by at least $1/3$ with each iteration, so we need only a logarithmic number of iterations relative to the ratio between the starting interval size and the desired accuracy.

[TODO: Example code]

Integer ternary search

When ternary searching over an interval of *integers*, much of the same advice applies as for binary search (see the previous section). However, care must be taken with the stopping conditions; depending on exactly how the recursion works it is possible to end up in a scenario where it loops infinitely on an interval of size 1 or 2. Even if it is possible to come up with an elegant design that does not require any special cases, it may be easiest to simply stop the loop when the interval has size 2 or smaller, and then simply check the few remaining items manually.

Chapter 6

Graphs

6.1 Graph basics

- Every edge in a *directed* graph has an orientation, *i.e.* a "from" vertex and a "to" vertex. Edges in an *undirected* graph have no orientation.

- *Simple* graphs have at most one edge between any two vertices, and no self-loops. Most graph problems feature simple graphs. Sometimes, however, there can be loop edges from a vertex back to itself and/or multiple edges between the same two vertices.

[TODO: Directed, undirected, weighted, unweighted, self loops, multiple edges] [TODO: characterization of trees] [TODO: New virtual source/sink node trick]

> ⚙ chopwood

6.2 Graph representation

[TODO: Adjacency matrix, adjacency maps. Edge objects. Implicit graphs.]

Figure 6.1 has a sample solution for ⚙ horrorlist which builds an adjacency map representation of an undirected graph.

[TODO: State space search with complex states: make a class, implement Comparable, use TreeMap]

6.3 Breadth-First Search

> ⚙ ballsandneedles, brexit, buttonbashing, collapse, erdosnumbers, grapevine, horrorlist, mazemakers

Breadth-first search (BFS) can be used to find single-source shortest paths (*i.e.* shortest paths from a particular starting vertex to all other vertices) in an unweighted graph. BFS comes up often in many different guises, so it's worth being very familiar with BFS and its variants. Below is pseudocode showing a generic BFS implementation. Important invariants:

- Every vertex in Q has already been marked visited. (This is important since it prevents vertices from being added to Q multiple times.)

- Q only contains vertices from at most two (consecutive) levels at a time.

Algorithm BFS

1: $s \leftarrow$ starting vertex
2: Mark s visited
3: $Q \leftarrow$ new queue containing only s
4: $level[s] \leftarrow 0$
5: **while** Q not empty **do**
6:     $u \leftarrow Q.remove$
7:     **for** each neighbor v of u **do**
8:        **if** v is not visited **then**
9:           $level[v] \leftarrow level[u] + 1$ ▷ Optionally mark level
10:           Add v to Q
11:           Mark v visited
12:           $parent[v] \leftarrow u$ ▷ Optionally record parent

---

Some options/variants:

- The *level* array shown above is optional, and can be omitted if not needed. Sometimes it makes sense to have the *level* array do double-duty to also track visited vertices: if the *level* of every vertex is initialized to some nonsensical value such as -1 or ∞, then a vertex is visited iff its *level* is not equal to the initial value.

  Figure 6.1 shows a sample solution for 🔀 horrorlist , exhibiting a BFS with level labelling.

- The parent map is also optional, and can be used to reconstruct an actual shortest path from s to any vertex, by starting with the end vertex and iteratively following parents backwards until reaching s.

- If you want to compute shortest paths from *any* of a set of starting vertices, simply replace the initialization of s with the desired set (*i.e.* mark them all visited, add them all to Q, and set their *level* to 0 before starting the loop; the loop itself does not change) (🔀 zoning).

- Replacing Q with a stack results in a depth-first rather than breadth-first search (although often it makes more sense to implement a DFS recursively; see (§6.4, page 32)).

[TODO: Applications of BFS: identify reachable vertices; identify (weakly) connected components; identify bipartite graphs/odd cycles (detect cross-edges with map of level sets)]

6.4 DFS and SCCs

[TODO: Code for DFS, start/finish labelling, top sorting, Tarjan's SCC algorithm]

6.5 Topological sorting

🔀 builddeps, easyascab, eatingeverything, excavatorexpedition, mravi, promotions, reactivity, runningmom, succession

A *topological sort* of a directed graph G is a list of vertices such that whenever there is an edge from u to v, u comes before v in the list; G has a topological sort if and only if it is acyclic. Topological sorting can thus be used to detect the presence of cycles. It is also often used in conjunction with dynamic programming (🔀 eatingeverything, excavatorexpedition, mravi): if we need to compute some value of each vertex such that the value can be computed once we already know the values for all the outgoing (or incoming) neighbors, topological sort gives us the right order for computing the values.

There are two main methods to do a topological sort. Method 1 (Kahn's Algorithm) is to repeatedly remove nodes with no incoming edges (or dually, nodes with no outgoing edges). Empirically this seems to be faster than Method 2, but is perhaps a bit more code. Pseudocode is as follows:

```java
import java.util.*;

public class horrorList {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int movie_num = in.nextInt();
        int horror_num = in.nextInt();
        int linked_num = in.nextInt();
        int[] scores = new int[movie_num];
        Queue<Integer> hi = new ArrayDeque<Integer>();
        for(int i = 0; i < movie_num; i++) {
            scores[i] = Integer.MAX_VALUE;
        }
        for(int i = 0; i < horror_num; i++) {
            int a = in.nextInt();
            scores[a] = 0;
            hi.add(a);
        }
        HashMap<Integer, HashSet<Integer>> graph = new HashMap<>();
        int j = 0;
        while (j < linked_num) {
            int a = in.nextInt();
            int b = in.nextInt();
            if(! graph.containsKey(a)) graph.put(a, new HashSet<Integer>());
            if(! graph.containsKey(b)) graph.put(b, new HashSet<Integer>());
            graph.get(b).add(a);
            graph.get(a).add(b);
            j++;
        }
        while(! hi.isEmpty()) {
            int temp = hi.remove();
            if(graph.containsKey(temp)) {
                for(int i: graph.get(temp)) {
                    if( scores[i] == Integer.MAX_VALUE) {
                        scores[i] = scores[temp] + 1;
                        hi.add(i);
                    }
                }
            }
        }
        int output = 0;
        int max = 0;
        for(int i = 0; i < movie_num; i++) {
            if( scores[i] > max) {
                max = scores[i];
                output = i;
            }
        }
        System.out.println(output);
    }
}
```

Figure 6.1: Sample solution for `horrorlist` (Adjacency set representation; BFS with level labelling)

Algorithm TopSort(G)

Require: Directed graph $G = (V, E)$.
1: $T \leftarrow$ empty list (to store topsort)
2: $Z \leftarrow$ empty queue (to store nodes with 0 indegree)
3: $in \leftarrow$ dictionary mapping all vertices to their indegree
4: Put all vertices with indegree 0 into Z
5: **while** Z is not empty **do**
6:      $v \leftarrow Z.dequeue$
7:      append v to T
8:      **for** each u adjacent to v **do**
9:          decrement $in[u]$
10:          **if** $in[u] = 0$ **then**
11:              add u to Z

If the queue becomes empty before all vertices have been added to the topsort, then a cycle exists.

For a sample implementation of this algorithm, see the solution to 🔵 succession at https://github.com/Hendrix-CS/programming-team/blob/master/solved/Succession.java.

The second method is to do a recursive DFS: simply add each vertex to a list just *after* recursively processing all its neighbors; this yields a topsort in reverse order.

Algorithm Topological sort via DFS

1: **function** TopSort-DFS(G)
2:      $T \leftarrow$ empty list/stack to hold topsort
3:      **for all** $v \in V$ **do**
4:          **if** v is not visited **then** DFS(v, T)
5:      **return** T
6:
7: **function** DFS(x, T)
8:      Mark x visited
9:      **for all** $(x, y) \in E$ **do**
10:          **if** y is not visited **then** DFS(y, T)
11:      Add x to T

6.6 Single-source shortest paths (Dijkstra)

🔵 bigtruck, blockcrusher, coffeedate, george, getshorty, kitchen, shortestpath1, shortestpath2, showroom, walkway

Dijkstra's algorithm is the standard algorithm for solving the *single-source shortest path* problem in weighted, directed graphs. That is, given a graph with (possibly) directed edges and a weight on each edge, Dijkstra's algorithm can find the shortest directed path from a single chosen start vertex to every other vertex in the graph (where the length of a path is the sum of the weights on the edges). If you want to find the shortest path in a weighted, undirected graph, just make a directed graph with edges going both directions between each pair of vertices. Figure 6.2 has a basic implementation.

☞ Since Java's ☕ PriorityQueue class does not have a "decrease key" method, on line 28 we have to instead do a **remove** followed by an **add**; but **remove** is $O(n)$, making the whole algorithm $O(VE)$. If you really need $O(E \lg V)$ performance (🔵 flowerytrails), you can use an Adjustable priority queue (§4.5, page 24). In some situations you can also simply call **add** without calling **remove**; see the discussion below.

```java
import java.util.*;

public class Dijkstra {
    static int INF = Integer.MAX_VALUE;

    // Dijkstra's algorithm.  Assumes vertices are numbered 0 .. n-1.
    // Parameters = # of vertices, start vertex, and adjacency map
    // describing the (directed, weighted) graph.
    public static int[] dijkstra(int n, int s, Map<Integer, ArrayList<Edge>> g) {
        // parent is optional. If you need access to both dist and
        // parent after dijkstra runs, just make them both global
        // static variables.
        int[] parent = new int[n], dist = new int[n];
        Arrays.fill(dist, INF); dist[s] = 0;

        PriorityQueue<Integer> pq
            = new PriorityQueue<>(n, (Integer u, Integer v) -> dist[u] - dist[v]);
        pq.add(s);

        while (!pq.isEmpty()) {
            int cur = pq.remove();
            if (!g.containsKey(cur)) continue;

            for (Edge e : g.get(cur)) {
                int next = e.to;
                int nextDist = dist[cur] + e.weight;
                if (nextDist < dist[next]) {
                    dist[next] = nextDist; parent[next] = cur;
                    pq.remove(next); pq.add(next);
                }
            }
        }
        return dist;
    }
}

class Edge {
    public int to, weight;
    public Edge(int to, int weight) { this.to = to; this.weight = weight; }
}
```

Figure 6.2: $O(VE)$ Dijkstra's algorithm

[TODO: using adjust, you have to decide whether to 'add' or 'adjust']

There are many possible variants of this basic template; here are a few.

- The given code explores the *entire* graph. However, if you have a particular target vertex in mind you can stop early once you find it: just `break` out of the loop if removing the next node from the priority queue yields the target node, since at that point we are guaranteed that we know the shortest path from the start node to the target node.

- If the vertices of your graph are not naturally represented as integers in the range $0 \ldots n-1$, one could modify the algorithm to use `Map`s in place of the `parent` and `dist` arrays. Alternatively, it may be easier to deal with this outside of Dijkstra's algorithm: just arbitrarily assign indices to vertices and use a `Map` or two to keep track of the assignment. Then run Dijkstra using the assigned vertex indices and translate the result back to the original vertices.

- If the priority queue contains objects whose priority never changes once they are put in the priority queue (note that the example code in Figure 6.2 does *not* have this property, since `Integer`s in the PQ are compared by the value stored in the external array `dist`, which can change) then it can be an optimization to simply call `pq.add(next)` without calling `pq.remove(next)` first. The priority queue will end up with multiple copies of the same node, each with a different priority, but this is not a problem; when removing the next node from the PQ just ignore it if it has already been visited. (🎲 `nikola`)

- Dijkstra's algorithm uses addition to combine the weights of consecutive edges and `min` to pick the shortest path among parallel options. However, there are other pairs of operations one can use with the same basic algorithm template.[1]

  - Using `min`/`max` in place of `+`/`min` yields an algorithm which finds the path with the maximum possible minimum weight (🎲 `vuk`, `crowdcontrol`, `muddyhike`). For example, if the edge weights are thought of as capacities, and the capacity of a path is equal to the minimum capacity of any of its edges (*i.e.* the bottleneck) then this corresponds to finding maximum-capacity paths. One must be careful to:
    * update the comparison operation for the priority queue to use `min` instead of `max` (*e.g.* by switching to `dist[v] - dist[u]` instead of `dist[u] - dist[v]`),
    * initialize all the entries of `dist` to an appropriate identity value for `max` such as `0`, `-1`, or `-INF` instead of `INF`,
    * change the definition of `nextDist` to use `min` instead of `+`, and
    * change the comparison of `nextDist` and `dist[next]` to use `>` instead of `<`.
  - If we have a directed graph with edge weights corresponding to probabilities, where the probability of a path is defined as the product of the probabilities of its edges, then Dijkstra's algorithm with `*`/`max` finds highest-probability paths. Similar modifications have to be made as in the previous example.

- One can modify the basic algorithm to keep track of extra information, such as the *number* of shortest paths from the start to any given node: add to the count when finding a new path equal in weight to the previous best-known path; reset the count when finding a shorter path than previously known (🎲 `visualgo`).

- Dijkstra can also deal with edges whose weight depends on the time they are reached (think of *e.g.* bus routes, where you may have to wait a while for the next bus to come depending on what time you reach the stop). (🎲 `coffeedate`)

---

[1] The details of which properties of the operations are needed for this to work are too far outside the scope of this document; see [TODO: XXX]

6.7   All-pairs shortest paths (Floyd-Warshall)

crosscountry, allpairspath, shoppingmalls, transportationplanning, units

The *all-pairs shortest path* problem is to find the shortest path in a (directed, weighted) graph between *any* pair of vertices. Typically the idea is to precompute some table(s) and then be able to quickly look up any pair of vertices to find the distance between them. This could be done by running *e.g.* Dijkstra's algorithm once from every vertex, but that takes at least $O(VE \log V)$ (which is $O(V^3 \log V)$ for a dense graph) and doesn't work if there are negative edge weights. The Floyd-Warshall algorithm runs in $O(V^3)$ no matter how many edges there, can handle negative edge weights, and is just a few lines of code.

Note this only works when $|V|$ is small enough for a cubic algorithm to fit in the time limits, typically something like $|V| \leq 400$, though I have seen examples with $|V|$ even up to 1000 that work. Each individual loop of Floyd-Warshall is only a few operations so the constant factor is very small.

Assume the vertices in G are labelled $\{0, \ldots, n-1\}$. Create a 2D matrix of distances d and initialize it like so:

$$d[i][j] = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if there is an edge } i \to j \\ \infty & \text{otherwise} \end{cases}$$

If there can be multiple edges from i to j, be sure to set $d[i][j]$ to the *minimum* of all the edge weights. In practice, for ∞, just use a value that is much larger than any other values that could occur in the problem.

Then the Floyd-Warshall algorithm is as follows. We iterate k from 0 to $n-1$; k represents the intermediate vertex we will consider. Then for every possible pair of vertices u and v, we check if there is a way to get from u to k and a way to get from k to v, and the sum of these distances is less than the current shortest distance from u to v. If so, we update it.

```
for (int k = 0; k < n; k++)
  for (int u = 0; u < n; u++)
    for (int v = 0; v < n; v++)
      if (d[u][k] < INF && d[k][v] < INF)
        d[u][v] = Math.min(d[u][v], d[u][k] + d[k][v]);
```

After running the above loops, $d[i][j]$ will contain the length of the shortest path from i to j. If there is no path from i to j then the length will be ∞. Analyzing the running time of this algorithm is a Data Structures student's dream: there are literally three nested loops which each iterate exactly $|V|$ times, so the running time is $O(V^3)$.

Variants:

- If you want to detect the presence of negative cycles, you can use the fact that after running the main algorithm, $d[i][i] < 0$ if and only if vertex i is contained in some negative cycle. Hence if there are no negative numbers along the diagonal of the matrix, the graph is negative-cycle-free. However, if there are negative cycles then the distances found may not be valid (if there is a negative cycle along a path from u to v then one could travel around the cycle any number of times before finally going to v).

- If you actually want to distinguish negative-cycle-free paths (for which the computed minimum distance is valid) from others (allpairspath), you can run the following additional code, which propagates information about negative cycles:

```
    for (int u = 0; u < n; u++)
      for (int v = 0; v < n; v++)
        for (int k = 0; d[u][v] != -INF && k < n; k++)
          if (d[u][k] < INF && d[k][k] < 0 && d[k][v] < INF)
            d[u][v] = -INF;
```

  For each pair of vertices u, v, we check all possible intermediate nodes k. If there is a path $u \to k$ and a path $k \to v$, and k is part of some negative cycle, then we set the distance from u to v to $-\infty$ to signify that the length of a path from u to v can be arbitrarily small.

- Sometimes you want to know not only the *length* of the shortest path, but the actual shortest path itself. This can be accomplished by keeping a 2D array *next* such that $next[i][j]$ stores the next vertex along the shortest path from i to j. Initialize $next[i][j]$ to j whenever there is an edge from i to j (it does not matter what value it has otherwise). Then update the `if` statement in the inner loop as follows:

  ```
  if (d[u][k] < INF && d[k][v] < INF && d[u][k] + d[k][v] < d[u][v]) {
    d[u][v] = d[u][k] + d[k][v];
    next[u][v] = next[u][k];
  }
  ```

  Now after running the algorithm, the shortest path from u to v can be recovered by looking up $u_2 = next[u][v]$, then $u_3 = next[u_2][v]$, and so on, until v is reached.

6.8   Min spanning trees (Kruskal)

> ♦ `drivingrange`, `islandhopping`, `jurassicjigsaw`, `lostmap`, `minspantree`, `treehouses`

Kruskal's algorithm is the go-to algorithm for computing a minimum spanning tree (MST). It is relatively straightforward to code, given an implementation of a Union-find (§4.3, page 23) data structure.

- Create an initial union-find structure `uf` with one entry corresponding to each vertex.

- Sort the edges of the graph by weight. Typically, one makes a small `class` to store an edge (it may store *e.g.* the two endpoints of the edge and its weight), which implements `Comparable` in such a way that `compareTo` compares the weights. Then one can simply make an `ArrayList` of edge objects and call `Collections.sort` on it.

- Iterate through the edges from smallest to largest weight.

- For each edge, check whether its endpoints are already connected (`uf.find(x) == uf.find(y)`). If not, connect them (`uf.union(x,y)`) and add the edge to the MST. (If so, discard the edge and move on to the next.)

- Stop as soon as the number of chosen edges is one less than the number of vertices.

Given an efficient union-find implementation, the running time is dominated by the time to sort the edges, $O(E \lg E)$.

An example solution for ♦ `minspantree` is shown in Figure 6.3.

6.9   Eulerian paths

> ♦ `railroad2`, `eulerianpath`, `catenyms`

An *Eulerian path* is one which traverses every edge exactly once (but may visit vertices multiple times). An *Eulerian circuit* is an Eulerian path which starts and ends at the same vertex.

Checking whether an Eulerian path/circuit *exists* (♦ `railroad2`) is relatively simple:

- An undirected graph has an Eulerian path if and only if it is connected, and exactly zero or two vertices have odd degree, and all the rest have even degree. If all vertices have even degree then it the Eulerian path is actually a cycle.

- A *directed* graph has an Eulerian path if and only if it is strongly connected, and every vertex has equal in- and out-degrees, except possibly two, one of which must have one more incoming edge than outgoing, and the other has one more outgoing edge than incoming. If all vertices have equal in- and out-degree then the Eulerian path is actually a cycle.

```java
import java.util.*;

class Edge implements Comparable<Edge> {
    int from, to, weight;
    public Edge(int _from, int _to, int _weight) {
        if (_from < _to) { from = _from; to = _to; }  // problem requires vertices sorted
        else             { from = _to; to = _from; }
        weight = _weight;
    }
    public int compareTo(Edge e) { return weight - e.weight; }
    public int getFrom() { return from; }
    public int getTo() { return to; }
}

public class MinSpanTree {
    public static void main(String[] args) {
        Kattio io = new Kattio(System.in, System.out);

        while (true) {
            int n = io.getInt(); int m = io.getInt();
            if (n == 0 && m == 0) break;

            ArrayList<Edge> edges = new ArrayList<>();
            for (int i = 0; i < m; i++) {
                edges.add(new Edge(io.getInt(), io.getInt(), io.getInt()));
            }

            // ----- Kruskal's algorithm -----
            Collections.sort(edges);  // sort by weight
            UnionFind uf = new UnionFind(n);
            ArrayList<Edge> mstEdges = new ArrayList<Edge>();

            for (Edge e : edges) {
                if (uf.find(e.from) != uf.find(e.to)) {
                    uf.union(e.from, e.to);
                    mstEdges.add(e);
                }
                if (mstEdges.size() == n-1) break;
            }

            // ----- Output -----
            if (mstEdges.size() != n - 1) {
                io.println("Impossible");  // No spanning tree exists
            } else {
                int total = 0;
                for (Edge e : mstEdges) total += e.weight;
                io.println(total);

                // sort edges lexicographically
                Collections.sort(mstEdges,
                    Comparator.comparing(Edge::getFrom).thenComparing(Edge::getTo));

                for (Edge e : mstEdges) io.println(e.from + " " + e.to);
            }
        }
        io.flush();
    }
}
```

Figure 6.3: Sample solution for minspantree

In other words, first check whether the graph is connected using DFS and SCCs (§6.4, page 32) or Breadth-First Search (§6.3, page 31). Then compute the (in/out) degrees of every vertex and check how many are even/odd (for undirected graphs) or how many have matching in/out degrees (for directed).

To *find* an Eulerian path, use Hierholzer's Algorithm. In an undirected graph, start at a vertex with odd degree, (or at any vertex if there are no odd-degree vertices); in a directed graph, start at the vertex whose outdegree is one greater than the indegree (or any vertex if all have equal in/out-degree). [TODO: Explain, example code.]

6.10 Max flow/min cut

>   copsandrobbers, escapeplan, gopher2, guardianofdecency, marblestree, maxflow, mincut, paintball, pianolessons, waif

A *flow network* is a directed, weighted graph where the edge weights (typically integers) are thought of as representing *capacities* (*e.g.* imagine pipes of varying sizes). The *max flow problem* is to determine, given a flow network, the maximum possible amount of *flow* which can move through the network between given source and sink vertices, subject to the constraints that the flow on any edge is no greater than the capacity, and the sum of incoming flows equals the sum of outgoing flows at every vertex other than the source or sink.

6.10.1 Flow network problem types

- Certain types of problems about optimally assigning items or resources subject to some constraints can be solved by finding a maximum flow in an appropriate flow network (escapeplan, gopher2, pianolessons, waif).

- Maximum matchings in a bipartite graph can be found by creating two new virtual nodes, a source node with a connection to every vertex in the left-hand set and a sink node with a connection from every vertex in the right-hand set. Set all edge capacities to 1; then a maximum flow corresponds to a maximum matching in the graph (guardianofdecency, paintball)

- A famous theorem asserts that the maximum flow on a network corresponds exactly to the *minimum cut*, which is the minimum "bottleneck", *i.e.* the minimum possible sum of capacities of a set of edges that splits the graph into two halves (mincut, copsandrobbers).

6.10.2 Flow network variants

- Flow networks must have a single source node and a single sink node. You can model multiple sources/sinks simply by adding a new virtual source and/or sink node and connecting it to all the source/sink nodes with infinite capacity edges.

- To model networks where the *vertices* have capacities, just split each vertex into two vertices with an edge in between them having the given vertex capacity. All the incoming edges connect to the first new vertex and all the outgoing edges emanate from the second new vertex.

- [TODO: Minimum-cost max flow: use Edmonds-Karp with Dijkstra?]

6.10.3 Dinitz' Algorithm

Dinitz' Algorithm[2] is probably the best all-around algorithm to use for solving max flow problems in competitive programming. It takes $O(V^2E)$ in theory (although is often much faster in practice). In the special case where we are modelling a bipartite matching problem, Dinitz' Algorithm reduces to the Hopcroft-Karp algorithm which runs in $O(E\sqrt{V})$.

Some general guidelines for using the max flow code below:

---

[2] You may also see it spelled "Dinic's Algorithm" but this is not the preferred spelling of its inventor, Yefim Dinitz.

- Be very careful to decide which edges should be directed and which should be undirected; this makes a big difference, and the code given below requires calling `addDirEdge` or `addEdge` appropriately (calling `addEdge` is *not* the same as calling `addDirEdge` once in each direction!).

- The vertices of the graph must be labelled $0 \ldots n-1$. Typically they have some other labels which are specified as part of the problem. You must carefully keep track of which entities in the problem map to which vertex indices, either via some formulas or using some lookup tables.

```java
class FlowNetwork {
    private static final int INF = ~(1<<31);
    int[] level;
    boolean[] pruned;
    HashMap<Integer, HashMap<Integer, Edge>> adj;

    public FlowNetwork(int n) {
        level = new int[n];
        pruned = new boolean[n];
        adj = new HashMap<>();

        for (int i = 0; i < n; i++)
            adj.put(i, new HashMap<>());
    }

    public void addDirEdge(int u, int v, long cap) {
        if (adj.get(u).containsKey(v)) {
            adj.get(u).get(v).capacity = cap;
        } else {
            Edge e = new Edge(u,v,cap);
            Edge r = new Edge(v,u,0);
            e.setRev(r);
            adj.get(u).put(v, e);
            adj.get(v).put(u, r);
        }
    }

    // Add an UNdirected edge u<->v with a given capacity
    public void addEdge(int u, int v, long cap) {
        Edge e = new Edge(u,v,cap);
        Edge r = new Edge(v,u,cap);
        e.setRev(r);
        adj.get(u).put(v, e);
        adj.get(v).put(u, r);
    }

    public long maxFlow(int s, int t) {
        if (s == t) return INF;
        else {
            long totalFlow = 0;
            while (bfs(s,t)) totalFlow += sendFlow(s,t);
            return totalFlow;
        }
    }

    private long sendFlow(int s, int t) {
        for (int i = 0; i < pruned.length; i++)
            pruned[i] = false;
        return sendFlowR(s, t, INF);
    }

```

```java
52        private long sendFlowR(int s, int t, long available) {
53            if (s == t) return available;
54
55            long sent = 0;
56            for (Edge e : adj.get(s).values()) {
57                if (e.remaining() > 0 && !pruned[e.to] && level[e.to] == level[s] + 1) {
58                    long flow = sendFlowR(e.to, t, Math.min(available, e.remaining()));
59                    available -= flow; sent += flow;
60                    e.flow += flow; e.rev.flow -= flow;
61                    if (available == 0) break;
62                }
63            }
64            if (sent == 0) pruned[s] = true;
65            return sent;
66        }
67
68        private boolean bfs(int s, int t) {
69            for (int i = 0; i < level.length; i++) level[i] = -1;
70
71            Queue<Integer> q = new ArrayDeque<>();
72            q.add(s); level[s] = 0;
73            while (!q.isEmpty()) {
74                int cur = q.remove();
75                for (Edge e : adj.get(cur).values()) {
76                    if (e.remaining() > 0 && level[e.to] == -1) {
77                        level[e.to] = level[cur]+1;
78                        q.add(e.to);
79                    }
80                }
81            }
82            return level[t] >= 0;
83        }
84    }
85
86    class Edge {
87        int from, to;
88        long capacity, flow;
89        Edge rev;
90        public Edge(int from, int to, long cap) {
91            this.from = from; this.to = to; this.capacity = cap; this.flow = 0;
92        }
93        public void setRev(Edge rev) { this.rev = rev; rev.rev = this; }
94        public long remaining() { return capacity - flow; }
95    }
```

[TODO: Include a sample solution using a flow network]

Chapter 7

Dynamic Programming

balanceddiet, drivinglanes, ticketpricing, walkforest

[TODO: subset sum] [TODO: knapsack, longest common subsequence] [TODO: longest increasing subsequence ($O(n^2)$ and $O(n \lg n)$), see https://stackoverflow.com/questions/2631726/how-to-determine-the-longest-inc

Chapter 8

Sequences and strings

8.1 Longest Increasing Subsequence (LIS)

> ♻ `increasingsubsequence`, `longincsubseq`, `manhattanmornings`, `signals`

A *subsequence* of a sequence is a subset of the elements, taken in order, but not necessarily contiguous. (By contrast, a contiguous subset of elements is often referred to as a *subinterval*.) For example, $[1, 3, 4, 7]$ is a subsequence of $[1, 2, 3, 4, 5, 6, 7, 8]$. Given a sequence of integers (or any elements which can be ordered), the *longest increasing subsequence* (LIS) problem is to find the longest subsequence which is in strictly increasing order. For example, given the sequence $[9, 2, 8, 10, 5, 4, 20, 16, 7, 1]$, one increasing subsequence is $[2, 5, 16]$, but it is not the longest. There are several increasing subsequences of length 4, such as $[2, 8, 10, 16]$, and it turns out that this is the longest possible.

Conceptually, to compute the LIS of a sequence, we first build a *downravel*, a set of nonincreasing subsequences which partition the original sequence. We keep these subsequences as a list of stacks, and maintain the invariant that their top elements are always sorted from smallest to biggest. We iterate through the elements of S and push each onto the leftmost possible stack whose top is $\geq$ the element being added.

Algorithm Building a downravel of a sequence S

1: **function** DOWNRAVEL(S)
2:     $D \leftarrow$ empty list of stacks
3:     **for all** $x \in S$ **do**
4:         $k \leftarrow$ first stack in D whose top is $\geq x$
5:         **if** no such k exists **then**
6:             Add a new singleton stack containing x to the end of D
7:         **else**
8:             Push x onto k
9:     **return** D

[TODO: add some pictures?]

The length of the LIS is the same as the length of the downravel D. Naïvely, this runs in $O(n^2)$ time, since for each of the the n elements in S, we have to search through up to $O(n)$ stacks in D to find the right one to push. However, there are several possible optimizations.

- First, although the stacks are conceptually helpful, we do not actually need to store them. It's enough to simply store the current top element of each stack. So instead of having a list of stacks we just have a list of elements.

- Second, since this list will always be sorted from smallest to biggest, we can use a binary search to find the proper place to push each new element. This brings the running time down to a very respectable $O(n \lg n)$.

Of course, we might want not just the *length* of the LIS but an actual LIS itself.

- First of all, we need to modify D so it stores not the elements themselves but their indices in S. This requires a bit of extra indirection while doing a binary search for the correct place to put each new element.

- We keep an extra array *back* such that *back*[i] stores the index of an element that could come before $S[i]$ in a LIS up to and including $S[i]$. Every time we put a new element $S[i]$ into D, we set *back*[i] to the previous value in D—that is, the index of the element currently on top of the previous stack in the list.

- After running the algorithm we start with the item represented by the last entry in D, and then keep following *back* links to get each previous item. This yields a LIS in reverse order.

[TODO: include some examples and pictures]

```java
public class LIS {
    public static int[] LIS(int[] arr) {
        if (arr.length == 0) return new int[0];
        int[] prev = new int[arr.length];
        ArrayList<Integer> downravel = new ArrayList<>();

        for (int i = 0; i < arr.length; i++) {
            int opt = 0, lo = 1, hi = downravel.size();
            while (lo <= hi) {
                int mid = (lo + hi)/2;

                // For longest *non-decreasing* sequence, change < to <=
                if (arr[downravel.get(mid-1)] < arr[i]) { opt = mid; lo = mid + 1; }
                else { hi = mid - 1; }
            }

            if (opt < downravel.size()) downravel.set(opt, i);
            else downravel.add(i);
            prev[i] = opt == 0 ? -1 : downravel.get(opt-1);
        }

        int[] incseq = new int[downravel.size()];
        int j = downravel.size() - 1, cur = downravel.get(j);
        while (cur != -1) { incseq[j] = cur; cur = prev[cur]; j--; }
        return incseq;
    }
}
```

8.2   LCS via LIS

> ⟳ inflagrantedelicto, princeandprincess

Given two sequences, the *longest common subsequence* (LCS) problem is to find the longest sequence which is a subsequence of both. This comes up in quite a few real-world applications including DNA processing and diffing (*i.e.* what Github does when it shows you which lines have changed).

Most generally, the LCS of two sequences with lengths m and n can be computed via Dynamic Programming (§7, page 43) in $O(mn)$ time. However, in the special case that the sequences do not have too many repeated elements, it is possible to solve it more quickly, as follows.

Suppose the sequences are called A and B.

- Construct all pairs of indices (i, j) such that $A[i] = B[j]$, that is, all pairs of locations where A and B agree.

- Sort these pairs lexicographically, that is, sort them first by i and break ties by j.

- Now make an array consisting of just the j values from these sorted pairs.

- A longest increasing subsequence in this array of j values gives the length of a LCS of A and B. (Exercise for the reader: why does this work?)

8.3 Z-algorithm

8.4 Suffix arrays

Chapter 9

Mathematics

9.1   GCD/Euclidean Algorithm

The *Euclidean algorithm* can be used to compute the greatest common divisor of two **nonnegative** integers. (If you need it to work for negative numbers as well, just take absolute values first.) It runs in logarithmic time. The *extended Euclidean algorithm* not only finds the GCD g of a and b, but also finds integers x and y such that $ax + by = g$.

> ☘ fairwarning, jughard, kutevi, candydistribution, diagonalcut

```java
public class GCD {
    public static long gcd(long a, long b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    public static EGCD egcd(long a, long b) {
        if (b == 0) return new EGCD(a, 1, 0);
        EGCD e = egcd(b, a % b);
        return new EGCD(e.g, e.y, e.x - a / b * e.y);
    }
}

class EGCD {  // For storing result of egcd function
    long g, x, y;
    public EGCD(long _g, long _x, long _y) {
        g = _g; x = _x; y = _y;
    }
}
```

9.2   Rational numbers

> ☘ bikegears, jointattack, prosjek, prsteni, rationalarithmetic, wheels, zipfsong

Occasional problems may require dealing with explicit rational values rather than using floating-point approximations. If a problem involves non-integer values but requires being able to test values for equality *exactly*, then likely rational numbers are required. The below code for a `Rational` class is not difficult but it's nice to have it as a reference. Of course in a real contest situation you may not need all the methods.

```java
1  class Rational implements Comparable<Rational> {
2      long n, d;
3      public Rational(long _n, long _d) {
4          n = _n; d = _d;
5          if (d < 0) { n = -n; d = -d; }
6          long g = gcd(Math.abs(n),d); n /= g; d /= g;
7      }
8      private long gcd(long a, long b) {
9          return b == 0 ? a : gcd(b, a % b);
10     }
11     public Rational(long n) { this(n,1); }
12
13     public Rational plus(Rational other) {
14         return new Rational(n * other.d + other.n * d, d * other.d);
15     }
16     public Rational minus(Rational other) {
17         return new Rational(n * other.d - other.n * d, d * other.d);
18     }
19     public Rational negate() {
20         return new Rational(-n, d);
21     }
22     public Rational times(Rational other) {
23         return new Rational(n * other.n, d * other.d);
24     }
25     public Rational divide(Rational other) {
26         return new Rational(n * other.d, d * other.n);
27     }
28     public boolean equals(Object otherObj) {
29         Rational other = (Rational)otherObj;
30         return (n == other.n) && (d == other.d);
31     }
32     public int compareTo(Rational r) {
33         long diff = n * r.d - d * r.n;
34         if (diff < 0) return -1;
35         else if (diff > 0) return 1;
36         else return 0;
37     }
38     public String toString() {
39         return d == 1 ? ("" + n) : (n + "/" + d);
40     }
41 }
```

9.3   Modular arithmetic

> ♻ crackingrsa, modulararithmetic, pseudoprime, reducedidnumbers

> ☞ Java's mod operator % behaves strangely on negative numbers. In many other languages (*e.g.* Python, Haskell) a % b always returns a result between 0 and $b - 1$; however, in Java (as in C/C++), if a is negative then a % b will also be negative. Try adding b first if you need a nonnegative result.

For example, suppose i is an index into an array of length n and you need to shift by an offset o, wrapping around in case the index goes off the end of the array. The obvious way to write this would be

```
i = (i + o) % n;
```

however, this is **incorrect if o could be negative!** If we assume that `o` will never be larger in absolute value than `n`, then we could write this correctly as

```
i = (i + o + n) % n;
```

If `o` could be arbitrarily large then we could write

```
i = (((i + o) % n) + n) % n;
```

(the first mod operation reduces it to lie between $-n \ldots n$; adding n ensures it is positive; and the final mod reduces it to the range $[0, n)$).

Modular exponentiation and modular inverses

Sometimes one needs to compute the modular exponentiation $b^e \bmod m$ for some base b, exponent e, and modulus m. Using repeated squaring, it is possible to do this efficiently even for very large exponents e. Relatedly, if b is relatively prime to m, it is possible to compute $b^{-1} \bmod m$, the *modular inverse* of b, that is, the unique number $0 < b' < m$ such that $bb' \equiv 1 \pmod{m}$.

In Java, probably the easiest way to compute these is using the `modPow` method from the `BigInteger` class (§2.15, page 14). If `b`, `e`, and `m` are `BigInteger`s, then `b.modPow(e, m)` is a `BigInteger` that represents $b^e \bmod m$. The exponent `e` can also be negative; in particular, if `e` is -1 then `b.modPow(e,m)` will compute the inverse of `b` modulo `m`.

It is also useful to know how to compute modular exponentiation and inverses manually, in case you need some sort of variant version, or if `BigInteger` is not fast enough.

Modular exponentiation can be computed by repeated squaring. The basic idea is to compute b^e by splitting up e into a sum of powers of two (according to its binary expansion), raising b to each power of two and taking the product. This can be done efficiently since we can get from b^{2^k} to $b^{2^{k+1}}$ just by squaring.

> ☞ Even if you need the answer modulo an `int` value such as $10^9 + 7$, it is important to use `long` in the method below: the product of two `int` values does not necessarily fit in an `int`, even if the very next step will reduce it modulo m back into the range of an `int`.

```
2  public static long modexp(long b, long e, long m) {
3      long res = 1;
4      while (e > 0) {
5          if ((e & 1) == 1) res = (res * b) % m;  // include current power of b?
6          b = (b * b) % m;                         // square to get next power of b
7          e >>= 1;                                  // shift out rightmost bit of e
8      }
9      return res;
10 }
```

Note this correctly computes $0^0 = 1$. It would be possible to add a special case for when $b = 0$ and $e \neq 1$, to avoid multiplying 0 by itself a bunch of times, but it's hardly worth it.

Modular inverses can be computed using the extended Euclidean algorithm (§9.1, page 49). In particular, suppose a and b are relatively prime, that is, their GCD is 1. In that case the `egcd` algorithm will compute numbers x and y such that $ax + by = 1$. Taking this equation $(\bmod b)$ yields

$$ax + by \equiv ax \equiv 1 \pmod{b},$$

and so x is the modular inverse of a modulo b (in practice one may want to reduce $x \bmod b$ so x is between 0 and $b - 1$).

Alternatively, for a prime p, Fermat's Little Theorem says that

$$a^{p-1} \equiv 1 \pmod{p}$$

and hence a^{p-2} is the modular inverse of a modulo p, which can be computed using modular exponentiation.

9.4 Primes and factorization

Methods for primality testing and prime factorization that may show up in a contest can be put in two main classes. First, methods based on *trial division* are relatively simple to code and work well for testing just one or a few numbers. *Sieve* based methods construct a whole table of primes or factors all at once, and are often more efficient when many numbers need to be factored or tested for primality.

9.4.1 Trial division

> ♻ almostperfect, candydivision, crypto, enlarginghashtables, flowergarden, goldbach2, happyprime, iks, listgame, olderbrother, pascal, primalrepresentation

To test whether a single number is prime, you can use the following function which performs (somewhat optimized) trial division. Note that although there are faster primality testing methods (*e.g.* Miller-Rabin, Baille-PSW), it is highly unlikely that a contest would ever require anything more sophisticated than divisibility testing: Miller-Rabin is not hard to code but it is probabilistic, so a program using it may give different results on subsequent runs, hardly suitable for a competitive programming environment; Baille-PSW is known to be deterministic for numbers up to 2^{64}, but is much more complex to code.

Note that isPrime has runtime $O(\sqrt{n})$ and is hence appropriate for numbers up to the maximum size of an int ($\approx 2 \cdot 10^9$); running it on inputs up to the maximum size of a long is likely to be too slow.

```
2   public static boolean isPrime(int n) {
3       if (n < 2) return false;
4       if (n < 4) return true;
5       if (n % 2 == 0 || n % 3 == 0) return false;
6       if (n < 25) return true;
7       for (int i = 5; i*i <= n; i += 6) // O(√n)
8           if (n % i == 0 || n % (i + 2) == 0) return false;
9       return true;
10  }
```

The following method takes $O(\sqrt{n})$ to factor a number into its prime factorization, also using trial division. The returned prime factors will be sorted from smallest to biggest.

```
4   public static ArrayList<Integer> factor(int n) {
5       ArrayList<Integer> factors = new ArrayList<>();
6       while ((n & 1) == 0) { factors.add(2); n >>= 1; }  // get factors of 2
7       int d = 3;                    // get odd factors
8       while (d*d <= n) {            // O(√n)
9           if (n % d == 0) {
10              factors.add(d);       // found a factor
11              n /= d;
12          } else {                  // try next odd divisor
13              d += 2;
14          }
15      }
16      if (n != 1) factors.add(n);   // don't forget final prime
17      return factors;
18  }
```

9.4.2 Sieving

> ♻ industrialspy, nonprimefactors, primereduction, primesieve, reseto

The term *sieve* comes from the ancient *Sieve of Eratosthenes*, a very effective method for generating all the primes up to a certain bound. The basic idea is to make a table of all the numbers from 1 up to some upper bound n and iterate through the table. Each time we discover a prime p we "cross out" all the multiples of p in the table; we know a number is prime if it hasn't yet been crossed out by the time we get to. This takes time $O(n \log \log n)$ (essentially linear time) to construct a table for $1 \ldots n$. The code below uses a ⚖ BitSet which uses less memory than an array of `boolean`s. Constructing a `PrimeSieve` of size 10^8 should take about a second and use only about 12 MB of memory; constructing smaller prime sieves should be quite fast. Even a `PrimeSieve` of size `Integer.MAX_VALUE`, *i.e.* $\approx 2 \cdot 10^9$, will fit quite easily in memory, although constructing it will probably take too long for most contest problems. (However, there may be occasional problems that require building a sieve of this size in order to precompute some data offline—*i.e.* writing a program that runs for a few minutes in order to precompute some kind of set or lookup table to be included in the submitted solution.)

```java
import java.util.*;

public class PrimeSieve {
    BitSet prime;
    public PrimeSieve(int MAX) {
        prime = new BitSet(MAX+1);
        prime.set(2,MAX+1,true);            // initialize all to true
        for (int p = 2; p*p <= MAX; p++)    // iterate up to √MAX
            if (prime.get(p))               // found a prime p
                for (int m = p*p; m <= MAX; m += p)  // cross out multiples of p
                    prime.set(m,false);
    }
    public boolean isPrime(int n) {  // Once sieve is built, test primality in O(1)
        return prime.get(n);
    }
}
```

Instead of simply storing a boolean indicating whether each number is prime or not, we could also store the smallest prime factor. We can still use this to test whether a given number is prime, by checking whether `smallest[n] == n`. But we can also use it to quickly factor any composite `n`: simply divide `n` by `smallest[n]` and repeat. We can construct the smallest factor array using a sieving method similar to `PrimeSieve`. The tradeoff is that this uses much more memory: instead of one bit per number, we use an entire `int`, that is, 32 bits. A `FactorSieve` of size 10^8 will take up around 380 MB.

The `FactorSieve` class below includes a trivial `isPrime` method as well as a `factor` method, which is carefully written to work even for `int` values which are bigger than the lookup table.

```java
import java.util.*;

public class FactorSieve {
    int[] smallest;
    public FactorSieve(int MAX) {
        smallest = new int[MAX+1];
        smallest[1] = 1;
        int p = 2;
        for (; p*p <= MAX; p++) {      // Sieve up to √MAX
            if (smallest[p] == 0) {
                smallest[p] = p;
                for (int m = p*p; m <= MAX; m += p)
                    if (smallest[m] == 0) smallest[m] = p;
            }
        }
        for (; p <= MAX; p++)          // Fill in remaining primes
```

```
17              if (smallest[p] == 0)
18                  smallest[p] = p;
19      }
20
21      public boolean isPrime(int n) {
22          return n > 1 && smallest[n] == n;
23      }
24
25      public ArrayList<Integer> factor(int n) {
26          ArrayList<Integer> factors = new ArrayList<>();
27          while ((n & 1) == 0) { factors.add(2); n >>= 1; }
28          int d = 3;
29          // Pull out factors until n is small enough to look up
30          while (d*d <= n && n >= smallest.length) {
31              if (n % d == 0) {
32                  factors.add(d);
33                  n /= d;
34              } else {
35                  d += 2;
36              }
37          }
38          // Now just look up remaining factors in the table
39          if (n < smallest.length) {
40              while (smallest[n] != n) {
41                  factors.add(smallest[n]);
42                  n /= smallest[n];
43              }
44          }
45          if (n != 1) factors.add(n);
46          return factors;
47      }
48  }
```

9.5   Divisors and Euler's Totient Function

> ⊛ farey, relatives

[TODO: Number of divisors. Euler's φ function: computing directly and by sieving.]

9.6   Factorial

> ⊛ eulersnumber, factstone, howmanydigits, lastfactorialdigit, inversefactorial, loworderzeros, factovisors

$n! = 1 \cdot 2 \cdots n$ is the number of ways of arranging n things in a sequence. Computing $n!$ is straightforward with a loop, although note that

- $12! = 479001600$ is the largest factorial that fits in a 32-bit `int`, and

- $20! = 2432902008176640000$ is the largest factorial that fits in a 64-bit `long`.

Note that $\log(n!) = \log(1 \cdot 2 \cdots n) = \log 1 + \log 2 + \cdots + \log n$ which is occasionally handy. For example, the number of base-10 digits needed to represent a number n is $\lfloor \log_{10} n \rfloor$, so by summing logs instead of computing a factorial and then taking the log, you can figure out how many digits are in very large factorials even when the numbers themselves would not fit in a `long` (⊛ howmanydigits).

9.7 Combinatorics

☘ `insert`, `anagramcounting`, `nine`, `secretsanta`, `kingscolors`, `howmanyzeros`

Some basic principles of combinatorics:

- If two sets of choices are completely disjoint, add their sizes to get the total number of choices. For example, the number of subsets of $\{1, \ldots, n\}$ is equal to the number of subsets that do contain 3 plus the number that don't.

- If two sets of choices are independent, multiply their sizes to get the total number of combinations. For example, if we can pick one of five different shirts and independently pick one of seven different hats, we have 35 possible outfit choices.

- Often, the answer to a combinatorics problem will be very large, so the problem asks for the answer modulo $10^9 + 7$ (the smallest prime bigger than 10^9), which fits in a 32-bit `int`. Since taking remainders commutes with addition and multiplication, just reduce via mod at every step to make sure that the intermediate values never overflow.

> ☞ Although the *sum* of two values under $10^9 + 7$ will fit in a 32-bit `int`, their *product* will not. If you need an answer modulo $10^9 + 7$ but computing the answer involves multiplication, you must use 64-bit (`long`) values to make sure the intermediate steps do not overflow.

For example, to compute $n!$ (mod $10^9 + 7$), make a `long` accumulator initialized to 1, and then loop from 1 to n, on each step multiplying by the current index and then taking the remainder mod $10^9 + 7$.

9.7.1 Binomial and multinomial coefficients

The binomial coefficient

$$\binom{n}{k} = \binom{n}{n-k} = \frac{n!}{k!(n-k)!}$$

, aka the kth entry in the nth row (both counting from 0) of Pascal's triangle, counts the number of ways to choose a set of k things out of n possibilities, and can be computed using the following code, which works up to $n = 60$ (higher values of n will cause overflow):

```
3   public static long choose(int n, int k) {
4       if (n < 0 || k < 0 || k > n) return 0;
5
6       k = Math.min(k,n-k);
7       long res = 1;
8       for (int i = 1; i <= k; i++) {
9           res = res * (n-i+1) / i;
10      }
11      return res;
12  }
```

Some useful identities:

- $\binom{n}{0} = \binom{n}{n} = 1$ (there is only one way to choose none of the items, or all of the items).

- $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

The *multinomial* coefficient

$$\binom{n}{k_1 \ k_2 \ \ldots \ k_l} = \frac{n!}{k_1! k_2! \cdots k_l!},$$

where $n = k_1 + k_2 + \cdots + k_l$, represents the number of ways of partitioning a set of n things into groups of sizes $k_1, k_2, \ldots, k_l$. Note that the usual binomial coefficient $\binom{n}{k}$ can be thought of as the multinomial coefficient $\binom{n}{k\ (n-k)}$.

[TODO: Computing multinomial coefficients.]

[TODO: Large binomial coefficients modulo a prime (modular inverse factorial tables, Lucas's theorem).]

[TODO: Heap's Algorithm for generating all permutations; next permutation. See Bit Tricks for generating all subsets.]

[TODO: PIE]

9.8   Probability

[TODO: Write me]

Chapter 10

Bit Tricks

int values are represented as a sequence of 32 bits; long values are 64 bits. Sometimes it is useful to think about/work with such values directly as a sequence of bits rather than as a number. We typically think of the bits as indexed from 0 starting at the rightmost (least significant) bit. For example,

$$974_{10} = \underset{9\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0}{1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0}$$

In general, a 1 bit at index i has value 2^i.

One frequently useful point of view is to think of a value of type int/long as representing a particular subset of a given set of up to 32/64 items. The bit at index i indicates whether item i is included in the subset or not.

Java has built-in operators to manipulate values at the bit level:

- & represents bitwise logical AND. That is, the index-i bit of the result is the logical AND of the index-i bits of the inputs; each bit index is considered separately. It is often useful to think of & as a "masking" operation: given values v and mask, evaluating v & mask will only "let through" the bits of v which correspond to 1 bits in mask; all other bits will be "turned off". For example, if you want to extract only the last three bits of a value v, you can compute v & 7 (since bitwise AND with $7 = 111_2$ will turn off all bits except the last three).

  If values are thought of as representing subsets, then & corresponds to set intersection.

- | represents bitwise logical OR. This can be used to "turn on" certain bits: v & on will result in a value which is the same as v except that the bits which are set to 1 in on will be turned on.

  If values are thought of as representing subsets, then | corresponds to set union.

- ^ represents bitwise logical XOR. This can be used to "toggle" bits: v ^ toggle will result in a value which is the same as v except that the bits in positions corresponding to the 1 bits in toggle have been flipped.

  If values are thought of as representing subsets, then ^ corresponds to symmetric difference: a ^ b represents the set of elements which are in a or b but not both.

- n >> k shifts n right by k bits, chopping off the rightmost k bits. This corresponds to (integer) division by 2^k. n << k shifts n left by k bits, adding k zeros on the right; this corresponds to multiplying by 2^k.

  Note that right shifting uses something called *sign extension* so that it fills in bits on the left according to whatever the leftmost bit was initially: a value starting with a zero bit (*i.e.* a positive value) will have zeros filled in on the left, but a (negative) value beginning with a one bit will have ones filled in

on the left. If you don't want this (it rarely matters!) you can use `n >>> k` which does a right shift by k bits *without* sign extension, that is, it always fills in zero bits on the left regardless of the initial bit of n.

Here is a list of some non-obvious but sometimes useful things that can be done with bitmasks:

- To iterate through all possible subsets of an n-element set (represented by an n-bit mask), just use a counter:

  ```
  for (int S = 0; S < (1 << n); S++) {
      // process subset S
  }
  ```

  As the value of S progresses from 0 through $2^n - 1$, it will take on every possible pattern of n bits.

- To check whether a particular bit is turned on, mask out everything except that particular bit and check whether the result is 0. For example, to check whether bit j is set to 1 in S:

  ```
  if ((S & (1 << j)) != 0) ...
  ```

  Be careful: the precedence of `&` is actually lower than that of `!=`, so you need a bunch of parentheses.

- The *least significant bit* (LSB) of a value `S` can be extracted using the expression `S & (-S)`. The result is a value with only a single bit set, corresponding to the LSB of S. For example, if $S = 10001011000$, then `S & (-S)` will be `00000001000`. It's worth taking a minute to convince yourself that this works, keeping in mind that to negate a value in 2's complement representation, you invert all the bits and then add one.

  One way this can be used is when iterating over all subsets of a set: for each subset, instead of iterating over all elements and checking whether each one is in the subset, one can quickly iterate through only the elements which are actually in the subset. In some cases this can yield a big constant-factor speedup.

  ```
  for (int S = 0; S < (1 << n); S++) {
      int T = S;  // Save a copy of current subset S into T

      // Iterate through all elements of T
      while (T != 0) {
          int X = T & (-T);  // Find last element X of T
          ...                // process X
          T ^= X;            // Remove X to move on to the next element
      }
  }
  ```

  Another place this technique is used is in the implementation of Fenwick trees (§12.3.5, page 66).

- The *least significant zero* (LSZ) can be computed by first inverting all the bits, then finding the LSB.

- There is no quick way to compute the *most significant bit* (MSB), which amounts to finding the logarithm base 2 (rounded down). The simplest is to keep shifting right until reaching 1, keeping a count of the number of shifts required.

- The *popcount* operation counts the number of 1 bits in a number, and sometimes comes in useful (🎲 bits, iboard, enviousexponents, pebblesolitaire). It can be accessed via the `Long.bitCount(...)` or `Integer.bitCount(...)` functions. Note that processors typically have a special popcount instruction, so this should be very fast—certainly much faster than manually looping through the bits of a number and counting how many are set to 1.

- Iterating through all the sub-subsets of a subset https://cp-algorithms.com/algebra/all-submasks.html

[TODO: iterating through sub-subsets] `https://cp-algorithms.com/algebra/all-submasks.html`
[TODO: BitSet instead of array of booleans.]

ith

Chapter 11

Geometry

♻ alldifferentdirections, convexpolygonarea, cookiecutter, countingtriangles, cranes, glyphrecognition, hittingtargets, hurricanedanger, jabuke, janitortroubles, polygonarea, rafting

[TODO: Keep building above list—grep for geom. Next to look at is robotprotection.]

See also list of formulas.

[TODO: Points, vectors, angles. Degrees/radians. `atan2`. Dot product. Rotation. Vector magnitude, norm (squared), normalize. Perpendicular (generate, test).] [TODO: Cross product in 2D. Signed area (parallelogram, triangle, Heron's formula), polygon area, right/left turn, inside/outside testing.] [TODO: Lines/rays (point + vector). Line intersection. Segment intersection. Closest point on a line/segment. Point/line distance.] [TODO: law of cosines.] [TODO: Convex hull.]

https://codeforces.com/blog/entry/48868

Chapter 12

Miscellaneous

12.1 2D grids

2D grids/arrays (of characters, numbers, booleans...) are a popular feature of many competitive programming problems.

- There is a trick for reading in a grid of characters which can save a bit of coding effort. The "traditional" way to read a grid of characters would be something like:

```
char[][] grid = new char[R][C];
for (int r = 0; r < R; r++) {
    String line = in.nextLine();
    for (int c = 0; c < C; c++) {
        grid[r][c] = line.charAt(c);
    }
}
```

  However, it is possible to assign each row of the 2D array all at once, like so:

```
char[][] grid = new char[R][C];
for (int r = 0; r < R; r++)
    grid[r] = in.nextLine().toCharArray();
```

- In many cases the grid should be thought of as a graph where each cell is a vertex which is connected by edges to its neighbors. Note that in these cases one rarely wants to explicitly construct a different representation of the graph, but simply use the grid itself as an (implicit) graph representation.

- It is often useful to be able to assign a unique number to each cell in the grid, so we can store ID numbers of cells in data structures rather than making some class to represent a pair of a row and column index. The easiest method is to number the first row from 0 to $C-1$ (where C is the number of columns), then the second row C to $2C-1$, and so on.

0	1	2	...	$C-1$
C	$C+1$	$C+2$	...	$2C-1$
$2C$	$2C+1$	$2C+2$	...	$3C-1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$(R-1)C$	$(R-1)C+1$	$(R-1)C+2$	...	$RC-1$

- Using this scheme, to convert between (r,c) pairs and ID numbers n, one can use the formulas

$$(r,c) \mapsto r \cdot C + c \qquad n \mapsto (n/C, n\%C)$$

- To list the four neighbors of a given cell (r, c) to the north, east, south, and west, one can of course simply list the four cases manually, but sometimes this is tedious and error-prone, especially if there is a lot of code to handle each neighbor that needs to be copied four times.

  Instead, one can use the following template. The idea is that (dr, dc) specifies the *offset* from the current cell (r, c) to one of its neighbors; each time through the loop we rotate it counterclockwise by 1/4 turn using the mapping $(dr, dc) \mapsto (-dc, dr)$ (see Geometry (§11, page 61)).

```
1   int dr = 1, dc = 0;  // starting offset of (1,0); nothing special about this choice
2   for (int k = 0; k < 4; k++) {
3       int nr = r + dr, nc = c + dc;
4       // process neighbor (nr, nc)
5
6       int tmp = dr; dr = -dc; dc = tmp;  // rotate offset ccw
7       // to get cw instead, switch the negative sign
8   }
```

12.2 Hexagonal grids

> ⊛ beehouseperimeter, honey, settlers2, beeproblem, honeyheist

Occasionally a problem will involve a 2D grid of tiled hexagons instead of a grid of squares. (Typically such problems involve a story about bees.) They are often not too hard (*e.g.* some kind of straightforward application of Breadth-First Search (§6.3, page 31)) other than the fact that dealing with hexagonal grids can be annoying, unless you know a few tricks for working with them elegantly.

[TODO: Write about hexagonal grids, storage, coordinate systems, etc.] Reference: https://www.redblobgames.com/grids/hexagons/

12.3 Range queries

Suppose we have a 1-indexed array $A[1 \ldots n]$ containing some values, and there is some operation $\oplus$ which takes two values and combines them to produce a new value. Given indices i and j, we want to quickly find the value that results from combining all the values in the range $A[i \ldots j]$, *i.e.* $A[i] \oplus A[i+1] \oplus \cdots \oplus A[j]$.

For example, A could be an array of integers, and $\oplus$ could be max, that is, we want to find the maximum value in the range $A[i \ldots j]$. Likewise $\oplus$ could be sum, or product, or GCD. Or A could be an array of booleans, and we want to find the AND, OR, or XOR of the range $A[i \ldots j]$.

- For this to make sense, the combining operation must typically be *associative*, *i.e.* $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. (This is called a *semigroup*.)

- Sometimes there is also an inverse operation $\ominus$ which "cancels out" the effects of the combining operation, that is, $(a \oplus b) \ominus b = a$ (this is called a *group*). For example, subtraction cancels out addition. On the other hand, there is no operation that can cancel out the effect of taking a maximum.

- If we only need to find the value of combining a *single* range $A[i \ldots j]$, then ignore everything in this section and simply iterate through the interval, combining all the values in $O(n)$ time.

- More typically, we need to do many queries, and $O(n)$ per query is not fast enough. The idea is to preprocess the array into a data structure which allows us to answer queries more quickly, *i.e.* in $O(1)$ or $O(\lg n)$.

- Sometimes we also need to be able to *update* the array in between queries; in this case we need a more sophisticated query data structure that can be quickly updated.

Each of the below subsections outlines one approach to solving this problem; for quick reference, each subsection title says whether an inverse operation is required, how fast queries are, and whether the technique can handle updates.

12.3.1 Prefix scan (inverse required; $O(1)$ queries; no updates)

In a situation where we have an inverse operation and we do not need to update the array, there is a very simple solution. First, make a *prefix scan array* $P[0 \ldots n]$ such that $P[i]$ stores the value that results from combining $A[1 \ldots i]$. ($P[0]$ stores the unique "identity" value $a \ominus a$, *e.g.* zero if the combining operation is sum.) P can be computed in linear time by scanning from left to right; each $P[i] = P[i-1] \oplus A[i]$. Now the value of $A[i \ldots j]$ can be computed in $O(1)$ time as $P[j] \ominus P[i-1]$. That is, $P[j]$ gives us the value of $A[1] \oplus \cdots \oplus A[j]$, and then we cancel $P[i-1] = A[1] \oplus \cdots \oplus A[i-1]$ to leave just $A[i] \oplus \cdots \oplus A[j]$ as desired.

Note that having $P[0]$ store the identity value is not strictly necessary, but it removes the need for a special case. If A is already 0-indexed instead of 1-indexed, then it's probably easier to just put in a special case for looking up the value of $A[0 \ldots j]$ as $P[j]$, without the need for an inverse operation.

For example, suppose we are given an array of 10^5 integers, along with 10^5 pairs (i, j) for which we must output the sum of $A[i \ldots j]$. Simply adding up the values in each range would be too slow. We could solve this with the following code:

```java
import java.util.*;
public class PrefixSum {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // Read array
        int n = in.nextInt();
        int[] A = new int[n+1];
        for (int i = 1; i <= n; i++) {
            A[i] = in.nextInt();
        }

        // Do prefix scan
        int[] P = new int[n+1];
        for (int i = 1; i <= n; i++) {
            P[i] = P[i-1] + A[i];
        }

        // Answer queries
        int Q = in.nextInt();
        for (int q = 0; q < Q; q++) {
            int i = in.nextInt(), j = in.nextInt();
            System.out.println(P[j] - P[i-1]);
        }
    }
}
```

More commonly, a prefix scan is a necessary first step in a more complex solution.

> 🔗 divisible, dvoniz, srednji, subseqhard

12.3.2 Kadane's Algorithm

As an aside, suppose we want to find the subsequence $A[i \ldots j]$ with the *biggest* sum. A brute-force approach is $O(n^3)$: iterate through all (i, j) pairs and find the sum of each subsequence. Using the prefix scan approach, we can cut this down to $O(n^2)$, since we can compute the sums of the $O(n^2)$ possible subsequences in $O(1)$ time each. However, there is an even better $O(n)$ algorithm which is worth knowing, known as *Kadane's Algorithm*.

The basic idea is simple: scan through the array, keeping a running sum in an accumulator, and also keeping track of the biggest total seen. Whenever the running sum drops below zero, reset it to zero. Below

is a sample solution to ⊛ `commercials`. Note that subtracting P from each input is specific to the problem, but the rest is purely Kadane's Algorithm.

```java
import java.util.*;

public class Commercials {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int N = in.nextInt(); int P = in.nextInt();

        int max = 0, sum = 0;
        for (int i = 0; i < N; i++) {
            sum += in.nextInt() - P;
            if (sum < 0) sum = 0;       // or  sum = Math.max(sum, 0);
            if (sum > max) max = sum;   // or  max = Math.max(max, sum);
        }
        System.out.println(max);
    }
}
```

12.3.3   2D prefix scan

[TODO: make pictures]

It is possible to extend the prefix scan idea to two dimensions. Given a 2D array A, we create a parallel 2D array P such that $P[i][j]$ is the result of combining all the entries of A in the rectangle from the upper-left corner to (i, j) inclusive. The simplest way to do this is to compute

$$P[i][j] = A[i][j] + P[i-1][j] + P[i][j-1] - P[i-1][j-1]$$

Including $P[i-1][j]$ and $P[i][j-1]$ double counts all the entries in the rectangle from the upper left to $(i-1, j-1)$ so we have to subtract them.

Given P, to compute the combination of the elements in some rectangle from (a, b) to (c, d), we can compute

$$P[c][d] - P[a-1][d] - P[c][b-1] + P[a-1][b-1]$$

⊛ `prozor` can be solved by brute force, but it's a nice exercise to solve it using the above approach.

12.3.4   Doubling windows (no inverse; $O(1)$ queries; no updates)

[TODO: Include link to discussion in CP3]

12.3.5   Fenwick trees (inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates)

> ⊛ `fenwick`, `supercomputer`, `turbo`, `moviecollection`, `dailydivision`

We can use a *Fenwick tree* to query the range $A[i..j]$ (*i.e.* get the combination of all the values in the range $A[i] \dots A[j]$ according to the combining operation $\oplus$) in $O(\lg n)$ time. We can also dynamically update any entry in the array in $O(\lg n)$ time. If dynamic updates are required and we have an invertible combining operation, a Fenwick tree should definitely be the first choice because the code is quite short. (Segment trees (§12.3.6, page 67) can also handle dynamic updates, and work for any combining operation, even with no inverse, but the required code is a bit longer.)

The code shown here stores `int` values and uses addition as the combining operation, so range queries return the *sum* of all values in the range; but it can be easily modified for any other type of values and any other invertible combining operation: change the type of the array, change the + operation in the `prefix` and `add` methods, change the subtraction in the `range` method, and change the assignment `s = 0` in `prefix` to the identity element instead of zero.

☞ Note that this `FenwickTree` code assumes the underlying array is 1-indexed!

```
1   class FenwickTree {
2       private long[] a;
3       public FenwickTree(int n) { a = new long[n+1]; }
4
5       // A[i] += delta. O(lg n).
6       public void add(int i, long delta) {
7           for (; i < a.length; i += LSB(i)) a[i] += delta;
8       }
9
10      // query [i..j]. O(lg n).
11      public long range(int i, int j) { return prefix(j) - prefix(i-1); }
12
13      private long prefix(int i) {   // query [1..i]. O(lg n).
14          long s = 0; for (; i > 0; i -= LSB(i)) s += a[i]; return s;
15      }
16      private int LSB(int i) { return i & (-i); }
17  }
```

- The constructor creates a `FenwickTree` over an array of all zeros.

- To create a `FenwickTree` over a given 1-indexed array A, simply create a default tree and then loop through the array, calling `ft.add(i, A[i])` for each `i`. This takes $O(n \lg n)$.

- `ft.add(i, delta)` can be used to update the value at a particular index by adding `delta` to it.

- If you want to simply replace the value at index i instead of adding something to it, you could use `ft.add(i, newValue - ft.range(i,i))`.

- `ft.range(i,j)` returns the sum $A[i] + \cdots + A[j]$.

[TODO: Discuss CP3 presentation of Fenwick trees; explain how Fenwick trees work]

12.3.6 Segment trees (no inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates)

[TODO: Segment trees.]

12.4 Cycle finding

[TODO: Floyd's algorithm, Brent's algorithm]

Chapter 13

Formulas

- **Ceiling division** (🌐 `soylent`, `wordcloud`, `amultiplicationgame`). If `p` and `q` are positive values of type `int` or `long`, then `p/q` computes $\lfloor p/q \rfloor$, the quotient (rounded down). If you want the quotient rounded *up*, that is, $\lceil p/q \rceil$, compute

  (p + q - 1) / q.

  Note that `-((-p)/q)` does not work in Java since Java truncates the result of integer division towards zero, instead of always taking the floor.

- **Derangements** (🌐 `secretsanta`). The number of permutations of n objects such that no object is left in its original place is

  $$!n = n{\cdot}!(n-1) + (-1)^n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!} = \left[\frac{n!}{e} \right],$$

  where $!1 = 0$, and $[x]$ denotes the closest integer to x. The first few values of $!n$ are

  $$0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961.$$

- **Heron's Formula**. The area of a triangle with side lengths a, b, c is

  $$\sqrt{s(s-a)(s-b)(s-c)} \qquad \text{where } s = (a+b+c)/2.$$

- **Brahmagupta's Formula** (🌐 `janitortroubles`). The area of a quadrilateral with side lengths a, b, c, and d, with all vertices lying on a common circle, is

  $$\sqrt{s(s-a)(s-b)(s-c)(s-d)} \qquad \text{where } s = (a+b+c+d)/2.$$

  This is also the maximum possible area of a quadrilateral with the given side lengths.

Chapter 14

Advanced topics

This is a list of advanced topics that may eventually be included in this document, but for now you can go read up on them if you are interested! (And then of course write up what you have learned for inclusion in this document.)

- Chinese Remainder Theorem (⬤ heliocentric, generalchineseremainder)

- Divisors of $n!$ (⬤ factovisors)

- Gauss-Jordan elimination (*i.e.* row reduction *i.e.* solving linear systems) (⬤ primonimo)

- Exact Set Cover with Algorithm X/dancing links (⬤ programmingteamselection)

- Matrix powers

  > ⬤ diceandladders, driving, linearrecurrence, mortgage, overlappingmaps, squawk, timing

- Markov chains

  > ⬤ lostinthewoods, gruesomecave

- Min cost max flow

- Max flow with minimum and maximum capacities

- Discrete logarithms with baby step/giant step (⬤ discretelogging)

- Faster primality testing with Miller-Rabin (*e.g.* testing with $a = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41$ makes it deterministic).

- Divide & conquer algorithm for counting inversions.

  > ⬤ excursion, froshweek, ultraquicksort

- 2-SAT

- SAT with DPLL?

- LCA queries: Tarjan's OLCA; via RMQ; binary lifting (⬤ tourists)

- Convolutions with FFT/NTT (⬤ tiles, aplusb, kinversions)

Appendix A

Reference

A.1 Primes

All primes up to 1000:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397
401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499
503 509 521 523 541 547 557 563 569 571 577 587 593 599
601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787 797
809 811 821 823 827 829 839 853 857 859 863 877 881 883 887
907 911 919 929 937 941 947 953 967 971 977 983 991 997

A.2 Pascal's Triangle

```
            1
         1     1
      1     2     1
    1     3     3     1
  1    4     6     4    1
1    5    10    10    5    1
```

Appendix B

Resources

Some good resources for further learning/reference:

- Problems/online judges
  - Of course, Open Kattis has a collection of over 1000 great problems ranging from trivial to very difficult.
  - The UVa Online Judge has been around much longer than Kattis and also has a huge collection of problems, mostly disjoint from those on Kattis.
  - The CP3 website has a Methods to Solve page with a huge annotated list of problems from Kattis and UVa, grouped by topic (corresponding to sections in CP3) with small hints for each one.

- Books
  - Competitive Programming, 3rd edition (aka CP3) by Steven and Felix Halim is amazing. Anyone serious about competitive programming should get a copy.
  - Programming Challenges by Skiena and Revilla is also good.

- Reference
  - [TODO: Geeksforgeeks]
  - [TODO: Topcoder]
  - [TODO: Codeforces]
  - [TODO: cp-algorithms.com]