

Hendrix Programming Team Reference

February 20, 2019



Contents

1	Limits	5
2	Java Reference	7
2.1	Template	7
2.2	Scanner	7
2.3	String/StringBuilder	8
2.4	Arrays	9
2.5	ArrayList	9
2.6	Stack	10
2.7	Queue/ArrayDeque	10
2.8	Comparator	11
2.9	PriorityQueue	11
2.10	Set	11
2.11	Map	12
2.12	BigInteger	12
2.13	Sorting	12
2.14	BitSet	13
2.15	Fast I/O	13
3	Python Reference	15
3.1	Template	15
4	Data Structures	17
4.1	Bag	17
4.2	Union-find	18
4.3	Tries	19
4.4	Adjustable priority queue	19
4.5	Segment trees and Fenwick trees	20
5	Search	21
5.1	Complete search	21
5.2	Binary search	21
5.3	Ternary search	22
6	Graphs	23
6.1	Graph basics	23
6.2	Graph representation	23
6.3	BFS	23
6.4	DFS, SCCs, topological sorting	24
6.5	Single-source shortest paths (Dijkstra)	24
6.6	All-pairs shortest paths (Floyd-Warshall)	24
6.7	Min spanning trees (Kruskal)	24
6.8	Max flow	24

7	Dynamic Programming	29
8	Strings	31
8.1	Z-algorithm	31
8.2	Suffix arrays	31
9	Mathematics	33
9.1	GCD/Euclidean Algorithm	33
9.2	Rational numbers	33
9.3	Modular arithmetic	34
9.4	Primes and factorization	36
9.4.1	Trial division	36
9.4.2	Sieving	36
9.5	Divisors and Euler's Totient Function	38
9.6	Factorial	38
9.7	Combinatorics	38
10	Bit Tricks	41
11	Geometry	43
12	Miscellaneous	45
12.1	2D grids	45
12.2	Range queries	46
12.2.1	Prefix scan (inverse required; $O(1)$ queries; no updates)	46
12.2.2	Kadane's Algorithm	47
12.2.3	2D prefix scan	47
12.2.4	Doubling windows (no inverse; $O(1)$ queries; no updates)	48
12.2.5	Fenwick trees (inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates)	48
12.2.6	Segment trees (no inverse required; $O(\lg n)$ queries; $O(\lg n)$ updates)	49
13	Formulas	51
14	Advanced topics	53
15	Resources	55

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 (§6.6, page 24): 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 (§6.5, page 24), 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 23), so you can run a BFS/DFS (§6.4, page 24) 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 38)
25	$O(2^n)$	Generating all subsets (§10, page 41)
100	$O(n^4)$	Some brute force algorithms
400	$O(n^3)$	Floyd-Warshall (§6.6, page 24)
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 `ints` 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](#) (§2.12, page 12) or just use [Python](#) (§3, page 15); see also [Combinatorics](#) (§9.7, page 38).

 different


Chapter 2


Java Reference

2.1 Template

```
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** (§2.15, page 13).

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

```
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 String/StringBuilder

 [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*.
- **toArray()** 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).

```

1  import java.util.Scanner;
2
3  public class sumOfTheOthers {
4      public static void main(String[] args) {
5          Scanner in = new Scanner(System.in);
6          while (in.hasNext()) {
7              String[] s = in.nextLine().split(" ");
8              int out = 0;
9              for (String i : s) {


```



```

10         out += Integer.parseInt(i);
11     }
12     System.out.println(out / 2);
13 }
14 in.close();
15 }
16 }

```

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.

[TODO: Example of using **StringBuilder**.]

 [itsasecret](#)


2.4 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 $1 \dots n$. 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 \(§2.13, page 12\)](#).
- 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 \(§5.2, page 21\)](#).
- Other methods from the  **Arrays** class may also occasionally be useful.

2.5 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 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** ([§2.7, page 10](#)).)





```

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

```

2.6 Stack




 [backspace](#), [islands](#), [pairingsocks](#), [reservoir](#), [restaurant](#), [symmetricorder](#), [throws](#), [zgrade](#)

 **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 24) as well as dealing with parentheses, or nesting more generally ( [pairingsocks](#),  [islands](#),  [reservoir](#)).

2.7 Queue/ArrayDeque

 [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 BFS (§6.3, page 23) and in simulations of various sorts (for examples of the latter, see the selection of problems above).



[TODO: Sample code using Queue/ArrayDeque]

2.8 Comparator


[TODO: Constructing Comparators via lambda; constructing via things like `comparing`, `thenComparing`, `Collections.reverseOrder()`. Use for sorting, PQs, TreeSet/Map.]

2.9 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 (§6.5, page 24), 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 *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]);
```


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 (§6.5, page 24). 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](#) (§4.4, page 19).





2.10 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 `Sets` 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. However, it has several other advantages over a **HashSet**:
 - 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 typically much 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!

2.11 Map

 **awkwardparty**, **administrativeproblems**, **snowflakes**, **pizzahawaii**

[TODO: HashMap, TreeMap] [TODO: Iterating over keys, values, both (MapEntry)] [TODO: Note for 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. Much easier to use custom classes as keys in a TreeMap (just implement Comparable) than in a HashMap (implement hashCode and equals).]

2.12 BigInteger

[TODO: Examples. Useful methods, constructors (gcd, mod, base conversion!).]

 **basicremains**

2.13 Sorting

[TODO: Basic template for implementing Comparable] [TODO: Arrays.sort, Collections.sort] [TODO: Sorting with a custom Comparator]

Explicit sorting algorithms


[TODO: Include code for basic sorting implementations (in case it's useful to code them up explicitly so they can be enhanced with extra info): bubble sort, insertion sort?, mergesort?, quicksort?]

Radix sort

[TODO: radix sort]


2.14 BitSet


[TODO: Basic examples of BitSet use.]

 [primesieve](#)

2.15 Fast I/O

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.

 [avoidland, cd](#)

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

```

1  /* Example usage:
2  *
3  * Kattio io = new Kattio(System.in, System.out);
4  *
5  * while (io.hasMoreTokens()) {
6  *     int n = io.getInt();
7  *     double d = io.getDouble();
8  *     double ans = d*n;
9  *
10 *     io.println("Answer: " + ans);
11 * }
12 *
13 * io.flush();    // DON'T FORGET THIS LINE!
14 */
15
16 import java.util.*;
17 import java.io.*;
18
19 class Kattio extends PrintWriter {
20     public Kattio(InputStream i) {
21         super(new BufferedOutputStream(System.out));
22         r = new BufferedReader(new InputStreamReader(i));
23     }
24     public Kattio(InputStream i, OutputStream o) {
25         super(new BufferedOutputStream(o));
26         r = new BufferedReader(new InputStreamReader(i));

```

```
27     }
28
29     public boolean hasMoreTokens() {
30         return peekToken() != null;
31     }
32
33     public int getInt() {
34         return Integer.parseInt(nextToken());
35     }
36
37     public double getDouble() {
38         return Double.parseDouble(nextToken());
39     }
40
41     public long getLong() {
42         return Long.parseLong(nextToken());
43     }
44
45     public String getWord() {
46         return nextToken();
47     }
48
49     private BufferedReader r;
50     private String line;
51     private StringTokenizer st;
52     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:

```
1  import sys
2
3  if __name__ == '__main__':
4
5      n = int(sys.stdin.readline()) # Read an int on a line by itself
6      for _ in range(n):           # Do something n times
7
8          # Read all the ints on a line into a list
9          xs = map(int, sys.stdin.readline().split())
10
11         # Read a known number of ints into variables
12         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 Bag

 `cookieselection`

A *bag* 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 \(§2.11, page 12\)](#), 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.

```
1 import java.util.*;
2
3 public class TreeBag<E extends Comparable<E>> implements Iterable<E> {
4     private TreeMap<E, Integer> map;
5     private int totalSize;
6     public TreeBag() { map = new TreeMap<>(); }
7     public void add(E e) {
8         if (!map.containsKey(e)) map.put(e, 0);
9         map.put(e, map.get(e) + 1);
10        totalSize++;
11    }
12    public void remove(E e) {
13        if (map.containsKey(e)) {
14            if (map.get(e) == 1) map.remove(e);
15            else map.put(e, map.get(e) - 1);
16            totalSize--;
17        }
18    }
19    public int size() { return totalSize; }
20    public boolean isEmpty() { return map.isEmpty(); }
21    public int count(E e) { return map.containsKey(e) ? map.get(e) : 0; }
22    public E first() { return map.firstKey(); }
23    public E last() { return map.lastKey(); }
24    public E pollFirst() { E e = first(); remove(e); return e; }
25    public E pollLast() { E e = last(); remove(e); return e; }
26    public Iterator<E> iterator() {
```

```

27     return new Iterator<E>() {
28         Iterator<E> it = map.keySet().iterator();
29         E cur; int count = 0;
30         public boolean hasNext() { return it.hasNext() || count > 0; }
31         public E next() {
32             if (count == 0) { cur = it.next(); count = map.get(cur); }
33             count--; return cur;
34         }
35     };
36 }
37 }

```

4.2 Union-find

 [forestfires](#), [kastenlauf](#), [ladice](#), [numbersetseasy](#), [unionfind](#), [virtualfriends](#), [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 (§6.7, page 24), and can also be useful on its own (see the above Kattis problems for examples). `find` and `union` both take essentially constant amortized time.

```

1  class UnionFind {
2      private byte[] r; private int[] p; // rank, parent
3
4      // Make a new union-find structure with n items in singleton sets,
5      // numbered 0 .. n-1 .
6      public UnionFind(int n) {
7          r = new byte[n]; p = new int[n];
8          for (int i = 0; i < n; i++) {
9              r[i] = 0; p[i] = i;
10         }
11     }
12
13     // Return the root of the set containing v, with path compression. O(1).
14     // Test whether u and v are in the same set with find(u) == find(v).
15     public int find(int v) {
16         return v == p[v] ? v : (p[v] = find(p[v]));
17     }
18
19     // Union the sets containing u and v. O(1).
20     public void union(int u, int v) {
21         int ru = find(u), rv = find(v);
22         if (ru != rv) {
23             if (r[ru] > r[rv]) p[rv] = ru;
24             else if (r[rv] > r[ru]) p[ru] = rv;
25             else { p[ru] = rv; r[rv]++; }
26         }
27     }
28 }

```

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`).

4.3 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]


```

1  class Trie<K,V> {
2      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 (§2.13, page 13), 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 is helpful to think about a problem in terms of a trie, but never actually implement/materialize the trie at all ( `herkabe`).

4.4 Adjustable priority queue

 `flowerytrails`

As discussed in `PriorityQueue` (§2.9, page 11), 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 (§6.5, page 24), 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;
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 }

```


4.5 Segment trees and Fenwick trees

See [Range queries \(§12.2, page 46\)](#).

Chapter 5

Search

5.1 Complete search

 [bing](#), [classpicture](#), [coloring](#), [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 \(§10, page 41\)](#). 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.

[TODO: Example code?]


Binary search on an integer interval

 [guess](#), [freeweights](#), [inversefactorial](#), [reservoir](#)

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

[TODO: When searching over the integers, make sure you're very explicit about whether the lo and hi bounds are included or excluded. Probably easiest to include.]

5.3 Ternary search

 [brocard](#), [euclideanasp](#), [infinitieslides](#), [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 $m_R = (L + 2R)/3$.

- 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]

Chapter 6

Graphs


6.1 Graph basics

[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 BFS

 brexit, collapse, 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).
- 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 24)).

[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, SCCs, topological sorting

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

6.5 Single-source shortest paths (Dijkstra)

6.6 All-pairs shortest paths (Floyd-Warshall)

6.7 Min spanning trees (Kruskal)

 **drivingrange**, **islandhopping**, **jurassicjigsaw**, **lostmap**, **minspantree**, **treehouses**

6.8 Max flow

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


```

1  import java.util.*;
2
3  public class horrorList {
4      public static void main(String[] args) {
5          Scanner in = new Scanner(System.in);
6          int movie_num = in.nextInt();
7          int horror_num = in.nextInt();
8          int linked_num = in.nextInt();
9          int[] scores = new int[movie_num];
10         Queue<Integer> hi = new ArrayDeque<Integer>();
11         for(int i = 0; i < movie_num; i++) {
12             scores[i] = Integer.MAX_VALUE;
13         }
14         for(int i = 0; i < horror_num; i++) {
15             int a = in.nextInt();
16             scores[a] = 0;
17             hi.add(a);
18         }
19         HashMap<Integer, HashSet<Integer>> graph = new HashMap<>();
20         int j = 0;
21         while (j < linked_num) {
22             int a = in.nextInt();
23             int b = in.nextInt();
24             if(! graph.containsKey(a)) graph.put(a, new HashSet<Integer>());
25             if(! graph.containsKey(b)) graph.put(b, new HashSet<Integer>());
26             graph.get(b).add(a);
27             graph.get(a).add(b);
28             j++;
29         }
30         while(! hi.isEmpty()) {
31             int temp = hi.remove();
32             if(graph.containsKey(temp)) {
33                 for(int i: graph.get(temp)) {
34                     if( scores[i] == Integer.MAX_VALUE) {
35                         scores[i] = scores[temp] + 1;
36                         hi.add(i);
37                     }
38                 }
39             }
40         }
41         int output = 0;
42         int max = 0;
43         for(int i = 0; i < movie_num; i++) {
44             if( scores[i] > max) {
45                 max = scores[i];
46                 output = i;
47             }
48         }
49         System.out.println(output);
50     }
51 }

```


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

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 outgoing flows at every vertex other than the source or sink. Flow networks can be used to model a wide variety of problems.

[TODO: Enumerate a few problem types: item assignment; max bipartite matching; min cut]

[TODO: choose directed/undirected edges carefully!]

[TODO: Requires vertices $0 \dots n - 1$: either carefully keep track of which numbers are for which vertices, or use lookup tables]

 [copsandrobbbers](#), [escapeplan](#), [gopher2](#), [guardianofdecency](#), [marblestree](#), [maxflow](#), [mincut](#), [paintball](#), [waif](#)

Dinitz' Algorithm 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 $O(E\sqrt{V})$.

```

1  class FlowNetwork {
2      private static final int INF = ~(1<<31);
3      int[] level;
4      boolean[] pruned;
5      HashMap<Integer, HashMap<Integer, Edge>> adj;
6
7      public FlowNetwork(int n) {
8          level = new int[n];
9          pruned = new boolean[n];
10         adj = new HashMap<>();
11
12         for (int i = 0; i < n; i++)
13             adj.put(i, new HashMap<>());
14     }
15
16     public void addDirEdge(int u, int v, long cap) {
17         if (adj.get(u).containsKey(v)) {
18             adj.get(u).get(v).capacity = cap;
19         } else {
20             Edge e = new Edge(u,v,cap);
21             Edge r = new Edge(v,u,0);
22             e.setRev(r);
23             adj.get(u).put(v, e);
24             adj.get(v).put(u, r);
25         }
26     }
27
28     // Add an UNdirected edge u<->v with a given capacity
29     public void addEdge(int u, int v, long cap) {
30         Edge e = new Edge(u,v,cap);
31         Edge r = new Edge(v,u,cap);
32         e.setRev(r);
33         adj.get(u).put(v, e);
34         adj.get(v).put(u, r);
35     }
36
37     public long maxFlow(int s, int t) {
38         if (s == t) return INF;

```

```

39     else {
40         long totalFlow = 0;
41         while (bfs(s,t)) totalFlow += sendFlow(s,t);
42         return totalFlow;
43     }
44 }
45
46 private long sendFlow(int s, int t) {
47     for (int i = 0; i < pruned.length; i++)
48         pruned[i] = false;
49     return sendFlowR(s, t, INF);
50 }
51
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]

[TODO: Variants: Multiple sources/sinks? Use trick of adding a new source/sink with infinite capacity edges. Vertex capacities? Turn each vertex into a new edge.]

Chapter 7

Dynamic Programming

[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-increasing-subsequence-using-dp>]

Chapter 8

Strings

8.1 Z-algorithm


8.2 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](#)

```
1 public class GCD {
2     public static long gcd(long a, long b) {
3         return b == 0 ? a : gcd(b, a % b);
4     }
5
6     public static EGCD egcd(long a, long b) {
7         if (b == 0) return new EGCD(a, 1, 0);
8         EGCD e = egcd(b, a % b);
9         return new EGCD(e.g, e.y, e.x - a / b * e.y);
10    }
11 }
12
13 class EGCD { // For storing result of egcd function
14     long g, x, y;
15     public EGCD(long _g, long _x, long _y) {
16         g = _g; x = _x; y = _y;
17     }
18 }
```

9.2 Rational numbers

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.


 [jointattack](#), [prosjek](#), [prsteni](#), [rationalarithmetic](#), [wheels](#), [zipfsong](#)


```

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

 [crackmap](#), [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 $-n \dots n$; 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 (§2.12, page 12). 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 33). 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 $1 \dots n$. The code below uses a `BitSet` (§2.14, page 13), which uses less memory than an array of `booleans`. 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.)

```

1  import java.util.*;
2
3  public class PrimeSieve {
4      BitSet prime;
5      public PrimeSieve(int MAX) {
6          prime = new BitSet(MAX+1);
7          prime.set(2,MAX+1,true);           // initialize all to true
8          for (int p = 2; p*p <= MAX; p++)    // iterate up to  $\sqrt{\text{MAX}}$ 
9              if (prime.get(p))              // found a prime p
10                 for (int m = p*p; m <= MAX; m += p) // cross out multiples of p
11                     prime.set(m,false);
12      }
13      public boolean isPrime(int n) { // Once sieve is built, test primality in  $O(1)$ 
14          return prime.get(n);
15      }
16  }
```

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.

```


1  import java.util.*;
2
3  public class FactorSieve {
4      int[] smallest;
5      public FactorSieve(int MAX) {
6          smallest = new int[MAX+1];
7          smallest[1] = 1;
8          int p = 2;
9          for (; p*p <= MAX; p++) { // Sieve up to  $\sqrt{\text{MAX}}$ 
10              if (smallest[p] == 0) {
11                  smallest[p] = p;
12                  for (int m = p*p; m <= MAX; m += p)
13                      if (smallest[m] == 0) smallest[m] = p;
14              }
15          }
16          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

[TODO: Computing factorials; size using logs, etc]

9.7 Combinatorics

 insert, anagramcounting, nine, secretsanta, kingscolors, howmanyzeros

[TODO: Basic principles of combinatorics. Code for computing binomial coefficients. Multinomial coefficients.]

[TODO: mod $10^9 + 7$.]


☞ Remember to use `long` if you need an answer $\text{mod}(10^9 + 7)$ (which would fit in an `int`) but computing the answer requires *multiplying* $\text{mod}(10^9 + 7)$.

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

[TODO: PIE?]

Chapter 10

Bit Tricks

 [bits](#), [classpicture](#), [flipfive](#), [font](#), [gepetto](#), [hypercube](#), [mazemakers](#), [pagelayout](#), [pebblesolitaire](#), [satisfiability](#), [turningtrominos](#)

`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,

$$974_{10} = \begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & & \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & & \end{array}$$

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 = 1112` 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 .

Bit strings for states


[TODO: Using bitstrings to compactly represent sets/states/adjacent neighbors. Building a set, iterating through all subsets with counter.]

[TODO: LSB, LSZ, MSB, pop count, iterating through sub-subsets]

[TODO: BitSet instead of array of booleans.]

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: Convex hull.]

Chapter 12

Miscellaneous

12.1 2D grids

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

- 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

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

- 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 \(§11, page 43\)](#)).

```
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 Range queries

Suppose we have a 1-indexed array $A[1 \dots n]$ 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 $A[i \dots j]$, i.e. $A[i] \oplus A[i+1] \oplus \dots \oplus A[j]$.

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 $A[i \dots j]$. 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 $A[i \dots j]$.

- 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 $A[i \dots j]$, 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.

12.2.1 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* $P[0 \dots n]$ such that $P[i]$ stores the value that results from combining $A[1 \dots i]$. ($P[0]$ 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 $A[i \dots j]$ can be computed in $O(1)$ time as $P[j] \ominus P[i-1]$. That is, $P[j]$ gives us the value of $A[1] \oplus \dots \oplus A[j]$, and then we cancel $P[i-1] = A[1] \oplus \dots \oplus A[i-1]$ to leave just $A[i] \oplus \dots \oplus A[j]$ 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 $A[0 \dots j]$ 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 $A[i \dots j]$. Simply adding up the values in each range would be too slow. We could solve this with the following code:

```

1  import java.util.*;
2  public class PrefixSum {
3      public static void main(String[] args) {
4          Scanner in = new Scanner(System.in);
5
6          // Read array
7          int n = in.nextInt();
8          int[] A = new int[n+1];
9          for (int i = 1; i <= n; i++) {
10             A[i] = in.nextInt();
11         }


```

```

12
13     // Do prefix scan
14     int[] P = new int[n+1];
15     for (int i = 1; i <= n; i++) {
16         P[i] = P[i-1] + A[i];
17     }
18
19     // Answer queries
20     int Q = in.nextInt();
21     for (int q = 0; q < Q; q++) {
22         int i = in.nextInt(), j = in.nextInt();
23         System.out.println(P[j] - P[i-1]);
24     }
25 }
26


```

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

 [divisible](#), [dvoniz](#), [srednji](#), [subseqhard](#)

12.2.2 Kadane's Algorithm

As an aside, suppose we want to find the subsequence $A[i \dots j]$ 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.

```

1  import java.util.*;
2
3  public class Commercials {
4      public static void main(String[] args) {
5          Scanner in = new Scanner(System.in);
6          int N = in.nextInt(); int P = in.nextInt();
7
8          int max = 0, sum = 0;
9          for (int i = 0; i < N; i++) {
10             sum += in.nextInt() - P;
11             if (sum < 0) sum = 0;           // or sum = Math.max(sum, 0);
12             if (sum > max) max = sum;     // or max = Math.max(max, sum);
13         }
14         System.out.println(max);
15     }
16 }

```

12.2.3 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.

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

[TODO: Include link to discussion in CP3]

12.2.5 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 (§12.2.6, page 49) 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 $A[i] + \dots + A[j]$.


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

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

[TODO: Segment trees.]

Chapter 13

Formulas

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


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

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

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

- **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

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


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.

- Chinese Remainder Theorem


 [heliocentric](#), [generalchineseremainder](#)

- Exact Set Cover with Algorithm X/dancing links ( [programmingteamselection](#))
- Matrix powers

 [diceandladders](#), [driving](#), [linearrecurrence](#), [mortgage](#), [overlappingmaps](#), [squawk](#), [timing](#)

- 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](#)

- 2-SAT
- LCA queries: Tarjan's OLCA; via RMQ; binary lifting ( [tourists](#))

Chapter 15

Resources

[TODO: methodstosolve] [TODO: UVa] [TODO: CP3] [TODO: Geeksforgeeks, topcoder, codeforces] [TODO: cp-algorithms.com]