Hendrix Programming Team Reference

January 3, 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 (§5.6, page 17): 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 (§5.5, page 17), 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 (§5.1, page 17), so you can run a DFS (§5.4, page 17) to find a 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 (§8.7, page 30)
25	$O(2^n)$	Generating all subsets (§9, page 33)
100	$O(n^4)$	Some brute force algorithms
400	$O(n^3)$	Floyd-Warshall (§5.6, page 17)
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 (§2.11, page 9) or just use Python (§12, page 43); see Combinatorics (§8.7, page 30).

Chapter 2

Java Reference

2.1 Template

```java
// *Don't* include a package declaration!

import java.util.*;
import java.math.*;

public class ClassName {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // Solution code here

        System.out.println(answer);
    }
}
```

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 (§2.14, page 9).

```java
import java.util.*;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // All these read a single token (ignore leading whitespace,
        // then read up until but not including the next whitespace)
        String s = in.next();
        int    n = in.nextInt();
        long   l = in.nextLong();
        double d = in.nextDouble();

        // WARNING!! A previous call to nextXXX will read up to a
        // newline character but leave it unconsumed in the input, so
        // the next call to nextLine() will just read that newline and
        // return an empty string!
        in.nextLine();   // throw away the empty line to get ready for the next

        // Read a whole line up to the next newline character.
        // Consumes the newline but does not include it in the
        // returned String.
        String line = in.nextLine();

        // Read until end of input
        while (in.hasNext()) {
            line = in.nextLine();
        }
    }
}
```

2.3 String

The `String` type can be used in Java to represent sequences of characters.

[TODO: Useful `String` methods: concatenation, substring, charAt, . . . ?] [TODO: Converting between `String` and `char[]`, advantages and disadvantages of each]

`battlesimulation`, `bing`, `connectthedots`, `itsasecret`, `shiritori`, `suffixarrayreconstruction`

2.4 Arrays

[TODO: Basic array template/examples.] [TODO: Useful `Arrays` methods: sort, copyOf, copyOfRange, fill, binarySearch]

`falcondive`, `freefood`, `traveltheskies`

2.5 ArrayList

[TODO: Basic template and examples. Useful `ArrayList` methods: `add`, `get`, `set`. Advantages/disadvantages compared to arrays.]

2.6 Stack

[TODO: Example using `Stack` class.] [TODO: Mention balanced parentheses, DFS]

`backspace`, `islands`, `pairingsocks`, `reservoir`, `restaurant`, `symmetricorder`, `throwns`, `zagrade`

2.7 Queue/Deque

[TODO: `Queue` interface, `ArrayDeque` class]

`brexit`, `coconut`, `ferryloading4`, `integerlists`, `shuffling`

2.8 PriorityQueue

[TODO: Examples of using `PriorityQueue`. `Collections.reverseOrder()` for max PQ. Show how to construct with custom `Comparator`, eg. using lambda notation; things like `Double.compare`, etc.] [TODO: Note lack of decreaseKey operation (Dijkstra), use remove + add, not as fast]

`bank`, `ferryloading3`, `guessthedatastructure`, `knigsoftheforest`, `vegetables`

2.9 Set

[TODO: HashSet, TreeSet]

2.10 Map

[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).]

`awkwardparty`, `administrativeproblems`, `snowflakes`

2.11 BigInteger

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

`basicremains`

2.12 Sorting

[TODO: Basic template for implementing Comparable] [TODO: Arrays.sort, Collections.sort] [TODO: Sorting with a custom `Comparator`] [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): insertion sort, mergesort, quicksort)]

2.13 BitSet

[TODO: Basic examples of BitSet use.]

`primesieve`

2.14 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!

```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;

    private String peekToken() {
        if (token == null)
            try {
                while (st == null || !st.hasMoreTokens()) {
                    line = r.readLine();
                    if (line == null) return null;
                    st = new StringTokenizer(line);
                }
                token = st.nextToken();
            } catch (IOException e) { }
        return token;
    }

    private String nextToken() {
        String ans = peekToken();
        token = null;
        return ans;
    }
}
```

[TODO: Add `getLine()` method]

Chapter 3

Data Structures

3.1 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.

[TODO: hashset and treeset-based bag implementations]

3.2 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 (§5.7, page 17), and can also be useful on its own (see the above Kattis problems for examples). `find` and `union` both take essentially constant amortized time.

```java
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`).

3.3 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.

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

    public Trie() {
        children = new HashMap<>(); mark = false;
    }
    public void add(String s) { addR(s); }
    public void addR(String s) {
        if (s.equals("")) mark = true;
        else ensureChild(s.charAt(0)).addR(s.substring(1));
    }
    public Trie getChild(Character c) { return children.get(c); }
    public Trie ensureChild(Character c) {
        Trie t = getChild(c);
        if (t == null) {
            t = new Trie();
            children.put(c, t);
        }
        return t;
    }
}
```

3.4 Segment trees and Fenwick trees

See Range queries (§11.2, page 38).

Chapter 4

Search

4.1 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 (§9, page 33). 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.

4.2 Binary search

`bottles`, `cheese`, `guess`, `insert`, `speed`, `suspensionbridges`, `tetration`

[TODO: Binary search on an array; binary search on unbounded function on the integers; binary search on real interval] [TODO: Point out `Arrays.binarySearch`]

[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.]

4.3 Ternary search

`brocard`, `euclideantsp`, `infiniteslides`, `janitortroubles`

[TODO: Write about ternary search.]

Chapter 5

Graphs

5.1 Graph basics

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

chopwood

5.2 Graph representation

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

5.3 BFS

[TODO: Code for BFS with level labelling, parent map.]

brexit

5.4 DFS, SCCs, topological sorting

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

5.5 Single-source shortest paths (Dijkstra)

5.6 All-pairs shortest paths (Floyd-Warshall)

5.7 Min spanning trees (Kruskal)

drivingrange, islandhopping, jurassicjigsaw, lostmap, minspantree, treehouses

5.8 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 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]

copsandrobbers, 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 $O(E\sqrt{V})$.

```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);
    }

    private long sendFlowR(int s, int t, long available) {
        if (s == t) return available;

        long sent = 0;
        for (Edge e : adj.get(s).values()) {
            if (e.remaining() > 0 && !pruned[e.to] && level[e.to] == level[s] + 1) {
                long flow = sendFlowR(e.to, t, Math.min(available, e.remaining()));
                available -= flow; sent += flow;
                e.flow += flow; e.rev.flow -= flow;
                if (available == 0) break;
            }
        }
        if (sent == 0) pruned[s] = true;
        return sent;
    }

    private boolean bfs(int s, int t) {
        for (int i = 0; i < level.length; i++) level[i] = -1;

        Queue<Integer> q = new ArrayDeque<>();
        q.add(s); level[s] = 0;
        while (!q.isEmpty()) {
            int cur = q.remove();
            for (Edge e : adj.get(cur).values()) {
                if (e.remaining() > 0 && level[e.to] == -1) {
                    level[e.to] = level[cur]+1;
                    q.add(e.to);
                }
            }
        }
        return level[t] >= 0;
    }
}

class Edge {
    int from, to;
    long capacity, flow;
    Edge rev;
    public Edge(int from, int to, long cap) {
        this.from = from; this.to = to; this.capacity = cap; this.flow = 0;
    }
    public void setRev(Edge rev) { this.rev = rev; rev.rev = this; }
    public long remaining() { return capacity - flow; }
}
```

[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 6

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-usin

Chapter 7

Strings

7.1 Z-algorithm

7.2 Suffix arrays

Chapter 8

Mathematics

8.1 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`

```
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;
    }
}
```

8.2 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`

```java
class Rational implements Comparable<Rational> {
    long n, d;
    public Rational(long _n, long _d) {
        n = _n; d = _d;
        if (d < 0) { n = -n; d = -d; }
        long g = gcd(Math.abs(n),d); n /= g; d /= g;
    }
    private long gcd(long a, long b) {
        return b == 0 ? a : gcd(b, a % b);
    }
    public Rational(long n) { this(n,1); }

    public Rational plus(Rational other) {
        return new Rational(n * other.d + other.n * d, d * other.d);
    }
    public Rational minus(Rational other) {
        return new Rational(n * other.d - other.n * d, d * other.d);
    }
    public Rational negate() {
        return new Rational(-n, d);
    }
    public Rational times(Rational other) {
        return new Rational(n * other.n, d * other.d);
    }
    public Rational divide(Rational other) {
        return new Rational(n * other.d, d * other.n);
    }
    public boolean equals(Object otherObj) {
        Rational other = (Rational)otherObj;
        return (n == other.n) && (d == other.d);
    }
    public int compareTo(Rational r) {
        long diff = n * r.d - d * r.n;
        if (diff < 0) return -1;
        else if (diff > 0) return 1;
        else return 0;
    }
    public String toString() {
        return d == 1 ? ("" + n) : (n + "/" + d);
    }
}
```

8.3 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 $-n \dots n$; 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 (§2.11, page 9). 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`.

```java
public static long modexp(long b, long e, long m) {
    long res = 1;
    while (e > 0) {
        if ((e & 1) == 1) res = (res * b) % m; // include current power of b?
        b = (b * b) % m;                       // square to get next power of b
        e >>= 1;                               // shift out rightmost bit of e
    }
    return res;
}
```

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 (§8.1, page 25). 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.

8.4 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.

8.4.1 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), they tend to be probabilistic and hence inappropriate for a contest environment. Hence, it is highly unlikely that a contest would ever require anything more sophisticated than divisibility testing. 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.

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

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.

```java
public static ArrayList<Integer> factor(int n) {
    ArrayList<Integer> factors = new ArrayList<>();
    while ((n & 1) == 0) { factors.add(2); n >>= 1; }  // get factors of 2
    int d = 3;                  // get odd factors
    while (d*d <= n) {          // O(√n)
        if (n % d == 0) {
            factors.add(d);     // found a factor
            n /= d;
        } else {                // try next odd divisor
            d += 2;
        }
    }
    if (n != 1) factors.add(n);  // don't forget final prime
    return factors;
}
```

8.4.2 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` (§2.13, page 9), 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.)

```
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.

```
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
            if (smallest[p] == 0)
                smallest[p] = p;
    }

    public boolean isPrime(int n) {
        return n > 1 && smallest[n] == n;
    }

    public ArrayList<Integer> factor(int n) {
        ArrayList<Integer> factors = new ArrayList<>();
        while ((n & 1) == 0) { factors.add(2); n >>= 1; }
        int d = 3;
        // Pull out factors until n is small enough to look up
        while (d*d <= n && n >= smallest.length) {
            if (n % d == 0) {
                factors.add(d);
                n /= d;
            } else {
                d += 2;
            }
        }
        // Now just look up remaining factors in the table
        if (n < smallest.length) {
            while (smallest[n] != n) {
                factors.add(smallest[n]);
                n /= smallest[n];
            }
        }
        if (n != 1) factors.add(n);
        return factors;
    }
}
```

8.5 Divisors and Euler's Totient Function

farey, relatives

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

8.6 Factorial

eulersnumber, factstone, howmanydigits, lastfactorialdigit, inversefactorial, loworderzeros

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

8.7 Combinatorics

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 $\bmod(10^9+7)$ (which would fit in an `int`) but computing the answer requires *multiplying* $\bmod(10^9+7)$.

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

[TODO: PIE?]

Chapter 9

Bit Tricks

[TODO: Basic bit manipulation: and, or, xor, shift. Right shift w/o sign extension. Using bitstrings to compactly represent sets/states. Iterating through all subsets with counter. LSB, LSZ, MSB, pop count.]

flipfive, pagelayout

[TODO: BitSet instead of array of booleans.]

Chapter 10

Geometry

[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), 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 11

Miscellaneous

11.1 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	$\dots$	$C-1$
C	$C+1$	$C+2$	$\dots$	$2C-1$
$2C$	$2C+1$	$2C+2$	$\dots$	$3C-1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$(R-1)C$	$(R-1)C+1$	$(R-1)C+2$	$\dots$	$RC-1$

- 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 (§10, page 35)).

```
int dr = 1, dc = 0;  // starting offset of (1,0); nothing special about this choice
for (int k = 0; k < 4; k++) {
    int nr = r + dr, nc = c + dc;
    // process neighbor (nr, nc)

    int tmp = dr; dr = -dc; dc = tmp;  // rotate offset ccw
    // to get cw instead, switch the negative sign
}
```

11.2 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.* $A[i] \oplus A[i+1] \oplus \ldots \oplus A[j]$.

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.

11.2.1 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* $P[0 \ldots n]$ 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 $A[1] \oplus \ldots \oplus A[j]$, and then we cancel $P[i-1] = A[1] \oplus \ldots \oplus A[i-1]$ to leave just $A[i] \oplus \ldots \oplus A[j]$ 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

11.2.2 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.

```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);
    }
}
```

11.2.3 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.

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

[TODO: Include link to discussion in CP3]

11.2.5 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 $A[i] \ldots A[j]$ 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 (§11.2.6, page 41) 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!

```java
class FenwickTree {
    private long[] a;
    public FenwickTree(int n) { a = new long[n+1]; }

    // A[i] += delta. O(lg n).
    public void add(int i, long delta) {
        for (; i < a.length; i += LSB(i)) a[i] += delta;
    }

    // query [i..j]. O(lg n).
    public long range(int i, int j) { return prefix(j) - prefix(i-1); }

    private long prefix(int i) {    // query [1..i]. O(lg n).
        long s = 0; for (; i > 0; i -= LSB(i)) s += a[i]; return s;
    }
    private int LSB(int i) { return i & (-i); }
}
```

- 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 $A[i] + \ldots + A[j]$.

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

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

[TODO: Segment trees.]

Chapter 12

Python

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.

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 13

Advanced topics

This is a list of advanced topics that may eventually be included.

- Chinese Remainder Theorem (`heliocentric`)
- 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`

Chapter 14

Resources

[TODO: methodstosolve] [TODO: UVa] [TODO: CP3] [TODO: Geeksforgeeks, topcoder, codeforces]