Hendrix Programming Team Reference

December 15, 2018

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 (§9.4, page 27)
25	$O(2^n)$	Generating all subsets (§10, page 29)
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 `int`s requires < 4 MB—no big deal since the typical memory limit is 1 GB. Don't be afraid to make arrays with millions of elements!
- `int` holds 32 bits; the largest `int` value is `Integer.MAX_VALUE` $= 2^{31} - 1$, a bit more than $2 \cdot 10^9$.
- `long` holds 64 bits; the largest `long` value is `Long.MAX_VALUE` $= 2^{63} - 1$, a bit more than $9 \cdot 10^{18}$.
- If you need larger values, use BigInteger (§2.11, page 9) or just use Python (§13, page 35); see Combinatorics (§9.4, page 27).

Chapter 2

Java Reference

2.1 Template

```
// *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).

```
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`, `throwns`, `zagrade`

2.7 Queue

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

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

2.8 PriorityQueue

[TODO: Examples of using `PriorityQueue`. Show how to construct with custom `Comparator`, eg. using lambda notation] [TODO: Note lack of decreaseKey operation (Dijkstra), use remove + add, not as fast]

`bank`, `guessthedatastructure`, `knigsoftheforest`

2.9 Set

[TODO: HashSet, TreeSet]

2.10 Map

[TODO: HashMap, TreeMap] [TODO: Iterating over keys, values, both (MapEntry)]

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, some problems on Kattis cannot be solved in Java without using this. [TODO: Link to some examples.]

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

```
/* 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 Union-find

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. `find` and `union` both take essentially constant amortized time.

> `drivingrange`, `islandhopping`, `kastenlauf`, `lostmap`, `minspantree`, `numbersetseasy`, `treehouses`, `unionfind`, `virtualfriends`, `wheresmyinternet`

```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]++; }
        }
    }
}
```

3.2 Heaps

3.3 Tries

`boggle`, `heritage`, `phonelist`

3.4 Red-black trees

3.5 Segment trees

3.6 Fenwick trees

Chapter 4

Search

4.1 Complete search

[TODO: Complete search aka brute force]

4.2 Binary and ternary search

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

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]

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)

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 $0 \ldots n-1$: 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 $O(E\sqrt{V})$.

```
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;
        }
```

```java
    }

    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 availableFlow) {
        if (s == t) return availableFlow;

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

    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.remainingCapacity() > 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 remainingCapacity() { return capacity - flow; }
}
```

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

Chapter 6

Dynamic Programming

Chapter 7

Strings

7.1 Suffix arrays

Chapter 8

Divide & Conquer

8.1 Counting inversions

`excursion`, `froshweek`

Chapter 9

Mathematics

9.1 GCD/Euclidean Algorithm

9.2 Fractions

9.3 Primes and factorization

[TODO: Basic primality testing and factorization with trial division. Sieving (primes, factors, Euler totient).]

9.4 Combinatorics

[TODO: Basic principles of combinatorics. Code for computing binomial 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. See Bit Tricks for generating all subsets.]

[TODO: PIE?]

Chapter 10

Bit Tricks

[TODO: Basic bit manipulation. Using bitstrings to compactly represent sets/states. Iterating through all subsets with counter.]

`flipfive`

[TODO: BitSet instead of array of booleans.]

Chapter 11

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.] [TODO: Convex hull.]

Chapter 12

Miscellaneous

12.1 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 combine 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 a 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* ($a \oplus (b \oplus c) = (a \oplus b) \oplus c$). [TODO: Mention word "semigroup".]
- Sometimes there is also an inverse operation $\ominus$ which "cancels out" the effects of the combining operation. For example, subtraction cancels out addition. On the other hand, there is no operation that can cancel out the effect of taking a maximum. [TODO: Mention word "group"; link to more info?]
- If we only need to find the value of combining a *single* range $A[i \ldots j]$, then ignore everything in this section and simply iterate through the interval, combining all the values in $O(n)$ time.
- More typically, we need to do many queries, and $O(n)$ per query is not fast enough. The idea is to preprocess the array into a data structure which allows us to answer queries more quickly, *i.e.* in $O(1)$ or $O(\lg n)$.
- Sometimes we also need to be able to *update* the array in between queries; in this case we need a more sophisticated query data structure that can be quickly updated.

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

12.1.1 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 another *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 "identity" value, *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 $P[j] \ominus P[i-1]$.

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.

[TODO: Example code for sum]

[TODO: Prefix sum trick] [TODO: 2D prefix sum trick with PIE] [TODO: Kadena's Algorithm for max subsequence sum] [TODO: Segment trees, Fenwick trees]

12.2 2D grids

[TODO: Discussion of implicit graphs]

[TODO: Formulas for converting between pair of coordinates and single index] [TODO: Trick for listing neighbors with delta vector]

Chapter 13

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]