

Hendrix Programming Team Reference

December 9, 2018



Contents

1	Limits	5
2	Java Reference	7
2.1	Template	7
2.2	Scanner	7
2.3	String	8
2.4	Arrays	8
2.5	ArrayList	8
2.6	Stack	8
2.7	Queue	8
2.8	PriorityQueue	8
2.9	Set	8
2.10	Map	8
2.11	BigInteger	8
2.12	Sorting	8
2.13	Fast I/O	9
3	Data Structures	11
3.1	Union-find	11
3.2	Heaps	12
3.3	Tries	12
3.4	Red-black trees	12
3.5	Segment trees	12
3.6	Fenwick trees	12
4	Graphs	13
4.1	Representation	13
4.2	Traversal (DFS/BFS)	13
4.3	Single-source shortest paths (Dijkstra)	13
4.4	All-pairs shortest paths (Floyd-Warshall)	13
4.5	Min spanning tree (Kruskal)	13
4.6	DAGs, topological sort	13
4.7	Max flow	13
4.8	Miscellaneous	13
5	Dynamic Programming	15
6	Strings	17
6.1	Suffix arrays	17
7	Divide & Conquer	19
7.1	Counting inversions	19

8 Mathematics	21
8.1 GCD/Euclidean Algorithm	21
8.2 Fractions	21
8.3 Primes and factorization	21
8.4 Combinatorics	21
9 Bit Tricks	23
10 Geometry	25
11 Miscellaneous	27
11.1 Range queries	27
11.2 2D grids	27
12 Python	29

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: 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, 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, so you can run DFS 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 running time	Example
11	$O(n!)$	Generating all permutations (§8.4, page 21)
25	$O(2^n)$	Generating all subsets (§9, page 23)
100	$O(n^4)$	Some brute force algorithms
400	$O(n^3)$	Floyd-Warshall (§4.4, page 13)
10^4	$O(n^2)$	Testing all pairs
10^6	$O(n \lg n)$	BFS/DFS; sort+greedy



bing, transportationplanning, dancerecital, prozor, rectanglesurrounding, weakvertices

- $2^{10} = 1024 \approx 10^3$
- $2 \cdot 10^9$ fits in a 32-bit `int`.
- $9 \cdot 10^{18}$ fits in a 64-bit `long`.
- If you need larger values, use `BigInteger` (§2.11, page 8) or just use Python (§12, page 29); see Combinatorics (§8.4, page 21).

Chapter 2

Java Reference

2.1 Template

```
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 (§2.13, 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

2.4 Arrays

[TODO: Basic array template/examples.]

2.5 ArrayList

2.6 Stack

[TODO: Stack class]

2.7 Queue

[TODO: Queue interface, ArrayDeque class]

2.8 PriorityQueue

[TODO: Note lack of decreaseKey operation (Dijkstra), use remove + add, not as fast]

2.9 Set

[TODO: HashSet, TreeSet]

2.10 Map

[TODO: HashMap, TreeMap]

2.11 BigInteger

2.12 Sorting

[TODO: Basic template for implementing Comparable] [TODO: Arrays.sort, Collections.sort] [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 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.StringTokenizer;
import java.io.BufferedReader;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.OutputStream;

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 (§4.5, page 13), 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`

```
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****3.4 Red-black trees****3.5 Segment trees****3.6 Fenwick trees**

Chapter 4

Graphs

4.1 Representation

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

4.2 Traversal (DFS/BFS)

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

4.3 Single-source shortest paths (Dijkstra)

4.4 All-pairs shortest paths (Floyd-Warshall)

4.5 Min spanning tree (Kruskal)

4.6 DAGs, topological sort

4.7 Max flow

4.8 Miscellaneous

[TODO: New virtual source/sink node trick]

Chapter 5

Dynamic Programming

Chapter 6

Strings

6.1 Suffix arrays

Chapter 7

Divide & Conquer

7.1 Counting inversions

Chapter 8

Mathematics

8.1 GCD/Euclidean Algorithm

8.2 Fractions

8.3 Primes and factorization

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

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

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

[TODO: PIE?]

Chapter 9

Bit Tricks

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

Chapter 10

Geometry

Chapter 11

Miscellaneous

11.1 Range queries

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

11.2 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 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:

```
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]