Hendrix Programming Team Reference

December 6, 2018

Contents

1 Limits

- Rule of thumb: 10^8 operations per second. XXX table of runtimes, discussion of brute force
- $2^{10} = 1024 \approx 10^3$
- $2 \cdot 10^9$ fits in a 32-bit `int`.
- $9 \cdot 10^{18}$ fits in a 64-bit `long`. 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* $\bmod (10^9 + 7)$.

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 (2.12, page 4).

```
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

2.5 ArrayList

2.6 Stack

2.7 Queue

ArrayDeque

2.8 Set

HashSet, TreeSet

2.9 Map

HashMap, TreeMap

2.10 BigInteger

2.11 Sorting

2.12 Fast I/O

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 (4.4, page 6), and can also be useful on its own. `find` and `union` both take essentially constant amortized time.

Kattis: `10kindsofpeople`, `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]++; }
        }
    }
}
```