Contents

List of Algorithms

Chapter 1

Fundamentals

1.1 Input and Output

The first part of solving any programming contest problem is reading the input correctly. In this section, we'll briefly go over input and output in Java using `java.util.Scanner` and `java.io.PrintWriter`. There are two scenarios that you should be familiar with: `stdin/stdout` and file I/O. You encounter the former when entering input and seeing output for a program run in the commandline. You encounter the latter when you have two files (for example `input.txt` and `output.txt`) that you read from and write to.

For `stdin/stdout`, we read input from `System.in` using a Scanner and output our results using `System.out.println()`. Here's an example of a main method that takes two integers and prints their sum:

```
public static void main(String args[]){
  // hint: you should write "import java.util.*;" at the top of your code.
  Scanner sc = new Scanner(System.in);
  int x = sc.nextInt();
  int y = sc.nextInt();
  System.out.println(x + y);
}
```

File I/O is a touch more complicated. For our Scanner, we have to make a new File object and use it in the constructor. We do something similar for our PrintWriter. However, PrintWriter also comes with a couple more usage notes. First, we should include `throws IOException` after our main method, since Java requires that we acknowledge the possibility of an IOException. (We bravely assume that our file open will succeed.) After we finish printing, we must also close the PrintWriter to make sure that everything gets written. Here's a snippet showing how Scanner and PrintWriter work with files:

```java
public static void main(String args[]) throws IOException {
  // hint: for file I/O, you should also have "import java.io.*;"
  Scanner sc = new Scanner(new File("input.txt"));
  int x = sc.nextInt();
  int y = sc.nextInt();
  PrintWriter pw = new PrintWriter(new File("output.txt"));
  pw.println(x + y);
  pw.close();
}
```

Although more efficient methods of I/O exist, such as BufferedReader and BufferedWriter, what we've covered here should be sufficient for now. (It's possible to read in 10^5 integers with Scanner in a fraction of a second.)

Now that we can read and write, let's move on to one of the most basic subroutines in computer science: sorting.

1.2 Comparable

The interface `Comparable` is how Java imposes an ordering on a set of objects. Implementing `Comparable` requires support of the method `int compareTo(Object o)`. If `a.compareTo(b) < 0`, then `a` comes before `b`; if `a.compareTo(b) > 0`, then `a` comes after `b`; if `a.compareTo(b) == 0`, then `a` is considered equal to `b`. (`a.compareTo(b) == 0` should be equivalent to `a.equals(b)`.) For example, consider the following `class MyPair`:

```java
class MyPair implements Comparable {
    int x;
    int y;
    // sort by x-coordinate first
    public int compareTo(Object o) {
        MyPair c = (MyPair) o;
        if(x < c.x) return -1;
        if(x > c.x) return 1;
        // if x-coordinates equal, compare y
        if(y < c.x) return -1;
        if(y > c.x) return 1;
        return 0; // equal
    }
}
```

We can clean this up a bit using generics:

```java
class MyPair implements Comparable<MyPair> {
    int x;
    int y;
    public int compareTo(MyPair c) {
        if(x < c.x) return -1;
        if(x > c.x) return 1;
        if(y < c.x) return -1;
        if(y > c.x) return 1;
        return 0;
    }
}
```

Note that no casting is required anymore.

1.3 Sorts

1.3.1 Merge Sort

The idea behind merge sort is simple. If we are given two equally sized sorted stacks, we could easily combine them into one sorted stack by comparing the top of the two stacks only, since the smallest element remaining must be one of those two.

From here, we divide and conquer. Simply cut the array in half, recurse on each half, and merge the two back together. This runs in $O(n \log(n))$ time.

1.3.2 Quicksort

Quicksort also uses a divide and conquer strategy to achieve an $O(n \log(n))$ amortized bound. We take a random element from the array, move anything less than it to the left of the array, anything right of it to the right of the array, and recurse.

This was previously implemented by `Arrays.sort()` and `Collections.sort()`, but Java now uses either dual-pivot quicksort or timsort.

1.4 Binary Search

Before we begin studying data structures, it is necessary to first understand general search techniques. Given a list, it is straightforward to simply check every element in the list in a linear search.

One way we can make this faster is by imposing an ordering on the list. Then inserting elements becomes slower because we can no longer just pop the element to the the end of the list. However, if we are asked to find an element, we know roughly where in the list to search because we can compare the element to other elements in the list.

Consider an array of sorted elements. The array supports $O(1)$ access to any element in the array. We'll take a look at the middle element in the array. Depending on whether or not the element to be searched is larger, smaller, or equal to the middle element, we can guarantee we can eliminate half of the array. Then finding an element is an $O(\log n)$ operation on a sorted array. This can be done either iteratively or recursively.