Lecture 2

Introduction

The structure of the course is simple to state: train students to solve ACM-ICPC contest-style problems correctly and efficiently. The hope is that through problem solving students will have a deeper understanding of algorithms, an improved level of comfort with their language of choice (C++ or Java), and stronger general problem solving skills.

Each of the problems will have some sort of description giving the details of the question that must be answered. It will describe the format of the input to be read by your solution, and the format of the output that should be generated. What follows is a rough sketch of an algorithm to employ when solving the problems:

1. Read the problem and extract the key details.
    (a) This may require you to draw parts of the problem or simple instances on paper.
    (b) Make sure to determine: the constraints on the input, the allowed running time, how the input will be formatted, and how the output should be formatted.
    (c) Some key details may be missing (sorry; can ask for clarification).
2. Decide on an algorithm to solve the problem.
    (a) Can help to determine which category the problem is in.
    (b) May have to look at multiple different correct solutions to find best one based on: running time, memory usage (UVA has limit about 512MB; stack size about 8.9MB giving a recursion depth limit around 475K for a very simple function), and ease of implementation.
3. Implement the solution.
    (a) Sometimes harder than finding the correct algorithm.
    (b) Requires a good understanding of the libraries that come standard with each language (e.g., I/O routines).
    (c) Good to have well-practiced programming idioms at your disposal.
    (d) Having a library of your own code can be very useful.
4. Test the solution.
    (a) Must have a good method for testing the supplied input (I/O redirection and diff).

   (b) Run with java −Xss8900K −Xmx512M MyClass < Test.in. C++ is version 11 and compiled with g++ -O2.
   (c) Can help to generate your own test cases (e.g., look for edge cases).

5. Debug the solution.
   (a) Be comfortable adding helpful print statements, or using a debugger.
   (b) May have to look for performance bottlenecks in the code.

The lectures of this course are designed to help you execute the above algorithm. For the most part, we will focus on algorithm selection, and implementation details. Now for some sample problems.

Exercises:

For each of the following problems, where applicable, the input comes from standard input and the time limit is 1 second.

1. In each test case you are given some permutation of the numbers $1, \ldots, n$ as a space-delimited list with one number missing. Determine the missing number and output it using one line per test case. Here $2 \leq n \leq 10^6$. Each test case is on a separate line, and there are 5 cases. The input is terminated by EOF.

2. Suppose someone writes the following Java snippet to compute 100!:

   ```
   long result = 1;
   for (int i = 1; i <= 100; ++i) result *= i;
   System.out.println(result);
   ```

   What is written to the screen?

3. Given a set of up to 10 distinct integers given as a space-delimited list, output all distinct non-empty subsets that sum to zero. Each individual subset should be output as a space-delimited list in the order it occurred in the input, and the list of subsets should be output in lexicographic order, one per line. For example, the input 0 −4 4 will yield −4 4, 0, and 0 −4 4 on three separate lines.

4. A coin is flipped n times, where n is given on a single line. You win if there is ever a run of 7 heads or 7 tails in a row. What is the probability that you win? Here $1 \leq n \leq 100$. Output the result with 6 digits after the decimal.

Solutions:

1. We give 3 solutions.

(a) Sorting: Worst case runtime is roughly $30 \cdot 5 \cdot 10^6 \cdot \lg(10^6)$ns $\approx 3s$. Here we assume that each operation takes about 1ns, and use the book's estimate that we do 30 operations per item. The estimate is very rough for at least 2 reasons: 30 is somewhat arbitrary, and I/O takes a considerable amount of time.

```java
import java.io.*;
import java.util.*;

public class Sorting
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader in = new BufferedReader(new
           InputStreamReader(System.in));
        String line;
        ArrayList<Integer> al = new
           ArrayList<Integer>(1100000);
        while ((line = in.readLine()) != null)
        {
            al.clear();
            StringTokenizer st = new StringTokenizer(line);
            while (st.hasMoreTokens())
                al.add(Integer.parseInt(st.nextToken()));
            Collections.sort(al);
            boolean found = false;
            for (int i = 0; i < al.size() && !found; ++i)
            {
                if (al.get(i) != i+1)
                {
                    System.out.println(i+1);
                    found = true;
                }
            }
            if (!found) System.out.println(al.size()+1);
        }
    }
}
```

Note that instead of a linear search, we could have binary searched for the missing value once the array was sorted.

It is usually not needed, but if the problem happens to use extended Ascii characters instead of just regular characters, you should use the following line (which is almost never necessary):

```java
BufferedReader in = new BufferedReader(new
   InputStreamReader(System.in,"ISO-8859-1"));
```

(b) Adding: Worst case runtime roughly $30 \cdot 5 \cdot 10^6$ns $\approx$ 150ms. This estimate is inaccurate since we do far fewer than 30 operations, but I/O time is considerable. Compute sum of list and subtract double the sum from $n(n+1)$ (overflow doesn't matter; if you prefer to compute $n(n+1)/2$ use a long), and divide by 2.

(c) Xoring: Worst case runtime roughly $30 \cdot 5 \cdot 10^6$ns $\approx$ 150ms. Code snippet follows:

```
int result = 0, numValues = 0;
while (st.hasMoreTokens())
{
  numValues++;
  result ^= Integer.parseInt(st.nextToken());
}
for (int i = 1; i <= numValues+1; ++i) result ^= i;
System.out.println(result);
```

Note that xor is commutative, associative, $x \wedge x = 0$ and $x \wedge 0 = x$. Furthermore, notice that both the adding and xoring solution didn't need the auxilliary ArrayList, and thus avoided many extra operations (allocation, boxing, unboxing, memory access);

Even though our estimates for the runtime are fairly inaccurate, it is true that the sorting algorithm likely would not pass the judges, but the adding and xor code would. We will see more about I/O times later. For now we will estimate runtimes that ignore I/O processing. The point of this problem is to give you an example of parsing input, to show you how Collections.sort is helpful, to introduce you to runtime analysis, and to remind you about bitwise operators.

2. Let's compute the largest integer n such that 2^n divides 100!. Note that n is given by

$$\left\lfloor \frac{100}{2^1} \right\rfloor + \left\lfloor \frac{100}{2^2} \right\rfloor + \left\lfloor \frac{100}{2^3} \right\rfloor + \left\lfloor \frac{100}{2^4} \right\rfloor + \left\lfloor \frac{100}{2^5} \right\rfloor + \left\lfloor \frac{100}{2^6} \right\rfloor = 50 + 25 + 12 + 6 + 3 + 1 = 97.$$

Recall that the long data type in Java (long long in C++) is 64-bits and signed. When overflows occur, the lowest 64-bits are what remains. That is, after our computation the lowest 64-bits of 100! will be left in result. Thus 0 is printed to the screen.

3. Here 10 is a small number as $2^{10} = 1024$. Thus we can enumerate all subsets and test each one. We then sort the valid subsets into lexicographic order using a custom sort. This would have been easier in C++ as std::vector has an operator< defined which compares lexicographically. The runtime is dominated by the loop, which requires roughly $30 \cdot 2^{10}$ns $\approx$ .03ms. Below String.split is used as a slower but more convenient alternative to StringTokenizer.

```
import java.io.*;
import java.util.*;
```

```
public class SubsetSum
{
    static class LexicoComp implements
       Comparator<ArrayList<Integer>>
    {
        public int compare(ArrayList<Integer> a,
           ArrayList<Integer> b)
        {
            int m = Math.min(a.size(),b.size());
            for (int i = 0; i < m; ++i)
            {
                int av = a.get(i), bv = b.get(i);
                if (av != bv) return av < bv ? -1 : 1;
            }
            if (a.size() != b.size())
                return a.size() < b.size() ? -1 : 1;
            return 0;
        }
    }

    public static void main(String[] args) throws Exception
    {
        BufferedReader in = new BufferedReader(new
           InputStreamReader(System.in));
        String line = in.readLine();
        String[] toks = line.split("\\s+");
        int[] arr = new int[toks.length];
        for (int i = 0; i < toks.length; ++i)
            arr[i] = Integer.parseInt(toks[i]);
        ArrayList<ArrayList<Integer>> list = new
           ArrayList<ArrayList<Integer>>();
        for (int mask = 1; mask < (1<<arr.length); ++mask)
        {
            ArrayList<Integer> curr = new ArrayList<Integer>();
            int sum = 0;
            for (int i = 0; i < arr.length; ++i)
            {
                int currBit = 1<<i;
                if ( (mask & currBit) != 0 )
                {
                    sum += arr[i];
                    curr.add(arr[i]);
                }
            }
            if (sum == 0) list.add(curr);
        }
        Collections.sort(list, new LexicoComp());
        for (ArrayList<Integer> al : list)
        {
            for (int i = 0; i < al.size(); ++i)
            {
                if (i > 0) System.out.print(" ");
                System.out.print(al.get(i));
            }
            System.out.println();
        }
    }
}
```

It's not necessary here, but in problems that require a lot of output it is often better to minimize the number of calls to System.out.print and System.out.println. To do this we can buffer our output in a StringBuilder as is done below:

```
import java.io.*;
import java.util.*;

public class SubsetSum2
{
    static class LexicoComp implements
       Comparator<ArrayList<Integer>>
    {
        public int compare(ArrayList<Integer> a,
           ArrayList<Integer> b)
        {
            int m = Math.min(a.size(),b.size());
            for (int i = 0; i < m; ++i)
            {
                int av = a.get(i), bv = b.get(i);
                if (av != bv) return av < bv ? -1 : 1;
            }
            if (a.size() != b.size())
                return a.size() < b.size() ? -1 : 1;
            return 0;
        }
    }

    public static void main(String[] args) throws Exception
    {
        BufferedReader in = new BufferedReader(new
           InputStreamReader(System.in));
        String line = in.readLine();
        String[] toks = line.split("\\s+");
        int[] arr = new int[toks.length];
        for (int i = 0; i < toks.length; ++i)
            arr[i] = Integer.parseInt(toks[i]);
        ArrayList<ArrayList<Integer>> list = new
           ArrayList<ArrayList<Integer>>();
        for (int mask = 1; mask < (1<<arr.length); ++mask)
        {
            ArrayList<Integer> curr = new ArrayList<Integer>();
            int sum = 0;
            for (int i = 0; i < arr.length; ++i)
            {
                int currBit = 1<<i;
                if ( (mask & currBit) != 0 )
                {
                    sum += arr[i];
                    curr.add(arr[i]);
                }
            }
            if (sum == 0) list.add(curr);
        }
        Collections.sort(list, new LexicoComp());
        StringBuilder sb = new StringBuilder();
        for (ArrayList<Integer> al : list)
        {
            for (int i = 0; i < al.size(); ++i)
            {
                if (i > 0) sb.append(' ');
                sb.append(al.get(i));
            }
            sb.append('\n');
        }
        System.out.print(sb);
    }
}
```

The point of this problem is to show an example of brute force, how to estimate the runtime, how to use split, how to use bitmasks, and what lexicographic orderings are. In general you should know how to compute powers of two, numbers like 10!, and roughly estimate logarithms in your head.

4. We will postpone a discussion about the precision of floating point computations to a later lecture. Below is a solution that uses memoization:

```
import java.util.*;
import java.io.*;
```

```
public class CoinProb
{
    static double[][] cache;
    static double solve(int runLength, int tossesLeft)
    {
        if (runLength == 7) return 1;
        if (tossesLeft + runLength < 7) return 0;
        if (cache[runLength][tossesLeft] > -1)
            return cache[runLength][tossesLeft];
        return cache[runLength][tossesLeft] =
            (solve(runLength+1,tossesLeft-1)+solve(1,tossesLeft-1))/2.0;
    }
    public static void main(String[] args) throws Exception
    {
        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));
        int n = Integer.parseInt(in.readLine());
        cache = new double[7][n+1];
        for (int i = 0; i < 7; ++i) Arrays.fill(cache[i],-1.0);
        System.out.printf("%.06f\n",solve(0,n));
    }
}
```

The runtime will roughly be $30 \cdot 700\text{ns} \approx .021\text{ms}$. Interestingly, it is unfavorable to play the game with 89 flips but favorable with 90 flips. This problem illustrates the usage of printf, and the usage of memoization, one of the most prevalent problem solving techniques we will cover.

Languages and Performance

In this course you will use either C++ or Java to implement your solutions. There are tradeoffs to choosing one over the other:

1. Java is easier to learn than C++.
2. Java has array indexing exceptions that will show you where you have made a mistake.
3. Java has easy to use String tokenizing and parsing routines, along with similarly easy to use routines for changing primitives into Strings.
4. Java has BigInteger, HashMap, GregorianCalendar, and regular expressions built-in, along with some geometry routines.
5. C++ is faster for the types of computations performed in this class.
6. C++ has more built-in algorithms to work with (such as next_permutation).

7. C++ has #defines and typedefs to abbreviate commonly used idioms.

Despite the tradeoffs, both languages are fine for competitions and this course. For Java, some of the things to keep in mind are:

1. BigInteger, Random, ArrayList, Arrays, Collections, HashMap, HashSet, PriorityQueue, TreeMap, TreeSet, ArrayDeque, StringBuilder, StringTokenizer, BufferedReader, Comparator, String, and primitive parsing routines should be well understood.

2. Understand how 2d arrays work, and what boxing/unboxing are.

3. The immutability of Strings means that concatenation can be costly. Consider using a StringBuilder.

The following exercises highlight some interesting performance tuning concepts that are relevant to both languages.

Exercises:

1. Suppose your program repeatedly iterates over a list of numbers in sequence. Would it be faster to use an **int**[], ArrayList, or LinkedList and if so why?

2. Consider the following three snippets:

   (a)

   ```
   int M = 1<<25, arr[] = new int[M], res = 0;
   for (int i = 0; i < M; i = (i+1)&(M-1)) res += arr[i];
   ```

   (b)

   ```
   int M = 1<<25, arr[] = new int[M], res = 0;
   for (int i = 0; i < M; i = (i+32)&(M-1)) res += arr[i];
   ```

   (c)

   ```
   int M = 1<<25, arr[] = new int[M], res = 0;
   for (int i = 0, p = 1; i < M; ++i, p = (7*p)&(M-1))
       res += arr[p];
   ```

   Which will run faster, and why?

3. Consider the following two snippets:

   (a)

   ```
   int[][] arr = new int[10000000][2];
   ```

   (b)

```
int[][] arr = new int[2][10000000];
```

Which will run faster, and why?

4. Suppose you need to build a TreeMap of 10^6 strings and then query it 10^6 times. What is the estimated runtime of this operation?

Solutions:

1. The **int**[] is fastest, followed closely by the ArrayList and then the LinkedList. The array has slightly less overhead than the ArrayList. The LinkedList is not contiguous in memory, and thus can be slower to traverse due to inefficient cache prefetching, wasted memory on nodes, better load queue/out-of-order execution performance.

2. From fastest to slowest is first, second, third. For the second the program loads more cache lines than the first example. Since the program does very few CPU operations per cache line it is memory bottlenecked even though cache prefetching is successful. For the third program the prefetcher is defeated by randomly jumping around the array. As a result, the memory operations take even longer. Note that we use &(M−1) instead of %M as moding is a relatively expensive operation.

3. The second is faster. The first allocates 10000000 different int arrays of length 2.

4. You might say $10^6 \lg 10^6 \approx 20 \cdot 10^6$ giving roughly $.6s$ (up to a factor of 2 for building the tree) but this would be leaving out an important factor: the length of the strings. This would give a runtime of $O(mn \lg n)$, where m is the average length of the strings. If m is large, this can be prohibitively expensive.

When profiling keep the following things in mind (in general, not just this class):

1. Understand the memory hierarchy, and non-algorithmic runtime costs (like I/O).

2. Write small profiling tests, and run them many times. Look at the distribution of your times and plot them if possible.

3. Logging and instrumentation has a cost that should be considered.

4. Use tools to help profile your code, analyze system calls, and detect bottlenecks.

5. Modern compilers are very complex and it is hard to intuit the code they produce. Profiling is the best way to measure the performance of your code.

6. The rule of counting items and billing each one at 30ns is a rough estimate and has many issues with it. That said, it can be a good way to ballpark the runtime of your code and determine whether an algorithm is feasible or infeasible when the answer is clear (eg., it will take 10 seconds, or it will take 100 milliseconds on a problem that gives you 1 seconds).