Thomas Jefferson Invitational Open in Informatics 2012

Contest Part II (Practical—Programming)

Problem Analyses and Sample Solutions

In this document are explanation and Java solutions to all the programming problems from TJ IOI 2012. More sample solutions and test cases are available on the website. Feel free to contact us if you have any questions.

Programming Problem Practice 0

Addition Is Fun (add)

In this problem, we are asked to read in and sum a list of numbers. There are at most 10,000 values, and each is at most 100. So, the maximum possible sum is 10,000,000 - small enough to fit within a standard 32 bit integer.

We can use a loop to read in all of the numbers and add them on to the variable holding the sum. Once we process all the values, we print out the answer.

Java users are advised to use a BufferedReader instead of a Scanner for faster input.

Sample Solution

```
import java.io.*;

public class add {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int S = 0, N = Integer.parseInt(in.readLine());
        for(int n=0; n<N; n++)
            S+=Integer.parseInt(in.readLine());
        System.out.println(S);
    }
}
```

Programming Problem Practice 1

Programming Contest (tjioi)

We can solve this problem using what is known as a *Greedy Algorithm*. We start by observing that it is always more advantageous to solve a shorter problem before a longer one, since each has the same point value. Since we want the smallest values first, we want to read in the list of problem lengths and sort it.

Once the values are sorted, we look through them from smallest to largest, keeping track as we go of how much time has been used up for all of the problems solved so far. When we run out of time, we print out the answer: the number of problems solved.

Conveniently, most programming languages (Java, C++, etc.) have built in sorting methods for you to use.

Sample Solution

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

public class tjioi {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int N = Integer.parseInt(in.readLine()), T=Integer.parseInt(in.readLine()), A;
        int[] m = new int[N];
        for(int n=0; n<N; n++)
            m[n] = Integer.parseInt(in.readLine());
        Arrays.sort(m);
        for(A=0; A<N; A++)
            if(T>=m[A]) T-=m[A];
            else break;
        System.out.println(A);
        System.exit(0);
    }
}
```

Programming Problem 0

The Quest Begins (miles)

This problem asks us to convert from miles to feet, and then multiply by the number of ice cream cones per foot. First, we read in the number of miles and multiply by 5280 to convert to feet.

Then, we compute for each length the minimum and maximum number of cones by multiplying by 3 and 5, respectively. Finally, we print the answers. The solution uses Java's built-in round function, which most other languages also have. Many teams came up with alternative solutions to rounding (for example, adding 0.5 and then truncating the number).

Sample Solution

```
import java.io.*;

public class miles {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int N = Integer.parseInt(in.readLine());
        for(int n=0; n<N; n++) {
            double d = Double.parseDouble(in.readLine());
            System.out.println(Math.round(3*5280*d) + " " + Math.round(5*5280*d));
        }
        System.exit(0);
    }
}
```

Programming Problem 1

A Number Game (numbers)

To solve this problem, we maintain an array of numbers that tell us the digits written down by each player on each turn. We know that they both write '1' for the first turn. We also have rules telling us what they write on each subsequent turn, based on the digits from the previous turn.

After setting up the array, we use a loop to fill in the values from the second turn onward, based on the rules we were given. We can stop as soon as we reach the turn we were asked about. Finally, we print out the answer.

In implementing the rules, we have to be careful to make sure that the right player performs addition first.

Sample Solution

```
import java.io.*;

public class numbers{
    static int[][] num = new int[10000][2];
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int N = Integer.parseInt(in.readLine());
        int Q = in.readLine().equals("lbert")?1:0;
        int F = in.readLine().equals("lbert")?1:0;
        num[0][0] = num[0][1] = 1;
        for(int n=1; n<N; n++){
            num[n][F] = (num[n-1][0]+num[n-1][1])%10;
            num[n][1-F] = num[n-1][1-F];
            F = 1-F;
        }
        System.out.println(num[N-1][Q]);
        System.exit(0);
    }
}
```

Programming Problem 2

Ice Cream Race (overlap)

To solve this problem, we can simulate the situation described. We use two variables to keep track of how much ice cream each Glacier's stack has. We then use a loop to simulate the race. For each second, we add on the correct number of scoops to each pile, and subtract the scoops eaten by Sir Albert (if needed). Then, we check if the piles have an equal number of scoops. If so, we increase the answer by one. Finally, we print out the answer.

Sample Solution

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

public class overlap{
    static int[][] num = new int[10000][2];
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer st = new StringTokenizer(in.readLine());
        int A = Integer.parseInt(st.nextToken()), B = Integer.parseInt(st.nextToken());
        int K = Integer.parseInt(st.nextToken()), T = Integer.parseInt(st.nextToken());
        int ans = 1, a = 0, b = 0;
        for(int t=0; t<T; t++){
            a = (a+A)%K;
            b = (b+B)%K;
            if(a==b) ans++;
        }
        System.out.println(ans);
        System.exit(0);
    }
}
```

Programming Problem 3

Buying Ice Cream (buy)

This problem can be solved using a Greedy Algorithm. We observe that we will prefer to purchase cheaper flavors, since we are trying to maximize the amount of ice cream we get.

In order to take the flavors from cheapest to most expensive, we can sort all of them. Then, we iterate through and buy as much of each flavor as we can before moving on to the more expensive flavors.

In our sample Java implementation, we use a data structure known as a TreeMap to handle the sorting.

Sample Solution

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.StringTokenizer;

public class buy {

    public static void main(String[] args) throws IOException
    {
        BufferedReader f = new BufferedReader(new InputStreamReader(System.in));
        String line = f.readLine();
        StringTokenizer st = new StringTokenizer(line, " ");
        int n = Integer.parseInt(st.nextToken());
        int amt = Integer.parseInt(st.nextToken());
        HashMap<Integer,ArrayList<Integer>> vals = new HashMap<Integer,ArrayList<Integer>>();
        int[] keys = new int[n];
        for (int i = 0; i < n; i++)
        {
            String line1 = f.readLine();
            StringTokenizer s = new StringTokenizer(line1, " ");
            int price = Integer.parseInt(s.nextToken());
            int amount = Integer.parseInt(s.nextToken());
            if (vals.containsKey(price))
            {
                ArrayList<Integer> temp = vals.get(price);
                temp = sortedInsertion(temp,amount);
                vals.put(price,temp);
            }
            else
            {
                ArrayList<Integer> temp = new ArrayList<Integer>();
                temp.add(amount);
                vals.put(price,temp);
            }
            keys[i] = price;
        }
        f.close();
        Arrays.sort(keys);
```

```
        int finalPrice = 0;
        int index = 0;
        int count = 0;
        while (amt > 0 && index < keys.length)
        {
            int nextKey = keys[index];
            if (amt < vals.get(nextKey).get(0) * nextKey)
            {
                count+= amt/nextKey;
                amt = 0;
            }
            else
            {
                amt -= vals.get(nextKey).get(0) * nextKey;
                count+= vals.get(nextKey).get(0);
                ArrayList<Integer> temp = vals.get(nextKey);
                temp.remove(0);
                vals.put(nextKey,temp);
                index++;
            }
        }
        System.out.println(count);
        System.exit(0);
    }
    public static ArrayList<Integer> sortedInsertion(ArrayList<Integer> a, int b)
    {
        int i = 0;
        while(i < a.size() && b >= a.get(i))
            i++;
        if (i == a.size())
            a.add(a.size(), b);
        else if (i == 0)
            a.add(0,b);
        else
            a.add(i+1,b);
        return a;
    }

}
```

Programming Problem 4

Ice Cream Pizza (fractions)

This problem asks us to add up a list of fractions. We keep track of the total sum by using two variables, one for the numerator of our sum and another from the denominator. To start, we set the numerator to 0 and the denominator to 1.

Then, we read in all of the fractions one by one and add them to our sum. To add two fractions, we observe that $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$. However, there is a possibility that the numerator and denominator will share a common factor, which needs to be divided out. In order to do this, we find the greatest common factor of the numerator and denominator, and divide it out from both of them.

Finally, we need to print the answer as a mixed number. First, we divide the numerator by the denominator and drop any remainder to find the integer portion. Next, we print out the remainder with the appropriate denominator.

In order to find the greatest common divisor, we can iterate through all possibilities of numbers, checking for the largest number that divides the numerator and the denominator. There also exists a more efficient method, implemented below, called the Euclidean algorithm, but it was not necessary to pass within the time limits on this problem.

Sample Solution

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

public class fractions {
    static long gcd(long a, long b) {
        if(b==0) return a;
        return gcd(b, a%b);
    }
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int N = Integer.parseInt(in.readLine());
        long A = 0, B = 1;
        for(int n=0; n<N; n++) {
            StringTokenizer st = new StringTokenizer(in.readLine());
            long a = Integer.parseInt(st.nextToken());
            long b = Integer.parseInt(st.nextToken());
            A = A*b + a*B;
            B *= b;
            long g = gcd(A,B);
            A /= g;
            B /= g;
        }
        System.out.println(A/B);
        if(A%B != 0) System.out.println(A%B + " " + B);
        System.exit(0);
    }
}
```

```
//A/B + a/b = (Ab + aB)/(Bb)
```

Programming Problem 5

Ice Cream Fad (fad)

To solve this problem, we will use a matrix of characters to represent the city. Each cell gets a corresponding entry in the matrix.

For each 'day,' we look through all of the values in the grid and spread the fads outward from each cell, where appropriate. We stop checking as soon as we see a turn where no cell in the grid changes. One of the answers we are asked to print is the number of days before the grid stops changing, so we save this value.

Then, we look through all of the cells to count the number of people who are fans of both flavors. Finally, we print both answers.

Sample Solution

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

public class fad {
   static int n;
   static int m;
   static char grid[][];
   static boolean x[][], x2[][], y[][], y2[][];
   public static void main(String args[]) {
      int jdx[] = {1, -1, 0, 0};
      int jdy[] = {0, 0, 1, -1};
      int pdx[] = {-1, -1, 1, 1};
      int pdy[] = {-1, 1, -1, 1};
      Scanner cin = new Scanner(System.in);
      n = cin.nextInt();
      m = cin.nextInt();
      grid = new char[n][m];
      x = new boolean[n][m];
      y = new boolean[n][m];
      for(int i = 0; i < n; ++i) {
         String line = cin.next();
         for(int j = 0; j < m; ++j) {
            grid[i][j] = line.charAt(j);
            if(grid[i][j] == 'X') {
               x[i][j] = true;
            } else {
               x[i][j] = false;
            }
            if(grid[i][j] == 'Y') {
               y[i][j] = true;
            } else {
               y[i][j] = false;
            }
         }
      }

      int t = 0;
      boolean spread = true;
      while(spread) {
         spread = false;
```

```
        x2 = new boolean[n][m];
        y2 = new boolean[n][m];
        for(int i = 0; i < n; ++i) {
           for(int j = 0; j < m; ++j) {
              if(x[i][j]) {
                 for(int k = 0; k < 4; ++k) {
                    if(i + jdx[k] < 0 || i + jdx[k] >= n || j + jdy[k] < 0 || j + jdy[k] >= m) {
                       continue;
                    }
                    if(grid[i + jdx[k]][j + jdy[k]] != 'E' && x[i + jdx[k]][j + jdy[k]] == false) {
                       spread = true;
                       x2[i + jdx[k]][j + jdy[k]] = true;
                    }
                 }
              } if(y[i][j]) {
                 for(int k = 0; k < 4; ++k) {
                    if(i + pdx[k] < 0 || i + pdx[k] >= n || j + pdy[k] < 0 || j + pdy[k] >= m) {
                       continue;
                    }
                    if(grid[i + pdx[k]][j + pdy[k]] != 'E' && y[i + pdx[k]][j + pdy[k]] == false) {
                       spread = true;
                       y2[i + pdx[k]][j + pdy[k]] = true;
                    }
                 }
              }
           }
        }
        for(int i = 0; i < n; ++i) {
           for(int j = 0; j < m; ++j) {
              x[i][j] |= x2[i][j];
              y[i][j] |= y2[i][j];
           }
        }
        t++;
     }
     int count = 0;
     for(int i = 0; i < n; ++i) {
        for(int j = 0; j < m; ++j) {
           if(x[i][j] && y[i][j]) {
              count++;
           }
        }
     }
     System.out.println(count);
     System.out.println(t - 1); // takes one more of unit of time to verify
     System.exit(0);
  }
}
```

Programming Problem 6

Scrambled Flavors (scramble)

To quickly check whether two flavor names match, we can sort the words and then check if the resulting strings are identical.

Thus, we sort all of the strings we are given. Then, we read in each potential flavor name and check it against all of the original flavor names to see if it matches any of them.

Sample Solution

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

public class scramble {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        StringTokenizer st = new StringTokenizer(br.readLine(), " ");
        int N = Integer.parseInt(st.nextToken());
        int M = Integer.parseInt(st.nextToken());
        String[][] orig = new String[N][];

        for(int i = 0; i < N; i++) {
            ArrayList<String> current = new ArrayList<String>();
            StringTokenizer s = new StringTokenizer(br.readLine(), " ");
            while(s.hasMoreTokens()) {
                current.add(s.nextToken());
            }
            orig[i] = current.toArray(new String[1]);
            Arrays.sort(orig[i]);
        }

        for(int i = 0; i < M; i++) {
            String flav = br.readLine();
            ArrayList<String> current = new ArrayList<String>();
            StringTokenizer s = new StringTokenizer(flav, " ");
            while(s.hasMoreTokens()) {
                current.add(s.nextToken());
            }
            String[] arr = current.toArray(new String[1]);
            Arrays.sort(arr);

            for(String[] a : orig) {
                if(Arrays.equals(a, arr)) {
                    System.out.println(flav);
                    break;
                }
            }
        }
        System.exit(0);
    }
}
```

Programming Problem 7

Store Managers (boss)

When we compute the five best flavors for any specific manager, it is sufficient to know the five best flavors for each of his direct underlings. We also immediately know the answer for any manager without any underlings. Thus, we process them bottom up, keeping the top five flavors for each manager. Then, we print the answers in the original order given.

We can implement the bottom up process by using a recursive method; see our sample code for details. The code below uses a class to represent each node. A single call to the senior manager's node results in recursive calls to each of its children, and eventually their children, and so on. An ArrayList stores the values within each subtree.

Sample Solution

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

public class boss
{
    static int N;
    static Node[] nodes;
    public static void main(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int N = Integer.parseInt(in.readLine());
        nodes = new Node[N + 1];
        for(int i = 1; i <= N; i++)
            nodes[i] = new Node();
        for(int i = 2; i <= N; i++)
            nodes[i].process(in.readLine());
        nodes[1].solve();
        for(int i = 1; i <= N; i++)
            System.out.println(nodes[i].answer);
        System.exit(0);
    }
    public static class Node
    {
        int parent, answer;
        ArrayList<Integer> vals;
        ArrayList<Node> children;
        public Node()
        {
            children = new ArrayList<Node>();
            vals = new ArrayList<Integer>();
            parent = -1;
        }
        public void process(String input)
        {
            StringTokenizer st = new StringTokenizer(input);
            parent = Integer.parseInt(st.nextToken());
            nodes[parent].addChild(this);
            if(st.hasMoreTokens())
                for(int i = 0; i < 5; i++)
                    vals.add(Integer.parseInt(st.nextToken()));
        }
```

```
        public void addChild(Node other)
        {
            children.add(other);
        }
        public void solve()
        {
            for(Node n : children)
            {
                n.solve();
                for(Integer i : n.vals)
                    vals.add(i);
            }
            Collections.sort(vals);
            answer = vals.get(vals.size() - 5);
        }
    }
}
```

Programming Problem 8

Ice Cream Grids (rotate)

In this problem, the bounds are small enough for us to check every possible location in the grid. We must also be careful to consider rotations of the arrangement we are searching for. We can do this by implementing a function to rotate a given grid by 90 degrees, and calling it 1, 2, or 3 times to achieve all possible rotations. As we check each of the four possible orientations for each location, we keep track of the number of exact matches we find.

See the rotate function below for details on how to implement rotation of an arbitrary square grid. The implementation is slightly tricky. Note that the value at row i and column j is moved to row j and column $N - i - 1$, where rows and columns are numbered starting from 0 and N is the length of the grid.

Sample Solution

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

public class rotate
{
    static int R, C, N;
    static char[][] grid, mini, temp;
    // 90 degrees clockwise
    public static void rotate()
    {
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                temp[j][N - i - 1] = mini[i][j];
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                mini[i][j] = temp[i][j];
    }
    public static void main(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer st = new StringTokenizer(in.readLine());
        R = Integer.parseInt(st.nextToken());
        C = Integer.parseInt(st.nextToken());
        grid = new char[R][C];
        for(int i = 0; i < R; i++)
            grid[i] = in.readLine().toCharArray();
        N = Integer.parseInt(in.readLine());
        mini = new char[N][N];
        temp = new char[N][N];
        for(int i = 0; i < N; i++)
            mini[i] = in.readLine().toCharArray();

        int answer = 0;
        for(int i = 0; i + N <= R; i++)
            for(int j = 0; j + N <= C; j++)
            {
                boolean good = false;
                for(int k = 0; k < 4; k++)
                {
                    boolean yes = true;
```

```
                    for(int a = 0; a < N; a++)
                        for(int b = 0; b < N; b++)
                            if(mini[a][b] != grid[i + a][j + b])
                                yes = false;
                    if(yes)
                        good = true;
                    rotate();
                }
                if(good)
                    answer++;
            }
        System.out.println(answer);

        System.exit(0);
    }
}
```

Programming Problem 9

Toppings (toppings)

One might try a greedy strategy, where you take the toppings with the highest tastiness to space ratio in order to use the space most efficiently. Unfortunately, this does not work. For example, suppose we have $S = 5$, and toppings with $C = [2, 2, 5]$ and $T = [4, 4, 9]$. The greedy strategy would take the first two toppings, since $4/2 > 9/5$. But that would only give a total tastiness of 8, while we can achieve 9 by simply taking the last topping.

To solve this problem, we need to use dynamic programming. We start by making a matrix dp, where $dp[n][s]$ will contain the amount of tastiness we can fit into s space if we only consider the first n toppings.

We can express values in this array in terms of previous values if we split possible solutions into two cases: the case where we use the nth topping, and the case where we do not. If we do not use the nth topping, the maximum tastiness would simply be $dp[n-1][s]$. If we do use the last topping, then the maximum tastiness would be $T[n] + dp[n-1][s-C[n]]$, where $C[n]$ is the cost of the nth topping and $T[n]$ is its tastiness. We want the greater of these two values, so $dp[n][s] = \max(dp[n-1][s], T[n] + dp[n-1][s-C[n]])$.

We can use two loops to find the value of $dp[n][s]$ for every n and s up to the N and S given in the problem input. Since we need to use previous values to calculate new values of $dp[n][s]$, we have to be careful to make sure the loop indices are increasing. For the final answer, we print out the value of $dp[N][S]$.

Sample Solution

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

public class toppings {
    static int s, n;
    static int c[], t[], dp[];
    public static void main(String args[]) {
        Scanner cin = new Scanner(System.in);
        s = cin.nextInt();
        n = cin.nextInt();
        c = new int[n + 1];
        t = new int[n + 1];
        dp = new int[s + 1];
        for(int i = 0; i < n; ++i) {
            c[i] = cin.nextInt();
            t[i] = cin.nextInt();
        }
        dp[0] = 0;
        for(int i = 1; i <= s; ++i) {
            dp[i] = 0;//1000 * 100 + 1;
        }
        for(int j = 0; j < n; ++j) {
            for(int i = s; i >= c[j]; --i) {
```

```
                dp[i] = Math.max(dp[i], t[j] + dp[i - c[j]]);
            }
        }
        int ans = 0;
        for(int i = s; i >= 0; --i) {
            if(dp[i] != 1000 * 100 + 1) {
                ans = Math.max(ans, dp[i]);
            }
        }
        System.out.println(ans);
        System.exit(0);
    }
}
```

Programming Problem 10

Too Many Flavors (manyflav)

The first step to solving this problem is to find out the bin number where each flavor ends (the index of the last bin containing that flavor). We can store this in an array. Whenever we get a query, we search this array for the first "last bin" that is less than the query bin, and then take the next flavor (the flavor after the bin that just ended).

Thus, there exists an iterative solution that goes through each of the flavors, searching for the query. However, this is too slow. One observation is that the array is an increasing sequence of bin indices. It is already sorted. Thus, we can search for the target bin using a binary search. See the solution below for details. Note that Java's built-in binary search returns a negative index when the element is not found, so some small manipulations are necessary to convert these negative indices to flavor numbers.

Sample Solution

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

public class manyflav{
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        StringTokenizer st = new StringTokenizer(br.readLine(), " ");
        long N = Long.parseLong(st.nextToken());
        long M = Long.parseLong(st.nextToken());
        long[] amts = new long[(int)N+1];
        for(int i = 1; i <= N; i++) {
            amts[i] = Long.parseLong(br.readLine()) + amts[i-1];
        }
        for(int i = 0; i < M; i++) {
            long loc = Long.parseLong(br.readLine());
            int index = Arrays.binarySearch(amts, loc);
            if(index < 0){
                index += 1;
                index *= -1;
            }
            System.out.println(index);
        }
        System.exit(0);
    }
}
```