Auburn University

2014 ICPC Notes

Table of Contents

Dijkstra's Algorithm

Dijkstra's Algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex.

Fun fact: if you let all edge weights equal 1, you simply have breadth-first-search.

Pseudocode

```
1   function Dijkstra(Graph, source):
2         dist[source]  := 0   // Distance from source to source
3         for each vertex v in Graph:       // Initializations
4             if v ≠ source
5                 dist[v]  := infinity// Unknown distance function from source to v
6                 previous[v]  := undefined   // Previous node in optimal path from source
7             end if
8             add v to Q             // All nodes initially in Q (unvisited nodes)
9         end for
10
11        while Q is not empty:                  // The main loop
12            u := vertex in Q with min dist[u]  // Source node in first case
13            remove u from Q
14
15            for each neighbor v of u: // where v has not yet been removed from Q.
16                alt := dist[u] + length(u, v)
17                if alt < dist[v]:         // A shorter path to v has been found
18                    dist[v]  := alt
19                    previous[v]  := u
20                end if
21            end for
22        end while
23        return dist[], previous[]
24    end function
```

Minimum Spanning Trees

Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

One example would be a telecommunications company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths (eg. along roads), then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight — there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, thus would represent the least expensive path for laying the cable.

There are two different algorithms that we've talked about that create a minimum spanning tree. **Prim's Algorithm** and **Kruskal's Algorithm**. They're slightly different, but can be used for the same applications. Pretty much just pick one.

Prim's Algorithm

Informal Description

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Technical

If a graph is empty then we are done immediately. Thus, we assume otherwise.

The algorithm starts with a tree consisting of a single vertex, and continuously increases its size one edge at a time, until it spans all vertices.

- Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize: V_{new} = {x}, where x is an arbitrary node (starting point) from V, E_{new} = {}
- Repeat until V_{new} = V:
- Choose an edge {u, v} with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)

Add v to V_{new}, and {u, v} to E_{new}
Output: V_{new} and E_{new} describe a minimal spanning tree

Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Description

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty and F is not yet spanning
    - remove an edge with minimum weight from S
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Pseudocode

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6       A = A ∪ {(u, v)}
7       UNION(u, v)
8 return A
```

Example

The best example of this was done in the problem "Underground Cables."

Underground Cables asks "Given a list of points, what is the least amount of cable necessary to make sure that every pair of points is connected, either directly, or indirectly through other points?"

Here is the Java solution to this problem, using **Kruskal's Algorithm**.

```
import java.util.*;
import java.text.*;
```

```
class UndergroundCables {
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        DecimalFormat df = new DecimalFormat("#.00");

        for (;;) {
            int n = in.nextInt();
            if (n == 0) return;

            ArrayList<Point> points = new ArrayList<Point>(n);

            for (int i = 0; i < n; i++) {
                points.add(new Point(in.nextInt(), in.nextInt(), i));
            }

            // revisit this
            ArrayList<Edge> edges = new ArrayList<Edge>();

            for (int i = 0; i < points.size(); i++) {
                for (int j = i + 1; j < points.size(); j++) {
                    edges.add(new Edge(points.get(i).distance(points.get(j)),
                                       points.get(i),
                                       points.get(j)));
                }
            }

            Collections.sort(edges);
            int current = 0;
            double cost = 0;

            while(!sameColor(points)) {
                Edge e = edges.get(current++);

                if (e.p1.color == e.p2.color) {
                    continue;
                } else {
                    int oldColor = e.p2.color;
                    e.p2.color = e.p1.color;
                    recolor(points, oldColor, e.p1.color);

                    cost += e.weight;
                }
            }
            System.out.println(df.format(cost));
        }

    }

    private static boolean sameColor(ArrayList<Point> points) {
        for (int i = 1; i < points.size(); i++) {
            if (points.get(i).color != points.get(i-1).color) {
                return false;
```

```
            }
        }
        return true;
    }

    private static void recolor(ArrayList<Point> points, int oldColor, int newColor) {
        for (Point p : points) {
            if (p.color == oldColor) {
                p.color = newColor;
            }
        }
    }

    private static class Point {
        public int x;
        public int y;

        public int color;

        public Point(int x, int y, int color) {
            this.x = x;
            this.y = y;
            this.color = color;
        }

        public double distance(Point p) {
            int x = this.x - p.x;
            int y = this.y - p.y;
            return Math.sqrt(x * x + y * y);
        }
    }

    private static class Edge implements Comparable<Edge> {
        public double weight;
        public Point p1;
        public Point p2;

        public Edge(double weight, Point p1, Point p2) {
            this.weight = weight;
            this.p1 = p1;
            this.p2 = p2;
        }

        public int compareTo(Edge other) {
            if (this.weight < other.weight) {
                return -1;
            } else if (this.weight == other.weight) {
                return 0;
            } else {
                return 1;
            }
        }
    }
}
```

Topological Sort | Cycle Detection

Topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Pseudocode

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

Minimization / Maximization Problems

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. More generally, optimization includes finding "best available" values of some objective function given a definedomain (or a set of constraints), including a variety of different types of objective functions and different types of domains.

Maximum Subarray Sum

A maximum subarray sum is the largest sum of a subsection of an array that may or may not contain negative values.

Example

One example of a Maximum Subarray Sum problem is the problem **Profits**. In this problem, we're presented with profits of a company over a span of many days. Some of these profits are negative and some are zero or positive. The problem is to find the largest profit made by the company over a span of several days. For example, over the following period:

Day 1: -3
Day 2: 4
Day 3: 9
Day 4: -2
Day 5: -5
Day 6: 8

The largest span amount of profit generated is 14, generated over the span of Day 2 to Day 6 (inclusive)

```
#!/usr/bin/python3
def maxSubarray(vals):
	maxHere, maxAll = 0, 0
	for x in vals:
		maxHere = max(0, maxHere+x)
		maxAll = max(maxHere, maxAll)
	if maxAll == 0:
		return max(vals)
	else:
		return maxAll

while True:
	n = int(input())
	if n > 0:
		vals = [int(input()) for i in range(n)]
		print(maxSubarray(vals))
	else:
		break
```

Dynamic Programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure

Dynamic Programming: Memory-oriented algorithms

With Dynamic Programming, we use computer memory to store previous results that we've computed (this is called memoization) to avoid having to recompute results multiple times.

Rod-cutting problem

You have a rod of length n that you can cut into smaller pieces so long as both pieces have an integer length. A rod of length m will sell at price prices[m]. How do you cut your rods so that they sell for the highest price?

Pseudocode

```
def CutRod(n,prices):
       r = array of length n+1
       r[0] = 0
       for i from 1 to n+1:
           q = minimal possible price
           for j from 1 to i:
               q = max(q, prices[j] + r[i-j]
           r[i] = q
       return r[n]
```

Knapsack Problem

The Knapsack problem is a more sophisticated dynamic programming problem, where you enter a store with a certain amount of money. You want to buy as much value as you can with the money you have. What do you buy?

Example

A good example of the Knapsack problem is the problem **Candy Store**. In this problem, we enter a store. The store has n types of candy, and we have m dollars. Each candy has a cost in dollars and a number of calories. Our challenge is to figure out how many calories we can get with the money we have.

```java
import java.util.*;

public class CandyStore {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        for (;;) {
            int n = scan.nextInt();
            int m = (int) (scan.nextDouble() * 100);

            if (n == 0) {
                break;
            }

            int[] a = new int[m + 1];
            for (int i = 0; i < n; i++) {
                int c = scan.nextInt();
                int p = (int) (scan.nextDouble() * 100);
                a[p] = Math.max(c, a[p]);
            }
            for (int i = 0; i <= m; i++) {
                for (int j = 0; j <= i / 2; j++) {
                    int k = i - j;
                    a[i] = Math.max(a[j] + a[k], a[i]);
                }
            }
            System.out.println(a[m]);
        }
    }
}
```

Tree Traversal

Tree traversal (also known as **tree search**) is a form of graph traversal and refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

There are three types of depth-first traversal: pre-order, in-order, and post-order. For a binary tree, they are defined as display operations recursively at each node, starting with the root node, whose algorithm is as follows:

Pre-order

1. Display the data part of root element (or current element).
2. Traverse the left subtree by recursively calling the pre-order function.
3. Traverse the right subtree by recursively calling the pre-order function.

   Pseudocode

```
preorder(node)
        if node == null then return
        visit(node)
        preorder(node.left)
        preorder(node.right)
```

In-order (symmetric)

1. Traverse the left subtree by recursively calling the in-order function.
2. Display the data part of root element (or current element).
3. Traverse the right subtree by recursively calling the in-order function.

   Pseudocode

```
inorder(node)
          if node == null then return
          inorder(node.left)
          visit(node)
          inorder(node.right)
```

Post-order

1. Traverse the left subtree by recursively calling the post-order function.
2. Traverse the right subtree by recursively calling the post-order function.
3. Display the data part of root element (or current element).

   Pseudocode

```
postorder(node)
          if node == null then return
          postorder(node.left)
```

```
        postorder(node.right)
        visit(node)
```

For some cases, you'll want to generalize these out to a tree with n children per node.

To traverse *any* tree in **depth-first order**, perform the following operations recursively at each node:

1. Perform pre-order operation
2. For each *i* (with *i* = 1 to *n* − 1) do:
    1. Visit *i*-th, if present
    2. Perform in-order operation
3. Visit *n*-th (last) child, if present
4. Perform post-order operation

where *n* is the number of child nodes. Depending on the problem at hand, the pre-order, in-order or post-order operations may be void, or you may only want to visit a specific child node, so these operations are optional. Also, in practice more than one of pre-order, in-order and post-order operations may be required. For example, when inserting into a ternary tree, a pre-order operation is performed by comparing items. A post-order operation may be needed afterwards to re-balance the tree.

Xor Math

Xor (^) logic combines the bits of two numbers, zeroing bits that are both 1 or 0 and marking as 1 the bits that are different.
0^0 = 0
0^1 = 1
1^1 = 0
1^0 = 1

Example
This is used particularly in the problem **Ping!** In Ping, we must determine the number of satellites that transmit over a given interval. Satellites that transmit at the same time negate each other's signals, and xor logic is perfect for resolving that interference. Code for the problem can be seen below:

```
import java.util.*;
import java.math.*;

public class Ping {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        for (;;) {
            char[] str = (in.nextLine().trim()).toCharArray();
            if (str.equals("0")) {
                return;
            }
            String output = "";
            for (int = 1; i < str.length; i++) {
                if (str[i] == '1') {
                    output += i;
                    output += " ";
                    for (int j = i; j < str.length; j += i) {
                        str[j] = (char) (str[j] ^ (char) 1);
                    }
                }
            }
            System.out.println(output.trim());
        }
    }
}
```

Physics Computations

We will discuss several physics formulas related to the problem **Collision Detection**.

Position

The position of an object can be represented by a 2- or 3-vector (x, y) or (x, y, z). In collision detection, a car's location is represented by the vector (x, y).

Velocity

The velocity of an object can be represented by a 2- or 3-vector (vx, vy) or (vx, vy, vz). It can be computed as an object's change in position over a span of time. Velocity modifies position according to the following formula:

```
x += vx * timeStep;
y += vy * timeStep;
```

Where timeStep is the amount of time to apply to the operation.

Speed

Speed is the length of the velocity vector. It can be computed as the distance of the velocity vector from a vector composed exclusively of 0.

Acceleration

Acceleration is the change in an object's velocity over a span of time. Formulas for dealing with acceleration are similar to those for speed. In cases where acceleration is only along the velocity vector, such as in Collision Detection, it can be treated as a single value (a) as opposed to a vector (ax, ay).

Normalized Vector

A normalized vector is a vector divided by its length. The formula for this (specifically for normalizing a velocity vector) is as follows:

```
normalizedVX = vx / Math.sqrt(vx * vx + vy * vy);
normalizedVY = vy / Math.sqrt(vx * vx + vy * vy);
```

Dot Product

A vector Dot Product is the sum of the product of each element of a vector. A 2D dot product of vectors U and V is:

```
u * v = ux*vx + uy*vy;
```

A 3D vector dot product is computed by adding the product of uz*vz.

Cross Product

A vector's Cross Product is the vector normal to two vectors. A 2D cross product will give a scalar value, as there is no normal direction. The formula for 2D cross product for vectors U and V is:

```
assume u = (ux, uy)
```

```
assume v = (vx, vy)
u X v = ux * vy - uy * vx;
```

For a 3D cross product, a 3D vector is returned:

```
assume u = (ux, uy, uz)
assume v = (vx, vy, vz)
u X v = (uy*vz - uz*vy, uz*vx - ux*vz, ux*vy - uy*vx);
```

Distance

Distance is a measurement of cost to travel between two points. We will discuss Euclidean distance, although there are other metrics that can be used. The distance formula for a pair of 2-dimensional points is:

```
return Math.sqrt(Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2));
```

The formula for a pair of 3-dimensional points is:

```
return Math.sqrt(Math.pow(p2.x - p1.x, 2) +
                 Math.pow(p2.y - p1.y, 2) +
                 Math.pow(p2.z - p1.z, 2));
```

Example

In Collision Detection, we are given a pair of cars travelling in straight lines. They have a position, velocity, and acceleration, and the problem is to determine if they will collide. The result is determinable by simulation, as shown below:

```
import java.util.*;

public class CollisionDetection {
	static double timeStep = 0.0001;
	private double t, x, y, s, a, vx, vy;

	public Car(double t1, double x1, double y1, double s1,
		double t2, double x2, double y2, double s2) {
		t = t2;
		x = x2;
		y = y2;
		s = s2;
		a = (s2 - s1) / (t2 - t1);
		// Compute vx, vy as a Normalized Vector.
		// We can multiply by speed s to get the unnormalized vector.
		vx = (x2 - x1) / (t2 - t1) / s;
		vy = (y2 - y1) / (t2 - t1) / s;
	}

	public void step() {
		t += timeStep;
		x += s * vx * timeStep;
		y += s * vy * timeStep;
		s += a * timeStep;
		if (s < 0.0) {
```

```
                s = 0.0;
            } else if (s > 80.0) {
                s = 80.0;
            }
        }

        public double dist(Car c) {
            return Math.sqrt(Math.pow(c.x - x, 2) + Math.pow(c.y - y, 2));
        }

        public static void main(String[] args) {
            Scanner scan = new Scanner(System.in);

            for (;;) {
                double d = scan.nextDouble();
                if (d < 0) {
                    break;
                }
                Car car1 = new Car(d, scan.nextDouble(),
                        scan.nextDouble(),
                        scan.nextDouble(), scan.nextDouble(),
                        scan.nextDouble(), scan.nextDouble(), scan.nextDouble());
                Car car2 = new Car(scan.nextDouble(), scan.nextDouble(),
                        scan.nextDouble(),
                        scan.nextDouble(), scan.nextDouble(),
                        scan.nextDouble(), scan.nextDouble(), scan.nextDouble());
                double globeTime = Math.max(car1.t, car2.t);
                double finishTime = globeTime + 30.0;
                while (globeTime <= finishTime) {
                    if (car1.dist(car2) < 19) {
                        System.out.println("1");
                        break;
                    }
                    car1.step();
                    car2.step();
                    globeTime += timeStep;
                }
                if (globeTime > finishTime) {
                    System.out.println(0);
                }
            }
        }
}
```

String Manipulation

Example

An example of a string manipulation problem is the problem **Palindrometer.** The problem here is to determine how many miles must be driven until a car's odometer reads as a complete palindrome. In this case, we have to check that the string representation of the odometer's value is a palindrome.

```
import java.util.*;

public class Palindrometer {
    public static void main(String[] args) {
        (new Palindrometer()).main();
    }

    public void main() {
        Scanner in = new Scanner(System.in);

        for (;;) {
            String line = in.nextLine();
            if (line.equals("0")) break;

            int n = Integer.parseInt(line);
            int len = line.length();
            int original = n;

            while (!palin(n, len)) {
                n++;
            }
            System.out.println(n - original);
        }
    }

    public boolean palin(int n, int len) {
        char[] num = String.format("%0" + len + "d", n).toCharArray();

        for (int i = 0; i < len; i++) {
            if (num[i] != num[len-i-1]) return false;
        }
        return true;
    }
}
```

Maps

map, **symbol table**, or **dictionary** is an abstract data type composed of a collection of key, value pairs, such that each possible key appears at most once in the collection.

Operations associated with this data type allow:

- the addition of pairs to the collection
- the removal of pairs from the collection
- the modification of the values of existing pairs
- the lookup of the value associated with a particular key

The dictionary problem is a classic computer science problem: the task of designing a data structure that maintains a set of data during 'search' 'delete' and 'insert' operations. A standard solution to the dictionary problem is a hash table; in some cases it is also possible to solve the problem using directly addressed arrays, binary search trees, or other more specialized structures.

In Java, we typically use HashMap<Key, Value> to implement a map.

Example

In the practice programming competition, we looked at **Politics**, a problem that asked us to find for each candidate and list of supporters that supported those candidates. Here, we had to use two data structures, a list and a map because maps do not preserve order of the keys that are entered.

Here, it is advantageous to use a map, because HashMaps have O(1) lookup time, making the solution linear in performance.

```
import java.util.*;

public class Politics {

	public static void main(String[] args) {

		Scanner scan = new Scanner(System.in);
		String line = scan.nextLine();
		Scanner lineScan = new Scanner(line);
		int n = lineScan.nextInt();
		int m = lineScan.nextInt();

		while (n > 0) {
			HashMap<String, ArrayList<String>> map = new HashMap<>();
			ArrayList<String> names = new ArrayList<>();
			for (int i = 0; i < n; i++) {
				String candidate = scan.nextLine().trim();
				names.add(candidate);
				map.put(candidate, new ArrayList<String>());
			}
```

```
                for (int i = 0; i < m; i++) {
                        lineScan = new Scanner(scan.nextLine());
                        String supporter = lineScan.next();
                        String candidate = lineScan.next();
                        if (map.containsKey(candidate)) {
                                map.get(candidate).add(supporter);
                        } else {
                                ArrayList<String> name = new ArrayList<>();
                                name.add(supporter);
                                map.put(candidate, name);
                                names.add(candidate);
                        }
                }
                for (int i = 0; i < names.size(); i++) {
                        ArrayList<String> supporters = map.get(names.get(i));
                        for (int j = 0; j < supporters.size(); j++) {
                                System.out.println(supporters.get(j));
                        }
                }
                line = scan.nextLine();
                lineScan = new Scanner(line);
                n = lineScan.nextInt();
                m = lineScan.nextInt();
        }

    }

}
```

Comparator (Sorting) Problems

The sorting problem is a real problem in Computer Science; however, in Java, we have something called Comparator that allows us to sort a collection using a given object to compare the Objects within the collection.

A comparator consists of two methods, compare(T o1, T o2) and equals(Object o). For equals, always return false. This checks to see if the comparator is equal to another comparator, essentially useless for the programming competition. All of your work should go in the compare() method. It essentially takes the value of o1 and *subtracts* it from o2, returning a positive number of o1 is “larger” than o2, negative if o1 is “smaller” than o2, or 0 if they are equal.

Example

The best example of a problem where comparator is used was the **Sort Me** problem given during the second competition. Sort Me gave you an ordering of the alphabet, then asked you to sort the words based on that ordering instead of the standard a, b, c, ordering.

```
import java.util.*;

public class SortMe {

    public static void main(String[] args) {
        int count = 1;
        Scanner scan = new Scanner(System.in);
        for (int num = scan.nextInt(); num > 0; ) {
            char[] values = scan.next().toCharArray();
            final int[] alphabet = new int[256];
            for (int i = 0; i < values.length; i++) {
                alphabet[values[i]] = i;
            }
            ArrayList<String> strings = new ArrayList<>();
            for (int i = 0; i < num; i++) {
                strings.add(scan.next());
            }
            Collections.sort(strings, new Comparator<String>() {
                public int compare(String s1, String s2) {
                    int loop = Math.min(s1.length(), s2.length());
                    for (int i = 0; i < loop; i++) {
                        int diff = alphabet[s1.charAt(i)] - alphabet[s2.charAt(i)];
                        if (diff != 0) {
                            return diff;
                        }
                    }
                    return s1.length() - s2.length();
                }

                public boolean equals(Object o) {
                    return false;
```

```
                }
            });

            System.out.println("year " + count++);
            for (int i = 0; i < num; i++) {
                System.out.println(strings.get(i));
            }
            num = scan.nextInt();
        }

    }

}
```

Example

A more sophisticated example of sorting problems is the problem **Balloons.** In Balloons, we must find the optimum way for balloons to be run out of one of two rooms to a set of tables that are a given distance away from each room.

```
import java.util.*;

public class Balloons {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n, a, b;

        while(true){
            int distance = 0;
            ArrayList<Team> tourney = new ArrayList<Team>();
            n = scan.nextInt();
            a = scan.nextInt();
            b = scan.nextInt();

            if (n == 0) {
                break;
            }

            for (int i = 0 ; i < n; i++){
                tourney.add(new Team(scan.nextInt(), scan.nextInt(), scan.nextInt()));
            }

            Collections.sort(tourney);

            for (Team team : tourney){
                int k = team.k;
                int dA = team.dA;
                int dB = team.dB;
                if (dA < dB){
                    int bl = Math.min(k,a);
                    a -= bl;
                    k -= bl;
```

```
                    distance += bl*dA;
                } else if (dA > dB) {
                    int bl = Math.min(k,b);
                    b -= bl;
                    k -= bl;
                    distance += bl*dB;
                }
                while (k > 0) {
                    if (a > 0) {
                        int bl = Math.min(k, a);
                        a -= bl;
                        k -= bl;
                        distance += bl * dA;
                    } else {
                        int bl = Math.min(k, b);
                        b -= bl;
                        k -= bl;
                        distance += bl * dB;
                    }
                }
            }
            System.out.println(distance);
        }
    }

    public static class Team implements Comparable {
        int k, dA, dB;
        Team(int ik, int idA, int idB) {
            k = ik;
            dA = idA;
            dB = idB;
        }

        public int compareTo(Object other) {
            Team o = (Team) other;
            return -Math.abs(dA - dB) + Math.abs(o.dA - o.dB);
        }

        public String toString() {
            return k + " " + dA + " " + dB;
        }
    }
}
```

Union-Find *Useful for Graph Coloring*

A disjoint-set data structure, also called a **union-find** data structure or merge-find set, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It supports two useful operations:

- *Find*: Determine which subset a particular element is in. Find typically returns an item from this set that serves as its "representative"; by comparing the result of two Find operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.

Union-Find has three functions, MakeSet(), to make a set containing only one element, Union() to merge two sets, and Find(), to find the representative of the set that *x* belongs to.

```
function MakeSet(x)
     x.parent := x
     x.rank   := 0

function Union(x, y)
     xRoot := Find(x)
     yRoot := Find(y)
     if xRoot == yRoot
         return
     // x and y are not already in same set. Merge them.
     if xRoot.rank < yRoot.rank
         xRoot.parent := yRoot
     else if xRoot.rank > yRoot.rank
         yRoot.parent := xRoot
     else
         yRoot.parent := xRoot
         xRoot.rank := xRoot.rank + 1

function Find(x)
     if x.parent != x
        x.parent := Find(x.parent)
     return x.parent
```