Does God play Tetris? - Team reference document

Program submission checklist:

1. Works on sample inputs given.
2. Works on other sensible inputs.
3. Works on pathological inputs/corner cases.
4. Works in time on the largest possible inputs.
5. Works within memory limit (if given) use -Xmx128m for a limit of 128mb for example.
6. Compiles! (with warnings on! -Xlint)
7. No debug outputs!

Code

Big sample

```
import java.io.*;
import java.util.*;
import java.math.*;
public class samplecode {
  public static void debug(String s) {
    System.out.printf(">>>%s>>>\n", s); //Comment this out to kill n birds with two /
  }
  public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String s1 = br.readLine();
    int a = Integer.parseInt(s1.split(" ")[0]);
    String[] arr = s1.split(" ");

    // Does God play Tetris? used java.util.Collections, it's super effective!

    // A comparator can be defined by
    class MyClassCmp implements Comparator<MyClass> {
      // Should return a negative integer, zero, or a positive integer as the first
          argument is less than, equal to, or greater than the second respectively
      public int compare(MyClass a, MyClass b) {
        return a.a - b.a; }
      // As far as I can tell this may not be neccessary, but probably best to do anyway
      public boolean equals(MyClass a, MyClass b) {
        return a.a == b.a; }
    }
    // To change an array to a list we can do
    List<String> arrayaslist = Arrays.asList(arr);
    // Or make a general list
    List<MyClass> list = new LinkedList<MyClass>();
    List<MyClass> list2 = new Vector<MyClass>();
    // If we have a comparator already we can do
    Collections.sort(arrayaslist); // or maybe
    Collections.sort(list, new MyClassCmp());
    // If we have a sorted list we can do
    MyClass target = new MyClass(3);
    Collections.binarySearch(list, target, new MyClassCmp());
    SortedSet<MyClass> set = new TreeSet<MyClass>(new MyClassCmp());

    // We can work with arbitrary precision integers as follows:
    BigInteger numb = new BigInteger("12234237843295458912384712938123912546 51");
    numb = numb.add(BigInteger.valueOf(3));
    debug(numb.toString());
```

```
    // In places where code should never be reached we can debug (and submit) with
        assert(false); there, this way we will get an exception rather than dodge
        behaviour.
    debug(arr[0]);
  }
  // Custom classes declared within the main like this:
  static class MyClass {
    int a;
    MyClass(int A) {
      a = A; }
  }
}
```

Graphs

```
import java.util.*;
// This is wrapped up real nice in a class so it's super duper easy to use
// and because typing code of any length is O(1)
public class Graph {
  HashMap<String, LinkedList<String>> adjacents = new HashMap<String, LinkedList<String
      >>();
  // Add an edge (u, v)
  public void add(String u, String v) {
    if (u.equals(v)) {
      if (!adjacents.containsKey(u)) {
        LinkedList<String> newList = new LinkedList<String>();
        adjacents.put(u, newList);
        return;
      }
    }
    if (adjacents.containsKey(u)) {
      if (!adjacents.get(u).contains(v)) {
        adjacents.get(u).add(v);
      }
    } else {
      LinkedList<String> newList = new LinkedList<String>();
      newList.add(v);
      adjacents.put(u, newList);
    }
    if (adjacents.containsKey(v)) {
      if (!adjacents.get(v).contains(u)) {
        adjacents.get(v).add(u);
      }
    } else {
      LinkedList<String> newList = new LinkedList<String>();
      newList.add(u);
      adjacents.put(v, newList);
    }
  }
  // Tests whether (u, v) is an edge
  public boolean edge(String u, String v) {
    return (adjacents.containsKey(u) && adjacents.get(u).contains(v));
  }
  // Returns the degree of node u
  public int degree(String u) {
    return adjacents.containsKey(u) ? adjacents.get(u).size(): -1;
  }
  // BFS
  public void bfs(String root) {
    LinkedList<String> processed = new LinkedList<String>();
    String current = root;
    //while (adjacents.get(current)) {
      // ...
    //}
  }
}
```

One way of representing

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

```
public class graphsample {
  static void example() {
    HashMap<String, LinkedList<String>> adjacents = new HashMap<String, LinkedList<
        String>>();
    adjacents.put("x", new LinkedList<String>());
    adjacents.put("y", new LinkedList<String>());
    adjacents.get("y").add("x");
  }
}
```

Max-Flow

```
```

Shortest path

```
public class shortpath
{
  //Bellman ford
  //Dijsksra
}
```

Min spanning tree

```
```

Edmonds blossom algorithm for perfect matching (min-weight?)

```
```

DFS/BFS

```
public class search {
  static void dfs() {
  }
}
```

Colouring
Connectivity
Minor testing
Eulerian path
Ham path

Number theory

GCD

```
public class gcd {
  static int gcd(int a, int b) {
    int c = 0;
    while(a!=0 && b!=0) {
      c = b;
      b = a%b;
      a = c;
    }
    return a+b;
  }
  static int arrGCD(int[] a) {
    int g = a[0];
    for (int i = 0; i < a.length; i++) {
      g = gcd(a[i],g);
      if (g == 1) break;
    }
    return g;
  }
}
```

$$\mathrm{lcm}(a, b) = ab/\gcd(a, b)$$

Sieve of Eratosthenes

```
public class seive {
  public static boolean[] iscompslessthan(int n) {
    boolean[] iscomp = new boolean[n];
    for (int i = 2; i < Math.sqrt((double)n) + 1; i++) {
      if (iscomp[i]) continue;
      for (int j = i*i; j < n; j+=i) {
        iscomp[j] = true;
      }
    }
    return iscomp;
  }
}
```

Dynamic programming

Discrete knapsack problem

```
```

Combinatorics

Derangements, permutations, other bits

Logic

2-SAT (requires strongly connected components??)

```
```

Strings

Matching

```
public class kmp {
  static int[] createTable(char[] w)  {
    int[] t = new int[w.length];
    int i = 2;
    int j = 0;
    t[0] = -1;
    while (i < w.length) {
      if (w[i - 1] == w[j]) t[i++] = j++ + 1;
      else if (j > 0) j = t[j];
      else t[i++] = j = 0;
    }
    return t;
  }
  static int searchKMP(char[] w, char[] s, int[] t) {
    int m = 0;
    int i = 0;
    while ((m + i < s.length) && (i < w.length)) {
      if(s[m+i] == w[i]) i++;
      else {
        m += i - t[i];
         if (i > 0) i = t[i];
      }
    }
    return (i == w.length) ? m : -1;
  }
}
```

Suffix arrays!

```
```

Geometry

Centroid of set of point $C = (x_1 + x_2 + \cdots + x_k)/k$. Centroid of figure, triangulate into right triangles $X_1, \ldots, X_n$ and compute $C_x = (\sum C_{ix} A_i)/\sum A_i$, $C_y = (\sum C_{iy} A_i)/\sum A_i$ where the centroid of a right triangle perpendicular to the axis is $b/3, h/3$.

Simple data structures

```
public class Point implements Comparable<Point> {
  int x; int y;
  public int compareTo(Point p) {return (x-p.x == 0) ? y-p.y : x-p.x;}// left-bottommost
  public float cross(Point p) { return x*p.y - p.x*y; }
}
```

Convex hull, can be used for: furthest points, polygon containment (P inside Q iff hull(Q) = hull($P \cup Q$)),

```
import java.util.*;
public class convexhull
{
  static final double eps = 0.0000000001;
  static int isAnti(Point x0,Point x1,Point x2) {
    double a = (x1.x-x0.x)*(x2.y-x0.y)-(x2.x-x0.x)*(x1.y-x0.y);
    if (a > eps || -a > eps) return a > 0 ? -1 : 1;
    return 0;
  }
  static int isCloser(Point x0,Point x1,Point x2) {
    double d1 = (x0.x - x1.x)*(x0.x - x1.x) + (x0.y - x1.y)*(x0.y - x1.y);
    double d2 = (x0.x - x2.x)*(x0.x - x2.x) + (x0.y - x2.y)*(x0.y - x2.y);
    if (d1-d2 > eps || d2-d1 > eps) return d1 < d2 ? -1 : 1;
    return 0;
  }
  public static List<Point> hull(List<Point> points) {
    Collections.sort(points);
    final Point p0 = points.get(0);
    points.remove(p0);
    Collections.sort(points, new Comparator<Point>() {
        public int compare(Point p1, Point p2) {
        int a = isAnti(p0,p1,p2);
        if (a != 0) return a;
        return isCloser(p0,p1,p2);
        }});
    int m = points.size();
    for (int i = 1; i < m; i++) { // Remove colinears
      if (isAnti(p0,points.get(i-1),points.get(i)) == 0) {
        points.remove(i-1);
        m--;
      }
    }
    LinkedList<Point> hull = new LinkedList<Point>();
    if (m < 2) return hull; // All colinear, no hull
    hull.push(p0);
    hull.push(points.get(0));
    hull.push(points.get(1));
    for (int i = 2; i < m; i++) {
      while (isAnti(hull.get(0),hull.get(1),points.get(i))<=0) {
        hull.pop();
      }
      hull.push(points.get(i));
    }
    return hull;
  }
  //Andrew monotone chain is faster still...
}
```

Closest pair of points

```
import java.util.*;
public class closestpoints {
  public static Point[] closestPair(Point[] arr){
    Point[] ret = {arr[0],arr[1]};
    Arrays.sort(arr);
    return ret;
```

```
  }
}
```

Pseudocode

BFS

```
procedure BFS(G,v) is
  create a queue Q
  create a set V
  enqueue v onto Q
  add v to V
  while Q is not empty loop
    t <- Q.dequeue()
    if t is what we are looking for then
      return t
    end if
    for all edges e in G.adjacentEdges(t) loop
      u <- G.adjacentVertex(t,e)
      if u is not in V then
        add u to V
        enqueue u onto Q
      end if
    end loop
  end loop
  return none
end BFS
```

DFS

```
procedure DFS(G,v):
  label v as discovered
  for all edges e in G.adjacentEdges(v) do
    if edge e is unexplored then
      w <- G.adjacentVertex(v,e)
      if vertex w is unexplored then
        label e as a discovered edge
        recursively call DFS(G,w)
      else
        label e as a back edge
    label v as explored
```

Dijkstra

```
  function Dijkstra(Graph, source):
      for each vertex v in Graph:                            // Initializations
          dist[v]      := infinity;                          // Mark distances from
              source to v as not yet computed
          visited[v]   := false;                             // Mark all nodes as
              unvisited
          previous[v]  := undefined;                         // Previous node in optimal
              path from source
      end for

      dist[source]  := 0;                                    // Distance from source to
          itself is zero
      insert source into Q;                                  // Start off with the source
           node

      while Q is not empty:                                  // The main loop
          u := vertex in Q with smallest distance in dist[] and has not been visited;
              // Source node in first case
          remove u from Q;
          visited[u] := true                                 // mark this node as visited
```

```
        for each neighbor v of u:
            alt := dist[u] + dist_between(u, v);          // accumulate shortest dist
                from source
            if alt < dist[v] && !visited[v]:
                dist[v]  := alt;                          // keep the shortest dist
                    from src to v
                previous[v]  := u;
                insert v into Q;                          // Add unvisited v into the
                    Q to be processed
            end if
        end for
    end while
    return dist;
endfunction
```

Bellman-Ford

```
procedure BellmanFord(list vertices, list edges, vertex source)
   // This implementation takes in a graph, represented as lists of vertices and edges,
   // and fills two arrays (distance and predecessor) with shortest-path information

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then distance[v] := 0
       else distance[v] := infinity
       predecessor[v] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
```

Ford-Fulkerson

Inputs: Graph G with flow capacity c, a source node s, and a sink node t Output: A flow f from s to t which is a maximum $f(u,v) < -0$ for all edges (u,v) While there is a path p from s to t in G_f, such that $c_f(u,v) > 0$ for all edges (u,v) in p: Find $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$ For each edge $(u,v) \in p$ $f(u,v) < -f(u,v) + c_f(p)$ (Send flow along the path) $f(v,u) < -f(v,u) - c_f(p)$ (The flow might be "returned" later)

Toplogical Sort

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 insert n into 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)

Longest Common Substring

```
function LCSubstr(S[1..m], T[1..n])
    L := array(1..m, 1..n)
    z := 0
    ret := {}
    for i := 1..m
        for j := 1..n
            if S[i] == T[j]
                if i == 1 or j == 1
```

```
                    L[i,j] := 1
                else
                    L[i,j] := L[i-1,j-1] + 1
                if L[i,j] > z
                    z := L[i,j]
                    ret := {S[i-z+1..i]}
                elif L[i,j] == z
                    ret := ret union {S[i-z+1..i]}
            else L[i,j]=0;
    return ret
```

Point-in-Polygon Test

One simple way of finding whether the point is inside or outside a simple polygon is to test how many times a ray, starting from the point and going ANY fixed direction, intersects the edges of the polygon. If the point in question is not on the boundary of the polygon, the number of intersections is an even number if the point is outside, and it is odd if inside.

Polygon Stuff

A convex polygon is trivial to triangulate in linear time, by adding diagonals from one vertex to all other vertices. The total number of ways to triangulate a convex n-gon by non-intersecting diagonals is the $(n-2)$-th Catalan number

Delaunay Triangulation

The most straightforward way of efficiently computing the Delaunay triangulation is to repeatedly add one vertex at a time, retriangulating the affected parts of the graph. When a vertex v is added, we split in three the triangle that contains v, then we apply the flip algorithm. Done naively, this will take $O(n)$ time: we search through all the triangles to find the one that contains v, then we potentially flip away every triangle. Then the overall runtime is $O(n^2)$.

Edit Distance

len_s and len_t are the number of characters in string s and t respectively

```
int LevenshteinDistance(string s, int len_s, string t, int len_t)
{
  /* test for degenerate cases of empty strings */
  if (len_s == 0) return len_t;
  if (len_t == 0) return len_s;

  /* test if last characters of the strings match */
  if (s[len_s-1] == t[len_t-1]) cost = 0;
  else                          cost = 1;

  /* return minimum of delete char from s, delete char from t, and delete char from both
      */
  return minimum(LevenshteinDistance(s, len_s - 1, t, len_t    ) + 1,
                 LevenshteinDistance(s, len_s    , t, len_t - 1) + 1,
                 LevenshteinDistance(s, len_s - 1, t, len_t - 1) + cost);
}
```