Mid-Central US 2015 ICPC Problem Discussion

Mitch Price
Auburn University Programming Team

May 28, 2017

Contents

1 ACM Contest Scoring

1.1 Problem Description

View on Kattis
Tags: Implementation

In ACM Contest scoring, you are provided with the submission logs for a programming team during an ACM ICPC competition. These logs will consist of a variety of entries of the form: `<submission time> <problem> <status>`, where `<submission>` is the number of minutes into the competition at which the submission was made, `<problem>` is an uppercase letter (A-Z) denoting what problem the submission is for, and `<status>` will be either `right` or `wrong`, indicating whether or not the team got the problem correct. The submission logs will be sorted by non-decreasing submission time, and will be terminated with a single line containing only the value `-1`.

Your program must take these logs and output two numbers: the number of problems successfully solved throughout the contest, and the total penalty time assessed. As a reminder, penalty time is assessed as follows:

- No penalty time is assessed for a problem that is not solved during the contest.
- For each incorrect submission on a problem that is eventually solved that occurs prior to the given team's first correct submission for that problem, a penalty of 20 minutes is assessed.
- When the team correctly solves a problem for the first time, a penalty equal to the submission time of the submission time for the solution is assessed.
- The team's total penalty time will be the sum of the penalty times for every problem in the set.

1.2 Algorithm

This is a fairly straightforward implementation. We simply need to assess the rules as written, which can be done with the following approach:

Keep a running total of problems solved and penalty time, initialized to 0. Then, for each submission (in order of increasing submission time):

- If the problem has already been solved, do nothing.
- Otherwise, if the submission is incorrect, increment the number of incorrect solutions to this problem.
- If this problem has not already been solved, but the submission is correct, mark this problem as solved, increment the total number of problems solved, and add the number of incorrect submissions for this problem times 20, plus the submission time to the total penalty time.

After doing this for every problem, we just need to output the totals.

Time Complexity: $\mathcal{O}(N)$
Space Complexity: $\mathcal{O}(P)$
Where **N** is the number of entries and **P** is the number of different problems.

1.3 Implementation

```
import java.util.Scanner;

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

    int problemsSolved = 0, totalPenaltyTime = 0;
    boolean[] solved = new boolean[26];
    int[] incorrectSubmits = new int[26];

    while (true) {
      int submissionTime = in.nextInt();

      if (submissionTime < 0) break;

      // Convert A-Z to 1-26.
      int problem = in.next().charAt(0) - 'A';
      boolean correct = in.next().equals("right");

      // Ignore already solved problems.
      if (solved[problem]) continue;

      if (correct) {
        solved[problem] = true;
        problemsSolved++;
        totalPenaltyTime += 20 *
          incorrectSubmits[problem] + submissionTime;
      } else {
        incorrectSubmits[problem]++;
      }
    }

    System.out.println(problemsSolved + " " +
        totalPenaltyTime);
  }
}
```

2 Dance Recital

View on Kattis
Tags: Brute Force, Permutations

2.1 Problem Description

We are given a list of dance routines, each of which describe a cast of dancers that will be needed to perform each routine. We want to minimize the number of times a dancer will perform in back to back routines (called a "quick change"). For example, in the ordering:

```
ABC
ABEF
DEF
ABCDE
FGH
```

There are 6 quick changes (A and B between the 1st and 2nd dances, E and F between the 2nd and 3rd, and D and E between the 3rd and 4th dances).

However, if we use the ordering:

```
ABEF
DEF
ABC
FGH
ABCDE
```

There are only 2 quick changes (E and F between the 1st and 2nd dances). Given a list of **N** routines ($2 \leq N \leq 10$), consisting of dancers described using only uppercase A-Z letters, determine the minimum number of quick changes.

2.2 Algorithm

Since 10! = 3628800 is rather tractable, we can brute-force this. However, in order to make the computation performed at each step as fast as possible, we will pre-compute the number of quick changes between every pair of dances and store this ahead of time. Then, each step of the brute force calculation just requrires us to iterate over the permuted values in order.

Because brute-forcing all permutations of the given values is not the key part of this excercise, we will rely on our brute-force library to do the heavy lifting.

Time Complexity: $\mathcal{O}(N! + N \cdot P^2)$
Space Complexity: $\mathcal{O}(N(N + P))$
Where **N** is the number of routines ($2 \leq N \leq 10$), and **P** is the number of different dancers ($1 \leq P \leq 26$).

2.3 Implementation

```
import java.util.Scanner;

// From our libary, include the following:
// brute_force/Permutations.java

public class DanceRecital {

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

    // We will record which dancers are present in
    // each dance.
    boolean[][] present = new boolean[N][26];
    for (int i = 0; i < N; i++) {
      for (char c : in.next().toCharArray()) {
        present[i][c-'A'] = true;
      }
    }

    // quickChanges[i][j] will hold the number of
    // quick changes that need to occur between
    // dance i and dance j.
    int[][] quickChanges = new int[N][N];
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        // For each possible dancer:
        for (int dancer = 0; dancer < 26; dancer++) {
          // If the given dancer is present in
          // both dances, that means they must
          // perform a quick change.
          if (present[i][dancer] && present[j][dancer]) {
            quickChanges[i][j]++;
          }
        }
      }
    }

    // Create a PermutationWorker.
    Worker w = new Worker(quickChanges);
    // Run the permutation worker, performing our
    // computation for every dancer permutation.
    Permutations.runPermutations(w, N);
    System.out.println(w.best);
  }
}

class Worker implements Permutations.PermutationWorker {
```

```
  int[][] D;
  int best;

  Worker(int[][] D) {
    this.D = D;
    best = Integer.MAX_VALUE;
  }

  // The permutations library will call this
  // function for each permutation of [0...N].
  // Each time, the associated permutation will be
  // passed in as p.
  public void permutationWork(int[] p) {
    // Calculate the total # of quick changes.
    int cost = 0;
    for (int i = 1; i < p.length; i++) {
      cost += D[p[i]][p[i-1]];
    }
    // Update our best if necessary.
    best = Math.min(best, cost);
  }
}
```

3 Hidden Password

View on Kattis
Tags: Strings

3.1 Problem Description

We are given a password and a message that is supposed to secretly contain the given password. We must determine if the password is successfully hidden in the message. In order for this to be the case, the following must hold true:

- Denote the password as $P = \{c_1, c_2, ..., c_p\}$.
- c_1 should be the first character in the password to appear in the message.
- Let S^{c_i} denote all characters in the message following the character that corresponds to c_i.
- Then the first character in S^{c_i} that appears in $P[(i+1)...p]$ must equal c_{i+1}, for all $1 \leq i \leq p-1$.

3.2 Algorithm

Our algorithm will be a straight-forward implementation of the given check. We will keep a pointer, *passwordIndex* to which character in the password we must find next, and iterate over all characters in the message. For each character c, if $c = P[passwordIndex]$, then we increment *passwordIndex*. Once *passwordIndex* equals $|P|$, we know we have a match. However, if $c \neq P[passwordIndex]$ and $\exists \quad i \quad (passwordIndex < i \leq |P|) \quad s.t. \quad c = P[i]$, then the message and password do not match. Furthermore, if we reach the end of the message, and $passwordIndex < |P|$, then the message and password do not match.

Time Complexity: $\mathcal{O}(|P| \cdot |S|)$
Space Complexity: $\mathcal{O}(|P| + |S|)$
Where $|P|$ $(3 \leq |P| \leq 8)$ is the length of the password and $|S|$ $(10 \leq |S| \leq 40)$ is the length of the message.

3.3 Implementation

```
import java.util.Scanner;

public class HiddenPassword {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    char[] P = in.next().toCharArray();
    char[] S = in.next().toCharArray();

    System.out.println(valid(P, S) ? "PASS" : "FAIL");
  }

  static boolean valid(char[] password, char[] message) {
    int passwordIdx = 0;

    for (char c : message) {
      if (c == password[passwordIdx]) {
        passwordIdx++;
        if (passwordIdx == password.length) {
          return true;
        }
      } else {
        for (int i = passwordIdx + 1; i < password.length
            ; i++) {
          if (c == password[i]) {
            return false;
          }
        }
      }
    }

    return false;
  }
}
```

4 Kitchen Measurements

View on Kattis
Tags: Search, UCS

4.1 Problem Description

We are given a set of N cups ($2 \leq N \leq 5$) of decreasing capacities $c_1, c_2, ..., c_N$. The first (and largest) cup is initially full of water, and all other cups are empty. We are also given a target volume V. Our goal is to end up with exactly V units of water in the first cup, only by pouring water from one cup to another until either the source cup is empty or the destination cup is full.

If this is impossible, need to say so. Otherwise, we should determine the minimum amount of liquid that must be poured to achieve the goal.

4.2 Algorithm

The solution to this can be derived by performing a Uniform Cost Search, or Dijkstra's Algorithm. The focus of this exercise will be on determining what the state space will be, the successor function, and how we will determine whether or not we are at a goal state. Because performing the search is not the focus of this, we will be using the search library to perform the acutal search.

State Space
We will define a state as $\{w_1, w_2, ...w_N\}$, the amount of water in each cup in the given state. Two states $s_1 = \{u_1, ...u_N\}$ and $s_2 = \{v_1, ...v_N\}$ are identical if and only if $\forall\, i$, $1 \leq i \leq N$, $v_i = u_i$.

Goal State
A given state $s = \{w_1, w_2, ...w_N\}$ is a goal state if and only if $w_1 = V$.

Neighbor Function
We can use the following algorithm to find all neighbors:

Let *distance* be a map of neighbors to distance.
for $src \leftarrow 1$ to N **do**
  for $dst \leftarrow 1$ to N **do**
    if $src \neq dst$ **then**
      $pour \leftarrow \min(w_{src}, c_{dst} - w_{dst})$
      if $pour > 0$ **then**
        $\{w'_1, ...w'_N\} \leftarrow \{w_1, ...w_N\}$
        $w'_{src} \leftarrow w'_{src} - pour$
        $w'_{dst} \leftarrow w'_{dst} + pour$
        $distance(\{w'_1, ...w'_N\}) \leftarrow pour$
      end if
    end if
  end for
end for

With all of the following defined, we can rely on the UCS class in the search library to find the optimal procedure, if it exists.

Time Complexity: $\mathcal{O}(N^2 \cdot |S| \cdot \log(|S|))$
Space Complexity: $\mathcal{O}(|S|)$
Where N is the number of cups, and $|S|$ is the number of different states we can explore. This is loosely bound by $\prod_{i=1}^{N} c_i$, but is significantly less than this value in practice.

4.3 Implementation

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

// From our library, include the following:
// search/UCS.java

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

    // Read input.
    int N = in.nextInt();
    int[] sizes = new int[N];
    for (int i = 0; i < N; i++) {
      sizes[i] = in.nextInt();
    }
    int V = in.nextInt();

    // Make input accessible from state class.
    State.size = sizes;
    State.goal = V;

    // Build initial state.
    int[] startWater = new int[N];
    startWater[0] = sizes[0];
    State initial = new State(startWater, 0);

    // Perform UCS. doSearch will return -1
    // in the event of a failure.
    int res = UCS.doSearch(initial);
    System.out.println(res < 0 ? "impossible" : res);
  }
}
```

```
class State implements UCS.UCSState {

  // The capacity of each cup.
  static int[] size;
  // The target volume.
  static int goal;

  // The amount of water in each cup.
  int[] water;
  // The total distance this is from start.
  int dist;

  /**
   * Constructor for State.
   */
  State(int[] water, int dist) {
    this.water = water;
    this.dist = dist;
  }

  @Override
  /**
   * hashCode must be overwritten to allow
   * for the State to be used in HashMaps.
   * This is a necessary method of UCSState.
   */
  public int hashCode() {
    int out = 0;
    for (int w: water) {
      out *= 64;
      out += w;
    }
    return out;
  }

  @Override
  /**
   * equals returns true iff this state is
   * equal to the given object.
   * This is a necessary method of UCSState.
   */
  public boolean equals(Object o) {
    if (o instanceof State) {
      State s = (State) o;
      if (s.water.length != water.length) return false;
      for (int i = 0; i < water.length; i++)
        if (water[i] != s.water[i])
          return false;
      return true;
    }
```

```
    return false;
  }

  @Override
  /**
   * distance returns the total distance this
   * state is from the initial state.
   * This is a necessary method of UCSState.
   */
  public int distance() {
    return dist;
  }

  @Override
  /**
   * neighbors will return a list of all
   * UCSStates reachable from this state.
   * This is a necessary method of UCSState.
   */
  public List<UCS.UCSState> neighbors() {
    List<UCS.UCSState> out = new ArrayList<>();
    for (int src = 0; src < water.length; src++) {
      for (int dst = 0; dst < water.length; dst++) {
        if (src == dst) continue;
        int pour = Math.min(water[src], size[dst] - water
            [dst]);
        if (pour == 0) continue;
        int nDist = dist + pour;
        int[] nWater = Arrays.copyOf(water, water.length)
            ;
        nWater[src] -= pour;
        nWater[dst] += pour;
        out.add(new State(nWater, nDist));
      }
    }
    return out;
  }

  @Override
  /**
   * atDestination returns true iff this
   * state is a goal state.
   * This is a necessary method of UCSState.
   */
  public boolean atDestination() {
    return water[0] == goal;
  }
}
```

5 Line Them Up

View on Kattis
Tags: Strings

5.1 Problem Description

Given a list of unique, uppercase names, determine if the list is completely in increasing alphabetical order, completely decreasing alphabetically, or neither.

5.2 Algorithm

We can rely on Java's built-in String.compareTo function to solve this. First, we will get the result of compareTo between the 2nd and first list elements.

Then, for every other adjacent pair of strings, we can verify that the result of compareTo is on the same side of 0 as the initial comparison (i.e. they must be either both positive or both negative). If it isn't, the answer is neither. If however, the second value is larger than the first for all pairs, we are ascending. Finally, if the second value is smaller than the first for all pairs, we are descending.

Time Complexity: $\mathcal{O}(L \cdot N)$
Space Complexity: $\mathcal{O}(L \cdot N)$
Where L is the length of the longest string.

5.3 Implementation

```
import java.util.Scanner;

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

    int N = in.nextInt();
    String[] L = new String[N];
    for (int i = 0; i < N; i++) L[i] = in.next();

    int res = getOrder(L);

    if (res < 0) System.out.println("DECREASING");
    else if (res > 0) System.out.println("INCREASING");
    else System.out.println("NEITHER");
  }

  static int getOrder(String[] L) {
    int order = L[1].compareTo(L[0]);

    for (int i = 2; i < L.length; i++) {
      int comparison = L[i].compareTo(L[i-1]);

      // If they aren't both positive or both negative:
      if (comparison * order < 0) {
        return 0;
      }
    }

    return order;
  }
}
```

6 Mosaic

View on Kattis
Tags: Bit Manipulation, Search

6.1 Problem Description

We need to create a mosaic from black and white squares. There are three possible types of squares:

- A square that is completely black.
- A square that is completely white.
- A half-white, half-black square, divided along the diagonal of the square. These can be rotated as necessary.

We are given a diagram with the location of every completely black square. Additionally, each of these completely black squares may contain a number on it: the number of half-white, half-black squares that are adjacent (up, down, left, or right) to this square.

An example starting diagram is given in Figure 1.

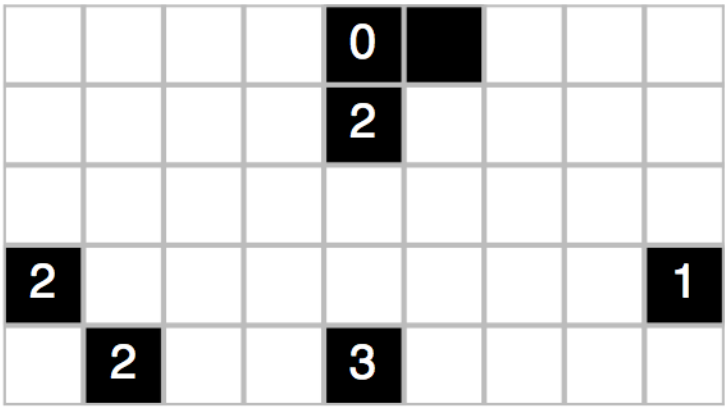

Figure 1: Sample Initial Diagram

Each starting diagram will correspond to exactly 1 completed valid mosaic, which must satisfy the following criteria:

- If a tile in the initial diagram has number on it, that number must be realized in the completed mosaic (i.e. if a black square has "2" in the initial diagram, that tile must be adjacent to exactly 2 half and half tiles).
- Every white shape must be a rectangle (possibly rotated).

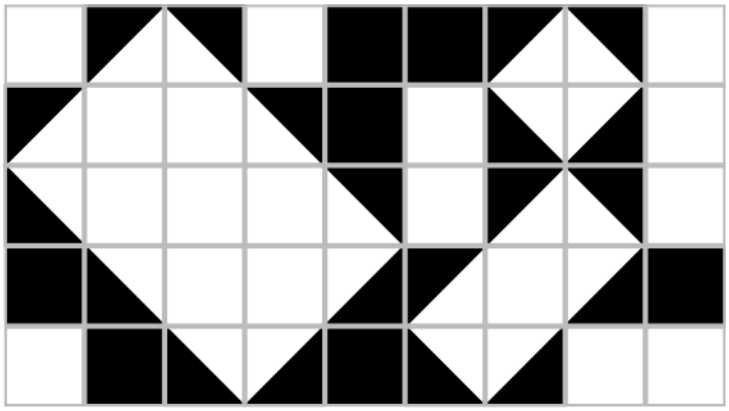

Figure 2: Completed mosaic for the starting diagram in Figure 1

6.2 Algorithm

The solution for this will be pretty straightforward: we will recursively try all placements of tiles until we find one that works. However, there is no way that a naive search will pass the time limit, so we will need to do some pruning.

One easy way of pruning the search space while also ensuring that all white areas in the result will be rectangular is to examine all 2x2 squares containing the tile you just placed, and ensure that they contain only 180 degree and 90 degree angles. Additionally, the triangle count contraints for black tiles adjacent to the tile just placed should also be satisfied.

To verify the integrity of a 2x2 square, we can traverse all interior sides of the selection (they should form a plus-sign), keeping track of whether they are black or white. At each shared edge, we will examine the color from both tiles. By doing this in order (either clockwise or counter-clockwise), we can look for any angles that are not 180 or 90 degrees. Those would have the format of `<black edge> <n white edges> <black edge>`, where n is either odd (45, 135, 225, or 315 degrees), or 6 (270 degrees).

For example, in Figure 3, we have the edges:

```
black, black, white, black, black, black, white, white
```

As we can see, starting at the second edge there is the subsequence:

```
black, white, black
```

Which, being a 45 degree angle, matches the pattern of an invalid square.

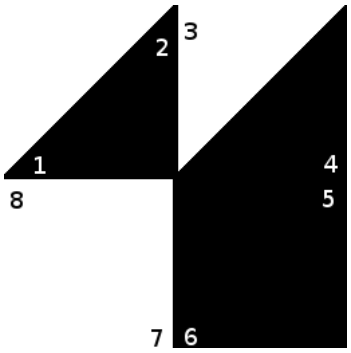

Figure 3: Example 2x2 selection

In order to improve performance even more, we can create a hash function that will convert any 2x2 square to a unique number and use that to precompute whether every combination is valid up front, before storing the results in a boolean array or set.

Another option to simplify our logic is to surround the entire puzzle in a region of black squares at the beginning, which can allow us to get rid of boundary checks.

Time Complexity: $\mathcal{O}(5^{R \cdot C})$
Space Complexity: $\mathcal{O}(R \cdot C)$
Where R is the number of rows and C is the number of columns. Due to our pruning, however, the actual implementation will be much faster than this time complexity, takes less than a second to process even the largest inputs.

6.3 Implementation

```
import java.util.*;
import java.io.*;
public class Mosaic {
  static final int WHITE = 0; // Completely black.
  static final int BLACK = 1; // Completely white.
  static final int TRITL = 2; // Top Left half is black.
  static final int TRITR = 3; // Top Right half is black.
  static final int TRIBR = 4; // Bottom Right half is
      black.
  static final int TRIBL = 5; // Bottom Left half is
      black.
  static final int UNSET = 6; // Un-determined tile.

  static int R; // Number of rows.
  static int C; // Number of columns.
  static int numTriangles = 0; // Count number of
      triangles.

  static final int TOP = 0;
  static final int RIGHT = 1;
  static final int BOTTOM = 2;
  static final int LEFT = 3;

  // Arbitary large number, the count of triangles
      adjacent
  // adjacent to a tile with an unknown count.
  static final int UNKNOWN_COUNT = 100;

  // Hardcoded listing of which sides are black.
  static int[] sides = new int[] {
    0b0000, 0b1111, 0b1001, 0b0011, 0b0110, 0b1100
  };

  // The locations of the top-left of all 2x2 tiles
  // containing a given tile, relative to that tile.
  static int[][] sqStart = new int[][] {
    {0, 0}, {-1, 0}, {0, -1}, {-1, -1}
  };

  // Up, Down, Left, and Right.
  static int[][] dirs = new int[][] {
    {0, -1}, {0, 1}, {-1, 0}, {1, 0}
  };

  // Will hold precomputed results of each 2x2 tile
      working.
  static boolean[] valid = new boolean[1<<12];
```

```
public static void main(String[] args) throws Exception
    {

  // Precompute valid 2x2 tiles.
  computeValid();

  // Read in the input, surrounding it with black
  // tiles.
  BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));
  String[] L = br.readLine().split(" ");
  C = Integer.parseInt(L[0]);
  R = Integer.parseInt(L[1]);

  int[][] G = new int[R+2][C+2];
  int[][] count = new int[R+2][C+2];

  for (int i = 0; i < G.length; i++) {
    Arrays.fill(G[i], BLACK);
    Arrays.fill(count[i], UNKNOWN_COUNT);
  }

  for (int i = 0; i < R; i++) {
    char[] input = br.readLine().toCharArray();
    for (int j = 0; j < C; j++) {
      char c = input[j];
      if (c == '.') {
        G[i+1][j+1] = UNSET;
      } else if (c != '*') {
        count[i+1][j+1] = c - '0';
      }
    }
  }

  // Fill in the board and print result.
  solve(G, count, 1, 1);
  System.out.println(numTriangles);
}
```

```
/**
 * solve will recursively attempt to fill g will a
 * set of tiles, working left to right, then top down
 * starting at (r, c).
 */
static boolean solve(int[][] g, int[][] count, int r,
    int c) {

  // Passed all rows. We did it!
  if (r > R) return true;

  // Compute what the next row and column will be.
  int nextRow = r;
  int nextCol = c + 1;

  if (nextCol > C) {
    nextCol = 1;
    nextRow++;
  }

  // If it is already determined to be black, move to
  // the next case.
  if (g[r][c] == BLACK) {
    return solve(g, count, nextRow, nextCol);
  } else {
    // Otherwise, try all non-black tiles.
    for (int t = WHITE; t < UNSET; t++) {
      if (t == BLACK) continue;

      // Set this tile.
      g[r][c] = t;
      if (t != WHITE) numTriangles++;

      // If it is invalid, recurse to keep filling.
      if (validPlacement(g, count, r, c)) {
        if (solve(g, count, nextRow, nextCol)) return
            true;
      }

      // If we reached this, this was not the
      // correct guess. Clear it.
      g[r][c] = UNSET;
      if (t != WHITE) numTriangles--;
    }
  }

  // Nothing we tried worked. Something before this
  // must be at fault.
  return false;
}
```

```
/**
 * validPlacement returns true if the tile at (r, c) in
 * the mosaic g, with the given counts is valid.
 */
static boolean validPlacement(int[][] g, int[][] count,
  int r, int c) {

  // Verify that all 2x2 squares containing (r, c)
  // either contain unset tiles or are valid.
  for (int[] sq : sqStart) {
    int h = hash(g, r + sq[0], c + sq[1]);
    if (h >= 0 && !valid[h]) {
      return false;
    }
  }

  // Verify that if there are any adjacent tiles with a
  // tile count requirement, that the requirement is
  // still satisfied.
  for (int[] move : dirs) {
    int nr = r + move[0];
    int nc = c + move[1];
    // For all adjacent tiles, if there is a
        requirement:
    if (count[nr][nc] != UNKNOWN_COUNT) {
      int adj = numTrianglesAdjacent(g, nr, nc);
      // Are there too many triangles adjacent to this
          tile?
      if (adj > count[nr][nc]) {
        return false;
      }

      // Are there not enough, even if all unset
          adjacent
      // to this become triangles?
      if (adj + numUnsetAdjacent(g, nr, nc) < count[nr
          ][nc]) {
        return false;
      }
    }
  }

  return true;
}
```

```
/**
 * numTrianglesAdjacent will return the number of
    triangle
 * tiles adjacent to (r, c) in g.
 */
static int numTrianglesAdjacent(int[][] g, int r, int c
   ) {
  int count = 0;
  for (int[] move : dirs) {
    int v = g[r + move[0]][c + move[1]];
    if (v >= TRITL && v <= TRIBL) count++;
  }
  return count;
}

/**
 * numUnknownAdjacent will return the number of unset
 * tiles adjacent to (r, c) in g.
 */
static int numUnsetAdjacent(int[][] g, int r, int c) {
  int count = 0;
  for (int[] move : dirs) {
    int v = g[r + move[0]][c + move[1]];
    if (v == UNSET) count++;
  }
  return count;
}

/**
 * hash returns a number uniquely identifying the 2x2
    set
 * of tiles in g starting at r, c if there are no unset
 * tiles. If there are any unset tiles, it will return
    a
 * negative number.
 */
static int hash(int[][] g, int r, int c) {
  int out = 0;
  for (int i = 0; i < 4; i++) {
    out = out << 3;
    int v = g[r + (i&1)][c + (i >> 1)];
    if (v == UNSET) return -1;
    out += v;
  }
  return out;
}
```

```
/**
 * computeValid will pre-compute which 2x2 tiles are
    valid.
 */
static void computeValid() {

  // Brute-force all combinations of 4 tiles.
  for (int q1 = 0; q1 < 6; q1++) {
    for (int q2 = 0; q2 < 6; q2++) {
      for (int q3 = 0; q3 < 6; q3++) {
        for (int q4 = 0; q4 < 6; q4++) {

          // Build a 2x2 grid from these 4 tiles.
          int[][] quad = new int[][] {
            {q1, q2}, {q3, q4}
          };

          // If it is valid, set the corresponding
              entry.
          if (quadIsValid(quad)) {
            valid[hash(quad, 0, 0)] = true;
          }
        }
      }
    }
  }
}

/**
 * whiteSide returns true if the given type of tile
 * is white on the given side.
 */
static boolean whiteSide(int tile, int side) {
  return (sides[tile] & (1 << side)) == 0;
}
```

```
  /**
   * quadIsValid returns true if the given 2x2 tile
   * arrangement is valid. Note that this may say
   * that the given arrangement is valid, when it
   * could contain an error, but that error is
   * guaranteed to be caught, due to an overlapping
   * 2x2 pattern being invalid.
   */
  static boolean quadIsValid(int[][] q) {

    boolean[] isWhite = new boolean[16];
    // Look at all 8 interior sides of the 2x2 group,
    // moving clockwise. (each side is looked at
    // once for both neighboring tiles).
    isWhite[0] = whiteSide(q[0][0], BOTTOM);
    isWhite[1] = whiteSide(q[0][0], RIGHT);
    isWhite[2] = whiteSide(q[0][1], LEFT);
    isWhite[3] = whiteSide(q[0][1], BOTTOM);
    isWhite[4] = whiteSide(q[1][1], TOP);
    isWhite[5] = whiteSide(q[1][1], LEFT);
    isWhite[6] = whiteSide(q[1][0], RIGHT);
    isWhite[7] = whiteSide(q[1][0], TOP);

    // Duplicate those 8, just to catch any issues
    // near the start/end of the pattern.
    for (int i = 8; i < 16; i++)
      isWhite[i] = isWhite[i-8];

    int wCount = 0, bCount = 0;
    // In the event of <black> <odd # white> <black>
        sides
    // we have an angle that is not 90/180 degrees.
    // In the event of <black> <6 white> <black> sides,
    // we have white components that form an 'L'.
    // Return false in either case.
    for (int i = 0; i < 16; i++) {
      if (isWhite[i]) {
        wCount++;
      } else {
        if (bCount > 0 && (wCount % 2 == 1 || wCount ==
            6))
          return false;
        bCount++;
        wCount = 0;
      }
    }
    // We found no other issues.
    return true;
  }
}
```

7 Pyro Tubes

View on Kattis
Tags: Bit Manipulation, Implementation

7.1 Problem Description

We are given a list of values of brightness to be achieved by illuminating a selection of tubes. The first tube has a brightness of 1, and each subsequent tube has double the brightness of the previous tube. For each value, we need to calculate the number of alternative brightness values. For an alternative value to be valid, it must satisfy the following:

- The alternate value must be greater than the original.
- The alternate must be another value in the given list.
- The alternate must be achieved by changing no more than 2 tubes from the original

7.2 Algorithm

The key to this is to realize that the tubes simply correspond to bits in the binary representation of each value. To solve this, we can iterate over all combinations of 1 or 2 bits in the given range (2^0 to 2^{17}, per the problem statement). We can simply xor the original value with these bits to change them. If the result is greater than the original, and is in a set containing all values in the list, we can increment the counter for this value.

Time Complexity: $\mathcal{O}(N \cdot \log^2 L)$
Space Complexity: $\mathcal{O}(N)$
Where N is the number of tubes, and L is the largest possible brightness value.

7.3 Implementation

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

public class PyroTubes {
  public static void main(String[] args)
    throws Exception {

    BufferedReader br = new BufferedReader(
      new InputStreamReader(System.in));

    PrintWriter out = new PrintWriter(
        new BufferedWriter(
        new OutputStreamWriter(System.out)));

    // Store all values here.
    List<Integer> nums = new ArrayList<>();
    Set<Integer> nSet = new HashSet<>();

    // Read in the input.
    int L = Integer.parseInt(br.readLine());
    while(L > 0) {
      nums.add(L);
      nSet.add(L);
      L = Integer.parseInt(br.readLine());
    }

    // Iterate over values.
    for (int l : nums) {

      int count = 0;

      // First bit to swap.
      for (int i = 0; i < 18; i++) {
        int nl = l ^ (1 << i);
        if (nl > l && nSet.contains(nl)) count++;

        // Second bit to swap.
        for (int j = i+1; j < 18; j++) {
          int nnl = nl ^ (1 << j);
          if (nnl > l && nSet.contains(nnl)) count++;
        }
      }
      // Output result.
      out.println(l + ":" + count);
    }
    out.flush();
  }
}
```

8 Square Deal

View on Kattis
Tags: Brute Force, Geometry

8.1 Problem Description

Given the width and height of 3 rectangles, can you arrange the rectangles to form a square?

8.2 Algorithm

Because there are exactly 3 rectangles, there are only 2 cases that we need to consider:

- All rectangles can be stacked in a line together (Figure 4).
- 2 rectangles are stacked together. The other will lie across them (Figure 5).

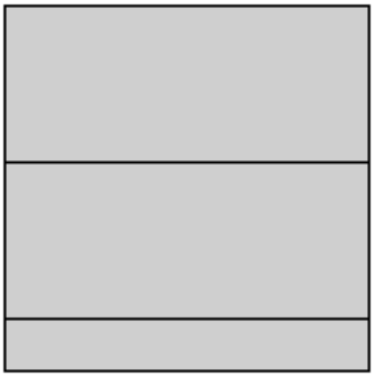

Figure 4: Case 1

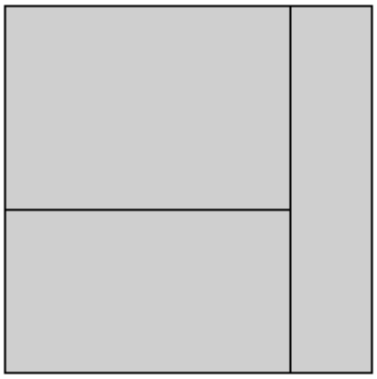

Figure 5: Case 2

We can brute force this quite easily. All we need to do is examine all 8 combinations of vertical/horizontal rotations, and then check both cases for each combination. The first case is one check, and for the second case, we will need to try all 3 rectangles as the one that isn't stacked with the others.

Time Complexity: $\mathcal{O}(2^N \cdot N)$
Space Complexity: $\mathcal{O}(N)$
Where N is the number of rectangles (3).

8.3 Implementation

```
import java.util.*;

public class SquareDeal {
  public static void main(String[] args) {
    Rect[] R = new Rect[3];
    Scanner in = new Scanner(System.in);
    // Read in input.
    for (int i = 0; i < 3; i++) {
      R[i] = new Rect(in.nextInt(), in.nextInt());
    }
    System.out.println(square(R) ? "YES" : "NO");
  }

  static boolean square(Rect[] R) {
    // Try all 8 combinations of rotations.
    for (int i = 0; i < 8; i++) {
      // T will hold a combination.
      Rect[] T = new Rect[3];
      for (int j = 0; j < 3; j++) {
        // Base the orientation of the jth triangle
        // on the jth bit of i.
        T[j] = R[j].rotate(i & (1 << j));
      }
      // Test if all 3 can be lined up.
      if (T[0].W + T[1].W + T[2].W == T[0].H &&
          T[0].H == T[1].H && T[1].H == T[2].H) {
        return true;
      }
      // Test if you can put 2 together and the
      // other can go across them.
      for (int j = 0; j < 3; j++) {
        // Wrap around.
        int r1 = j;
        int r2 = (j+1) % 3;
        int r3 = (j+2) % 3;

        if (T[r1].H == T[r2].H &&
            T[r1].W + T[r2].W == T[r3].H &&
            T[r1].H + T[r3].W == T[r3].H) {
          return true;
        }
      }
    }
    // Nothing worked. It's impossible.
    return false;
  }
}
```

```
class Rect {
  int W;
  int H;

  Rect(int W, int H) {
    this.W = W;
    this.H = H;
  }

  /**
   * Rotate this rectangle. If the given direction
   * is 0, it will return the original direction.
   * Otherwise, it will rotate it 90 degrees.
   */
  public Rect rotate(int direction) {
    if (direction == 0) {
      return this;
    } else {
      return new Rect(H, W);
    }
  }
}
```

9 The Agglomerator

View on Kattis
Tags: Algebra, Geometry

9.1 Problem Description

We are given the initial positions, initial velocities, and sizes of a collection of circular droplets in a 2-D grid. When two droplets touch one another, they will merge into a single droplet. This new droplet's location and velocity will be the area-weighted average of the 2 initial droplet's locations and velocities. We need to simulate the movement of the droplets until no more merges will occur. Luckily, the droplets have the following constraints:

- The original droplets do not touch each other.
- When a new droplet is formed from a merge, the new droplet will be a distance of at least 0.001 from all other droplets.
- Changing the radius of any drop by ± 0.001 will not affect whether it collides with any other drop.
- All collisions will be at least 0.001 seconds apart.
- No droplets will merge after $t = 10^9$ seconds.

Given these constraints, determine how many droplets will be left at the end, and the time of the last merge.

9.2 Algorithm

To find when (and if) two drops a and b will collide, we can use some basic algebra. We need to solve for the first time t where the drop's distance is equal to the sum of their radii:

$$\left| \left(\begin{pmatrix} x_a \\ y_a \end{pmatrix} + t \cdot \begin{pmatrix} v_{ax} \\ v_{ay} \end{pmatrix} \right) - \left(\begin{pmatrix} x_a \\ y_a \end{pmatrix} + t \cdot \begin{pmatrix} v_{ax} \\ v_{ay} \end{pmatrix} \right) \right| = r_a + r_b \tag{1}$$

To simplify calculatations, let us say $\begin{pmatrix} x_a \\ y_a \end{pmatrix} + \begin{pmatrix} x_b \\ y_b \end{pmatrix} = \begin{pmatrix} d_x \\ d_y \end{pmatrix}$, $\begin{pmatrix} v_{ax} \\ v_{ay} \end{pmatrix} + \begin{pmatrix} v_{bx} \\ v_{by} \end{pmatrix} = \begin{pmatrix} d_{vx} \\ d_{vy} \end{pmatrix}$, and $r_a + r_b = s_r$.

Then, we can reduce the given equality to a quadratic equation:

$$\left(d_{vx}^2 + d_{vy}^2\right) \cdot t^2 + \left(2d_x d_{vx} + 2d_y d_{vy}\right) \cdot t + \left(d_x^2 + d_y^2 - s_r^2\right) = 0 \tag{2}$$

and solve using the quadratic formula $\left(-b \pm \sqrt{b^2 - 4ac}\right)/2a$, where a is the coefficient of t^2, b is the coefficient of t, and c is the constant term.

Because of the constraint that there will be no droplets intersecting at the start or immediately following a merge, we know that if we look for collisions at those times, the two solutions for this equation will always have the same sign. Thus, we can always use the lower solution, which (because a must be positive) is guaranteed to be the solution in which the root of the determinant is subtracted. If the determinant or the solution is negative, then the pair of droplets will not intersect in the future. Otherwise, they will intersect after the given amount of time.

Now that we have a means for finding the time of intersection, we can simulate the droplets. This can be done as follows:

Let D be a set containing all droplets.
$time \leftarrow 0$
while Another intersection will occur **do**
    Let a, b be the droplets that will intersect next, after time t.
    for d in D **do**
        $d \leftarrow advance(d, t)$
    end for
    $time \leftarrow time + t$
    $D \leftarrow D - a$
    $D \leftarrow D - b$
    $D \leftarrow D + merge(a, b)$
end while
Return $|D|$, $time$

Advancing and merging drops simply involves updating the positions of the droplets as specified in the problem statement.

Time Complexity: $\mathcal{O}(N^3)$
Space Complexity: $\mathcal{O}(N)$
Where N is the number of droplets ($2 \leq N \leq 100$).

9.3 Implementation

```
import java.util.*;

public class Agglomerator {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    double time = 0;
    // Read input.
    int N = in.nextInt();
    List<Drop> D = new ArrayList<>();
    for (int i = 0; i < N; i++) {
      D.add(new Drop(in.nextInt(), in.nextInt(),
        in.nextInt(), in.nextInt(), in.nextInt(), 0));
    }
    while(true) { // Repeat until done.
      // The drops to merge, how long until merging.
      int d1 = -1, d2 = -1;
      double nextTime = Integer.MAX_VALUE;
      // Look at all pairs of drops for next merge.
      for (int i = 0; i < D.size(); i++) {
        for (int j = i+1; j < D.size(); j++) {
          double dt = Drop.collisionTime(D.get(i),
              D.get(j));
          if (dt > 0 && dt < nextTime) {
            nextTime = dt; d1 = i; d2 = j;
          }
        }
      }
      // No more merges. We are done.
      if (d1 < 0) break;
      // Update the list of drops.
      List<Drop> newD = new ArrayList<>();
      newD.add(Drop.merge(
          D.get(d1).afterTime(nextTime),
          D.get(d2).afterTime(nextTime)));
      time += nextTime;
      for (int i = 0; i < D.size(); i++) {
        // Skip the drops to be merged.
        if (i == d1 || i == d2) continue;
        newD.add(D.get(i).afterTime(nextTime));
      }
      D = newD;
    }
    System.out.format("%d %.3f\n", D.size(), time);
  }
}
```

```
/**
 * Drop represents a droplet.
 */
class Drop {
  double x;    // X Coordinate
  double y;    // Y Coordinate
  double vx;   // X Velocity
  double vy;   // Y Velocity
  double r;    // Radius
  double t;    // Time at this postion.

  /**
   * Constructor for drop.
   */
  Drop(double x, double y, double vx, double vy,
      double r, double t) {

    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.r = r;
    this.t = t;
  }

  /**
   * Combine 2 drops together, creating
   * a new drop.
   */
  static Drop merge(Drop a, Drop b) {
    double a1 = a.r * a.r;
    double a2 = b.r * b.r;
    double totalArea = a1 + a2;
    double nr = Math.sqrt(totalArea);
    double nx = (a.x * a1 + b.x * a2) / totalArea;
    double ny = (a.y * a1 + b.y * a2) / totalArea;
    double nvx = (a.vx * a1 + b.vx * a2) / totalArea;
    double nvy = (a.vy * a1 + b.vy * a2) / totalArea;
    return new Drop(nx, ny, nvx, nvy, nr, a.t);
  }

  /**
   * Produce a new drop, representing this drop
   * after dt seconds.
   */
  Drop afterTime(double dt) {
    return new Drop(x + vx * dt, y + vy * dt,
      vx, vy, r, t + dt);
  }
```

```
  /**
   * Given 2 drops, returns the first time at which
   * they will collide, or a negative value if they
   * won't collide.
   */
  static double collisionTime(Drop a, Drop b) {
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    double dvx = a.vx - b.vx;
    double dvy = a.vy - b.vy;
    double sr = a.r + b.r;
    double A = dvx * dvx + dvy * dvy;
    double B = 2 * (dx * dvx + dy * dvy);
    double C = dx * dx + dy * dy - sr * sr;
    // Quadratic formula.
    double det = B * B - 4 * A * C;
    if (det < 0) return -1;
    // a is positive, so the smaller solution is the
    // one with the determinant subtracted.
    // Per the problem statement, both solutions will
    // either be positive or negative.
    return (-1 * B - Math.sqrt(det)) / (2 * A);
  }
}
```

10 Word Clouds Revisited

View on Kattis
Tags: Dynamic Programming

10.1 Problem Description

We are given a list of text boxes, represented with a width and a height. We need to lay them out within a box of fixed width. An entry can either be placed horizontally to the right of the previous box (if it fits), or at the far left end of the next row. The height of each row is the sum of the heights of every entry in the row. Our goal is to minimize the total height of the word cloud.

As it turns out, a greedy solution, as shown in Figure 6 may not be the optimal solution, depicted in Figure 7

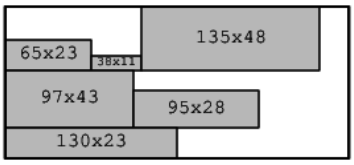

Figure 6: Greedy Strategy

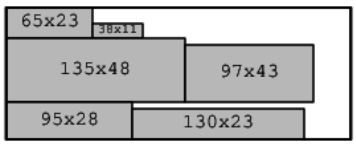

Figure 7: Optimal Solution

10.2 Algorithm

We can make this problem more tractable using dynamic programming. To approach this problem with that strategy, we need to define two things: an optimal substructure, and a data structure for memoizing the result.

Optimal Substructure
An optimal substructure simply represents some way of breaking the problem into smaller pieces, or some relation that we can use to build upon previous solutions to smaller portions of the problem. In this case, we can realize that we only need to track the smallest total height attainable for every possible width j of the final column after inserting the ith entry. Then, to calculate the $i+1$th entry's optimal values, we can iterate over all optimal solutions to the ith case and track the results of both inserting in that row (if possible) or inserting a new row.

Data Structure

If we are given N entries and a width of W, one option is to store everything in an $N \cdot (W+1)$ array. To compute the results of the ith row, we can iterate over all columns of the previous row. However, because many columns in the previous row may not be attainable, we can instead use a map of integers to optimal values, allowing us to cut down on the average case complexity. Additionally, because we only need the ith row to calculate the $i+1$th row, we only need to keep 2 maps, rather than N of them.

Armed with this, we can arrive at the following psuedocode:

Let *best* be a map of width to the optimal solution for that width.
for Rectangle r in the input **do**
  Let *nextBest* be a new map of width to the optimal solution.
  for *width*, *solution* in *best* **do**
    $nextBest(r.width) \leftarrow optimal(nextBest(r.width), addNewLine(best(width), r))$
    if $r.width + width \leq W$ **then**
      $newWidth \leftarrow r.width + width$
      $nextBest(newWidth) \leftarrow optimal(nextBest(newWidth), addSameLine(best(width), r))$
    end if
  end for
  $best \leftarrow nextBest$
end for
Return smallest height in *best*.

The optimal solution is whichever minimizes the total height, while $addNewLine$ and $addSameLine$ can be implemented by following the rules of the layout.

Time Complexity: $\mathcal{O}(N \cdot W)$
Space Complexity: $\mathcal{O}(W)$
Where N is the number of entries ($2 \leq N \leq 5000$) and W is the width of the word cloud ($150 \leq W \leq 1000$).

10.3 Implementation

```
import java.util.*;

public class WordCloudsRevisited {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int N = in.nextInt(), W = in.nextInt();

    Map<Integer, Size> memo = new HashMap<>();
    memo.put(0, new Size(0, 0));

    // Iterate over all text boxes.
    for (int i = 0; i < N; i++) {
      int w = in.nextInt(), h = in.nextInt();
      // Will hold the best values after inserting
      // this text box.
      Map<Integer, Size> nMemo = new HashMap<>();
      // Iterate over all trailing widths for the
      // previous insertion.
      for (int width: memo.keySet()) {
        Size prevSize = memo.get(width);
        // If we can insert this into the current
        // row, do so.
        if (width + w <= W) {
          int nCurrentRowHeight = Math.max(h,
              prevSize.currentRowHeight);
          Size nSize = new Size(nCurrentRowHeight,
            prevSize.previousRowsHeight);
          insert(nMemo, width + w, nSize);
        }
        // Always try creating a new row.
        Size nSize = new Size(h,
          prevSize.totalHeight);
        insert(nMemo, w, nSize);
      }
      // Copy over memoization.
      memo = nMemo;
    }
    // Find best at end.
    int best = Integer.MAX_VALUE;
    for (int width: memo.keySet()) {
      best = Math.min(best,
        memo.get(width).totalHeight);
    }
    System.out.println(best);
  }
```

```
  /**
   * Optimal Substucture - We want to store the
   * smallest total height (current row's current
   * height + height of any previous rows) attainable
   * for every possible width of the current row.
   */
  static void insert(Map<Integer, Size> m, int w,
      Size s) {

    if (!m.containsKey(w) || m.get(w).totalHeight >
        s.totalHeight) {
      m.put(w, s);
    }
  }
}

/**
 * Size represents the height of the word cloud
 * after a number of insertions.
 */
class Size {
  int currentRowHeight;
  int previousRowsHeight;
  int totalHeight;

  /**
   * Constructor for size.
   */
  public Size (int currentRowHeight,
    int previousRowsHeight) {

    this.currentRowHeight = currentRowHeight;
    this.previousRowsHeight = previousRowsHeight;
    totalHeight = currentRowHeight+previousRowsHeight;
  }
}
```

11 Tags Used

The following tags appeared throughout the contest:

- **Algebra**: The Agglomerator
- **Bit Manipulation**: Mosaic, Pyro Tubes
- **Brute Force**: Dance Recital, Square Deal
- **Dynamic Programming**: Word Clouds Revisited
- **Geometry**: Square Deal, The Agglomerator
- **Implementation**: ACM Contest Scoring, Pyro Tubes,
- **Permutations**: Dance Recital
- **Search**: Kitchen Measurements, Mosaic
- **Strings**: Hidden Password, Line Them Up
- **UCS**: Kitchen Measurements

12 Library Functions Used

12.1 Uniform Cost Search (UCS)

```
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

/**
 * UCS implements basic functionality for
 * performing Uniform Cost Searches, given an
 * implementation of the UCSState interface.
 */
public class UCS {

  /**
   * A contestant's implementation fo UCSState
   * can be used to perform a Uniform Cost Search.
   * It represents a state in the search's state
   * space, along with how far this search is from
   * the initial state, and if it is a goal state.
   */
  public interface UCSState {

    /**
     * distance should return the distance this
     * state is from the initial state.
     */
    public int distance();

    /**
     * neighbors should return a list of all
     * states reachable from the given state.
     * It does not need to keep track of whether
     * those states have been visited, provided
     * hashCode and equals are properly implemented.
     */
    public List<UCSState> neighbors();

    /**
     * atDestination should return true iff the
     * given state is a goal state.
     */
    public boolean atDestination();
```

```
  /**
   * equals should return true if the object
   * passed is an equivalent state to this
   * state. Note that this equivalence should
   * be independent of the distance the two
   * states have been reached at.
   *
   * For example, if s1 represents reaching
   * node 5 after 7 steps, and s2 represents
   * reaching node 5 after 9 steps, this should
   * still return true, because we only care
   * about what node we are at (the state),
   * not the distance.
   */
  public boolean equals(Object o);

  /**
   * s1.hashCode() must equal s2.hashCode() if
   * s1.equals(s2) is true. Otherwise, the less
   * occurrences where s1.equals(s2) is false, but
   * s1.hashCode() = s2.hashCode(), the faster
   * the search will be.
   */
  public int hashCode();
}

/**
 * UCSStateComparator is used to define a
 * comparison for the UCS's priority queue
 * between 2 states.
 */
private static class UCSStateComparator
  implements Comparator<UCSState> {

  public int compare(UCSState s1, UCSState s2) {
    return s1.distance() - s2.distance();
  }
}
```

```
  /**
   * doSearch will take an initial UCSState and
   * will return either the minimum distance
   * to a goal state or -1 if no path could be
   * found.
   */
  public static int doSearch(UCSState initial) {
    Map<UCSState, Integer> best = new HashMap<>();
    PriorityQueue<UCSState> Q = new PriorityQueue<>(128,
      new UCSStateComparator());

    Q.add(initial);
    best.put(initial, initial.distance());

    while(Q.size() > 0) {
      UCSState s = Q.poll();
      if (best.get(s) < s.distance()) continue;
      if (s.atDestination()) return s.distance();

      List<UCSState> neighbors = s.neighbors();
      for (UCSState neighbor : neighbors) {
        if (!best.containsKey(neighbor) ||
          best.get(neighbor) > neighbor.distance()) {

          best.put(neighbor, neighbor.distance());
          Q.add(neighbor);
        }
      }
    }

    return -1;
  }
}
```

12.2 Permutations

```
/**
 * The Permutations class contains utility functions for
 * brute forcing every permutation of a set of numbers.
 */
public class Permutations {

  /**
   * The PermutationWorker interface should be
   * implemented by the contestant, and should perform
   * some action in the permutationWork method, which
   * will be called by runPermutations, passing in all
   * necessary permutations in the process.
   */
  public interface PermutationWorker {
    public void permutationWork(int[] permutation);
  }

  /**
   * runPermutations will take a PermutationWorker and an
   * integer N and will repeatedly call the
   * PermutationWorker's permutationWork method, passing
   * in all permutations of [0...N] in the process. It is
   * the contestant's job to implement the
   * PermutationWorker's permutationWork method.
   */
  public static void runPermutations(PermutationWorker w,
      int N) {

    int[] o = new int[N];
    for (int i = 0; i < N; i++) o[i] = i;
    runPermutations(o, 0, w);
  }
```

```
  /**
   * Helper method used in brute-forcing all permutations
   */
  private static void runPermutations(int[] p, int d,
      PermutationWorker w) {

    if (d == p.length) {
      w.permutationWork(p);
    } else {
      for (int i = d; i < p.length; i++) {
        swap(p, d, i);
        runPermutations(p, d+1, w);
        swap(p, d, i);
      }
    }
  }

  /**
   * Swap two variables in an array.
   */
  private static void swap(int[] A, int i, int j) {
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
  }

}
```