Problem Tutorial: "ASCII Art"

Not much to say about this one. Just do it.

During contests, you will not have access to the PDF of the contest, so you may think that you will have to type all 8 pictures. But don't forget that we give you sample I/O in electronic form. All the images were included there. =)

Problem Tutorial: "Beginner's Training"

This problem absolutely stunned the judges! As we look at the scoreboard, no one at VCPC solved this problem, and only 3 teams around the region solved the problem. We had this pegged as an easier problem (about the 4th or 5th easiest).

The key insight is that if you are going to schedule the training session starting at time t_0, then t_0 might as well start at the beginning of (one of) *someone's* interval of availability. If it is not, just keep shifting it back in time until it does start at the beginning of someone's interval.

In this problem, there are only nk different intervals, so just try all of them! For each interval, check if each student is available for the entire session. Now, we haven't dealt with the coaches yet. But the easiest thing to do is to merge their intervals, then just pretend that they are one person.

Complexity

$O(n^2k^2)$.

Bonus for those who would like a bit of a challenge: This problem can be solved in $O(nk \log(nk))$. Can you find a solution with this complexity?

Problem Tutorial: "Chessboard Chaos"

This problem falls in the game theory category. What we need to do in games like this is look for winning-positions. A winning position means that if the token is here, the next player to play will win the game. A losing position means that no matter what they do, they cannot win the game.

Some Basics

Let's start filling in the 4×4 board with winning and losing positions – Note that any ordered pairs listed below are of the form (r, c), so row first, then column.

```
....
....
....
_...
```

Obviously, any of the squares in the bottom row, left column or main diagonal are winning positions (they can immediately move to the bottom-left corner).

```
W...W
W.W.
WW..
_WWW
```

Now, look at the $(3, 2)$ location. There are 4 possible places to move, but all of them are winning positions. Meaning that no matter what we do, the other person will win once we move there. This means that we are in a losing position. (a similar argument shows that $(2, 3)$ is a losing position.

```
W..W
WLW.
WWL.
_WWW
```

Now note that any cell that is in the same row, column or diagonal of those two losing positions are now winning positions because they can move to the losing position, forcing the other person to lose!

```
WWWW
WLWW
WWLW
_WWW
```

This completely fills in the 4×4 grid. In theory, we want to fill in the entire $r \times c$ grid, but because because r and c can be up to 10^6, this is way too slow.

Key Insights

Insight 1

In any row, there can be at most one losing position. (*Proof (by contradiction): Assume that there were 2 or more. The rightmost "losing"position could just move to any of the other losing positions in the row and force the other player to lose. So our assumption must have been wrong that there were 2 or more.*)

The same must be true about columns and diagonals.

Insight 2

In any row, there must be at least one losing position. (*Proof (by contradiction): Assume that row r has no losing positions. From Insight 1, we know that there is at most one losing position per row. So then there are at most $r-1$ losing positions in rows 1 through $r-1$. Let (R, C) be the rightmost losing position*

in rows 1 through $r-1$. You should be able to convince yourself that $(r, C+r+1)$ must be a losing position since it can only see winning positions from where it is. This is a contradiction, so our assumption that r has no losing positions is must be wrong.)

The same must be true about columns and diagonals.

Insights 1+2

This means that each row, column and diagonal has exactly one losing position.

Insight 3

(r, c) is a losing position if and only if (c, r) is a losing position. (This should be obvious.)

Putting it all together

We will keep track of an array $A[i]$, which records the losing positions in row i (this means that $(i, A[i])$ is a losing state). Initialize $A[i] = -1$. We will just concern ourselves with below the diagonal (the upper half is a mirror image by Insight 3). Label the diagonals in this way:

```
....0
...01
..012
.0123
01234
```

We will sweep r from bottom-to-top. If $A[r] \neq -1$, then we already know the answer for row r, so move to the next r. If $A[r] = -1$, then $A[r]$ is the lowest indexed diagonal that has not been covered previously (call this diagonal d). So $A[r] = r + d$. This also means that $A[r + d] = r$ by Insight 3. Note that we use the diagonals in order (0,1,2, ...).

Code Snippet

```
int d = 0;
for( int i=1 ; i<=MAX_N ; i++ ){
  if(A[i] != -1) continue;
  A[i] = i+d;
  A[i+d] = i;
  d++;
}
```

Complexity

$O(\max(r, c))$

For fun, there is also a $O(1)$ solution for this problem.

Let $\varphi = \frac{\sqrt{5}+1}{2}$. If $(r-1, c-1) = (\lfloor \varphi n \rfloor, \lfloor \varphi^2 n \rfloor)$ or $(r-1, c-1) = (\lfloor \varphi^2 n \rfloor, \lfloor \varphi n \rfloor)$, then it is a losing state (so output `Britney`), otherwise it is a winning state (so output `Alice`).

Problem Tutorial: "Dice"

Computing the probability of a particular pair

- The probability for a given pair of dice d_1, d_2 can be computed in $O(1)$. Assume without loss of generality that $d_1 \leq d_2$.
- Let $g(x)$ be the number of outcomes in which the total sum is exactly x. Make some observations:
  - The total number of outcomes of all possible rolls is $d_1 \cdot d_2$.
  - The number of outcomes in which the sum is exactly x can be computed by

$$g(x) = \min(d_1, x - 1, d_1 + d_2 + 1 - x)$$

  - The function g is increasing at a rate of 1 on the interval $[2, d_1 + 1]$, constant on the interval $[d_1 + 1, d_2 + 1]$ and decreasing at a rate of 1 on the interval $[d_2 + 1, d_1 + d_2]$ and zero elsewhere.
  - Therefore g can be split into three pieces: an increasing part, a constant part and a decreasing part. The sum of any interval can be calculated in .
- The sum $\sum_{x=A}^{B} g(x)$ can be computed in $O(1)$ by splitting it into the three mentioned parts (you will need the fact that $1 + 2 + \cdots + k = \frac{k(k+1)}{2}$).
- The probability is

$$\frac{\sum_{x=A}^{B} g(x)}{d_1 \cdot d_2}$$

Finding the optimal pair

We will choose one of the two dice (d_1) and search for its optimal partner. Define $f(d_2)$ to be the function that computes the probability that rolling d_1 and d_2 gives a good sum. It should not take much to convince yourself that f must have a particular shape (working from left-to-right):

- Initially, the probability is 0 since the dice may be too small so that all possible rolls are smaller than A. (Note, this interval may be empty).
- Then, as we increase the number of faces on d_2, the probability increases for a while, so f increases.
- Eventually, once the number of faces on d_2 is too large, the probability of rolling less than B gets smaller and smaller, so f decreases, and continues to decrease towards 0 as $d_2 \to \infty$.

This means that f can be ternary searched so long as you remove the section where the probability is 0. (the section of probability 0 can be found by binary search). You will also have to take care to process duplicate values and remove them before ternary searching, or they will break the shape of f.

Complexity

$O(n \log n)$

Bonus for those who would like a bit of a challenge: This problem can be solved using divide and conquer in $O(n \log n)$. Can you find this solution?

Problem Tutorial: "Environmental Concern IV"

This problem can be solved using a few different formulas, but all have the same underlying structure. Here, I will show a way to directly construct the answer. Then, I will hint at an alternative proof.

Note that the sequence a_i is basically the Fibonacci numbers (well, not quite since they don't necessarily start with $a_1 = 0$ and $a_2 = 1$).

The problem is essentially asking you for the sum of the first d Fibonacci numbers modulo m for all $L \leq m \leq R$. We will do each of these computations independently for each value of m, then take the minimum afterwards.

We will call $s_k = a_1 + a_2 + \cdots + a_k$. Note that you cannot compute s_d directly since $d \leq 10^{18}$. One way to compute the sum is to use matrices. Note that the following is true (expand out the multiplication if you are not convinced, and use the fact that $a_i + a_{i-1} = a_{i+1}$):

$$\begin{bmatrix} a_i \\ a_{i+1} \\ s_{i+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_{i-1} \\ a_i \\ s_i \end{bmatrix}$$

You can use this to show:

$$\begin{bmatrix} a_i \\ a_{i+1} \\ s_{i+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}^{i-1} \begin{bmatrix} a_1 \\ a_2 \\ s_1 \end{bmatrix}$$

You can do matrix exponentiation in $O(\log i)$ matrix multiplications (see `http://bit.ly/2euE11L` for more details). Do not forget to do modulo after every operation!

Complexity

$O(R \log d)$

Note that you can also show that $s_k = a_{k+2} - a_2$. Try to prove it! Then try to compute a_i using a 2×2 matrix instead of 3×3 as above.

Problem Tutorial: "Floor Plan"

There are many solutions to this problem. Here is a simple one:

- Give everyone a seat for the practice contest (it doesn't matter how you give everyone a seat for this step).
- For the real contest, keep their seat number the same, but put them in the next lab. For example, if you are in seat `B5` for the practice, then you would be in seat `C5` for the real contest and if you are in seat `D3` for the practice, then you would be in seat `A3` for the real contest.

Be careful because seat `A7` does not exist! So if you are in seat `D7` in the practice contest, the *next* lab is actually lab B, so you would be in seat `B7`.

Complexity

$O(n)$

Problem Tutorial: "Guardians of the Galaxy"

Imagine that R2D2 has placed the ship at (x_0, y_0) and the distance to the farthest planet is d. This means that all of the planets must be inside (or on) the circle of radius d centred at (x_0, y_0). The problem is asking you to find the smallest circle that satisfies this property. This is called the *minimum enclosing circle*. Here is one way to solve this problem:

If there is only one planet, the answer is easy, so we will assume $n \geq 2$. You will need the following insight: The minimum enclosing circle must touch at least 2 planets. To see this, you can do it in a few steps:

- If the circle doesn't touch any points, then you can slowly shrink the circle until it does touch at least 1.
- If the circle only touches one point (called p), then you can slowly shrink the circle while making sure that p remains on the circle until the circle touches another point.
- If the circle only touches two points (called p and q), then those points must be on opposite sides of the circle (otherwise, you can slowly shrink the circle while making sure that p and q remain on the circle until you either (a) touch another point or (b) p and q are on opposite sides of the circle).

The first naive solution (too slow):

- For every pair of points (p and q), check if the circle where p and q are on opposite sides encloses all the other points.
- For every triple of points (p, q and r), check if there is a circle that goes through all 3 points, and if there is one, check if it encloses all of the other points.
- For every quadruple of points, check if there is a circle that goes through all 4 points, and if there is one, check if it encloses all of the other points.
- (...Continue with 5-tuples, 6-tuples, etc...)

Out of all circles that enclose all of the points, output the minimum radius. However, note that there is a unique circle that goes through any 3 non-collinear points, so all of the circles found in the 4-tuple, 5-tuple, 6-tuple, etc. steps will have already been found in the triple step!

So the correct answer is just the first 2 steps:

- For every pair of points (p and q), check if the circle where p and q are on opposite sides encloses all the other points.
- For every triple of points (p, q and r), check if there is a circle that goes through all 3 points, and if there is one, check if it encloses all of the other points.

To find the unique circle through 3 points just takes a little bit of math using the formula for circles. We will leave that as an exercise.

Complexity

$O(n^4)$

There are many more efficient algorithms for this problem. There is a $O(n)$ solution to this problem. There is also a simple algorithm that has *expected time* $O(n)$. This means that, on average, it takes only linear time, but in the worst-case takes $O(n^2)$. Be careful in contests about using algorithms like this, as it is not guaranteed that it will run fast! Although, even $O(n^2)$ for this problem was plenty fast enough!

Problem Tutorial: "Hacking Passwords"

This problem is pretty straightforward. Just read every word and check if it satisfies all of the required constraints. It helps to know your language's built-in libraries to write problems like this. For example, in C++, here is some quick code:

```
bool f(const string& s){
  if(s.length() < 6) return false;

  bool upper = false, lower = false, digit = false, punct = false;
  for(int i=0;i<(int)s.length();i++){
    if(isupper(s[i])) upper = true;
    if(islower(s[i])) lower = true;
    if(isdigit(s[i])) digit = true;
    if(ispunct(s[i])) punct = true;
  }

  return upper && lower && digit && punct;
}
```

Complexity

$O(n)$

Problem Tutorial: "Interesting Signs"

The Wrong Answer

Any naive greedy algorithm will not work (it is probably true that no greedy algorithm will work). For example, consider the following code:

```
bool doable(string s){
  int length = s.length();
  if(length <= 1) return true; // Length 0 or 1 is good!

  if(s[0] == s[1])
    return doable(s.substr(2)); // This removes the first two characters

  if(s[length-1] == s[length-2])
    return doable(s.substr(0,length-2)); // This removes the last two characters

  if(s[0] == s[length-1])
    return doable(s.substr(1,length-2)); // This removes the first and last character

  return false; // If nothing else works, it must be bad
}
```

However, consider this string: `AACABBA`. If you run the above algorithm, you will do: `AACABBA` $\rightarrow$ `CABBA`, but then you cannot do anything. However, here is a decomposition: `AACABBA` $\rightarrow$ `ACABB` $\rightarrow$ `ACA` $\rightarrow$ `C`.

You may think "Oh, I can just switch the order of the `if` statements and it will work"(and it does work for the above string). However, none of the 6 orders will work for this string: `AABBAABABABBA`.

The Correct Answer

This problem is actually a reasonably straightforward Dynamic Programming problem.

Let $A[L][R]$ be true if the substring from L to R is decomposable. To compute $A[L][R]$, you need to check if any of the following are true: (a) $A[L+2][R]$ (if the first two characters are the same), (b) $A[L][R-2]$ (if the last two characters are the same), or (c) $A[L+1][R-1]$ (if the first and last characters are the same). If any of them are true, then $A[L][R]$ is true. If all are false, then $A[L][R]$ is false. The answer is $A[0][length]$.

Complexity

$O(n^2)$

Problem Tutorial: "Juggling Errands"

This problem is a reasonably hard one! I'd like to thank Max Ward-Graham for submitting this problem.

Time Limit Exceeded

First, I would like to put your minds at ease: The answer is NOT *for each errand, compute the shortest path in the graph.* Any such algorithm (Dijkstra, BFS, Floyd-Warshall, A*, etc.) will be too slow.

The Correct Answer

In a general graph, this problem is quite hard, so we must use something about the structure of the graph.

We will be using the triangle inequality in this problem:

$$\text{distance}(x, y) \leq \text{distance}(x, z) + \text{distance}(z, y)$$

Intuitively, $\text{distance}(x, z) + \text{distance}(z, y)$ computes the distance from x to y assuming you must go through z along the way.

First, set the distance of every errand to ∞ (as we run the algorithm below, we will incrementally find better answers for each errand). Next, run BFS from every cell in the centre row and column and compute the distance from that cell to every other cell in the grid.

```
.......v.......
.......v.......
.......v.......
.......v.......
.......v.......
.......v.......
vvvvvvvvvvvvvvv
.......v.......
.......v.......
.......v.......
.......v.......
.......v.......
.......v.......
```

For each errand (from x to y) and for each BFS (from z), check if $\text{distance}(x, z) + \text{distance}(z, y)$ is better than the current best distance you have computed from x to y. If it is, update the best answer.

Now, there are 2 cases:

Case 1: If x and y are in different quadrants, then at least one shortest path MUST go through one of the cells that were BFSed from. So at this point, you must have the correct answer for this query and you can stop considering it in all future computations.

Case 2: If x and y are in the same quadrant, then at this point, we know the shortest path from x to y assuming that the path leaves the quadrant (but we know nothing about paths that are fully contained in the quadrant). To find the shortest path from x to y that is fully contained in the quadrant, you can simply recurse on that quadrant (which is a quarter of the size of the original graph).

Pseudocode

The following code computes the distance of each errand, assuming that all paths cannot leave a certain sub-square of the grid. Originally, `f(all_errands, 0, n-1, 0, n-1)` is called.

```
f(list_of_errands, top, bottom, left, right):
  if list_of_errands is empty:
    return // There is nothing to do...

  for each cell (row,col) on the middle row or middle column:
    dist = BFS(row,col) // dist[r][c] is the distance from (row,col) to (r,c)
                        // such that you stay in the subsquare:
                        // rows: 'top' to 'bottom' and columns: 'left' to 'right'
    for each errand from [a,b] to [c,d] in list_of_errands:
      if dist[a][b] + dist[c][d] < errand.best_answer:
        errand.best_answer = dist[a][b] + dist[c][d]

  for each quadrant q:
    errands_in_quadrant = [all errands that start and end in this quadrant]
    T = topmost row in q
    B = bottommost row in q
    L = leftmost column in q
    R = rightmost column in q
    f(errands_in_quadrant, T, B, L, R)
```

Complexity

At first glance, it doesn't seem like this algorithm is necessarily fast enough. First, let's ignore the errands and just consider the time complexity of the BFSs. I am assuming that to compute BFS on a grid of size $k \times k$ takes $5k^2$ steps (k^2 vertices and $4k^2$ edges).

- On the first iteration, there are $2n$ BFSs on a grid of size $n \times n$.
  Amount of work: $10n^3$
- On the second iteration, there are $4n$ BFSs on grids of size $\frac{n}{2} \times \frac{n}{2}$.
  Amount of work: $5n^3$
- On the third iteration, there are $8n$ BFSs on grids of size $\frac{n}{4} \times \frac{n}{4}$.
  Amount of work: $\frac{5}{2}n^3$
- ...
- On the i^{th} iteration, there are 2^i BFSs on grids of size $\frac{n}{2^{i-2}} \times \frac{n}{2^{i-2}}$.
  Amount of work: $\frac{5}{2^{i-2}}n^3$

Note that $10n^3 + 5n^3 + \frac{5}{2}n^3 + \cdots + \frac{5}{2^k}n^3 = 10n^3\left(1 + \frac{1}{2} + \cdots + \frac{1}{2^k}\right) \leq 10n^3\,(2) = 20n^3$. So the total for this portion of the algorithm is $O(n^3)$.

Now consider each errand. In the worst case, the start point and the end point are the same, so they are always in the same quadrant no matter what.

- On the first iteration, the distance check is performed $2n$ times.
- On the second iteration, the distance check is performed n times.
- On the third iteration, the distance check is performed $\frac{n}{2}$ times.
- ...
- On the i^{th} iteration, the distance check is performed $\frac{n}{2^{i-1}}$ times.

Note that $2n + n + \frac{n}{2} + \cdots + \frac{n}{2^k} = 2n\left(1 + \frac{1}{2} + \cdots + \frac{1}{2^k}\right) \leq 2n\,(2) = 4n$. So each errand adds $O(n)$. So the total for this portion of the algorithm is $O(E \cdot n)$.

Thus, the total complexity is: $O(n^3 + E \cdot n)$.

Note that it is not required that you necessarily need to use quadrants. Instead, you could just use the middle column every time and split the map in half. If you re-run the above computations, you will see that this gives a complexity of $O(n^3 \log n + E \cdot n \log n)$, which was also accepted.

Question: Is there an answer faster than $O(n^3 + E \cdot n)$? I honestly don't know. So if you have an idea, send me (Darcy) an email!

Problem Tutorial: "Kreative Konstruction"

This problem requires a decent amount of background knowledge, but it is all stuff that you should know!

A *bridge* is an edge that increases the number of connected components in your graph when it is removed. In this problem, since the graph is originally connected, a bridge disconnects the graph if it is removed. Note that if you cut any edge that is not a bridge, then the level of inconvenience is 0.

A *2-edge connected component* is a maximal set of vertices such that for every pair of vertices (u and v) in the set, there are two edge-disjoint paths from u to v. (This is the same as saying that there is a cycle that contains both u and v).

Merge all vertices in each 2-edge connected component into one super-vertex, and make a super-edge between two super-vertices if there is an edge in the original graph between the 2-edge connected components. The super-edges will correspond directly to bridges in the original graph. The *size* of a super-vertex is the number of vertices from the original graph in this 2-edge connected component.

The super-vertices and super-edges will form a tree (if there were a cycle in the super-graph, the bridges wouldn't be a bridge...). This tree can be computed in $O(n+m)$ using Tarjan's Bridge-finding algorithm. `http://bit.ly/2eSg7hn`.

Root the tree at an arbitrary super-vertex. The *size* of a subtree is the sum of the sizes of the super-vertices in the subtree. If you cut a bridge in the graph, the level of inconvenience is the product of k and $n-k$, where k is the size of the subtree under the bridge in the rooted tree.

Pseudocode

After you have built the tree, here is the DFS code. `f(v,p)` returns the size of the subtree rooted at vertex `v` where `p` is the parent of `v` in the tree. You call the original function with `f(root,-1)`. Note that you need the `parent` parameter to make sure that you don't move back up the tree.

```
f(v,parent):
  size_of_subtree = size[v]
  for each u adjacent to v:
    if u != parent:
      s = f(u,v) // This is the size of the subtree.
      if(s*(n-s) > best) // s*(n-s) is the level of inconvenience of cutting this bridge
        best = s*(n-s)
      size_of_subtree += s

  return size_of_subtree
```

Complexity

$O(n+m)$