Problem Tutorial: "Atomic Mass"

Difficulty: Easy
Category: Parsing
Intended Complexity: $O(n)$
Author: Darcy Best

This is a fairly straightforward problem. You just need to parse the string appropriately and add up the corresponding atomic masses.

Note (rant) from the Judges:

We decided to be nice by only asking for an approximate answer to this problem. We could easily have asked for exactly 2 decimal points in the output. However, we guessed that too many teams would incorrectly use floating point numbers in their solutions to this, so we decided to ask for an approximation. Why are floating point numbers an issue? Because they are not stored as exact values (for example, 1.80 in binary is $1.1100110011001100\ldots$, so it cannot be stored exactly). You may say, "Well, I can just round the answer to the nearest hundredth." And you would be right for this problem, but for harder problems, it may not be so easy. Get used to using integers whenever possible.
To read a real number with 2 digits of precision:

```
int whole, fraction;
char dot;
cin >> whole >> dot >> fraction;
int input = whole * 100 + fraction;
```

Do NOT do this:

```
double x;
cin >> x;
int input = x * 100;
```

since the rounding error has already occurred when reading in `x`.

Problem Tutorial: “Bake-off”

Difficulty: Easy-Medium
Category: Exhaustive Search and Bit Manipulation
Intended Complexity: $O(n + 2^6 m)$
Author: Andrew Haigh

At first, the input of this problem seems a bit daunting, but after seeing what is being asked, it is actually quite straightforward.

1. Partition the cakes into $2^6 = 64$ different lists corresponding to the subset of flavours that are in them (the lists should be sorted by deliciousness).
2. When a customer arrives, you do not need to check all of the cakes. You only need to check 64 cakes: one from each of the lists made above. Simply check which of those 64 cakes satisfy the subset requirement and take the largest one of those. Remove that item from the list and continue.

To check if one set is a subset of another, you can use the following code. Think of the binary representation of `x` and `y` being which elements of the set you are choosing.

```
// Returns true if x is a subset of y
bool subset(int x,int y){
  return (x & y) == x;
}
```

Problem Tutorial: "Clever Title"

Difficulty: Medium
Category: Dynamic Programming
Intended Complexity: $O(2^n nk)$
Author: Darcy Best

Since each output can be as large as 10!, computing each answer naively is too slow. Instead, we will use Dynamic Programming. Fix a valid word, w.

We will sweep through the characters of w from left to right. At iteration k, let S be the set of author names that have already been used (the size of S is k). We now compute the number of ways to fill in the last $n - k$ entries in the permutation. Consider the following code:

```
int f(S){
  if S == all_authors:
    return 1
  k := size of S
  ans := 0
  for(name in authors)
    if (name not in S) and (name contains w[k]):
      ans += f(S union {name})
  return ans
}
```

This will compute the answer, but will take $O(n!)$ time to compute. However, there are only 2^n states, so with Dynamic Programming, this can be computed in $O(2^n n)$.

Problem Tutorial: "Deck of Cards"

Difficulty: Hard
Category: Graph Theory
Intended Complexity: $O(n^{2.5})$
Author: Daniel Anderson

First of all, we will think about the game somewhat differently. Add the card that is on the table to Richard's hand and force him to play that card on the first turn. This is an equivalent game to the one asked, but will help us in finding a solution.

Define the bipartite graph G with a vertex for each card in Richard's hand on one side and each card in Malcolm's hand on the other side. Put an edge between a card in Richard's hand and a card in Malcolm's hand if they share a colour or number. Each move in the game corresponds to traversing an edge in this graph.

If you are familiar with alternating paths in maximum matching, then the solution is probably easier to come by. I would recommend reading up on alternating paths first.

Lemma 1: If every maximum matching contains the card that was on the table, then Malcolm has a winning strategy.
Proof. Malcolm can arbitrarily choose any maximum matching, M. Malcolm will always take the move suggested by the maximum matching. No matter what move Richard makes in response at each step, he will move to a vertex that is part of the maximum matching. Why? Because if Richard ever moves to a vertex that is not part of M, then consider the set of edges from M with the moves that Malcolm made replaced by the moves that Richard made (call this M'). Note that M' is a maximum matching that does not contain the card on the table—which we assumed was not possible! Thus, Malcolm must make the last move, so Malcolm wins.

Lemma 2: If there is some maximum matching that does not contain the card that was on the table, then Richard has a winning strategy.
Proof. Richard can arbitrarily choose any maximum matching that does not contain the card on the table, M. Richard will always take the move suggested by the maximum matching. No matter what move Malcolm makes in response at each step, he will move to a vertex that is part of the maximum matching. Why? Because if Malcolm ever moves to a vertex that is not part of M, then consider the set of edges from M with the moves that Richard made replaced by the moves that Malcolm made (call this M'). Note that M' is a matching that is bigger than the one we started with (so the original matching could not have been maximum)—which is not possible!. Thus, Richard must make the last move, so Richard wins.

Thus, Malcolm wins if and only if every maximum matching contains the card that was on the table.

To check if every maximum matching contains that vertex, first compute the size of the maximum matching of G, then remove that special vertex from the graph. If the size of the matching of this graph is the same, then there is some maximum matching that does not contain that vertex. If the size does change, then every maximum matching of the graph contains that vertex.

Note that you can solve this problem in $O(n + d^3)$, where d is the number of different cards since we can collapse each of the same type of card into one vertex and use a more general max flow rather than bipartite matching.

Problem Tutorial: "Extra Judicial Operation"

Difficulty: Medium-Hard
Category: Graph Theory
Intended Complexity: $O(n + m)$
Author: Mike Cameron-Jones

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 remove any edge that is not a bridge, then the the graph is still connected, so it doesn't affect the number of judging servers needed.

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 bridges...). This tree can be computed in $O(n + m)$ using Tarjan's Bridge-finding algorithm. `http://bit.ly/2eSg7hn`.

Now we just need to look at this tree. If there is only one vertex in the tree, then we only need the one judging server. Now consider a non-trivial tree. Obviously, each leaf (vertices of degree 1) in this tree needs a server because if that one edge that is connected to this vertex is deleted, then all of the vertices in this super-vertex would not be connected to a judge server. This is also enough: no matter which bridge is deleted in the graph, it will split the graph into two pieces, each of which will contain at least one leaf from the super-graph.

Problem Tutorial: “Front Nine”

Difficulty: Medium
Category: Dynamic Programming
Intended Complexity: $O(nh)$
Author: Andrew Haigh

At the start of the contest, AbishforesT jumped well past the easy problems in the set to solve this medium problem in a mere 7 minutes. This problem is quite straightforward for those who do competitive programming frequently.

Consider the following, where `expected(i,j)` returns the expected area under the terrain for all $x \geq j$ assuming we are currently at height `i`.

```
double expected(int i,int j){
  if(j == n-1)
    return 0;
  double ans = 0;

  // First, let's do P_{-1}. Note that (2*i-1)/2.0 is the area under
  //    the terrain for x=j to x=j+1.
  if(i != 0)
    ans += P_neg1 * ( expected(i-1,j+1) + (2*i-1)/2.0 )
  else
    ans += P_neg1 * ( expected(i,j+1) + i )

  // Now do P_0
  ans += P_0 * (expected(i,j+1) + i)

  // Last, do P_1. Note that (2*i+1)/2.0 is the area under
  //    the terrain for x=j to x=j+1.
  if(i != h)
    ans += P_1 * ( expected(i+1,j+1) + (2*i+1)/2.0 )
  else
    ans += P_1 * ( expected(i,j+1) + i )

  return ans
}
```

Note that only $(n+1) \cdot (h+1)$ different values can be input to this function, so once you have computed it once, then you should never compute it again (this is Dynamic Programming).

Problem Tutorial: "Greedy Generosity"

Difficulty: Easy-Medium
Category: Simulation
Intended Complexity: $O(T)$
Author: Mike Cameron-Jones

In this problem, you just simply need to simulate the process described. There are no tricky corner cases.

Note that the last line of the problem ("It is guaranteed that the largest coin a customer will put in will exceed the value of the correct change.") ensures that there is always enough change left in the machine–since you can simply give that coin back in the worst case. Also, this implies that the largest amount of change that can be given is 50c.

There were a number of small mistakes made by contestants. Most notably: (a) forgetting to take multiple copies of a coin in the greedy approach (40c should be two 20c pieces) and (b) not restocking back to the standard level after extra change was given.

Problem Tutorial: "Hole-In-One"

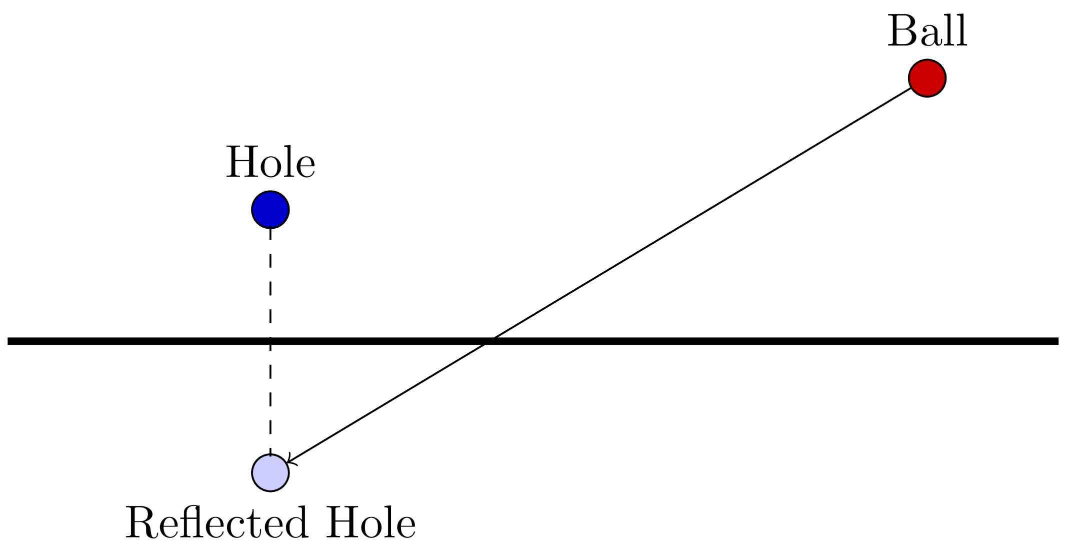
Difficulty: Hard
Category: Geometry
Intended Complexity: $O(n^2)$
Author: Daniel Anderson

I have given this problem a *hard* tag solely because it is a geometry problem. Conceptually, this problem is extremely easy. However, to solve this problem properly (especially without using floating point numbers) takes some care.

The general idea:

- Check if you can hit the ball straight into the hole (no bounce).
- For each object, try bouncing the ball off that object and check if it hits any other wall along the way.

Let's fix an object W. How do you determine how to bounce the ball of off W to get into the hole? The easiest way to do this is to reflect the hole over the wall and hit the ball directly at that point.



Now what you can do is hit the ball along this line and make sure that W is the first object that you hit. Any point along this line can be written in the form $\mathbf{b} + a(\mathbf{b} - \mathbf{r})$. We can ignore all points where $a < 0$ since that is in the opposite direction. For each line, compute the a value and check to make sure that W is the wall with the smallest value (since we want to hit that one first).

You now repeat this process by trying to hit the hole towards the ball and check that W is the first wall that you hit.

I will leave most of the underlying math as an exercise (or you can look at the official solutions).

Question from the Judges
What is the worst case input for floating point numbers? We found cases where the path that the ball must take comes within approximately 4×10^{-7} of another wall (so it doesn't quite hit it, but it is close). I think that I can show that the closest we can get given the input bounds is approximately 10^{-7} (maybe $10^{-7}/2$). I'd be interested if anyone finds anything that is closer or has a proof that this is the closest.

Problem Tutorial: “Incomplete Book”

Difficulty: Easy
Category: Simple Math
Intended Complexity: $O(\log d)$
Author: Darcy Best

This problem was intended to be the easiest in the set (and the scoreboard seems to agree).

Simply count the number of books directly. Since the time that it takes to write each book doubles each time, the time that it takes grows super fast!

```
int books = 0;
while(k <= d) {
  books++;
  d -= k;
  k *= 2;
}
```

Problem Tutorial: "Jupiter Rock-Paper-Scissors"

Difficulty: Medium
Category: Game Theory
Intended Complexity: $O(n^4)$
Author: Darcy Best

This problem was the only problem that the judges disagreed with contestants in terms of the relative difficulty. We have it marked as our 6th or 7th easiest problem, but the scoreboard disagrees with this. The assistant coach at Monash yesterday pointed out that the image in the top-right corner made the problem harder—solely because it pushed the input bounds to the second page!

The short answer to this question is "try everything". In my explanation below, I am going to ignore `Draw` (it can be added in easily afterwards given the same logic—so let's assume that a Draw is impossible).

Think about Phase 1: Alice only has $n - k + 1$ different choices to make. Let's go through each of these moves and ask "if Alice makes this move, can she guaranteed win?" If there is some move that the answer to that question is yes, then Alice should take that move. If there is no move that she can take that guarantees a win, then Bob is going to win the game no matter what Alice does. But how do we determine the answer to the question?

Well, let's think about Phase 2 (assuming we know the move that Alice decided to make). We are now playing as Bob. Bob also has $n - k + 1$ different choices to make. Go through each of these moves and ask "if Bob makes this move, is he guaranteed to win?" If there is some move that the answer to that question is yes, then Bob should take that move! If there is no move that Bob can take that guarantees a win, then Alice will win no matter what Bob does. How do we answer that question?

Now let's think about Phase 3 (assuming we know the moves that Alice and Bob took in Phases 1 and 2). We are again play as Alice. Alice has $k - \ell + 1$ choices for this step. She should try them all and determine if she wins in any of them (you can check this directly since this is the last phase). If any move is a win for her, she should take that. If there is no move where she wins, then Bob must win!

This handles all cases.

To implement this, we will use three values: -1 means that Bob wins, 0 means draw and 1 means that Alice wins. This means that Bob is trying to minimize the outcome of the game and Alice is trying to maximize the output of the game.

In the following, `morph` morphs the ℓ moves starting at index i and the function `win` determines who wins the Play Phase.

```python
def phase1(s,t):
    return max(phase2(s[i:i+k],t) for i in range(n-k+1))

def phase2(s,t):
    return min(phase3(s,t[i:i+k]) for i in range(n-k+1))

def phase3(s,t):
    return max(win(morph(s,i),t) for i in range(k-l+1))
```

Problem Tutorial: "Keeping Cool"

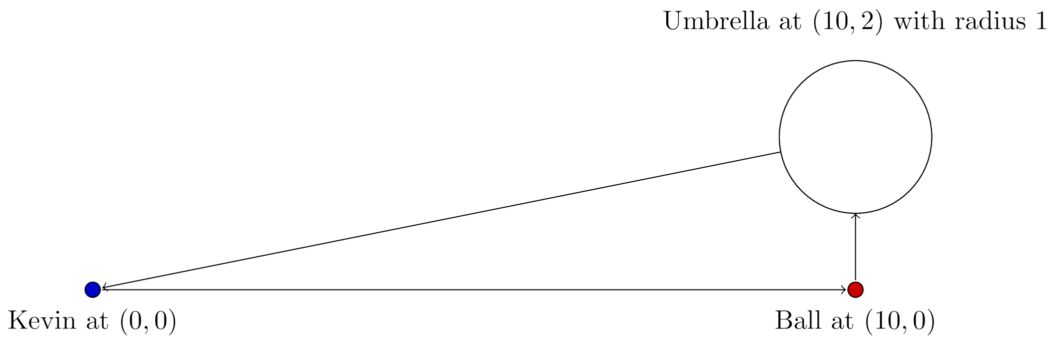
Difficulty: Medium
Category: Graph Theory
Intended Complexity: $O(n^2)$
Author: Darcy Best

This problem should scream graph theory at you!

If the distance between two umbrellas is at most k, then draw an edge between those umbrellas with the distance between the two umbrellas being the weight of that edge. You are trying to get to the ball and back as quickly as possible.

But note a couple things:

- If you want to get to the ball, you need to ensure that you can also get safely to an umbrella after grabbing it! Imagine the ball is 3 metres away, but Kevin can only run 5 metres. Then you cannot go to the ball and back, so this is impossible.
- The way you go TO the ball is not necessarily the way you should come BACK from the ball. Consider the following example where it is impossible to run to the ball, then back. The best strategy may be to get the ball, then go to the umbrella, then run from the umbrella back to the car.



Instead, compute the shortest distance in the graph above to each of the umbrellas (call these distances $d_1, d_2, \ldots, d_n$). Then for all pairs of umbrellas, u and v (including an umbrella and itself), check if you can run from u to the ball to v within the allotted timeframe. If you can grab the ball in p seconds ($p \leq k$), then then one potential solution is $d_u + p + d_v$ (since the distance from the start to v is the same as the distance from v to the start).

Note that the distances can be computed in $O(n^2)$ with Dijkstra (if you use the heap version, it is $O(n^2 \log n)$). We also allowed Floyd-Warshall in $O(n^3)$ if you wanted to.

Problem Tutorial: "Lights in the Morning"

Difficulty: Easy
Category: Math
Intended Complexity: $O(N)$
Author: Daniel Anderson

For this problem, you can treat each traffic light independently. For each traffic light, if $x < a$, then the light is still read, so the answer is NO.

Otherwise, the traffic light starts cycling with a cycle length of $g + r$. You can make it through the traffic light if you get there in the first g minutes within one of these cycles. You can check this with this simple formula:

```
// Returns true if you can make it through the traffic light
bool make_through(int x,int a,int r,int g){
  if(x < a)
    return false
  if( (x-a) % (r+g) > g )
    return false
  return true
}
```