Florida High School Programming Series
2015 Championship Round

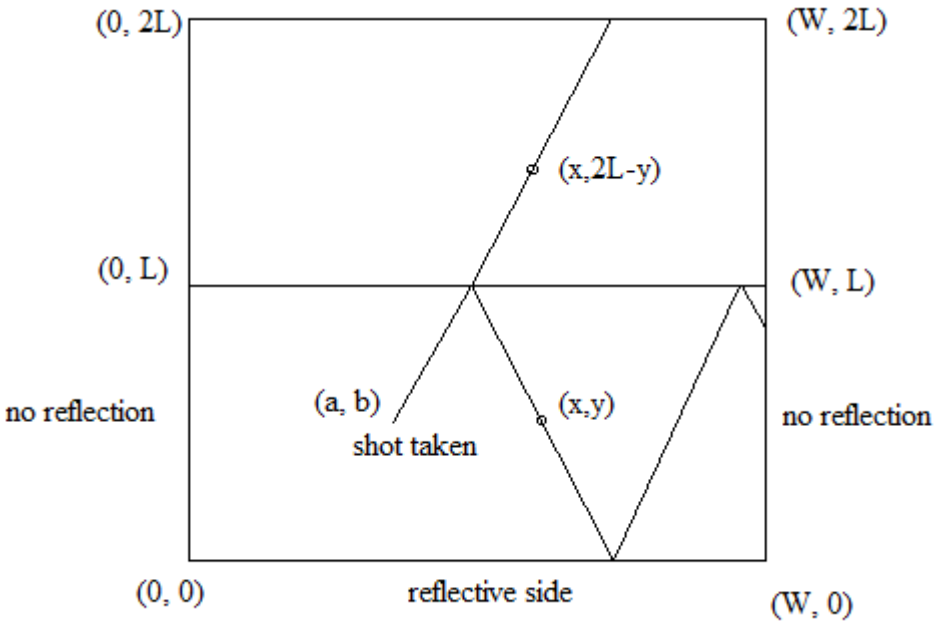
Problem Writer's Analysis by Arup Guha

Problem 1: Orlando Eye

The key to this problem is realizing that you can work out the movement in the vertical axis (defined as the y-axis for the purpose of this problem) separate to the horizontal axis. Thus, the key is using the initial velocity of the penny in the vertical axis coupled with the equation given that determines the height of the penny at time t. Once a value is assigned to this velocity, a quadratic equation in t must be solved. Based on the input restrictions, it's guaranteed that the quadratic will have one positive root and one negative root. The positive root reflects the time at which the penny will hit the ground. For example, if the initial vertical velocity is 10 ft/sec, and the initial height of the penny is 100 feet, then the height of the penny at time t is $100 + 10t - 16t^2$. For this case, the value of t we desire is $\frac{10+\sqrt{100-4(16)(-100)}}{32}$ seconds. Once we know when the penny will hit the ground, we can calculate how far the penny moves from its initial x-coordinate by multiplying the horizontal velocity component with the time previously calculated. The trickiest test case for this problem involved an actual answer that was slightly negative, but greater than -0.5. In this situation, printf with the code %.0f produces the output -0, when rounding to the negative integer. One can specifically check for this case. Alternatively, one can round the result to an integer and store it in an integer variable before printing.

Problem 2: Laser Tag

I thought that this was one of the hardest problems in the set and was very happy that a team solved it! Though the code is very simple for it, the key realization to get to that code is not. when a laser bounces off either wall parallel to the x-axis, we can imagine the path of the laser after the bounce being reflected over the wall which it bounced off of. Consider the situation for just one bounce illustrated below:



The key here is noting that in order for our shot to hit the point (x, y) shown from (a, b), the straight line segment from (a, b) heading in the same direction must hit the point (x, 2L-y), based on the law of reflection. Thus, a brute force solution that will run within the given time limit is taking the target point, (x, y) and reflecting it over and over again (some care has to be taken when doing this, since the mathematics works in two different cases, depending on whether it's the odd or even reflection) and continuing to do this until the reflected point isn't within the specified angle range of the shooter. This must be done in both directions, up and down.

A more efficient solution would be to take the range of angles the shooter has and use the tangent function to calculate the maximum and minimum y values reachable with a single shot. Once these values are obtained, one can set up an equation to solve for how many "reflections" in both directions represent the shooter's range.

The idea for the question was taken from a more difficult question, Pool Table, posed in the 2009 South East Regional Programming Contest. In that problem, a pool ball was allowed to bounce off all four cushions and one had to determine the "shortest shot" that made contact with the target ball.

Problem 3: Dinner and a Movie

This question was the easiest question in the set. It was the question I put in to make sure that all teams solved at least one question. I decided to make it slightly more difficult than a straight formula question by requiring an if statement for a conditional calculation, mimicing policies that many places have for bulk buying. Judging by the scoreboard, most teams found this problem fairly quickly, except for two teams that took a bit more than an hour to solve it. My guess is that those teams simply worked on other problems first before discovering this one.

Problem 4: Radio Prizes

In my opinion, this was the hardest problem in the set, though the scoreboard doesn't justify that claim, since one team did solve it while other problems remained unsolved. Many students may look at this problem and gravitate towards a greedy solution. Unfortunately, no simple greedy algorithm correctly solves the problem. A quick analysis of various possible strategies (take the first item you can, take the largest individual prize, etc.) yields algorithms that don't properly solve some cases. In fact, the initial sample case disproves the notion of taking the first available item, since that can block future items that are better.

The key to solving the problem is to calculate the best schedule of prizes ending at a specific prize, and doing this for each prize. For example, if we wanted to calculate the best schedule of prizes ending at prize 10, we'd have try each possible previous prize, which are prizes 1 through 9. It may be the case that for some of these prizes, we would not be available to obtain prize 10. Say that for prizes 2, 3, 6, and 8, after we collect them, we would be able to collect prize 10. Then, our best answer would look like:

best(10) = value(10) + maximum(best(2), best(3), best(6), best(8))

In short, we set our best value to the sum of the current prize plus the maximum value that we are allowed to build off of. The critical section of code in my solution is below. Note that the if statement in the inner for loop screens out invalid cases to build off of.

```
for (int i=1; i<dp.length; i++) {

    dp[i] = prizes[i].value;
    for (int j=0; j<i; j++) {

        if (prizes[j].wait+prizes[j].day > prizes[i].day)
            continue;

        dp[i] = Math.max(dp[i], dp[j]+prizes[i].value);
    }

    res = Math.max(res, dp[i]);
}
```

The variable res is keeping track of the largest item in the dp array, so that at the end of the segment of code above, res stores the correct answer for the test case.

Problem 5: Science Center Membership

This problem was supposed to be the second easiest in the set, but it seems as if it gave most teams much more trouble than I had hoped. Only one of the Timber Creek teams got the problem on their first try. Every other team had multiple submissions.

If I had used a ficticious science center and created my own membership rules, my guess is that more students would have solved the problem. Instead, I thought it would be fun to mirror the rules of the Orlando Science Center. Since we have a nanny for our 2 1/2 year old daughter, the rule for nannies made sense to me. Perhaps since most high school students haven't dealt with a nanny, they had some difficulty making sense of the rules.

Ultimately, there are really three cases here: a membership for a single person, a membership for any two people, or a family membership. Thus, the key to the solution is counting the number of people for the membership and applying a different rule for the cases of 1 person, 2 people, and more than 2 people. My guess is the case that I allowed that threw people off was 1 adult and 1 nanny, since technically, according to the rules given, this is a couple. If there are more than two people, we simply add the cost of the family membership to the cost for the nannies and the cost for the extras.

Problem 6: Orlando City Soccer

A common problem first year students learn in computer science is sorting. This problem required a custom sort. Since both Java and C++ have built in sorts, this problem was essentially testing if students could write a custom comparison method/function based on the rules given. Again, I opted to use the actual rules in the professional league MLS, with a simplification, allowing for only the first four tie-breaking conditions. The data I created made sure to test each level of comparison.

This problem lends itself to a nice object-oriented design. Here is my compareTo method:

```
public int compareTo(team other) {

    if (this.points() != other.points())
        return other.points() - this.points();

    if (this.wins != other.wins)
        return other.wins - this.wins;

    if (this.goalDiff() != other.goalDiff())
        return other.goalDiff() - this.goalDiff();

    return other.goalsFor - this.goalsFor;
}
```

Notice that its structure very closely resembles the paragraph describing the rules, in the order that each rule is mentioned.

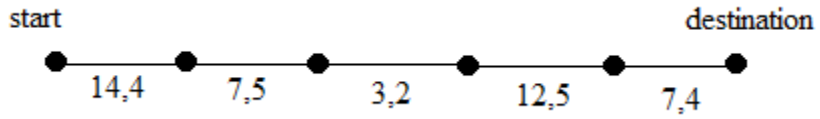
Problem 7: Theme Park Speed

I felt that this was the second most difficult problem in the set. In past years I put in a shortest distance problem, but this year there was a twist! Each edge seems to have two possible weights, and the weight that counts isn't obviously clear. Furthermore, after taking some time to make test cases, one realizes that you can't simply alternate running and walking. In some instances, it may make sense to walk twice in a row, though it never makes sense to walk three times in a row. (In any sequence with three consecutive walks, we can convert the middle walk to a run and improve our result, thus the preceding result could not have been minimal.)

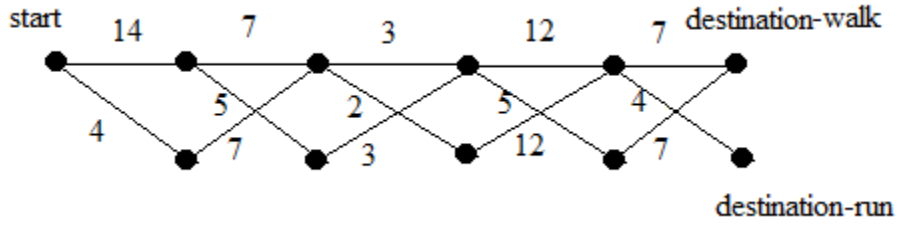
One way to handle this apparent issue is to split each location into two vertices. All edges that indicate running will go from a "walking" vertex to a "running" vertex. All edges that indicate walking will go from both the "walking" and "running" starting vertex to the corresponding "walking" vertex. In the end, we can simply check to see the shortest distance from the start vertex to either version of the destination vertex. The problem with storing the graph as originally given is that no information is stored indicating which edge weight is available to take upon arrival at a vertex. By "splitting" each vertex into two, we can "store" the important information of whether or not someone ran to this vertex. The really neat thing about "splitting" the vertex for this problem (and many others), is once we do it, then we can run our typical algorithm for solving for the shortest distance between two points.

Based on the bounds of this question, one can run either a modified breadth first search, Dijkstra's algorithm or Floyd-Warshall's algorithm. (Other algorithms will work as well.)

To more clearly illustrate the graph transformation for splitting a vertex, consider the following original graph:



Our new graph would look like the following:



For this graph, the shortest route from the start to either of the destination vertices would be to run(4), followed by walking twice(7, 3), running(5) and finishing by walking(7), for a time of 26 minutes.

Note: for simplicity of drawing, this example used a "line" graph, but the technique does generalize to any graph.

Problem 8: Swan Boats

Unlike a few of the other problems in the set where the actual problem was well-disguised, this problem was relatively straight-forward. The set-up of the problem and the bounds of the input strongly suggested using a brute force solution investigating every permutation of visiting the landmarks given. I was a bit surprised that no team in the contest solved the problem, given that comparable teams had solved similar questions in UCF's High School Programming Contest. The only extra difficulty was that the points to visit were given in polar coordinates, with an angle and a distance from the origin. This format is easily convertible to (x, y) coordinates since:

$$x = d\cos\theta$$
$$y = d\sin\theta$$

where d is the distance from the origin and θ is the typical angle measure in radians.

The key to these types of questions is writing the same permutation code and using an integer permutation array to index into a separate array of positions.

Here is the key recursive code from my solution to the problem (in the non-terminating case):

```
double res = 2*RADIUS*(n+20);

for (int i=0; i<n; i++) {
    if (!used[i]) {
        used[i] = true;
        perm[k] = i;
        res = Math.min(res, solve(k+1, used, perm));
        used[i] = false;
     }
}

return res;
```

Note that the initial result is simply a value guaranteed to be larger than the real answer. Here, k is an input parameter that represents the slot in the permutation array that is currently being filled. The for loop goes through each possible item that could go in slot k. The if statement screens out any items that have already been placed in the permutation in progress. Thus, we only enter the if for values that have yet to be placed in the permutation, correctly trying each possible value in slot k. It's very important after finishing a recursive call to unmark an item (used array) so that item can be used later in a different permutation.