Judged response for input C-small: Correct!

code jam
hello, world!

Practice Mode

Contest scoreboard | Sign in

Round 1B 2012

A. Safety in Numbers

B. Tide Goes In, Tide Goes Out

C. Equal Sums

Contest Analysis

Questions asked

- Submissions

Safety in Numbers

10pt	Not attempted **2687/5608 users** correct (48%)
11pt	Not attempted **2008/2680 users** correct (75%)

Tide Goes In, Tide Goes Out

18pt	Not attempted **682/892 users** correct (76%)
18pt	Not attempted **619/670 users** correct (92%)

Equal Sums

6pt	Not attempted **2257/2531 users** correct (89%)
37pt	Not attempted **149/853 users** correct (17%)

- Top Scores

Gennady.Korotkevich	100
bmerry	100
hansonw	100
marcina	100
ZhukovDmitry	100
random.johnnyh	100
yeputons	100
rng..58	100
pashka	100
mikhailOK	100

Contest Analysis

Overview | Problem A | Problem B | Problem C

There is a lot going on in this problem, and it is difficult to keep track of everything. Only seven hundred contestants managed to get it right, so you know it can't have been easy! Let's go over it.

The first thing you should notice is that this is a shortest-path problem. We want to get from point A (the start) to point B (the exit) as fast as possible. There are a number of classical shortest-path algorithms available; we will try to adapt Dijkstra's algorithm here.

Dijkstra's algorithm works on a weighted graph, so we need to identify a set of vertices and edges between them to use it. This is pretty easy if you have done graph theory before - we take the squares to be the vertices, and we put an edge between every two adjacent cells. We encounter problems when trying to assign weights to the edges: the cost (in seconds) of travelling from one square to another is not fixed - it can be one or ten seconds (and we haven't touched on the issue of moving around before the tide starts going down yet).

A fact that might be somewhat surprising is that this is not a problem for Dijkstra's algorithm. Let us briefly recall how it works - it maintains a tentative cost for each vertex, and at each step it chooses the vertex with the smallest cost, fixes that to be the real cost of visiting that vertex, and updates the neighbors accordingly. Of course in our case the cost of reaching a vertex is simply the time it takes.

This means that we need to consider the time of going from some square **A** to an adjacent square **B** only once, when we are recalculating the time of reaching **B** due to having fixed the time for **A**. But since we fixed the time of reaching **A**, we can easily calculate the water level at this moment - and so we will know the cost of travelling this edge. (Note that the earlier we can start moving from **A**, the faster the move will be. Therefore, we really should be moving as soon as possible instead of waiting for a better moment.)

There are also other constraints that we have. Let's name two of them - the water level needs to be at least 50 centimeters below the ceiling of the square we want to enter, and the "gap condition" needs to be satisfied. For the latter, we can just check - this does not depend on the time we want to travel - and remove the edge from the graph if the condition is not satisfied (we can either do this up-front, or lazily at the moment we try to use this edge). For the former, we need to look at what Dijkstra's algorithm needs again. The cost of travelling the edge **AB** is used to determine the shortest path to **B** that goes through **A** and then directly through the edge

AB to **B**. Well, if we really want to take this path, and the water level is too high to enter **B** when we arrive at **A**, we have only one choice - we have to wait until the water level drops to $\mathbf{C_B}$-50 before moving. Remember that this may cause us to have to drag the kayak!

All this means that we can calculate the cost of each edge at the moment in which we need it, and we would have a complete solution for the problem if not for the extra possibility of moving around before the tide starts going down. One might be tempted to solve this issue by a preprocessing phase, where we use a breadth-first search to find all the squares that are reachable from the start in time zero.

A simpler solution to code, however, is to insert this phase into the Dijkstra's algorithm that works so well for us so far. All this extra movement means is that if we want to traverse the edge **AB**, we are at **A** in time zero, and **B** is accessible from **A** in time zero, then the cost of making this move is zero - as we can make it before the tide starts going down. This means that we insert one extra condition in our function that calculates the cost, and we are done!

Notice that the outcome here is just a standard implementation of Dijkstra's algorithm and nothing more, with all the problem-specific logic inserted into the function that calculates the weight of a given edge at the time it is needed. To convince oneself that this works, however, an understanding of the inner workings of the algorithm is needed.