Results from UVA

0.1 Problem A: Amalgamated Artichokes (page 2)

17246735	1709 Amalgamated Artichokes	Accepted	JAVA	4.410	2016-04-22 19:22:25

0.2 Problem C: Catering (page 5)

17319298	1711 Catering	Accepted	JAVA	1.250	2016-05-06 04:29:57

0.3 Problem D: Cutting Chese (page 12)

17274629	1712 Cutting Cheese	Accepted	JAVA	1.700	2016-04-27 22:19:28

0.4 Problem E: Evolution in Parallel (page 16)

17311732	1713 Evolution in Parallel	Accepted	JAVA	0.620	2016-05-04 17:06:22

0.5 Problem F: Keyboarding (page 22)

17306383	1714 Keyboarding	Accepted	JAVA	13.780	2016-05-04 01:50:35

0.6 Problem I: Ship Traffic (page 27)

17334126	1717 Ship Traffic	Accepted	JAVA	1.780	2016-05-09 03:42:21

1 Problem A: Amalgamated Artichokes

Results:

17246735	1709 Amalgamated Artichokes	Accepted	JAVA	4.410	2016-04-22 19:22:25

Background:

Fatima Cynara is an analyst at Amalgamated Artichokes (AA). As with any company, AA has had some very good times as well as some bad ones. Fatima does trending analysis of the stock prices for AA, and she wants to determine the largest decline in stock prices over various time spans. For example, if over a span of time the stock prices were 19, 12, 13, 11, 20 and 14, then the largest decline would be 8 between the first and fourth price. If the last price had been 10 instead of 14, then the largest decline would have been 10 between the last two prices.

Fatima has done some previous analyses and has found that the stock price over any period of time can be modeled reasonably accurately with the following equation:

$$price(x) = p \cdot (sin(a \cdot x + b) + cos(c \cdot x + d) + 2)$$

where $p, a, b, c,$ *and* d are constants. Fatima would like you to write a program to determine the largest price decline over a given sequence of prices. You have to consider the prices only for integer values of x.

Input:

The input file contains several test cases. Each test case is on a single line containing 6 integers, p $(1 \leq p \leq 1000)$, a, b, c, d $(0 \leq a, b, c, d \leq 1000)$, *and* n $(1 \leq n \leq 10^6)$. The first 5 integers are described above. The sequence of stock prices to consider are $price(1), price(2), ..., price(n)$.

Output:

For each test case, display the maximum decline in stock prices. If there is no decline, display the number '0'. Your output should have an absolute or relative error of at most 10^{-6}.

Sample Input:

42 1 23 4 8 10
100 7 615 998 801 3
100 432 406 867 60 1000

Sample Output:

104.855110477
0.00
399.303813

1.1 Mathematical Formulation

Given an input of integers $p, a, b, c, d,$ *and* n, the formula $f(x) = p \cdot (sin(a \cdot x + b) + cos(c \cdot x + d) + 2)$ where $x \in [1, n]$, determine the largest decrease between the integer values x_i, x_j where $i<j$ and $x_i \geq x_j$ and there does not exist another pair x_k, x_l where $k<l$ and $x_k \geq x_l$ but $x_k - x_l > x_i - x_j$.

1.2 Solution

The main functionality of this algorithm is to plug in each point keeping track of the highest seen point, h, the lowest seen point occuring after l, and the largest difference, $d = h - l$. It should be noted that since we are always taking the difference between the two values, we can factor out the $\cdot p$ as well as neglect the $+2$ portions of the formula. Also, to cut down on run time, it works in the java system if you % pi each of the entries before putting them into the sine and cosine functions. For whatever reason the larger the input, the more costly the operation is.

Algorithm 1 Main

```
procedure F(x)
    ab ← (a*x+b) % pi,
    cd ← (c*x+d) % pi;
    return (Math.sin(ab) + Math.cos(cd))
procedure SOLVE(p, a, b, c, d, n)
    val, h, l ← f(1); diff ← 0
    for x ∈ [2, n] do // if n = 1, do not execute
        val ← f(x)
        if val>h then // higher than current highest
            h, l ← val;
        else if val<l then // lower than current lowest
            l ← val; curDiff ← h - l;
            if curDiff>diff then diff ← curDiff;
    PRINT(p · diff)
```

1.3 Correctness

Proposition 1.
We will determine the value of largest price decline over the interval $[1, n]$*, only considering* $f(1), f(2), ..., f(n)$.

Proof.
We do this by keeping track of the largest price decline seen thus far, $diff$, the current highest point seen, $h = f(x_i)$, and the current lowest point seen, $l = f(x_j)$, such that $x_i \leq x_j$, *and* $f(x_i) \geq f(x_j)$. Therefore whenever we see a higher point,

$f(x_k)>f(x_i), x_k>x_i$, we update our $h = f(x_k)$ and reset our lowest point to be $l = h$ since we are searching for the largest decline $\implies$ l must occur after h. Now every time that we see a number $f(x_m) \leq l$, we update l and check to see if our $h-l \geq diff$, if so we update diff, else we continue to the next point. If we see a higher point than h we will repeat this process. Therefore we will be looking at each subsequent highest corresponding following lowest points $\implies$ we will see this largest price decline. □

1.4 Analysis

Proposition 2. *The space complexity of this algorithm is* ***O(1)***

Proof.

This is due to the fact that we will only store the values $p, a, b, c, d, n, anddiff$ as integer variables O(1):

Giving us a space complexity of **O(1)**

□

Proposition 3. *The time complexity of this algorithm is* ***O(N)***

Proof. This is the case because our algorithm goes throughh the points $1, 2, ..., n$ once and only calculates each value one time.

Giving us a time complexity of **O(N)**

□

1.5 An Example

Given the input of: 42 1 23 4 8 10, we will read this in as $p = 42, a = 1, b = 23, c = 4, d = 8, and\ n = 10$. Then we will initialize our $h = l = f(1)$ and $diff = 0$. Then starting with the second point until the 10th we will read throughh and record the values of what h, l, curDiff, and diff are:

x =	1	2	3	4	5	6	7	8	9	10
f(x)	-0.061724	-1.090011	1.170641	1.380555	-0.691700	0.170589	-1.115995	-1.070976	1.551270	0.359768
h	-0.061724	-0.061724	1.170641	1.380555	1.380555	1.380555	1.380555	1.380555	1.551270	1.551270
l	-0.061724	-1.090011	1.170641	1.380555	-0.691700	-0.691700	-1.115995	-1.115995	1.551270	0.359768
curDiff	–	1.028286	–	–	2.072255	–	2.496550	–	–	1.191502
diff	0	1.028286	1.028286	1.028286	2.072255	2.072255	2.496550	2.496550	2.496550	2.496550

Now multiplying our $diff$ by p $\implies$ $2.496550 \cdot 42 = 104.855110$ which is our solution.

2 Problem C: Catering

Results:

17319298	1711 Catering	Accepted	JAVA	1.250	2016-05-06 04:29:57

Background:

Paul owns a catering company and business is booming. The company has k catering teams, each in charge of one set of catering equipment. Every week, the company accepts n catering requests for various events. For every request, they send a catering team with their equipment to the event location. The team delivers the food, sets up the equipment, and instructs the host on how to use the equipment and serve the food. After the event, the host is responsible for returning the equipment back to Paul's company.

Unfortunately, in some weeks the number of catering teams is less than the number of requests, so some teams may have to be used for more than one event. In these cases, the company cannot wait for the host to return the equipment and must keep the team on-site to move the equipment to another location. The company has an accurate estimate of the cost to move a set of equipment from any location to any other location. Given these costs, Paul wants to prepare an Advance Catering Map to service the requests while minimizing the total moving cost of equipment (including the cost of the first move), even if that means not using all the available teams. Paul needs your help to write a program to accomplish this task. The requests are sorted in ascending order of their event times and they are chosen in such a way that for any i <j, there is enough time to transport the equipment used in the i^{th} request to the location of the j^{th} request.

Input:

The input file contains several test cases, each of them as described below.

The first line of input contains two integers n ($1 \leq n \leq 100$) and k ($1 \leq k \leq 100$) which are the number of requests and the number of catering teams, respectively. Following that are n lines, where the i^{th} line contains $n - i + 1$ integers between 0 and 1000000 inclusive. The j^{th} number in the i^{th} line is the cost of moving a set of equipment from location i to location $i + j$. The company is at location 1 and the n requests are at locations 2 to $n + 1$.

Output:

For each test case, display the minimum moving cost to service all requests. (This amount does not include the cost of moving the equipment back to the catering company.)

2.1 Mathematical Formulation

Given an input of k teams and n jobs that must be fulfilled, we are given the order that the jobs occur in chronological order such that j_1 occurs before j_2...before j_n. We are also given every cost of moving equipment from job j_i to j_k where that job j_i occurs before j_k. The task is to determine what the minimum moving cost to service all of the jobs is.

2.2 Solution

The main functionality of this algorithm is to construct a min-cost bipartite graph which has a total of $k + 2 \cdot n + 1$ nodes which are representative of:

1. **1**: source (s)
2. **k + (n - 1)**: for the left side of the graph representing a node for each team and a node for each job except for the last one since they occur chronologically and therefor not possible to come from that node. (iFrom)
3. **n**: for the right side as these are the destinations that each team can go to (starting point) as well as coming from one of the previous jobs. (iTo)
4. **1**: sink (t)

Giving us a total of $[1]+[k+(n-1)]+[n]+[1] = k+(n+n)-1+1+1 = k+2\cdot n+1$. We use an adjacency matrix of integers denoting weight to go from one node to another to represent our graph and initialize the costs of going from the source to each of the left side, "origin", nodes to be 0 and each of the nodes from the right side, "destination", nodes to the sink to be 0.

Algorithm 2 Build

```
procedure CONTROL
    for each test case do
        buildGraph( )
        print(match( ))
procedure buildGraph
    N, K, numNodes, s, iFrom, iTo, t ← initialized
    int[numNodes][numNodes] graph ← initialized with all infinity values initially
    for each origin node n, n ∈ [iFrom, iTo − 1] do
        connect(s, n, 0)
    for each origin node n, n ∈ [iFrom, iTo − 1] do
        for each destination node m, m ∈ [iTo, t − 1] do
            connect(n, m, costFromInput)
    for each destination node n, n ∈ [iFrom, iTo − 1] do
        connect(n, t, 0)
```

Once we have created this graph, we note that it this is a min-cost bipartite graph problem as we seek to minimize the cost of getting from the source to the sink while visiting each node (although this is not a true bipartite graph this is discussed in the correctness section). To preform this we make a copy of our graph to which we will be altering while referencing our original (very important) then we will define a cost array to store the cost of each node. From here we find our shortest path using dijkstra's algorithm and reverse all edges along that path found; then we update the cost of each vertex, the flow on edges going from our origin nodes to our destination nodes, then the flow on edges going from our destination back to our origin nodes. We do this process for each request so that way we do this n times. Finally we will sum the total cost of the edges that go from the origin to the destination nodes.

Algorithm 3 Min Cost Bipartite Matching

```
procedure MATCH
    defCopyG, cost ← initialized
    for each Request do
        int[numNodes] distTo, from ← initialized
        IndexMinPQ<Integer> mpq ← initialized with size numNodes
        perform dijkstra altering distTo and from arrays
        // reverse all edges along the path found
        while cNode = s do
            int index ← from[cNode]
            defCopyG[cNode][index] = defCopyG[index][cNode]
            defCopyG[cNode][index] = Infinity
            cNode = index
        // update cost of each vertex
        for each node n ∈ numNodes do
            cost[n] += distTo[n]
        // update the flow on edges going from origin to destination
        for each node n ∈ origin and m ∈ destination do
            defCopyG[n][m] = cost[n] + graph[n][m] - cost[m]
        // update the flow on edges foring from destination to origin
        for each node n ∈ origin and m ∈ destination do
            defCopyG[n][m] = cost[m] + graph[m][n] - cost[n]
    int minimum = 0
    for each origin node n, n ∈ [iFrom, iTo − 1] do
        for each node m, m ∈ [1, numNodes] do
            if graph[n][m] != Infinity && defCopyG[n][m] != Infinity then
                minimum += graph[n][m]
```

2.3 Correctness

Proposition 4.
Solving min-cost bipartite matching on our graph construction will give us the correct answer.

Proof.
The formulation that we used for this graph was that we can start at any of the jobs which are not the last one and make connections to any job that occurs after it and not before. Then with the teams, they are allowed to go to any job first but cannot go to the same job at the same time since this would not match each place once. The destination nodes are each of the jobs since each one must be fuffilled and it only contains inEdges from jobs that can go to it or directly from each of the teams. Though this is not a full bipartite graph, we can see that it has the correct formulation if we include any non present nodes (teams in the destination section or the last job in the origin section) to have all indegrees and outdegrees of Infinity and not to count them in our searches. Though this only takes up space and time so we simply neglect them in our implementation, though they are technically present. □

2.4 Analysis

We note that the number of nodes in our graph is $(k + 2 \cdot n + 1)$ which is equal to O(k+n)

Proposition 5. *The space complexity of this algorithm is* $\boldsymbol{O((k+n)^2)}$

Proof.
This is due to the fact that all of our information is stored in either one dimensional or two dimensional integer arrays or a minimum priority queue, each of which has the size comparable to our number of nodes. The largest of these therefore is our two dimensional array of size $(numNodes)^2$. Since our number of nodes is $(k + 2 \cdot n + 1)$ this is $(k + 2 \cdot n + 1)^2$ s

Giving us a space complexity of $\mathbf{O((k+n)^2)}$

□

Proposition 6. *The time complexity of this algorithm is* $\boldsymbol{O(n \cdot (k+n)^2)}$

Proof. We break the algorithm into a two main cases, our construction of the graph and performance of our min-cost bipartite algorithm (for purposes of readability we will use N to represent $(k+n)$). We note that since this is a bipartite graph it is fully connected (essentially) so we can simplify that $V+E$ in normal graph representations is actually $N + N^2$. Therefore we note that the construction of the graph will take V+E time which is O(N^2). The next part is the performance of our min-cost bipartite algorithm which will go trough for number of jobs (n) + and perform dijkstra's

$(E+V \cdot log(V)) \implies (N^2)+N \cdot log(N)$, $O(N^2)$ then we will update costs which each time (4) will at most go throughh the entire graph once $O(N^2) \implies n \cdot (N^2+4 \cdot N^2)$ $= O(n \cdot N^2)$. Comparing the two parts then gets us $O(N^2) + O(n \cdot N^2)$.

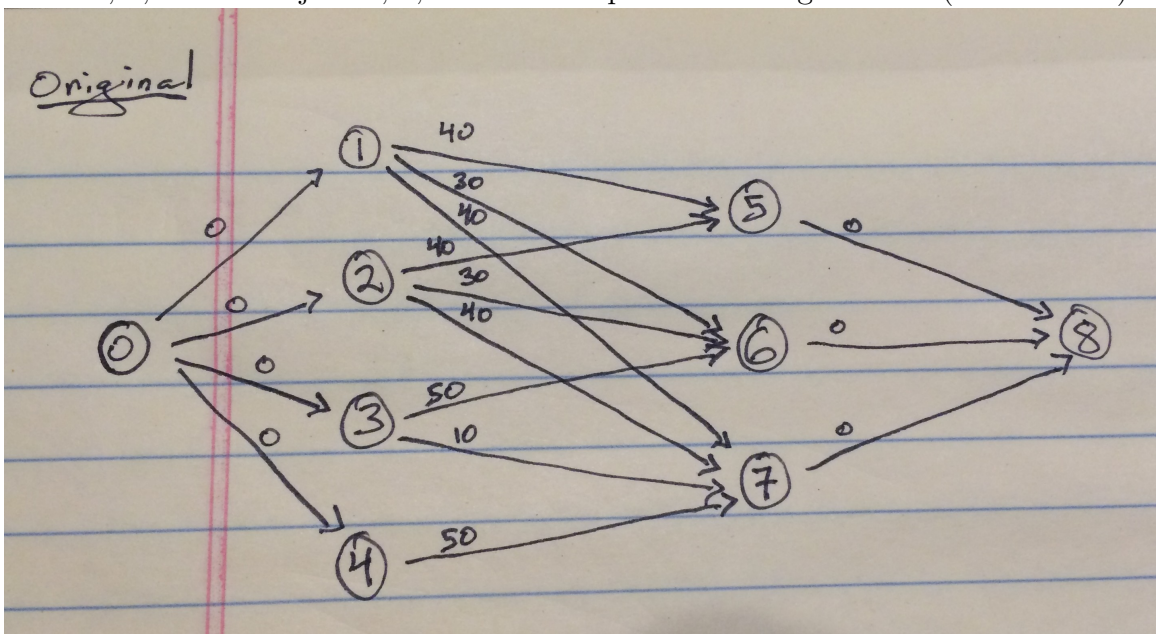
Giving us a time complexity of $\mathbf{O(n \cdot (k+n)^2)}$

□

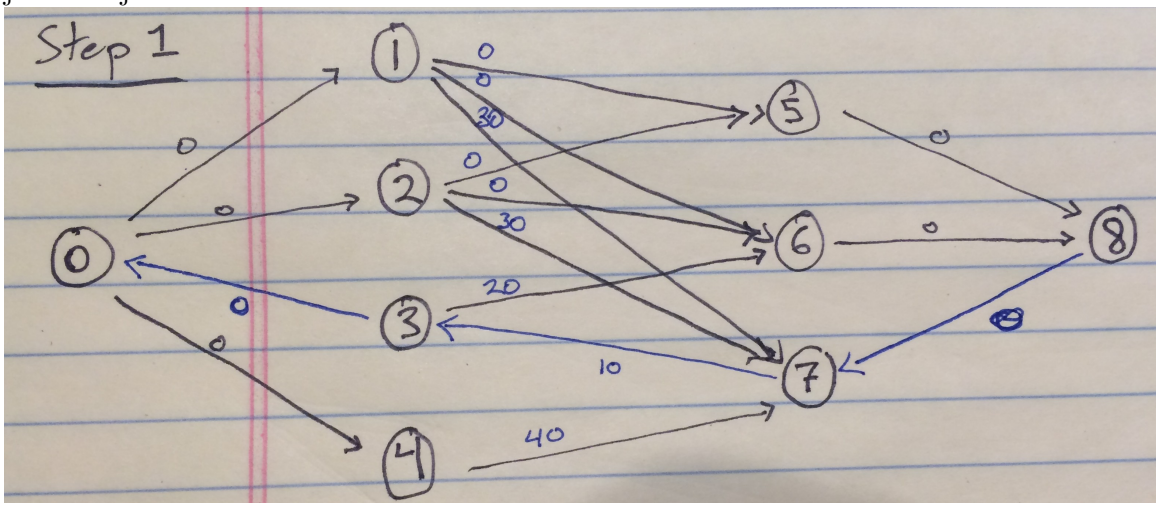
2.5 An Example

Given the input of:
3 2
40 30 40
50
50

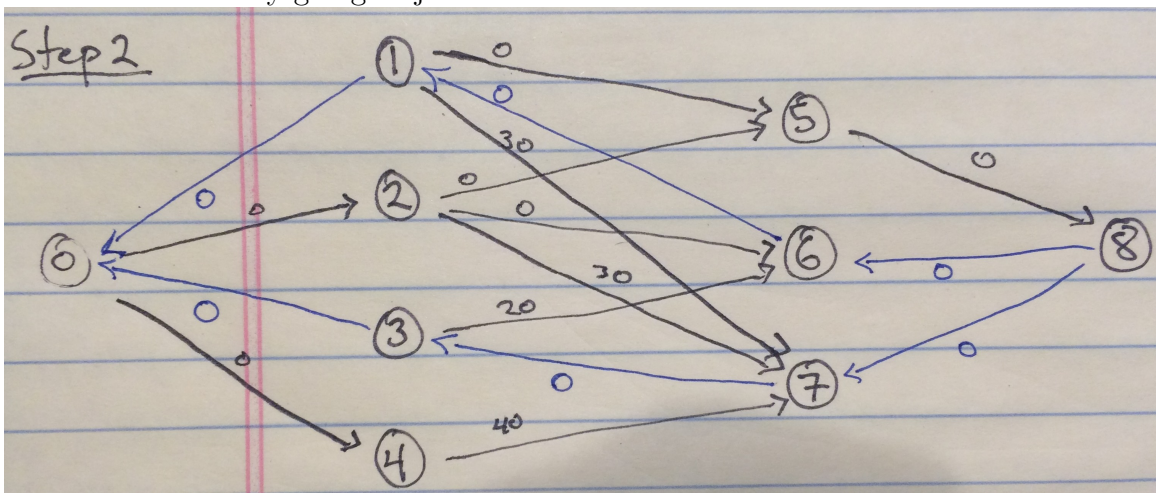
We Build the following graph where indices 0 and 8 are the source and sink, 1 and 2 represent the teams, 3 and 4 are jobs 1 and 2 and symbolize leaving them, and nodes 5, 6, and 7 are jobs 1, 2, and 3 and represent coming to them (destinations).



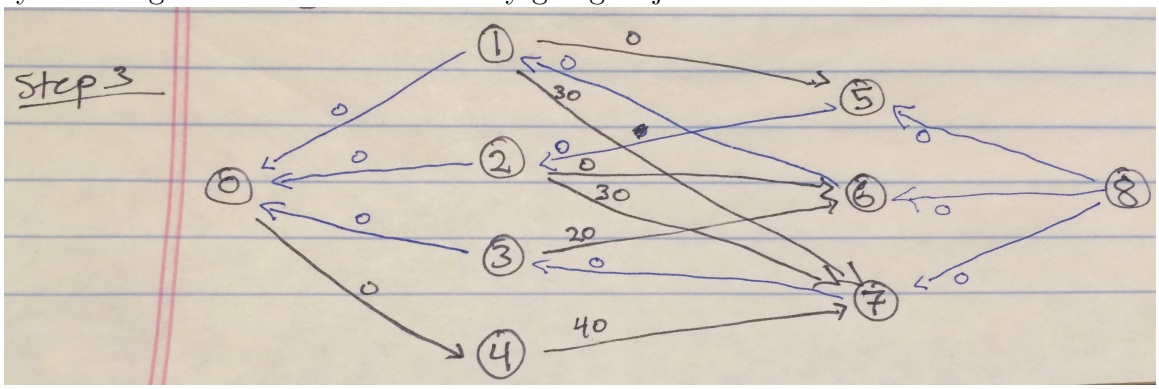
We then perform our dijkstra and update our copy of the graph such that it looks as below where we have reversed the edges $0 \to 3 \to 7 \to 8$ symbolizing going from job 1 to job 3.



This is followed by finding the path that goes $0 \to 1 \to 6 \to 8$, symbolizing that team 1 starts out by going to job 2.



Then for our last augmentation we find the path that goes $0 \to 2 \to 5 \to 8$, symbolizing that team 2 starts out by going to job 1.



Therefore our paths found can be understood as team 1 goes first to job 1 then to job 3 and team 2 goes only to job 2 which will produce the minimum cost of 80.

3 Problem D: Cutting Cheese

Results:

17274629	1712 Cutting Cheese	Accepted	JAVA	1.700	2016-04-27 22:19:28

Background:

Of course you have all heard of the International Cheese Processing Company. Their machine for cutting a piece of cheese into slices of exactly the same thickness is a classic. Recently they produced a machine able to cut a spherical cheese (such as Edam) into slices – no, not all of the same thickness, but all of the same weight! But new challenges lie ahead: cutting Swiss cheese.

Swiss cheese such as Emmentaler has holes in it, and the holes may have different sizes. A slice with holes contains less cheese and has a lower weight than a slice without holes. So here is the challenge: cut a cheese with holes in it into slices of equal weight.

By smart sonar techniques (the same techniques used to scan unborn babies and oil elds), it is possible to locate the holes in the cheese up to micrometer precision. For the present problem you may assume that the holes are perfect spheres.

Each uncut block has size $100x100x100$ where each dimension is measured in millimeters. Your task is to cut it into s slices of equal weight. The slices will be $100mm$ wide and $100mm$ high, and your job is to determine the thickness of each slice.

Input:

The input contains several test cases, each of them as described below:

The first line of the input contains two integers n and s, where $0 \leq n10000$ is the number of holes in the cheese, and $1 \leq s \leq 100$ is the number of slices to cut. The next n lines each contain four positive integers, $r, x, y,$ *and* z are the coordinates of the center, all in micrometers. The cheese block occupied the points (x, y, z) where $0 \leq x, y, z \leq 100000$, except for the points that are part of some hole. The cuts are made perpendicular to the z-axis.

You may assume that holes do not overlap but may touch, and that the holes are fully contained in the cheese but may touch its boundary.

Output:

For each test case, display the s slice thicknesses in millimeters, starting from the end of the cheese with $z = 0$. Your output should have an absolute or relative error of at most 10^{-6}.

3.1 Mathematical Formulation

Given an input of n totally encapsulated, non-overlapping spheres, each with a x position and r, radius we can determine where to make cuts in a block of cheese 100000 x 100000 x 100000

3.2 Solution

The main functionality of this algorithm is to compute the total volume of the block of cheese, then determine what the weight should be of each equally sliced piece and perform a binary search of segments (from the left side, $z = 0$) to find the segments which are within 10^{-6} of this target weight. We calculate the volumes of each block by using spherical segment calculations of all spheres within the range in question.

The only data structure we used was a two dimensional array **holes**$[n][2]$ such that each entry $holes[i][0]$ corresponds to the center z coordinate and $holes[i][1]$ is the radius of hole i. Aside from this we use the instance variable $v, goal, numHoles, numSlices$ to keep track of the total volume, the goal weight for evenly cut slices of cheese, the number of holes and the number of slices.

Algorithm 4 Main

```
procedure MAIN
    v, numHoles, numSlices, holes[numHoles][2] ← initialized
    for hole i ∈ [1..numHoles] do
        store i_z, i_r in index i
        update v
    goal ← v/numSlices
    Sort(holes) based off of left-most point on sphere on z-axis
    BINARYSEARCH( )
procedure BINARYSEARCH
    (double) low, high, last, cut ← 0, 100000, 0.0
    (int) slicePerformed ← 1
    while true do
        cut ← (h + l)/2
        (double) diff ← goal - ALLVOL(last, cut)
        if diff == 0 then
            PRINT((cut-last)/1000)
            last ← call; l ← last; h ← 100000;
            if slicePerformed++ == numSlices-1 then break;
        else if goal - val > 0 then l ← cut
        elseh ← cut
    PRINT((100000 - last)/1000);
```

Since we have sorted by left most part of each sphere, as soon as the leftmost point of the i^{th} hole is past the mark b, then the points $j \geq i$ are not contained in the bounds so we can break out of the loop and not calculate any more spheres. Otherwise we calculate the spherical segments of each sphere for which there is some portion of it in the band $[a, b]$. The link for equations used can be found here: http://mathworld.wolfram.com/SphericalSegment.html.

Algorithm 5 Computations

```
procedure allVol(double a, double b)
    vol ← DIM · DIM · (b − a)
    for i ∈ 1..numHoles do
        if i_z − i_r > b then break;
        if i_z + i_r < a then continue;
        val− = volInRange(a, b, i);
    return val;
procedure volInRange(double a, double b, int i)
    Computes the spherical segment based off of the formulas in the link above.
```

3.3 Correctness

Corollary 7.

It is sufficient for us to simply view the spheres on a 1-dimensional plane because we are ensured that each sphere is fully encapsulated within the block of cheese and that no two spheres are overlapping, therefore if we take the band of [a,b], if some potion of sphere i is within this we can calculate the volume displaced using only the portion (i_a, i_b) that overlaps with sphere i and its radius, i_r.

Proposition 8.

Given the correct process for determining the volume of portions of spheres, a binary search for cuts in the cheese will give us the correct cuts to make.

Proof.

We know that binary search is a viable option for when we know the stopping criteria and can calculate or lookup each intermediate stage. Therefore if we perform s binary searches, decreasing our range appropriately each time we find a cut, and we know our stopping criteria as *goal*; we can calculate the volume displaced by each portion of a sphere in intermediate ranges $[a, b]$, ultimately giving us the appropriate cut coordinates along the z-axis. □

3.4 Analysis

Proposition 9. *The space complexity of this algorithm is* ***O(N)***

Proof.
This is due to the fact that all we store are the z-coordinate and the radius of each hole.

Giving us a space complexity of **O(N)**

□

Proposition 10. *The time complexity of this algorithm is* ***O(N·log(N) + N·S)***

Proof. This is the case because our initial sorting of the holes takes $N \cdot log(N)$, then within the binary search ($S \cdot log(100,000)$) where S is number of slices asked to perform and 100,000 is the range that we perform our binary search on. Since log(100,000) $\sim$ 16, we can just say that this is O(S) which we do at most (worst-case if every interval contains every hole) N volume calculations.

Giving us a time complexity of **O(N·log(N) + N·S)**

□

3.5 An Example

Given the input of:
1 2
10000 10000 10000 50000

Which is asking for 2 slices with one hole positioned at (x, y, z) = (10000, 10000, 50000) and the radius is 10000. As we read in we learn:

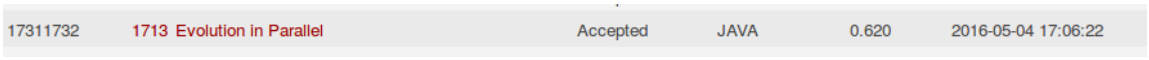
$$vol = 9.958112 \cdot 10^{14} \implies goal = 4.979056 \cdot 10^{14}.$$

For this case we record everything and make our initial cut at 50000, for which we calculate the sphere takes up $2.094395 \cdot 10^{12}$ $micrometers^3$ which gives us our goal of $4.979056 \cdot 10^{14}$. Therefore we print out:

50.000000
50.000000

4 Problem E: Evolution

Results:



Background:

It is 2178, and alien life has been discovered on a distant planet. There seems to be only one species on the planet and they do not reproduce as animals on Earth do. Even more amazing, the genetic makeup of every single organism is identical!

The genetic makeup of each organism is a single sequence of nucleotides. The nucleotides come in three types, denoted by 'A' (Adenine), 'C' (Cytosine), and 'M' (Muamine). According to one hypothesis, evolution on this planet occurs when a new nucleotide is inserted somewhere into the genetic sequence of an existing organism. If this change is evolutionarily advantageous, then organisms with the new sequence quickly replace ones with the old sequence.

It was originally thought that the current species evolved this way from a single, very simple organism with a single-nucleotide genetic sequence, by way of mutations as described above. However, fossil evidence suggests that this might not have been the case. Right now, the research team you are working with is trying to validate the concept of "parallel evolution" – that there might actually have been two evolutionary paths evolving in the fashion described above, and eventually both paths evolved to the single species present on the planet today. Your task is to verify whether the parallel evolution hypothesis is consistent with the genetic material found in the fossil samples gathered by your team.

Input:

The input file contains several test cases, each of them as described below.

The input begins with a number $n, (1 \leq n \leq 4000)$ denoting the number of nucleotide sequences found in the fossils. The second line describes the nucleotide sequence of the species currently living on the planet. Each of the next n lines describes one nucleotide sequence found in the fossils.

Each nucleotide sequence consists of a string of at least one but no more than 4 000 letters. The strings contain only upper-case letters 'A', 'C', and 'M'. All the nucleotide sequences, including that of the currently live species, are distinct.

Output:

For each test case, display an example of how the nucleotide sequences in the fossil record participate in two evolutionary paths. The example should begin with one line containing two integers s_1 and s_2, the number of nucleotide sequences in the fossil record that participate in the first path and second path, respectively. This should be followed by s_1 lines containing the sequences attributed to the first path,

in chronological order (from the earliest), and then s_2 lines containing the sequences attributed to the second path, also in chronological order. If there are multiple examples, display any one of them. If it is possible that a sequence could appear in the genetic history of both species, your example should assign it to exactly one of the evolutionary paths.

If it is impossible for all the fossil material to come from two evolutionary paths, display the word 'impossible'.

4.1 Mathematical Formulation

Given a currently present string s_g and N strings of average length S, determine if the N strings can be grouped into 2 sequences of evolutionary paths sub_1 and sub_2. A valid evolutionary path constitutes of having each string to be a sub-sequence [defined below] of the proceeding string. Both sub_1 and sub_2 should be sub-sequences of s_g and if there is a string which does not belong to either sub_1 or sub_2 or they are not a sub-sequence of s_g, then the algorithm will return "impossible".

4.2 Solution

The main functionality of this algorithm is to first order the given strings in ascending order in terms of length of string. Then we will build sub_1 and sub_2 top down, first appending s_g then ensuring that every subsequent string added to either sub_1 or sub_2 must be a sub-sequence [see helpers section for this] of the currently smallest length element on them.

Algorithm 6 Main

```
procedure MAIN
    for each test case do
        INITIALIZE // see below
        boolean failed, shared ← false; String lastS1, lastS2 ← initialized
        sub1.PUSH(sg); sub2.PUSH(sg);
        for each string i from N..1 do
            token ← sequence[i]
            if shared then
                if isSubSequence(token, sharedList.peekFirst()) then
                    sharedList.addFirst(token);
                else if isSubSequence(token, lastS1) then
                    sub1.PUSH(token)
                    sub2.PUSH(sharedList)
                    shared ← true;
                else if isSubSequence(token, lastS2) then
                    sub2.PUSH(token)
                    sub1.PUSH(sharedList)
                    shared ← true;
                elsefailed ← true
            else
                (boolean) inS1, inS2 ← isSubSequence(token, sub1,2.peek())
                if inS1&&inS2 then
                    shared ← true; lastS1, lastS2 ← sub1,2.peek();
                    sharedList.ADDFIRST(token)
                else if inS1 then sub1.PUSH(token)
                else if inS2 then sub2.PUSH(token)
                elsefailed ← true
        if failed then print("impossible");
        else
            if shared then sub1.PUSH(sharedList)
            print(sub1.size() sub2.size()); print(sub1); print(sub2);
```

A string, s_1 is called a sub-sequence of s_2 if every letter in s_1 is present in s_2 and they occur in the same order (though there can be different letters in between them).

Algorithm 7 Helpers

```
procedure isSubSequence(String s1, String s2)
    (int) i ← 0
    for j ∈ [0..s2.length] do
        if s1.charAt(i) == s2.charAt(j) then
            if ++i == s2.length() then return true;
    return false;
procedure initialize( )
    N ← number of strings; sequence[N] ← initialized and filled;
    s_g ← goal string (currently present string)
    sort(sequence) by ascending order
    Stack<String> sub_1, sub_2 ← initialized;
    LinkedList<String> sharedList ← initialized;
```

4.3 Correctness

Proposition 11.
If ∃ two valid sub-sequence sets, sub_1 and sub_2, for the current species s_g, this algorithm will determine an instance of them and report them in chronological order. i.e. ($\forall$ $|s_i|<|s_j| \in sub_1$ or sub_2).

Proof.

We know that if two sub-sequence sets exist then every string s_i, $i \in [1..N]$ must be a sub-sequence of s_g. Also ∃ at most two strings of the same length within the entirety of the set of strings. If ∃ more then it is impossible to make two sub-sequences as the strings given are unique and therefore cannot be sub-sequences of one another if they are the same length by definition of a sub-sequence. Therefore the only way that a string s_i can be a sub-sequence of another string s_j is if $|s_i|<|s_j|$. So therefore we can build from the longest string and adhere to the following rules:

$$s_i \; sub-sequence \; of \begin{cases} both \; sub_1 \; and \; sub_2 \\ only \; sub_1 \\ only \; sub_2 \\ neither \end{cases}$$

The cases then are simple, if ONLY sub_1 or sub_2, then the string gets placed in the subsequent one, if neither then the sequences are seen to be impossible. Therefore the only tricky situation is if $s_i \in sub_1$ *and* sub_2. If this is the case, we will continue to read in strings appending them to this shared sub-sequence set until we either reach a

string which is not a sub-sequence of them, or run out of strings. If we reach a string s_k which is not a sub sequence of the shared set, then we check $s_k \in sub_1$ *or* sub_2; if it is, we place it in the subsequent set and place the shared set on the other, if not then we know that it is impossible to have this $s_k \in sub_1$ *or* sub_2. This follows by the fact that, if there are 2 valid sub-sequences, every string must belong to one or the other, therefore s_k must exist in one of the sets, so we determine which one and then likewise, the sub-sequence must also exist on one of the sets, however since it can be placed on either, it is not constrained to which until we see an instance that must be on a specific one so we can decide at that time which it must be on. □

4.4 Analysis

Here we will refer to N being number of strings inputed and S as the average length of all the strings.

Proposition 12. *The space complexity of this algorithm is* $\boldsymbol{O(N \cdot S)}$

Proof.

This is due to the fact that we store every string in our $sequence[N]$, our two stacks, and the queue. Worst case the sum of the sizes of the two stacks and the queue will be N because each string is only present at one at a time. Therefore, since each of these contain every string, it becomes $S \cdot (N + N)$.

Giving us a space complexity of $\mathbf{O(N \cdot S)}$

□

Proposition 13. *The time complexity of this algorithm is* $\boldsymbol{O(N \cdot log(N) + N \cdot S)}$

Proof. This is the case because we first sort all of the strings based off of length $(N \cdot log(N))$. Then we will perform the $isSubSequence$ method on each string a maximum of 4 times. Twice to determine sub for the top of s_1 and s_2 then possibly to the string that may be place on top of it and then possibly again if they are the end of a split as described in the proof above therefore giving us $N \cdot 4 \cdot S$ as worst case.

Giving us a time complexity of $\mathbf{O(N \cdot log(N) + N \cdot S)}$

□

4.5 An Example

Given the input of:
5
AACMMAA
C
A
AA
AAAA
ACMAA

We note have $s_g = AACMMAA$ and note that we should have a valid subsequence here. So we begin by setting $sub_1 = \{AACMMAA\}$ and $sub_2 = \{AACMMAA\}$, now we read in our longest string and put it on $shared = \{ACMAA\}$ then read in the next string and see that $AAAA \notin ACMAA$ so we check sub_1 and put it there such that $sub_1 = \{AACMMAA, AAAA\}$ and $sub_2 = \{AACMMAA, ACMAA\}$. Next string we note can go on either so we have $shared = \{AA\}$, and with the next string we get $A \in shared$, so $shared = \{AA, A\}$. However the next and final string $C \notin shared$, but is in sub_2 so we place it in there so $sub_1 = \{AACMMAA, AAAA, AA, A\}$ and $sub_2 = \{AACMMAA, ACMAA, C\}$. Then we print out such that we get:

3 2
A
AA
AAAA
C
ACMAA

5 Problem F: Keyboarding

Results:

17306383	1714 Keyboarding	Accepted	JAVA	13.780	2016-05-04 01:50:35

Background:

How many keystrokes are necessary to type a text message? You may think that it is equal to the number of characters in the text, but this is correct only if one keystroke generates one character. With pocket-size devices, the possibilities for typing text are often limited. Some devices provide only a few buttons, significantly fewer than the number of letters in the alphabet. For such devices, several strokes may be needed to type a single character. One mechanism to deal with these limitations is a virtual keyboard displayed on a screen, with a cursor that can be moved from key to key to select characters. Four arrow buttons control the movement of the cursor, and when the cursor is positioned over an appropriate key, pressing the fifth button selects the corresponding character and appends it to the end of the text. To terminate the text, the user must navigate to and select the Enter key. This provides users with an arbitrary set of characters and enables them to type text of any length with only five hardware buttons. In this problem, you are given a virtual keyboard layout and your task is to determine the minimal number of strokes needed to type a given text, where pressing any of the five hardware buttons constitutes a stroke. The keys are arranged in a rectangular grid, such that each virtual key occupies one or more connected unit squares of the grid. The cursor starts in the upper left corner of the keyboard and moves in the four cardinal directions, in such a way that it always skips to the next unit square in that direction that belongs to a different key. If there is no such unit square, the cursor does not move.

Input:

The input file contains several test cases. The first line of the input contains two integers r and c ($1 \leq r, c \leq 50$), giving the number of rows and columns of the virtual keyboard grid. The virtual keyboard is specified in the next r lines, each of which contains c characters. The possible values of these characters are uppercase letters, digits, a dash, and an asterisk (representing Enter). There is only one key corresponding to any given character. Each key is made up of one or more grid squares, which will always form a connected region. The last line of the input contains the text to be typed. This text is a non-empty string of at most 10,000 of the available characters other than the asterisk.

Output:

For each test case, display the minimal number of strokes necessary to type the whole text, including the **Enter** key at the end.

5.1 Mathematical Formulation

Given an input of a keyboard of size RxC and a string of length S that can be typed out on the keyboard, determine the minimum number of keystrokes that are required to type in the string that is used which consist of starting in the top left corner of the keyboard, visiting each letter in the string and pushing each button, then moving to and pressing the "$*$" button.

5.2 Solution

The main functionality of this algorithm is to perform $S + 1$ bfs's for each one of the letters $s_1, s_2, ...s_S \in s$ the input string. We move onto the next level once we have found the corresponding goal letter for that level, however each level is fully explored unless we have found the terminating key and for each time that we find the corresponding key for that level we move up.

The only data structures we use are the $keyboard[R][C]$ as a character array, the input string $target$, $|target| = S$ and the three dimensional $seen[S][R][C]$ to keep track of which keys have already been visited for each level.

Algorithm 8 Main

```
procedure MAIN
    for each case do
        R, C ← sizes of array
        keyboard[R][C] ← filled from input
        target.concat(*) ← from input
        seen[target.length()][R][C] ← initialized
        dr[4], dc[4] ← right, left, up, down
        PRINT(bfs(keyboard, target)+target.length())
```

Algorithm 9 BFS

```
procedure BFS(keyboard, target)
    q ← initialize Integer Queue for bfs; (int) temp = 0
    while target.charAt(temp) == keyboard[0][0] do temp++;
    q.add(temp, 0, 0, 0); for depth(index in string), row, col, distance
    seen[0][0][0] = true
    while !q.empty() do
        index, row, col, l++ ← 4x(q.pop())
        char goal ← target.charAt(index);
        for direction d_i ∈ {left, right, up, down} do
            dr, dc ← row + d_i, col + d_i
            char lastLetter ← keyboard[dr − d_i][dc − d_i];
            while dr, dc inBounds do
                char thisLetter ← keyboard[dr][dc]
                if thisLetter == lastLetter then
                    dr, dc += d_i
                    continue;
                if !seen[index][dR][dC] then
                    seen[index][dR][dC] = true
                    if thisLetter == goal then
                        temp = 0
                        while target.charAt(temp) == keyboard[0][0] do
                            temp++
                            if index + temp == length then return l
                            q.add(index + temp)
                            seen[index + temp][dR][dC] = true
                    elseq.add(index)
                    q.add(dr, dc, l)
                break;
```

5.3 Correctness

Proposition 14.

We will determine the minimum number of keystrokes required to type in the string $s = s_1 s_2 ... s_S$ on the given keyboard.

Proof.

First use the fact that we can determine the shortest distance from any one key k_i to another k_j on the keyboard throughh a bfs originating at k_i. Therefore we enact a S depth bfs on the keyboard such that we go from depth $1 \to 2$ after we find s_1 as one of the keys k_i on the first keyboard and start searching from key k_i on lever 2 for s_2. The entire time however we continue to search on keyboard 1 until we either

have explored the entire board; if we come across s_1 on say k_l we will move up to board 2 searching from k_l unless it has already been seen. The reason this works is that if the original key is not the optimal first key to explore from, then we will have not seen the optimal route on one of levels currently exploring which will be explored from another searching origin. □

5.4 Analysis

For purposes of this problem we will let S be the length of the input string and RxC be the dimensions of the given keyboard.

Proposition 15. *The space complexity of this algorithm is* ***O(S·R·C)***

Proof.

This is due to the fact that we are storing a three dimensional array or S 2-D boolean array of the keyboard of size RxC

Giving us a space complexity of $\mathbf{O(S \cdot R \cdot C)}$

□

Proposition 16. *The time complexity of this algorithm is* ***O(S·R·C)***

Proof. This is the case because the worst case for this algorithm is to do a complete bfs for each of the S keyboards.

Giving us a time complexity of $\mathbf{O(S \cdot R \cdot C)}$

□

5.5 An Example

Given the input of:
6 4
AXYB
BBBB
KLMB
OPQB
DEFB
GHI*
AB

We begin by appending $*$ to the target so target $= AB*$ and $|target| = 3$ then we begin our search. noting that we immediately find 'A' so we move on starting at level 2, searching for 'B'. We note that the distances from A can be as followed: (- $\implies \infty$)

A X Y B $\rightarrow$ **0** 1 2 3
B B B B $\rightarrow$ 1 2 3 -
K L M B $\rightarrow$ 2 3 4 -
O P Q B $\rightarrow$ 3 4 - -
D E F B $\rightarrow$ 4 - - -
G H I * $\rightarrow$ - - - -

For sakes of making this as painless as possible I will fill in the 3rd level in one swoop to show you what we will do, bolding where we have come up from level 2 to this level 3.

A X Y B $\rightarrow$ 2 3 4 **3**
B B B B $\rightarrow$ **1 2 3** -
K L M B $\rightarrow$ 2 3 4 -
O P Q B $\rightarrow$ 3 4 - -
D E F B $\rightarrow$ 4 - - -
G H I * $\rightarrow$ - - - 4

Depending on the implementation, the arrays may be different, however we see here that the shortest route would be starting A $\rightarrow$ top right B $\rightarrow *$ which is reachable in a path length of 4. This is what the bfs would return and therefore will 4 and $4 + 3 = 7$ which is the least amount of key-strokes required.

6 Problem I: Ship Traffic

Results:

17334126	1717 Ship Traffic	Accepted	JAVA	1.780	2016-05-09 03:42:21

Background:

Ferries crossing the Strait of Gibraltar from Morocco to Spain must carefully navigate to avoid the heavy ship traffic along the strait. Write a program to help ferry captains and the largest gaps in strait traffic for a safe crossing.

Your program will use a simple model as follows. The strait has several parallel shipping lanes in eastwest direction. Ships run with the same constant speed either eastbound or westbound. All ships in the same lane run in the same direction. Satellite data provides the positions of the ships in each lane. The ships may have different lengths. Ships do not change lanes and do not change speed for the crossing ferry.

The ferry waits for an appropriate time when there is an adequate gap in the ship traffic. It then crosses the strait heading northbound along a north-south line at a constant speed. From the moment a ferry enters a lane until the moment it leaves the lane, no ship in that lane may touch the crossing line. Ferries are so small you can neglect their size. Your task is to find the largest time interval within which the ferry can safely cross the strait.

Input:

The input file contains several test cases, each of them as described below.

The first line of input contains six integers: the number of lanes n ($1 \leq n \leq 10^5$), the width w of each lane ($1 \leq w \leq 1000$), the speed u of ships and the speed v of the ferry ($1 \leq u, v \leq 100$), the ferry's earliest start time t_1 and the ferry's latest start time t_2 ($0 \leq t_1 \leq t_2 \leq 10^6$). All lengths are given in meters, all speeds are given in meters/second, and all times are given in seconds.

Each of the next n lines contains the data for one lane. Each line starts with either 'E' or 'W', where 'E' indicates that ships in this lane are eastbound and 'W' indicates that ships in this lane are westbound. Next in the line is an integer m_i, the number of ships in this lane ($0 \leq m_i \leq 10^5$ for each $1 \leq i \leq n$). It is followed by m_i pairs of integers l_{ij} and p_{ij} ($1 \leq p_{ij} \leq 1000$ and $-10^6 \leq p_{ij} \leq 10^6$). The length of ship j in lane i is l_{ij} and p_{ij} is the position at time 0 of its forward end, that is, its front in the direction it moves.

Ship positions within each lane are relative to the ferry's crossing line. Negative positions are west of the crossing line and positive positions are east of it. Ships do not overlap or touch, and are sorted in increasing order of their positions. Lanes are ordered by increasing distance from the ferry's starting point, which is just south of the first lane. There is no space between lanes. The total number of ships is at least 1 and at most 10^5.

Output:

For each test case, display the maximal value d for which there is a time s such that the ferry can start a crossing at any time t with $s \leq t \leq s + d$. Additionally the crossing must not start before time t_1 and must start no later than time t_2. The output must have an absolute or relative error of at most 10^{-3}. You may assume that there is a time interval with $d >$0.1 seconds for the ferry to cross.

6.1 Mathematical Formulation

Given a set of N ships with their position and their speed as well as a ferry's speed and an interval of time that it can begin traveling.

6.2 Solution

To store our important data and ensure sorting is possible, this algorithm uses a helper class called InvalidInterval which stores a *start* time and an *end* time as well as supports a Comparable of InvalidInterval such that the start times of two InvalidIntervals will be compared. These InvalidInterval start and end times are computed utilizing the basic function $time = \frac{distance}{velocity}$. We treat each east and west bound ship the same simply by multiplying the east bound boats starting position by -1. Once we have built up our set of InvalidIntervals, we sort them and then perform a greedy algorithm on them to determine the largest valid start interval for the ferry (see correctness for in depth of how this works).

Algorithm 10 Main

```
procedure MAIN
    for each case do
        numLanes, laneWidth, shipSpeed, ferrySpeed ← initialized
        fStart, fEnd, invalidIntervals[] ← initialized
        for lane ∈ [1, numLanes] do
            boolean eastBound ← initialized; int numShips ← initialized
            for numShips do
                front, back ← initialized
                if back<0 then continue
                timeForBoatFront = front / shipSpeed
                timeForFerryFront = (lane·laneWidth) / ferrySpeed
                start = timeForBoatFront - timeForFerryFront
                restOfBoat = length / shipSpeed + width / ferrySpeed
                end = start + restOfBoat
                if end < fStart || start > fEnd then continue;
                start = Math.max(start, fStart);
                end = Math.min(end, fEnd);
                invalidIntervals.add(new InvalidInterval(start, end))
    sort(invalidIntervals)
    determineMaxValidInterval(invalidIntervals)
```

Algorithm 11 Determine Valid Interval

```
procedure determineMaxValidInterval(invalidIntervals)
    if noships then print(fEnd - fStart); return
    largestGap ← invalidIntervals[0].getStart() - fStart
    endTime ← invalidIntervals[0].getEnd()
    curGap
    for curShip ∈ [1..numShips] do
        if invalidIntervals[curShip].getStart() >endTime then
            curGap ← invalidIntervals[curShip].getStart() - endTime
            largestGap ← Math.max(largestGap, curGap)
            endTime ← invalidIntervals[curShip].getEnd()
        else
            endTime = Math.max(endTime, invalidIntervals[curShip].getEnd())
    curGap ← fEnd-endTime;
    largestGap ← Math.max(largestGap, curGap);
    printf("
```

6.3 Correctness

Corollary 17.
Given both the ship and ferry, for each ship, we can determine an interval of time that the ferry cannot go due to this ship.

Proof.
We can determine the time at which each ship will be crossing the path of the ferry and therefore also the latest time that the ferry can leave before it would collide with this boat and then the earliest time that the ferry could leave such that this ship has passed. Therefore we can determine the interval of time that is between this and record that. □

Proposition 18.
Given a set of invalid intervals for the ferry to start at, we can determine the longest gap of time that the ferry is able to leave on.

Proof.
We do this through a greedy approach which the main idea is to sort all of the intervals in increasing order such that the earliest is first. Then we will essentially lay all of the intervals flat by keeping track of any overlap that occurs between the intervals and since they are sorted by start time then this implies that if we have a set of intervals i_1, i_2, i_3 (already sorted), then if i_3 overlaps with i_1, then i_1 and i_2 are guaranteed to also overlap. We linearly go throughh the list keeping track of our last known position as well as our largest gap so that way if there is no overlap between the last known position and the start of the next interval, we are guaranteed that

this is a gap, therefore we compare it to the longest gap that we know thus-far and if it is longer, we will know the longest gap. Thus when we go throughh every interval we are guaranteed to find the largest gap. □

6.4 Analysis

For purposes of this problem we will let N be the total number of ships per test case

Proposition 19. *The space complexity of this algorithm is* ***O(N)***

Proof.

This is due to the fact that we only store the start and end times for each ship in the form of a linked list first and then convert it to an array.

Giving us a space complexity of **O(N)**

□

Proposition 20. *The time complexity of this algorithm is* ***O(N·log(N))***

Proof. This is the case because our operations are as follows, for each ship O(N) we compute the start and stop times and enter them into a linked list O(1) $\implies$ O(N). Then we convert the linked list into an array O(N) and sort them by earliest start time O($N \cdot log(N)$). Then we go throughh each interval for each ship linearly keeping track of our largest gap seen thus-far O(N). Then we have $(N + N \cdot log(N) + N)$

Giving us a time complexity of **O(N·log(N))**

□

6.5 An Example

Given the input of:
1 100 5 10 0 200
W 4 100 100 100 300 100 700 100 900

We begin by understanding what the above numbers mean. We read this as there is 1 lane, each lane has a width of 100 m, each ships speed is 5 $\frac{m}{s}$ and the ferry's speed is 10 $\frac{m}{s}$, the earliest time our ferry can leave is time 0 and the latest is time 200. The next line tells us that in the first lane, there are 4 boats are traveling West-bound. The 1^{st} one is located at position 100 with a length of 100, the 2^{nd} located at position 300 with a length of 100, the 3^{rd} located at position 700 with a length of 100 and the 4^{th} located at position 900 with a length of 100.

We note here that none of the ships will be overlapping at any point since they are all in the same lane, with 100 units between each of them and are all traveling the same speed. We calculate then what time each one will intersect with the line that the ferry is traveling on and find that the intervals that they are crossing to be:

interval 1 = 10.0, 40.0
interval 2 = 50.0, 80.0
interval 3 = 130.0, 160.0
interval 4 = 170.0, 200.0

Though unessesary for this problem, we would ordinarily sort these by earliest start time and begin by saying our largest window is 10.0 - 0 = **10.0** and keep track of the farthest endpoint seen thus-far, 40.0. Then we check to see if our farthest endpoint and the start of interval 2 overlap, they do not so we check the gap = 10.0 which is the same so we update our farthest endpoint to be 80.0 and continue. Following this we check to see if 80.0 >start of interval 3, it is not so we check the gap which is 130.0 - 80.0 = **50.0** which is our largest gap thus-far so we record it as well as the farthest point 160.0. Then we check our interval 4, not overlapping so we check to see what the gap is there, it is 10.0 which is <50.0 so we record it's last position which is 200.0 and continue. Since we have run out of ships, now we check to see if the difference of our end time (200.0) and the last position is larger than our current largest gap i.e. 200.0 - 200.0 >50.0? No so we return 50.0 as our answer.