1 UVA Results

1.1 Problem A: Amalgamated Artichokes

17347468	1709 Amalgamated Artichokes	Accepted	JAVA	4.550	2016-05-11 13:29:21

1.2 Problem C: Catering

17316759	1711 Catering	Accepted	JAVA	1.360	2016-05-05 15:24:13

1.3 Problem D: Cutting Cheese

17349092	1712 Cutting Cheese	Accepted	JAVA	2.780	2016-05-11 21:55:16

1.4 Problem E: Evolution

17349302	1713 Evolution in Parallel	Accepted	JAVA	0.510	2016-05-11 23:23:16

1.5 Problem I: Ship Traffic

17347911	1717 Ship Traffic	Accepted	JAVA	1.760	2016-05-11 14:50:23

2 Problem A: Amalgamated Artichokes

2.1 Problem Specification

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 modelled reasonably accurately with the following equation:

$$price(k) = p(sin(ak + b) + cos(ck + d) + 2)$$

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 k.

2.2 Mathematical Formulation

In this problem, given N, we have to find the largest decline in $price(k)$ where k is in $[1, N]$. In other words, we need to find i, j where $i, j \in [1, N]$ and $i < j$ such that $price(i) - price(j)$ is the largest difference.

2.3 Problem Size

We have an unknown number of test cases, we have to find the largest decline in prices in under three seconds. However, we know that $1 \leq p \leq 1000$, $0 \leq a, b, c, d \leq 1000$, and $1 \leq n \leq 10^6$.

2.4 Solution Description

To solve this problem, we simply iterate from 1 to N, keeping track of two things: the largest decline we've seen so far, and the highest price we've seen so far. At every step, we compute the price and check to see if that price is greater than the highest price we've seen so far. If it is, we update the highest price. Then, we check to see if the difference between the highest price and the current price is greater than the largest decline we've seen so far. If it is, we update that value as well. Pseudocode for this process is below:

Algorithm 1 Main runner of our program.

```
procedure MAIN
    in ← new Scanner for system.in
    while in has more integers to read do
        p ← next integer from in
        a ← next integer from in
        b ← next integer from in
        c ← next integer from in
        d ← next integer from in
        n ← next integer from in
        maxPrice ← 0
        maxDecline ← 0
        for i ∈ [1, n] do
            currentPrice ← p * (sin((a * k + b)%(2π)) + cos((c * k + d)%(2π)) + 2)
            maxPrice ← max between maxPrice, currentPrice
            maxDecline ← max between maxDecline, maxPrice − currentPrice
```

A small note: when we run our price function, we take the sine and cosine of the remainder of the inner value with 2π. This provides the same value as it would if we didn't use the remainder (by the cyclic nature of sine and cosine), but it speeds up the computation.

2.5 Examples and Test Cases

Imagine, from some input, we produced this prices from 1 to N:

19 12 13 11 20 14 10

We would start at 19 and find that to be our highest price seen so far. Then, until 20, which is our next highest price seen so far we find the differences in prices to be 7, 6, and then 8. So, right before 20, our highest price seen so far is 19 and our largest difference seen so far is 8. Then, we run into 20, and so we update our highest price seen so far to 20.

From there until the end of the prices, we find the differences between 20 and the rest to be 6 and then 10. At the 14, we find the difference to be 6, which is less than our largest decline seen so far (8), so we don't update it. However, at the 10, our local decline from 20 is 10, which is greater than our largest decline seen so far, so we update our largest decline seen so far to 10. That's the end of input, so we find that our largest decline overall equals 10.

2.6 Correctness

Proposition 1. *Given N, our algorithm finds the largest decline between two points with N.*

Proof. Our algorithm keeps track of the highest price seen so far and the largest decline seen so far. We keep track of the highest price for a couple of reasons. Say we have three nonnegative integers a, b and c, such that $a > b$ and $c < a, c < b$. Then, we know that $a - c > b - c$. In our algorithm, a, b and c represent prices. Thus, if we find some low price c that occurs after both a and b, the larger of the two prices between a and b will have a larger decline. Thus, keeping track of the highest price seen so far alongside the largest delcine seen so far will lead to the right answer. □

2.7 Efficiency

Proposition 2. *Given N, our algorithm finds the maximum decline in $O(N)$ worst case time.*

Proof. Since all we do is iterate from 1 to N and perform a constant time operation at each step, our algorithm takes $O(N)$ time in the worst case. □

Proposition 3. *Given N, our algorithm uses up $O(1)$ space to find the maximum decline.*

Proof. Our algorithm takes constant worst case space since the only things we ever keep track of are p, a, b, c, d, the highest price seen so far, and the maximum decline, all of which are independent of the problem size. □

3 Problem C: Catering

3.1 Problem Specification

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.

3.2 Mathematical Formulation

3.3 Problem Size

We are given an unknown amount of test cases that we have to solve in under three seconds. However, our given variables do have bounds. The number of teams, k, and the number of requests, n, are both bounded by 0 and 100, inclusive. The ith request (counting from 1) has $n - i + 1$ connections to the requests after it.

3.4 Solution Description

To solve this problem, we model the input as a bipartite graph, and then try to find a minimum cost matching. Therefore, the main parts of our algorithm are: read in the input and build an adjacency matrix for a bipartite graph, and then, using that graph and a minimum cost bipartite matching algorithm with Dijkstra's single source shortest paths (with an indexed minimum priority queue), find the minimum cost matching.

The main runner for our program is written below. It is essentially the algorithm described above.

Algorithm 2 Main runner of our program.

```
procedure Main
    in ← new Scanner for system.in
    while in has more integers to read do
        n ← next integer from in
        k ← next integer from in
        g ← BuildGraphFromInput(in, n, k)
        minCost ← GetMinCostMatching(g, n, k)
        Print(minCost)
```

Before we can even solve the problem, we have to build a graph which represents it. When we call the BuildGraphFromInput method, we read in the input from standard input and we build our bipartite graph using that data. The nodes on the left part of the bipartite graph represent preceding locations, and nodes on the right part of the bipartite graph represent destinations. The algorithm for building the graph is detailed below:

Algorithm 3 Builds a bipartite graph from the input.

```
procedure BuildGraphFromInput(in, n, k)
    numNodes ← 1 + k + (2 × n)
    leftStart ← 1
    rightStart ← k + n
    g ← new array of size (numNodes × numNodes)
    Fill up all indices of g with ∞
    teamValues ← new array of size (numNodes)

    for i ∈ [0, numNodes) do
        teamValues[i] ← next integer from in
    for u ∈ [leftStart, k] do
        for i ∈ [0, n) do
            g[u][k + n + i] ← teamValues[i]
    for u ∈ [leftStart + k, rightStart) do
        for v ∈ [u + n, numNodes − 1) do
            g[u][v] ← next integer from in
    for i ∈ [leftStart, rightStart) do
        g[0][i] ← 0
    for i ∈ [rightStart, numNodes − 1) do
        g[i][numNodes − 1] ← 0

    return g
```

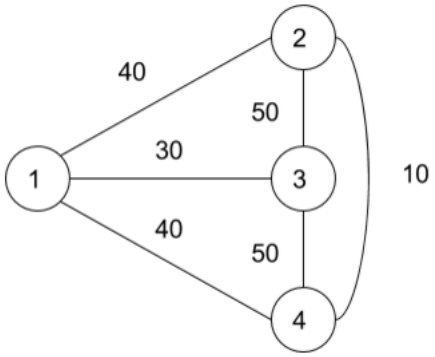
After building our bipartite graph, we simply run a minimum cost bipartite matching algorithm (our specific implementation utilizes Dijkstra's Single Source Shortest Paths algorithm with an indexed minimum priority queue) to find the minimum matching cost. This is done within the GetMinCostMatching method. After doing that, we just print out the value we found and, thus, the problem is solved.

3.5 Examples and Test Cases

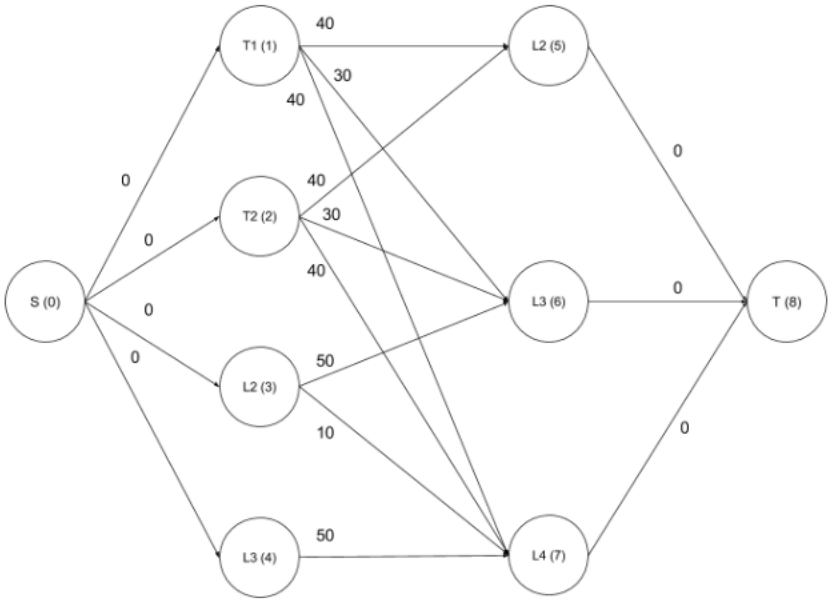
To show how our algorithm actually works, we'll run through a quick example using one of the sample test cases. Say our input is as follows:

```
3 2
40 30 40
50 10
50
```

This means that we have three requests ($n = 3$) and two teams ($k = 2$). We can draw this input as a location graph, showing us the connections between different locations and the costs to move from one location to another.



However, to actually solve this problem, we need to create a bipartite graph so we can use the minimum cost bipartite matching algorithm. The graph and the corresponding adjacency matrix can be found below (vertex numbers are in parentheses, T means team, and L means location):



$$\begin{bmatrix}
\infty & 0 & 0 & 0 & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & 40 & 30 & 40 & \infty \\
\infty & \infty & \infty & \infty & \infty & 40 & 30 & 40 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & 50 & 10 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 50 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty
\end{bmatrix}$$

When we run the minimum cost bipartite matching algorithm on this adjacency matrix, we find that the minimum cost is 80. That corresponds to sending team 1 to location 2 then 4, and sending team 2 to location 3.

3.6 Correctness

Lemma 4. *The minimum cost bipartite matching algorithm finds the minimum cost matching for graphs with only imperfect matchings.*

Proof. We know that the minimum cost bipartite matching algorithm finds the minimum cost *perfect* matching. However, it does so by making sure that at every step the matching at that step is the lowest costing one. Furthermore, at every step, the cardinality of the matching increases by one. Therefore, the algorithm finds the maximum cardinality minimum cost matching even for graphs where no perfect matching is possible. □

Proposition 5. *By modeling the catering problem as a bipartite graph, by running the minimum cost bipartite matching algorithm on it we get the minimum cost of servicing each request.*

Proof. In our formulation, we model the catering problem as a bipartite graph where the nodes on the left represent starting locations and the nodes on the right represent ending locations. Therefore, a match between a node on the right and a node on the left represents the movement of a team from one location to another location. The number of nodes on the right side of the graph is equal to the number of requests, so the highest number of matches we could have is equal to the number of requests, representing that each request is serviced. Therefore, by running the minimum cost bipartite matching on this graph, even though a perfect matching may be impossible, we find the minimum cost of servicing all requests (see the previous lemma). □

3.7 Efficiency

Proposition 6. *Given N requests and K teams, our algorithm finds the minimum cost of servicing all requests in $O((N^3)log(N+K))$ time in the worst case.*

Proof. In our formulation, we have a total of $1 + K + (2 * N)$ nodes (two for the source and the sink, one for each team, one for the last location, and two for all other locations). We also know that running Dijkstra's single source shortest paths with an indexed minimum priority queue takes $O(ElogV)$ time in the worst case, where E is the number of edges and V is the number of nodes. In our graph, we have the number of nodes found above, as well as $O(N+K)$ edges from source to the left nodes, $O(N^2)$ edges between the left nodes and the right nodes (each location has one less edge than the previous, and there are N locations so $N(N+1)/2)$, and $O(N-1)$ edges from each of the right nodes to the sink. Therefore, our Dijkstra's takes $O(N^2log(N+K))$ time. Then, we run Dijkstra's N times to match all requests, so our algorithm takes $O(N^3log(N+K))$ time in the worst case. □

Proposition 7. *Given N requests and K teams, our algorithm takes up $O((N+K)^2)$ to find the minimum cost of servicing all requests.*

Proof. As said before, there are $1 + K + (2 * N)$ nodes in our graph, which bounds us to $O(N+K)$ nodes. The big data structure we use is an adjacency matrix with $O(N+K)$ entries for each of the nodes, meaning that its size is $O((N+K)^2)$ in the worst case. □

4 Problem D: Cutting Cheese

4.1 Problem Specification

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 fields), 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 $100 \times 100 \times 100$ where each dimension is measured in millimeters. Your task is to cut it into s slices of equal weight. The slices will be 100 mm wide and 100 mm high, and your job is to determine the thickness of each slice.

4.2 Mathematical Formulation

Each block of cheese is $100 \times 100 \times 100$ measured in millimeters. There will be n holes and s slices to make. Each hole is a sphere. Each hole also has an x, y and z coordinate, which represents where the center of the hole is within the block of cheese. Each hole also has an r, which corresponds to the radius of the sphere. No holes overlap, but they touch, and each hole is fully contained by the block of cheese (or just touches its border). Each of these figures is measured in micrometers.

4.3 Problem Size

We have an unknown amount of test cases to solve, and three seconds to solve all of them. We know that $0 \leq n \leq 10000$, $1 \leq s \leq 100$, and $0 \leq x, y, z \leq 10000$.

4.4 Solution Description

To solve this problem, we have two main data structures: the block of cheese and the holes. The block of cheese containers spheres which represents the holes it has. To distill our algorithm down to a couple of basic steps, we first construct the block of cheese containing all of the holes. Then, we find the actual volume of the cheese by subtracting the volume of each hole from the total block's volume. Then, we find the volume each slice needs to have by subtracting the found volume by the number of slices we need. Then, we use binary search to find where we make slices such that

each slice is the same volume, leading to each slice taking up the same weight. We do this for every slice except for the last slice, which we do outside the loop.

Algorithm 4 Main function for our program.

```
procedure Main
    in ← new Scanner for system.in
    while in has more more to read do
        numHoles ← next integer from in
        numSlices ← next integer from in
        block ← new block of cheese
        holes ← read in hole from input

        Add holes to block, updating its volume for each hole
        Sort holes by bottom point closest to the bottom of block
        sliceVolume ← volume of block divided by numSlices
        lastCutEnd ← 0

        for i ∈ [1, numSlices − 1] do
            cutHeight ← NextSlice(block, lastCutEnd, sliceVolume)
            cutWidth ← cutHeight − lastCutEnd
            Print("%.9f\n", cutWidth/1000.0)
            lastCutEnd ← cutHeight

        Print("%.9f\n", (100000 − lastCutEnd)/1000.0)
```

Algorithm 5 Finds the next slice to cut.

```
procedure NextSlice(block, start, targetVolume)
    lo ← start
    hi ← 100000
    while hi > lo do
        mid ← (hi + lo)/2.0
        sliceVolume ← volumeOfSlice(block, start, mid)
        if sliceVolume is within 10000 to targetVolume then
            return mid
        else if sliceVolume > targetVolume then
            hi ← mid
        else
            lo ← mid
```

Algorithm 6 Finds the volume of the block between two points

```
procedure NextSlice(block, start, end)
    volume ← 100000 × 100000 × (end − start)
    for each hole within the block do
        if hole ends before start then continue
        if hole starts after end then break
        volume ← volume− VolumeBetween(hole, start, end)
    return volume
```

Algorithm 7 Finds the volume of a sphere between two points

```
procedure VolumeBetween(hole, start, end)
    if Entire hole is within start and end then
        return volume of hole
    else if Only bottom section of hole is within start and end then
        return volume of bottom section
    else if Only top section of hole is within start and end then
        return volume of top section
    else if Only middle section of hole is within start and end then
        return volume of hole− volume of top section − volume of bottom section
```

Using the four algorithms above, we can find the proper width of each slice so that each slice is the same weight. The only thing to note of is the fact that we break out of the loop in algorithm 6 if the current hole starts after our range. We do that because we sorted the holes before hand, and doing so saves us time.

4.5 Examples and Test Cases

To show how our algorithm actually works, we will go through a test case. Imagine our block of cheese had two holes in it, one positioned at (20000, 10000, 30000), and one positioned at (75028, 61234, 80000), both with a radius of 10000 (all of these values are measured in millimeters). Then, we know the volume of each sphere is 4.19×10^{12}.

Since our base volume is 10^{15}, we take away $2 * 4.19 \times 10^{12}$ away from it, reducing our block's volume to 9.9162×10^{14}. Then, imagine we had to make two slices. This means that each slice would have to have a volume of 4.9581×10^{14}.

When we run our binary search, we find that cutting at 50000 micrometers from the bottom of the block of cheese will give us this volume. Then, our two slices both have a width of 50000 micrometers, or 50 millimeters.

4.6 Correctness

Invariant 8. *We do not care about the x coordinate and y coordinate of each hole.*

Proof. We know this is true since we know that each hole does not overlap with any other hole. Thus, we can move each hole to any x coordinate and any y coordinate and our answer would not be affected. □

Proposition 9. *Using binary search finds the best spot to cut for another slice of the same weight.*

Proof. When we binary search in this algorithm, we know where we can start and the target volume. Our starting point is determined by the ending height of the last slice made, so no slices ever overlap. Then, we set our highest point to be the top of the block. If the volume of this cut is too large, we lower our cutting point to the midpoint between our starting position and the highest point. Then, if the volume of this slice is too large, we lower the cutting point again (by half). If its too large, we raise by half. Otherwise, if we are within 1000000 of the target volume (our error), we return the slice. Once we find our slicing point, we update the starting point of the next slice to that point. We continue this for all slices, thus finding best width to use for each slice. □

4.7 Efficiency

Proposition 10. *Given N holes and M slices, our algorithm takes up* $O(NlogN + MN)$ *time to find the slice widths in the worst case time.*

Proof. We have N holes, so the sorting step at the start of our algorithm takes $O(NlogN)$ time. Then, we perform a binary search for each slice, totaling M binary searches. Each binary search takes at most 30 iterations (to get under our error value), and also takes $O(N)$ time to compute the volume of the slice. Therefore, our worst case time is $O(NlogN + MN)$ time. □

Proposition 11. *Given N holes and M slices, our algorithm takes up* $O(N)$ *space.*

Proof. The only thing we really keep track of is each hole, since we output each slice width when we find it. There are N holes, so our algorithm takes up $O(N)$ space in the worst case. □

5 Problem E: Evolution in Parallel

5.1 Problem Specification

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.

5.2 Mathematical Formulation

We are given N strings and a source string. Our goal is to find if there exists two evolutionary paths so that we can reach the source string by two different sequences of the N strings, where each string is used by only one sequence.

5.3 Problem Size

We have an unknown amount of test cases and three seconds to solve all of them. We now that $1 \leq N \leq 4000$, and that each nucleotide sequence is between 1 and 4000 characters long. We also know that each nucleotide sequence is unique.

5.4 Solution Description

To solve this problem, we try to find two unique evolutionary sequences originating from our source string. To do so, we will start with our source string and go through our ancestors from the ancestor with the longest sequence to the ancestor with the shortest sequence. If we can use all of the sequences, with each one appearing in only one path, we know that two evolutionary paths are possible.

We move from the current sequence backwards because a sequence is an ancestor if it is a *subsequence* of the current sequence. Why this is true will be explained later.

Each evolutionary path starts with the current sequence. Then, we begin to go through our ancestor list. For each ancestor, we have a couple of possibilites:

1. The ancestor can be part of both evolutionary paths.
2. The ancestor can be part of one evolutionary path.
3. The ancestor can be part of neither of the evolutionary paths.

For each of these possibilities, we do a different thing.

For the first possibility, we keep track of ancestors that can appear in both paths. We first check to see if that list is empty. If it is, then we add it to the list of common ancestors. Then, if it isn't, we check to see if the current ancestor can be ancestor of the previous common ancestor. If it can, we also add it to the list of common ancestors. Otherwise, we just add it to one of the evolutionary paths and move on.

In the second possibility, we just add the ancestor to the respective evolutionary path.

If the third possibility happens, we know that there cannot be two evolutionary paths, since there is an ancestor that is not part of either of the two evolutionary paths, so we stop our search here and conclude that it is impossible.

Pseudocode for this algorithm is included below:

Algorithm 8 Determines if a given ancestor can be part of a given path.

procedure isPathAncestor($path, ancestor$)
    $latestPathMember \leftarrow$ latest member of $path$
    return if $ancestor$ is a subsequence of $latestPathMember$

Algorithm 9 Determines if we can add a new common ancestor.

procedure isPathAncestor($commonAncestors, ancestor$)
    $latestCommon \leftarrow$ latest member of $commonAncestors$
    return $lastestCommon$ is **null or** $ancestor$ is a subsequence of $latestCommon$

Algorithm 10 Runs our program.

```
procedure Main
    in ← new Scanner for system.in
    while in has more to read do
        n ← next integer from in
        current ← next string from in
        ancestors ← read ancestor sequences from in
        Sort ancestors from longest sequence to shortest

        pathOne ← new set containing current
        pathTwo ← new set containing current
        commonAncestors ← new empty set
        impossible ← false
        for each ancestor within ancestors do
            if ancestor is ancestor to both and can add new common ancestor then
                Add ancestor to commonAncestors
            else if ancestor is an ancestor for pathOne then
                Add ancestor to pathOne
                Add all within commonAncestors to pathTwo
            else if ancestor is an ancestor for pathTwo then
                Add ancestor to pathTwo
                Add all within commonAncestors to pathOne
            else
                impossible ← true
                break

        if not impossible then
            Add all within commonAncestors to pathOne
            Remove current from both paths
            Print out size of both paths on the same line
            Print out contents of both paths
        else
            Print out "impossible"
```

5.5 Examples and Test Cases

Imagine our source genome was "ACMA", and our other sequences were "ACM," "ACA," and "AMA." Then, we would see that the first genome "ACM" can be ancestors to both evolutionary paths, so we add it to the common path. We also find that the next genome "ACA" is common to both, but it is not an ancestor to "ACM", so we just add it to the first path, and add "ACM" to the second path.

Then, we move on to the third genome "AMA." We find that "AMA" isn't an

ancestor to both "ACM" and "ACA," so it isn't an ancestor to both paths, so we say that its impossible to have 2 evolutionary paths from these genomes.

5.6 Correctness

Lemma 12. *A given sequence is an ancestor of another one if it is a subsequence of the other one.*

Proof. By the problem specification, the species evolves by adding certain characters into its existing sequence, without changing the order in which the old sequence appears in the new sequence. This, by definition, is a subsequence. □

Proposition 13. *Given a source string and a set of ancestors, our algorithm determines whether or not two evolutionary paths exist.*

Proof. If two evolutionary paths exist, each gvien genome sequence must be a subsequence of the original sequence. If there is a subsequence that *isn't* part of the original sequence, then there is a sequence that cannot turn back into the original sequence, and thus a third path exists that that doesn't evolve into the original sequence, contradicting the fact that there is only one species on the planet.

If that does not happen, then by the previous lemma, if we go through the ancestors from the longest ancestor to the shortest ancestor, then each of the shorter ancestors are a subsequence of the previous ancestor on one of the paths. This is also due to the definition of a subsequence.

The only hard case is when an ancestor can be part of both paths. We handle this two ways: either we add it to the common ancestor path if it is compatible (subsequence), or if we can start a new common ancestor path (the current path is empty). Once we find an ancestor that isn't common to both paths, we add that ancestor to the path its compatible with, and add the common ancestors to the other path. We can do this since we know that the common path can belong to any of the two paths, and if the current ancestor isn't common to both paths, it also isn't common to the set of common ancestors, so we have to add the common ancestors to path that is incompatible with the current ancestor to keep the validity of both paths. □

5.7 Efficiency

Proposition 14. *Given N sequences, with the length of the longest sequence being M, we find two evolutionary paths, if possible, in $O(NlogN + NM)$ time in the worst case.*

Proof. It takes us $O(NlogN)$ time to sort the sequences from longest to shortest, and for each sequence, it takes us $O(M)$ time to check to see if it is an ancestor to one of the paths. Thus, our algorithm takes $O(NlogN + MN)$ worst case time. □

Proposition 15. *Given N sequences, our algorithm takes u $O(N)$ space in the worst case.*

Proof. We know that each of the N sequences has to appear in only one of the paths. We keep track of both paths, so we keep track of a maximum of N sequences, meaning our worst case space is $O(N)$. □

6 Problem I: Ship Traffic

6.1 Problem Specification

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 find 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.

6.2 Mathematical Formulation

We are given a set of ships, along with their position, speed, lane, and direction. We are also given the speed of the ferry. We have to find the largest interval of time the ferry has to cross safely.

6.3 Problem Size

There is an unknown amount of test cases, and we have three seconds to solve all of them. There will always be between 1 and 10^5 lanes, and the width of each lane will always between 1 and 1000. Each lane has the same width. Ships in the same lane travel in same direction.

The speed of the ferry is 1 and 100. The speed of the ships are also between 1 and 100. Each ship travels at the same speed. We are also given the earliest start time and the latest start time, both between 0 and 10^6.

Finally, we each lane we are given between 0 and 10^5 ships. Each ship has a length between 1 and 1000, and also has a starting position between -10^6 and 10^6.

6.4 Solution Description

To solve this problem, we calculate the intervals where the ferry cannot cross the strait. To do so, we calculate, for each lane, when the ferry cannot cross. We start with the closest lane, find where the ferry cannot cross, and then continue to the next

lane. Then, we merge all of those invalid times together and find the largest stretch of free time, after sorting the invalid times by starting time.

To find the times when the ferry cannot cross the lane, we have to look at each ship. We need to know when each ship begins to block the ferry from crossing (this includes when the ferry gets hit by the ship during crossing) and when each ship stops blocking the ferry. If these times are actually within when the ferry can move, we save the time.

Pseudocode for our algorithm can be found below:

Algorithm 11 Main Runner of our program.

```
procedure MAIN
    in ← new Scanner for system.in
    while in has more to read do
        numLanes ← next integer from in
        laneWidth ← next integer from in
        shipSpeed ← next integer from in
        ferrySpeed ← next integer from in
        earliestStart ← next integer from in
        earliestEnd ← next integer from in
        blockedTimes ← ∅

        for each lane ∈ [1, numLanes] do
            ships ← read in all ship information
            for each ship ∈ ships do
                blockStart ← when the ship begins blocking the ferry's path
                blockEnd ← when the ship stop blocking the ferry's path
                if the block is within our ferry's travel times then
                    add interval from blockStart to blockEnd to blockedTimes

        Sort blockedTimes by starting time
        Merge overlapping blocked times in blockedTimes together
        Print longest gap between blockedTimes
```

6.5 Examples and Test Cases

Imagine we had the sample input given to us:

```
3 100 5 10 0 100
E 2 100 -300 50 -100
W 3 10 60 50 200 200 400
E 1 100 -300
1 100 5 10 0 20
```

Then, we would have the following invalid intervals:

0.0 to 4.0,
10.0 to 30.0,
20.0 to 40.0,
30.0 to 60.0,
50.0 to 80.0,
60.0 to 100.0

The largest gap between non-overlapping intervals is 6.0, so we return 6.0.

6.6 Correctness

Proposition 16. *Our algorithm finds the longest interval of time where the ferry can cross the strait.*

Proof. By finding for each lane and for each ship within the lane the time where the ferry cannot cross that lane, we obtain a set of invalid times. During all of these times, the ferry cannot cross the strait at all, since it would be blocked at some point. Thus, if we combine all overlapping times together, we get continuous intervals of time where the ferry cannot cross the strait. Then, the remaining gaps begin these merged intervals represent the times when the ferry can cross the entire straight safe all the way. Thus, if we choose the largest out of these intervals, we will have the largest time interval during which the ferry can cross safely. □

6.7 Efficiency

Proposition 17. *Given N total ships, our algorithm finds the longest safe interval that the ferry can cross in $O(NlogN)$ time.*

Proof. We iterate through each lane, and for each lane we iterate through all of the ships in the lane. For each ship, we perform a constant time operation. Thus, to go through N ships, we take $O(N)$ time. Then, in the worst case, we have N invalid intervals, so it takes $O(NlogN)$ time to sort them and another $O(N)$ time to find the largest safe interval. Thus, our algorithm takes $O(N)$ time in the worst case. □

Proposition 18. *Given N total ships, our algorithm takes up $O(N)$ space to solve the problem in the worst case.*

Proof. We use a constant number of space for each ship, so that is already $O(N)$ space. Then, as said before, in the worst case we have N invalid intervals. We use a constant amount of space per invalid interval, so that takes up $O(N)$ time as well. Thus, in the worst case, our algorithm takes up $O(N)$ space in the worst case. □