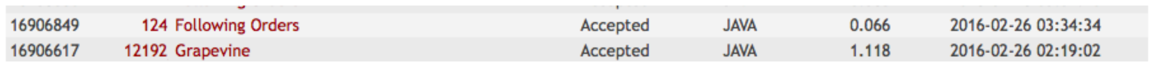
1 Results from UVA



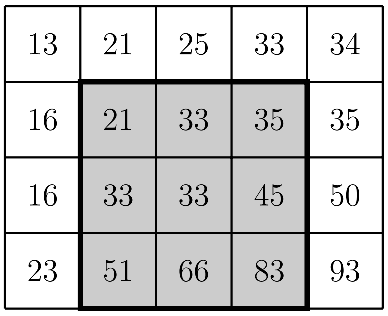
16906849	124 Following Orders	Accepted	JAVA	0.066	2016-02-26 03:34:34
16906617	12192 Grapevine	Accepted	JAVA	1.118	2016-02-26 02:19:02

2 UVA Problem 12192: Grapevine

In Quadradonia, all rural properties are square, all have the same area, all are perfectly flat and all have the sides aligned to the North-South and West-East axes. Since properties are flat, the hills in Quadradonia look like a series of huge stairs' steps, with different heights. In a certain mountain, an interesting situation occurs in a rectangular area of N ×M properties. Starting from anywhere within the region, traversing it in the West to East direction, the properties have non-descending heights. Similarly, traversing that region in the North to South direction, starting from anywhere, the properties have also non-descending heights.

A large wine company in Quadradonia wants to rent some properties from that region to grow wine grapes. The company is interested in some special varieties of wine grapes, which are productive only if grown in properties whose heights are within a certain interval. That is, the company is interested in renting properties whose heights are equal to or higher than a given altitude L, and equal to or lower than a given altitude U . To make it easier for harvesting, the rented properties must form a contiguous area. And since everyone in Quadradonia likes squares, the area to be rented must have the shape of a square.

The company has not yet decided which variety of grapes it will grow, and therefore it has a list of queries involving intervals, one for each grape variety. The figure below shows an area of interest of dimensions 4 × 5 (in number of properties) with examples of areas the company could rent to grow grapes in heights within the intervals given in the picture.



13	21	25	33	34
16	21	33	35	35
16	33	33	45	50
23	51	66	83	93

You must write a program that, given the description of the rectangular area of interest in the mountain, and a list of queries containing height intervals, determines, for each query, the largest side, in number of properties, of a contiguous square area with heights within the specified interval.

2.1 Mathematical Formulation

Given an N x M board, where each n_i ¡ n_{i+1} and m_j ¡ m_{j+1} so that the smallest value is in index (0, 0) and the largest is in (1, 1). We will determine the maximum square area that is within the user specified bounds (L, U).

2.2 Solution

In our main functionality we will be reading in the data and storing it in a 1-D representation of a 2-D topological map. Each index will represent a square lot in this **int[N*M]**, int array of size N*M. We use regular int values to store the Lower and Upper bounds (L,U) and number of cases (Q). For each case, we will go through each row executing a **modified binary search** to search for the index of the lot which is the lowest of the in-bound lots (left-most lowerbound). This is explained in Algorithm II. Once this value is retrieved we use the fact that each n_i ¡ n_{i+1} and m_j ¡ m_{j+1} meaning that we **only need to check the diagonal** from this lowerBound to ensure that this square is a valid one. We will record the maximum size and continue checking. We will terminate either on the last row or the last possible row. i.e. if we have found 3 to be the max size, once we reach the second to last row, we cannot find a better than 3 square so we are done.

Algorithm 1 GrapeVine main

```
procedure MAIN( )
    $reader \leftarrow$ BufferedReader
    $stringBuilder \leftarrow$ StringBuilder
    $line \leftarrow$ reader.READLINE( )
    while there is a new line do // get the dimentions of the board
        $N, M \leftarrow$ line.SPLIT(bySpace)
        Ensure N, M are in bounds
        $map \leftarrow$ int[N][M] // this is actually 1-Dimensional
        for $i \in length(N)$ do
            $splitLine \leftarrow$ reader.READLINE( ).SPLITLINE(bySpace)
            for $j \in length(M)$ do
                $map[i][j] \leftarrow$ Integer.PARSEINT(splitLine[j])
        $Q \leftarrow$ Integer.PARSEINT(reader.readLine())
        $stringBuilder \leftarrow$ new StringBuilder()
        while $Q \neq 0$ do
            $L, U \leftarrow$ Lower and Upper bounds
            $max \leftarrow 0$
            for $i \in N$ && $(N - i + 1) > max$ do
                $lowerIndex \leftarrow$ lowerBound(map, i, (M-1), L) // see below
                if lowerIndex == 0 then
                    continue
                for $j = max..<M$ do
                    $n \leftarrow$ i + j
                    $m \leftarrow$ lowerBound + j
                    if $n \geq N$ || $m \geq M$ || $map[n][m] > U$ then
                        break
                    if $j + 1 > max$ then
                        $max \leftarrow$ j + 1
            stringBuilder.APPEND(max)
            Q–
        stringBuilder.APPEND(-)
        $line \leftarrow$ reader.READLINE( )
```

The point of the following algorithm is to binary search a row within the map and returns the lowest possible valid lot.

Algorithm 2 GrapeVine get lower bound

```
procedure lowerBound(int[] map, int row, int size, int L)
    $lower \leftarrow 0$, $higher \leftarrow$ size, $answer \leftarrow$ -1, mid
    while $lower \leq higher$ do
        $mid \leftarrow$ lower + (higher - lower) / 2
        if $map[row][mid] \geq L$ then
            $answer \leftarrow$ mid
            $higher \leftarrow$ mid - 1
        else
            $lower \leftarrow$ mid + 1
```

2.3 Correctness

Proposition 1.

The lowerBound method will give us the index m_j of the lot which has the lowest elevation within the bounds (U, L) within the specified row n_i.

Proof.

We begin this by stating that the length of any given row is $k + 1$ and, since this method call is independent between rows we can say that we are only looking at the elements $m_0, m_1, ..., m_k$ in row n_i. We do a binary search then, our checking statement will be to see if the midpoint is within the bounds (U,L) s.t. first time through we check $m_{k/2} \in (U, L)$. If in bounds, our new upper bound will be $m_{k/2-1}$, otherwise our new lower bound will be $m_{k/2-1}$. This process is done until we have gone through the complete search (lg(k)). □

Proposition 2.

In any square area, the top left corner lot will have the lowest elevation and the bottom right will have the largest area.

Proof.

This is more or less given in the problem description but to put it into a mathematical formulation we say that given the square area

$$\begin{bmatrix} x_{i,j} & x_{(i+1),(j+1)} & \cdots & x_{(i),(j+k)} \\ x_{(i+1),j} & x_{(i+1),(j+1)} & \cdots & x_{(i+1),(j+k)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(i+k),(j)} & x_{(i+k),(j+1)} & \cdots & x_{(i+k),(j+k)} \end{bmatrix}$$

we are gauranteed that (1) $x_{i,j} < x_{i,(j+1)} < ... < x_{i,(j+k)}$ and (2) $x_{i,j} < x_{(i+1),j} < ... < x_{(i+k),j}$. This translates accross rows so that in (1) if we replace i with $(i + 1)...(i + k)$ the same relationship will hold and similarly, replacing j with $(j+1)...(j+k)$ in (2) will also hold.

$\therefore$ The lowest value will be $x_{i,j}$ and the highest will be $x_{(i+k),(j+k)}$.

□

Proposition 3.
Given the left most valid lot $x_{i,j}$ on a given row i, when we diagonally search through the topological map we will find the maximum size square possible originating from that row (not unique). i.e. look at $x_{(i+1),(j+1)}, x_{(i+2),(j+2)}, ..., x_{(i+p),(j+p)}$ where $0 \leq p \leq k = min(N, M)$.

Proof.
Given $x_{i,j}$, we know from proposition 2 that $x_{i,j}<x_{i,(j+1)}$, $x_{i,j}<x_{(i+1),j}$, and all three $< x_{(i+1),(j+1)} \implies$ as we go diagonally we are increasing in area covered s.t. the bottom right corner, $x_{(i+p),(j+p)}$ for $i \leq p \leq k$ is the biggest and $x_{i,j}$ is the smallest.Therefore the square matrix produced by this area is within the bounds (L,U) if both $x_{i,j}$ *and* $x_{(i+p),(j+p)} \in (L, U)$. Lets say the diagonal check will return that the $x_{(i+(p+1)),(j+(p+1))}$ fails $\implies$ that all lots $x_{(i+(p+2)),(j+(p+1))}, x_{(i+(p+3)),(j+(p+1))}$, ... ,$x_{(i+(p+k),(j+(p+1))}$ and $x_{(i+(p+1)),(j+(p+2))}, x_{(i+(p+1)),(j+(p+3))}, ..., x_{(i+(p+1)),(j+(p+k))}$ will fail as well as all $x_{(i+y),(j+z)}$ are also invalid $\forall\ (p+1) < y, z \leq\ k$ where $k \in min(N, M)$. This means that all lots in the matrix below will be invalid.

$$\begin{bmatrix} x_{(i+(p+1)),(j+(p+1))} & x_{(i+(p+1)),(j+(p+2))} & \cdots & x_{(i+(p+1)),(j+(p+k))} \\ x_{(i+(p+2)),(j+(p+1))} & x_{(i+(p+2)),(j+(p+2))} & \cdots & x_{(i+(p+2)),(j+(p+k))} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(i+(p+k)),(j+(p+1))} & x_{(i+(p+k)),(j+(p+2))} & \cdots & x_{(i+(p+k)),(j+(p+k))} \end{bmatrix}$$

Assume the valid lot matrix starting at $x_{i,(j+1)}$ as its top left corner is larger than the lot matrix we have just found. $\implies$ this lot must contain at least $x_{(i+(p+1)),(j+(p+2))}$ which would make the matrix size (p+1), one bigger than the lots previously found. However we just showed that $x_{(i+(p+1)),(j+(p+2))}$ which is a contradiction. we can replace this with any $x_{i,(j+m)})$ where $1 \leq m \leq k$ and the same contradiction will arrise. $\therefore$ Given the left most valid lot, this is the source from which we can find one of the biggest lots in any given map. □

Proposition 4.
Our algorithm will find the size of the largest square region. Note that this does not return the lot numbers, just the size $\therefore$ if there are several lots of the same largest size, the algorithm will return the size of them.

Proof.
From proposition 3, we know that we can find the largest size of valid lots for any given row. $\implies$ By saving the max size seen thusfar and stopping our search when it is impossible to get a size bigger than the current max, we will have the largest possible area at the end. □

2.4 Analysis

For the following analysis, we will say that the topological map is of size NxM and each entry can be represented using the notation $x_{i,j}$ where $i \in [0, N], j \in [0, M]$. As i and j increase, so does the value of each subsequent $x_{k,l}$ for $i, j<k, l$ respectively.

Proposition 5. *The space complexity of this algorithm is* ***O(N·M)***

Proof. This is due to the fact that all of our data can be infered from simply looking at the topological map which is a One-Dimensional array of size N·M, all other information is stored in integers so these are constant.

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

□

Proposition 6. *The time complexity of this algorithm is* ***O(N·min(N,M))***

Proof. Let us address first the min(N,M) portion: the reasoning behind this is that as we do our diagonalization, our worst case is that we must go to the other side of the matrix i.e. $x_{i,j}$ to $x_{k,k}$ where $k = min(M, N)$ which is k comparisons. This this would be the bottom right most corner of the square matrix contained in the map (without the bounds).
Now we say for each row (**N** of them):

- **O(log(M))** Binary searching to find the leftmost lot within the bounds (L,U)
- **O(min(N,M))** Diagonally searching from the found lot $\hat{t}$o the first out of bounds lot

Putting these together we get O(N·log(M) + N·min(N,M))

Giving us a time complexity of **O(N·min(N,M))**

□

2.5 An Example

See **grapeVine_example.txt** for a tracing with the input: 4 5
13 21 25 33 34
16 21 33 35 35
16 33 33 45 50
23 51 66 83 93
3
22 90
33 35
20 100

3 UVA Problem 124: Following Orders

Background

Order is an important concept in mathematics and in computer science. For example, Zorn's Lemma states: "a partially ordered set in which every chain has an upper bound contains a maximal element." Order is also important in reasoning about the x-point semantics of programs. This problem involves neither Zorn's Lemma nor x-point semantics, but does involve order.

The Problem

Given a list of variable constraints of the form x <y , you are to write a program that prints all orderings of the variables that are consistent with the constraints. For example, given the constraints x <y and x <z there are two orderings of the variables x, y, and z that are consistent with these constraints: x y z and x z y .

The Input

The input consists of a sequence of constraint specifications. A specification consists of two lines: a list of variables on one line followed by a list of contraints on the next line. A constraint is given by a pair of variables, where x y indicates that x < y. All variables are single character, lower-case letters. There will be at least two variables, and no more than 20 variables in a speci cation. There will be at least one constraint, and no more than 50 constraints in a speci cation. There will be at least one, and no more than 300 orderings consistent with the contraints in a speci cation. Input is terminated by end-of- le.

The Output

For each constraint speci cation, all orderings consistent with the constraints should be printed. Order- ings are printed in lexicographical (alphabetical) order, one per line. Characters on a line are separated by whitespace.
Output for different constraint specifications is separated by a blank line

3.1 Mathematical Formulation

Given an input of N distinct letters ($2 \leq \text{N} \leq 26$) and a set of R rules ($1 \leq \text{R} \leq 50$). The algorithm will enumerate all M anagrams of the letters adhearing to the rules. $Note: \ worst\ case\ M \simeq N!$

3.2 Solution

Important Confusing Data Structures:

- LinkedList<Integer>[n] **G** : Each array slot corresponds to the symbol that appears in that index in the sorted version of the input. Each slot holds a linked list of the symbols that must appear after the given symbol according to the user inputted rules.
- int[n] **inDegree** : This is an array that stores (at each recursive step) the number of symbols that must come before it. If the index is -1 $\implies$ that this was seen already in the recursion and has already been recorded within this branch of recursion.
- char[n] **result** : This is an array storing the symbols being used so far in the recursive step. It is being used throughout all of the different branches and never copied.

In the first algorithm, the program initializes all of the data structures that will be used, filling in the pertainant information. Once the structures have been properly formed, the program will then call it's recursive method which will enumerate the valid strings in alphabetic order. If there is another case, it will reinitialize everything appropriately and repeat, otherwise it will terminate.

Algorithm 3 FollowingOrders main

```
procedure MAIN( )
    $map \leftarrow$ new int[26] // to store the index of each letter
    while inFile.HASNEXT( ) do // each case
        $symbols \leftarrow$ sorted char[n] of the symbols
        $comparisons \leftarrow$ char[2*r] of all the rules, each pair is a rule
        $inverseMap \leftarrow$ char[n]
        for ($i \in symbols.length$ do
            Fill in map and inverseMap using symbols
        $G \leftarrow$ LinkedList<Integer>[n] all of the rules for each letter
        $inDegree \leftarrow$ int[n] // stores indegree for each symbol
        // Fill in G and inDegree
        for $i \in comparisons.length$ do
            $less \leftarrow$ map[comparisons[i]]
            $greater \leftarrow$ map[comparisons[i+1]]
            G[less].ADD(greater)
            inDegree[greater]++
        $results \leftarrow$ char[n] // to store the results thusfar in recursion
        DFSHASH(n, inverseMap, inDegree, G, result)) // See Below
```

The main functionality of dfsHash is to do a depth first search recursively on the valid strings in alphabetic order. It accomplishes this through using the combination of the data structures explained above, inDegree and G. The way that they work is that, the algorithm will look through inDegree for the first occurance of a 0 $\implies$ the first letter that can go first (unimpeeded by rules) and record it in the first index of result. Then the algorithm will look at all of the letters that must come after it (found in G). For each of these letters, their inDegree will be decrimented by 1 soas to suggest that one of the letters coming before it has been written down. Then a recursive call is made, attempting to find the second letter in the same manner...etc. Once the first word has been found (all given symbols have been used up) the algorithm will back track and (from the second to last letter) look for the next occuring 0 in the array. This will be the second word. If there is none, it will kill this branch and return back to the previous step. $Note$: $The\ recursive\ call\ will\ re-increment\ all\ decrimented\ indicies\ within\ inDegree$

Algorithm 4 FollowingOrders Recursive Method

```
procedure dfsHash(left,inverseMap[],inDegree[],G,result[])
    // base case, this means we have all n letters in result
    if left == 0 then
        print(result)
    for $i \in results.length$ do
        if inDegree[i] == 0 then
            $inDegree[i] \leftarrow$ -1
            $results[results.length - left] \leftarrow$ inverseMap[i]
            for letter $l \in G[i]$ do
                inDegree[letter]-
            dfsHash(left-1, im, inDegree, G, result)
            for letter $l \in G[i]$ do
                inDegree[letter]++
            $inDegree[i] \leftarrow$ -1
```

3.3 Correctness

Proposition 7.
One call of the $dfsHash$ method will be able to explore all possible anagrams from this position in alphabetical order while adhearing to the rules.

Proof.
Using the fact that the inDegree array corresponds to how many non-used letters are valid at this stage; we will exhibit this by saying that an entry can be denoted as $d_0, d_1, ...d_k$ for $(k+1)$ distinct letters and $d_i<0 \implies already\ used$, $d_i = 0 \implies valid\ letter\ and\ d_i = j>0 \implies j\ non-used\ letters\ must\ come\ before\ it$. Since inDegree is the representation of all of the letters in alphabetic order $\implies$ the first 0, is the first valid letter, the next 0 is the next valid letter and so on and so forth. $\therefore$ we see that we will do all of the first possible ones first and then (starting from the last letter) we will work our way back up continuing our depth first search. This will do an in-order traversal of all the valid anagrams that can be produced. □

3.4 Analysis

For the following analysis, we will say that Given an input of N distinct letters (2 $\leq$ N $\leq$ 26) and a set of R rules (1 $\leq$ R $\leq$ 50). The algorithm will enumerate all M anagrams of the letters adhearing to the rules. $Note:\ worst\ case\ M \simeq N!$

Proposition 8. *The* <u>*space complexity*</u> *of this algorithm is* ***O(N + R)***

Proof.
This is due to the fact that all of our data is stored in 7 data structures:

- map: int array of size 26 (worst case for N)
- inverseMap: char array of size N
- symbols: char array of size N
- comparisons: char array of size $2 \cdot R$
- G: Integer Linked List Array which has size N and R Nodes
- inDegree: int array of size N
- results: char array of size N

The only points of contension here have to do with our recursive calls on inverseMap, G, inDegree, and results. If a new copy were made each call, we would have at most N of each of these, however since we are passing the object as a parameter, we are only using the same pointer, pointing to a single object for each so the same one is being accessed and modified each time. This allows for the algorithm to use only one object for each. Therefore, summing all of these up, we have a space complexity of S$(N + N + N + 2 \cdot R + N + R + N + N) =$ S$(6 \cdot N + 3 \cdot R)$

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

□

Proposition 9. *The time complexity of this algorithm is **O(M)***

Proof. This is the case because our algorithm reads in all of the input in linear time O(N + R) which are both trivially small. Then, for every recursive call, we search through the list of symbols (N) once. So we look at the case now that we are enumerating a word which is $c_0c_1c_2...c_n$ (W.L.O.G.)lets assume the only rule is that $c_0 < c_k$ which would be the largest amount of anagrams possible. We can say that anagrams starting with c_0 number roughly $(k-1)!$ and for each letter, we will only do k^2 k's. $\implies$ going through all of the letters we would only go through k k^3 times $\implies$ k^4, replacing k with N we have N^4 yeilding a complexity of $M + N^4 + N + R$

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

□

3.5 An Example

I did not have enough time to finish this section. However I have the correct output for the given example by the UVA site.