# Results from UVA

## 0.1 Problem A (page 2)

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

## 0.2 Problem D (page 5)

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

## 0.3 Problem E (page 9)

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

## 0.4 Problem F (page 15)

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

# 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 modelled 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 correspoing 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 through 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 through 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 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 le 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 $10^{-6}$.

## 2.1 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

## 2.2 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 dimentional 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 $i$. Asside 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 2** 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 3** Computations

**procedure** allVol(double a, double b)
  $vol \leftarrow DIM \cdot DIM \cdot (b - a)$
  **for** $i \in 1..numHoles$ **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.

## 2.3 Correctness

**Corollary 4.**

*It is sufficient for us to simply view the spheres on a 1-dimentional 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 calcuate the volume displaced using only the portion $(i_a, i_b)$ that overlaps with sphere i and its radius, $i_r$.*

**Proposition 5.**

*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 calcualte 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 calcualte the volume displaced by each portion of a sphere in intemediate ranges $[a, b]$, ultimately giving us the appropriate cut coordinates along the z-axis. □

## 2.4 Analysis

**Proposition 6.** *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 7.** *The time complexity of this algorithm is* ***O(N·log(N) + N·S·log(S))***

*Proof.* This is the case because our initial sorting of the holes takes $N \cdot log(N)$, then within the binary search ($S \cdot log(S)$) we do at most (worstcase if every interval contains every hole) $N$ volume calculations.

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

□

## 2.5 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:

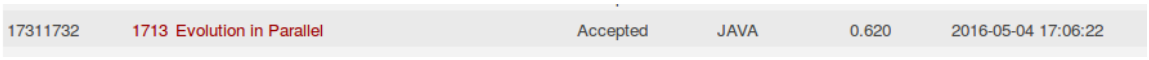
$$vol = 9.958112 \cdot 10^{14} \implies goal = 4.979056 \cdot 10^{14}.$$

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

# 3 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'.

## 3.1 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".

## 3.2 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 4** 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 5** 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 acending order
    Stack<String> sub_1, sub_2 ← initialized;
    LinkedList<String> sharedList ← initialized;
```

## 3.3 Correctness

**Proposition 8.**

*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. (∀ $|s_i|<|s_j| \in sub_1$ 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. □

## 3.4 Analysis

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

**Proposition 9.** *The space complexity of this algorithm is* ***O(N·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 $N$ becasue 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 **O(N·S)**

□

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

□

## 3.5 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

# 4 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 pocketsize devices, the possibilities for typing text are often limited. Some devices provide only a few buttons, signi cantly 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 fth 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 ve 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 ve 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.

## 4.1 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 keytsrokes 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.

## 4.2 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 correspodning 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 6** 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 7** 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;
```

## 4.3 Correctness

**Proposition 11.**

*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 through 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. □

## 4.4 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 12.** *The space complexity of this algorithm is* ***O(S·R·C)***

*Proof.*

This is due to the fact that we are storing a three dimentional 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 13.** *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 $\mathbf{O(S \cdot R \cdot C)}$

□

## 4.5 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.