Dynamic Programming

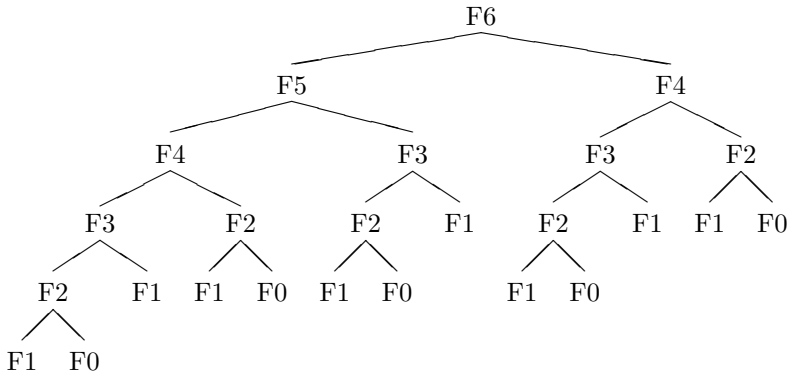
Due: April 27, 11:55 PM

Overview

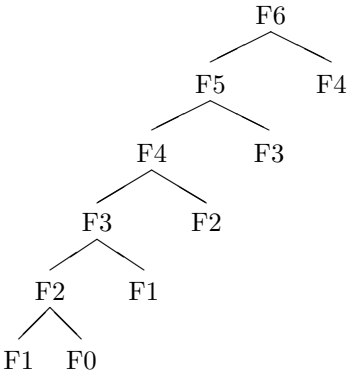
In this homework you will be solving several problems requiring the use of dynamic programming. Dynamic programming is an algorithmic technique which is usually based off of a recurrent formula, and some amount of starting states. Answers to sub-problems are constructed from smaller sub-problems (initially the starting states) until you have built up to your final answer. For example consider the famous Fibonacci sequence:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-2) + F(n-1) & \text{if } n \geq 2 \end{cases}$$

Here we have two starting states, and a recurrence relation. If we were to naively solve this using recursion we could end up with a solution with time complexity $O(2^n)$. Using this implementation calculating $F(100)$ on a computer able to perform $1,000,000,000,000$ operations per second would take approximately 40 billion years. Fortunately, using dynamic programming we can do *much* better. If we consider the recursion tree of the naive implementation, $F(6)$ will look as follows.



Here we are calculating $F(4)$ twice, $F(3)$ three times, and $F(2)$ five times. This tree will grow exponentially. If we try using a dynamic programming technique where we save the answers to our subproblems each time we calculate them, we can then perform a simple lookup to determine the answer of $F(n)$, after we have calculated it once. This will result in the following tree.



This tree will grow at a linear rate, rather than exponential, and makes Fibonacci much easier to solve for larger values. The concept of saving your subproblems as you calculate them is one of the main concepts of dynamic programming, and as you can see, it has a large impact on running time. Improving an algorithm that was $O(2^n)$ to $O(n)$.

General Specifications

We will provide an interface, containing the appropriate method headers for each method, including the extra credit methods. You will be creating the class Dynamic.java that will implement this interface. **Make sure you have all of these method headers in your program or it will not compile, if you choose not to write an extra credit problem stub it out!**

Imports from **java.util.*** are allowed, if for some reason you believe some other import would be useful to solve a problem (without making it trivial) feel free to ask.

Helper methods, and subclasses are also allowed.

For most of these problems there is a naive brute force implementation that will result in the correct answer; however, the point of this assignment is to implement more efficient algorithms using Dynamic Programming. So for all of these problems your solution should be returned within approximately 2 seconds for any valid input, the restrictions on valid input will be described within each problem's input section. As a rule of thumb about $1,000,000,000$ basic operations can be executed in a single second.

Within this assignment there will be several examples with the correct solutions provided to test each problem, these tests are **NOT** extensive, there will be cases that the examples do not test. Make sure you test out your code to ensure that it works, and runs in the appropriate time constraints.

Longest Common Subsequence

In this problem you will be given two strings, your task is to find the longest common subsequence within these two strings. Consider the strings: `"apple cider"`, and `"please"`. The longest common subsequence for this example would be `"plee"`. Now lets try to develop a dynamic programming solution for this problem.

We have two strings, A and B, of length m and n respectively. We will define the function $LCS(i, j)$ as the solution to the subproblem of the longest common subsequence of string A from $0 \rightarrow i$, $0 \leq i \leq m$, and of string B from $0 \rightarrow j$, $0 \leq j \leq n$.

Our goal is to solve $LCS(m, n)$ which can be done if we know the solution to $LCS(m-1, n-1)$, $LCS(m-1, n)$, and $LCS(m, n-1)$. If string A at the m_{th} character is the same character as string B at the n_{th} character, then we know that we can increase the length of the solution to $LCS(m-1, n-1)$ by one, because the next characters in each string match. Otherwise, the characters don't match and we will simply take the maximum of $LCS(m-1, n)$ and $LCS(m, n-1)$ to be the solution to $LCS(m, n)$. We now have a well defined set of subproblems we can use to build up to $LCS(m, n)$.

Determining our starting states is the next step. Lets look at $LCS(0, j)$ the answer will always be 0, because the empty string has no characters in common with any other string. Similarly $LCS(i, 0)$ will be 0 for all values of i. Now we have both our starting states, and a method for building up from our initial subproblems.

When two characters match, at (i, j), it will take the value of the diagonal to the upper-left, $(i-1, j-1)$, plus 1. If they do not match, we take the max of the value above it, and the value to the left as (i, j)'s value. Visually the algorithm will look as follows, for the strings: "sharpie" and "grape".

Initialization

		s	h	a	r	p	i	e
	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
r	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0

R is the first match

		s	h	a	r	p	i	e
	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
r	0	0	0	0	1	1	1	1
a	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0

Fill out the A row

		s	h	a	r	p	i	e
	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
r	0	0	0	0	1	1	1	1
a	0	0	0	1	1	1	1	1
p	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0

Fill out the P row

		s	h	a	r	p	i	e
	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
r	0	0	0	0	1	1	1	1
a	0	0	0	1	1	1	1	1
p	0	0	0	1	1	2	2	2
e	0	0	0	0	0	0	0	0

Fill out the E row

		s	h	a	r	p	i	e
	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
r	0	0	0	0	1	1	1	1
a	0	0	0	1	1	1	1	1
p	0	0	0	1	1	2	2	2
e	0	0	0	1	1	2	2	3

Final, LCS has a length of 3

		s	h	a	r	p	i	e
	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
r	0	0	0	0	1	1	1	1
a	0	0	0	1	1	1	1	1
p	0	0	0	1	1	2	2	2
e	0	0	0	1	1	2	2	3

Pseudo-code

```
String str1;
String str2;
int[][] array = ...  //set up array of appropriate size

for (i = 0 to array.length - 1)
    //set up base cases
for (i = 0 to array[0].length - 1)
    //set up base cases

for (j = 1 to array[0].length - 1)
    for (i = 1 to array.length - 1)
        if (str1.charAt(i - 1) == str2.charAt(j - 1))
            //put appropriate value into array[i][j]
        else
            //put appropriate value into array[i][j]

return ...; // return appropriate value here
```

Input

The input will consist of two Strings A and B of lengths m and n respectively. Where $1 \leq n, m \leq 5000$ the Strings will consist of only valid ASCII characters, note that **A** and **a** are not the same character, uppercase letters should not be treated as equal to their lowercase counterparts.

Output

You should output the length of the Longest Common Subsequence of Strings A and B, which should be the value located at:

```
array[array.length - 1][array[0].length - 1]
```

Example

Input	Output
sharpie grape	3
six Seven	0
The brown dog ate an apple yesterday. Once upon a time, the end.	14

Edit Distance

In this problem you will be given two strings, A and B, you will be finding the edit distance of these two strings. We will define the edit distance of two strings to be the minimum number of operations that must be performed on either string to make them the same.

For this problem there will be 3 operations we can perform:
Insert(i, c): Insert the character c into a string at position i
Remove(i): Remove the character at position i from a string
Change(i, c): Change the character at position i in a string to the character c

This problem will be very similar to the Longest Common Subsequence problem. We will define the subproblem edit(i, j) to be the minimum number of operations required to match the first i characters from A to the first j characters from B. This will be our subproblem, that we use to build up to our final answer. However, in this problem we are minimizing the entries in our array rather than maximizing them as we did in the last problem.

If we are considering the element in the array representing edit(i, j) then there are once again 3 subproblems we need to consider: edit($i-1$, j), edit(i, $j-1$) and edit($i-1$, $j-1$). Now we need to choose the minimum of all possible ways to reach edit(i, j). From edit($i-1$, j) and edit(i, $j-1$) it is easy to see that exactly 1 operation is required to reach edit(i, j). Additionally if we consider the subproblem edit($i-1$, $j-1$) then if the characters at i and j are the same then no operations are required to reach edit(i, j) because the next characters are the same. However if they are different then we must change one of these characters, which requires 1 operation. Considering all of these ways to reach the subproblem edit(i, j) we end up with this relation:

edit(i, j) = min(edit($i-1$, j) + 1, edit(i, $j-1$) + 1, edit($i-1$, $j-1$) + different(i, j))
where different(i, j) is equal to 0 if characters i and j are the same, 1 otherwise.

All that is left to this problem is figuring out the base cases. How many operations are required to change the empty string into another string? Use this to fill out edit(i, 0) for all i's and edit(0,j) for all j's.

Input

The input will consist of two Strings A and B of lengths m and n respectively. Where $1 \leq n, m \leq 5000$ the Strings will consist of only valid ASCII characters, note that **A** and **a** are not the same character, uppercase letters should not be treated as equal to their lowercase counterparts.

Output

You shuld output the Edit Distance of Strings A and B, which should be the value located at:

```
array[array.length - 1][array[0].length - 1]
```

Example

Input	Output
ababb bbab	2
sally sells seashells by the seashore	13

Shopping Spree

Congratulations! You've just won a shopping spree at your favorite store. Since it is your favorite store you already know all the prices of everything and their weights. You are given a single bag that can hold items with a total weight of up to W, the goal of this shopping spree is to maximize the value of the items that you can fit into your bag.

The size of the items do not matter, meaning that you can fit as many items into your bag as long as their total weights do not exceed W

Because this store is very popular they must always be well stocked. You may assume that you can take an infinite amount of any item. The store will not run out of stock.

Each item will have an integral weight w_j and an integral value v_j, additionally the maximum weight the bag can hold will be an integer W.

For this problem we will keep track of the max total value of items we can fit into smaller bags. shop(i) will be the maximum value of the items we can fit into a bag with max weight i. We will initialize an array containing W elements, one for each bag with a weight $\leq W$. We will then proceed to filling out this array. To determine each element, shop(i), we will iterate through each item to decide if it will improve our total value. To check if item j will improve our total value, consider shop($i - w_j$) + v_j, if it is greater than the current value, shop(i), then save it as the answer to the subproblem shop(i), otherwise continue to the next item.

Input

The input will consist of two arrays, `weight` and `value`. Each array will be of length n, representing the number of items. Item i will have a weight, $1 \leq$ `weight[i]` ≤ 5000, and a value, $1 \leq$ `value[i]` ≤ 5000.

Input will also consist of a maximum weight m, representing the maximum weight of the items your bag can hold.

$1 \leq n, m \leq 5000$

Output

You will return the maximum total value of the items you are able to fit into your bag.

Example

input	output

`weight = {3, 100, 7}`
`value = {5, 200, 9}`
`max = 197`
	`360`

`weight = {2, 6, 3, 3, 9, 7, 3}`
`value = {1, 3, 2, 8, 4, 2, 2}`
`max = 77`
	`201`

Chain Matrix Multiplication

Consider a list of matrices A_1, A_2, A_3, A_4, A_5. In this problem we will be determining the most efficient way to multiply these matrices together. For example if we have 3 matrices, A_1, A_2, A_3, which have the following dimensions: $(10 \times 100), (100 \times 5), (5 \times 50)$. There are several different ways to parenthesize these matrices, when multiplying matrices with dimensions $i \times j$, and $j \times k$ notice that it will take $i \cdot j \cdot k$ single multiplication operations. So for the parenthesization $((A_1A_2)A_3)$ it will take $(10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) = 7500$ operations. Lets consider another parenthesization: $(A_1(A_2A_3))$ this will take $(100 \cdot 5 \cdot 50) + (10 \cdot 100 \cdot 50) = 75000$ operations. As you can see the first parenthesization requires one tenth the number of operations the later takes, a rather significant difference.

In this problem you will be given an array D of length $n + 1$, which will represent the dimensions of a matrix chain containing n matrices. Where matrix A_i has D_{i-1} rows and D_i columns. Our goal will be to find the most efficient way to multiply matrices $A_1, A_2, \ldots, A_{n-1}, A_n$ together. Remember that matrix multiplication is associative, meaning that as long as you maintain the original ordering any parenthesization will yield the same final matrix. We will define our subproblem $m(i, j)$ as the optimal number of operations required to multiply together the matrices $A_i, A_{i+1}, \ldots, A_j$. To find the answer to $m(i, j)$ we will use the fact that at some point k we will be dividing $m(i, j)$ into two matrices to multiply together, $m(i, k) \cdot m(k + 1, j)$, to obtain $m(i, j)$. Notice that in this problem we are not considering a fixed number of subproblems, as in the previous problems.

To build up our subproblems we will need to start by finding the optimal parenthesizations containing $1, 2, 3, 4, \ldots, n$ matrices. This will be our outermost loop. Within each loop we will need to iterate through valid values of k that can divide the subproblem $m(i, j)$ and take the minimum.

$m(i, j) = \min(m(i, k) + m(k + 1, j) + (D_i \cdot D_k \cdot D_j))$ for all values of k such that $i \leq k < j$

The sum of the work required to obtain the two subproblems, plus the work required to combine them

Don't forget the basecase $m(i, i) = 0$

Input

An array D will be provided containing $n + 1$ integers. Matrix A_i will have D_{i-1} rows and D_i columns. You will compute the minimum number of operations required to compute $A_1 \times A_2 \times \cdots \times A_n$ by some parenthesization.

$1 \leq n, D_i \leq 500$

Output

Simply return the minimum number of operations required to compute, $A_1 \times A_2 \times \cdots \times A_n$ by some parenthesization, where multiplying matrices of dimensions $i \times j$ and $j \times k$ together takes $(i \cdot j \cdot k)$ operations.

Example

input	output
{5, 100, 70, 3}	22500
{1, 2, 3, 4, 500, 6, 7}	5060
{37, 22, 107, 8, 99, 17, 20}	47448

Extra Credit

For most of these problems, after you define the subproblems and recurrence it is not difficult to code the solution. The point of these extra credit problems is for you to think about and solve these problems, whereas the previous problems were meant to get you familiar with dynamic programming and how it is used to efficiently solve problems. Feel free to ask us for help, but do not expect the TAs to solve these problems/subproblems for you.

Robbers — Easy (5 Points)

You are a thief, currently in the process of a robbery. Unfortunately, you can't decide what items to take to maximize your profits. You do however happen to have your laptop with you and a detailed list containing every item's weight and value. So you decide to quickly code up a program to calculate the maximum total value of the items you could leave with if you can only carry a total weight of W.

In this problem when you decide to steal an item on the list it is gone, you may not keep choosing the same item. Once again the size of the item doesn't matter, you may keep choosing available items as long as the sum of their weights do not exceed your total weight W.

Input

The input will consist of two arrays, `weight` and `value`. Each array will be of length n, representing the number of items. Item i will have a weight, $1 \leq$ `weight[i]` ≤ 5000, and a value, $1 \leq$ `value[i]` ≤ 5000.

Input will also consist of a maximum weight m, representing the maximum weight of the items you can carry.

$1 \leq n, m \leq 5000$

Output

You will return the maximum total value of the items you are able to steal.

Example

input	output

`weight = {3, 100, 7}`
`value = {5, 200, 9}`
`max = 99`
	`14`

`weight = {2, 6, 3, 3, 9, 7, 3}`
`value = {1, 3, 2, 8, 4, 2, 2}`
`max = 20`
	`17`

Circus Tent — Medium (10 Points)

There is a circus coming to town! They will be building their square tent in a local forest. They would like to be able to bring their largest tent, so they need to know the maximum unobstructed area in this forest. This is where you come in, a description of the forest will be provided and your goal will be to find a square with the largest area containing no trees, boulders, or rivers. The area surrounding the tent has no impact on the circus, they don't mind being completely surrounded by boulders for example.

Input

You will be provided an array of Strings, where each element in this array describes a horizontal section of the forest. Within each string there are exactly 4 valid characters: (no spaces!)

. represents a section of grass (a clear area)
b represents a boulder
t represents a tree
r represents a river

Your goal is to find the maximum *square* containing *only* grass.

The input array will contain n Strings, all of these Strings will have the exact same length m where $1 \leq n, m \leq 3000$

Output

You will simply return a single integer representing the maximum area of the tent that the circus can bring.

Example

input	output

`rrrrrr`
`...bbt`
`......`
`......`
`rr...t`
	`9`

`.rr..t.t.b`
`.r......t.`
`.r.....btt`
`br.....t..`
`rr....t...`
`bb..b..ttr`
`........rr`
	`16`

Real Estate — Very Hard (20 Points)

Note: pay close attention to the time restrictions on this problem, that is what makes this problem difficult. You will receive points for this problem based on the efficiency of your algorithm, so even if you are not sure how to code this to run in under a few seconds give it a try anyway!

In an alternate universe called flat-land you are a real estate agent. You've noticed that there are ways to take advantage of how sections of land are sold in flat-land. Mainly the fact that after each sale only the bottom right corner and the perimeter of the land sold is recorded. This allows you to sell overlapping areas of land multiple time to result in some maximum profit.

When selling land each sale will be some rectangular section of flat-land. The price of each sale is based solely on the perimeter of the land sold (Not Area!). Your goal is to find the maximum number of sales you can make with their maximum perimeters. You are only able to sell sections of land that you own in it's entirety. If there is a single square within the land you are trying to sell not owned by you, you may not sell that rectangular area.

Flat-land will be described as an array of strings containing the numbers 0 through 9 (inclusive) you own all cells represented by a 0, cells represented by the digits 1 through 9 are owned by other real estate agents, and any areas containing these cells may not be sold by you.

Input

You will be given an array containing n Strings all of length m. The only valid characters within these Strings are 0 through 9 (inclusive). A 0 within the i_{th} String at position j means that you own the square cell in flatland at coordinates, (i, j). If this character is a digit other than 0 that means that some other real estate agent owns that cell.

$1 \leq n, m \leq 3000$

Output

You will return an array of integers, `Perimeters`. Where `Perimeters[`k`]` represents the number of sales made with perimeter k. The length of this array does not matter, as long as you are sure that the largest perimeter sold is accounted for within this array.

Example

input	output
000120 100000 100001 270700	{8, 0, 0, 0, 2, 0, 3, 0, 5, 0, 4, 0, 2}
0001010223400 1200000100002 1332151337000 1234000000001 0000000000000 0000000100000	{26, 0, 0, 0, 7, 0, 9, 0, 9, 0, 6, 0, 5, 0, 5, 0, 3, 0, 3, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1}

Deliverables

You are required to submit the following files. Make sure that they are all Javadocd appropriately.

- Dynamic.java

By submitting this assignment, you confirm that you understand the CS1332 collaboration policy, and that your submission complies with it.