CSE 101- Winter ‘18
Discussion Section
Week 6

Administrative

- Introducing 1:1 Sessions: https://docs.google.com/spreadsheets/d/1kgXT_RZbzLIbBDiJicZs_o1wXDWA9HHvXCcPrN8_bWk/edit?usp=sharing

  Please see the rules before signing up!

Topics for Today

- Dynamic programming
    - Motivation
    - How to approach the problem
    - Fibonacci (review)
    - How to analyze Runtime
    - ***Minimum Edit Distance***
    - ***Optimal Parenthesization***
    - Practice problems

Fibonacci

```
f(n):
    if n==1 or n == 2: return 1
    return f(n-1) + f(n-2)
```

Dynamic Programming

Design Strategy

1. Solve an “easy” sub-problem
2. Store the solution
3. Use stored solution to solve a more difficult sub-problem.
4. Repeat until you solve the “big” hard problem

Motivating Example: Edit Distance

- When you misspell a word in a text editor, the spell checker has to figure out what words are similar to the word you just typed
- To do so, you need some metric of how “close” any two words are, so that you can find the “closest” word
- The metric used for “closeness” is edit distance
- Actual spell-check is probably a lot more complicated than this, but let’s ignore that for this lesson

Edit Distance

The problem of finding an edit distance between two strings is as follows:

> Given an **initial string s**, and a **target string t**, what is the minimum number of changes that have to be applied to ***s*** to turn it into ***t*** **?**

The list of valid changes are:

1) Inserting a character
2) Deleting a character
3) Replace a character by another character.

Edit Distance

- The cost incurred by a pair of strings
- Think about it as the number of edits (insertions, deletions, character changes) that need to be done to get the strings to be equivalent
- Goal: Find the smallest possible edit distance achievable by inserting spaces into the strings

Examples

- What is the minimum edit distance obtainable for the following pairs of strings?
  - “carpet” and “carton”
  - “fluorine” and “fine”
  - “aaaaa” and “bbbbbb”

Examples

- What is the minimum edit distance obtainable for the following pairs of strings?
  - “carpet” and “carton” - 3
  - “fluorine” and “fine” - 4
  - “aaaaa” and “bbbbbb” - 6

How to Solve This Problem

- Theoretically, there are many combinations of ways that you could add spaces to the words
  - Trying them all is impossible
  - Answer: dynamic programming

Dynamic Programming

- Solving full problems takes a lot of time
- Idea: break problem up into subproblems
- Saving the results of those subproblems, you can save a lot of work

How to Approach the Problem

- Imagine a solution
- Write the solution such that it consists smaller solutions as its parts.
- Extract the subproblems and the decisions to be made by looking at the solutions.

How to Approach the Problem (Things that help)

- Think: What decisions do I have to make to come up with a solution to this problem?
- Does the problem have any natural ordering which I could use to differentiate subproblems?
- How can I efficiently find the **answer to one subproblem based on the answers to previous subproblems**?

How to Approach This Problem (Cont.)

Consider the last characters of strings s and t.

If we are lucky enough that they ALREADY match,

then we can simply "cancel" and recursively find the edit distance between the two strings left when we delete this last character from both strings.

Otherwise, we MUST make one of three changes:

1) Delete the last character of string s
2) Insert a character at the end of string s
3) Replace the last character of string s by the last character of string t.

How to Approach This Problem (Cont.)

Now, how do we use this to create a DP solution?

We simply need to store the answers to all the possible recursive calls.

Sub-Problems

Let s be of length n and t be of length m
What can be the sub-problems?

1. Delete the last character of string s
   editDistance(s[1…n-1],t[1…m])

2. Insert a character at the end of string s (Now, s[n+1]=t[m])
   editDistance(s[1…n],t[1…m-1])

3. Replace the last character of string s by the last character of string t. (or they are already equal)
   editDistance(s[1…n-1],t[1…m-1])

Edit Distance

Consider the following example with s="hello" and t="keep".

In order to fill in all the values in this table we will do the following:

1) Initialize values corresponding to the base case in the recursive solution.

		h	e	l	l	o
	0	1	2	3	4	5
k	1					
e	2					
e	3					
p	4					

Edit Distance

In order to fill in all the values in this table we will do the following:

2) Loop through the table from the top left to the bottom right. In doing so, simply follow the recursive solution.

If the characters you are looking at match,

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2				
e	3					
p	4					

Just bring down the up&left value.

Else the characters don't match, min (1+ above, 1+ left, 1+diag up)

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1			
e	3					
p	4					

Final Table

An entry in this table simply holds the edit distance between two prefixes of the two strings.

For example, the highlighted square indicates that the edit distance between the strings "he" and "keep" is 3.

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
e	3	3	2	2	3	4
p	4	4	3	3	3	4

Pseudocode

```
int editDistance(char s[1..n], char t[1..m])
        int d[0..n, 0..m]

        for i from 0 to n d[i, 0] := i
        for j from 1 to m d[0, j] := j

        for i from 1 to n
                for j from 1 to m
                        if s[i] = t[j] then cost := 0 else cost := 1

                        d[i, j] := minimum(
                                        d[i-1, j] + 1, // deletion
                                        d[i, j-1] + 1, // insertion
                                        d[i-1, j-1] + cost // Replacement )
        return d[n, m]
```

Runtime

- The matrix size is |s|x|t|
- Each cell takes O(1) time to fill
- Complexity- **O(|s|x|t|)**

Optimal Parenthesization

Suppose you have a mathematical equation of the form number operand number operand number … operand number

eg: 1 + 2 * 4

We have to perform the operations in an order that maximizes the value of the expression.

eg: ((1+2)*4) > (1+(2*4)) 12 > 9

Problem Specification

Given: arithmetic expression containing n real numbers and n -1 operators, each either + or *

Goal: Insert n - 1 pairs of parentheses into the expression so that its value is maximized. Return the maximum value.

Why n-1?

Sub-Problems

Let,

Operands: $a_1, a_2, \ldots., a_n$
Operators: $op_1, op_2, \ldots., op_{n-1}$

E(i,j) be the subexpression $a_i \; o_i \; \ldots.. \; o_{j-1} \; a_j$ of the input expression

- M(i,j)=maximum value obtainable from E(i,j) (by inserting parentheses)
- m(i,j)=minimum value obtainable from E(i,j) (by inserting parentheses)

We want M(1,n)

Building Recurrence

We must keep track of both the minimum and the maximum because the maximal value of an expression may result from multiplying two negative subexpressions.

Let $Eval_{\max}[i,j,k]$ denote the maximum value of the expression obtained after applying kth operator on *E(i,k)* and *E(k+1,j)*.

Similarly, $Eval_{\min}[i,j,k]$ denotes the minimum value of the expression obtained after applying kth operator on *E(i,k)* and *E(k+1,j)*.

Building Recurrence

$$
\begin{aligned}
Eval_{\max}[i,j,k] = \max\{ & M[i,k]\ op_k\ M[k+1,j], \\
& M[i,k]\ op_k\ m[k+1,j], \\
& m[i,k]\ op_k\ M[k+1,j], \\
& m[i,k]\ op_k\ m[k+1,j]\}.
\end{aligned}
$$

We have,

$$
M[i,j] = \begin{cases} a_i & \text{if } i = j \\ \max_{i \le k \le j-1} Eval_{\max}[i,j,k] & \text{otherwise} \end{cases}
$$

Building Recurrence

$$\begin{aligned} Eval_{\min}[i,j,k] = \min\{ & M[i,k]\ op_k\ M[k+1,j], \\ & M[i,k]\ op_k\ m[k+1,j], \\ & m[i,k]\ op_k\ M[k+1,j], \\ & m[i,k]\ op_k\ m[k+1,j])\}. \end{aligned}$$

We have,

$$m[i,j] = \begin{cases} a_i & \text{if } i = j \\ \min_{i \le k \le j-1} Eval_{\min}[i,j,k] & \text{otherwise} \end{cases}$$

Example

Given: $-3 * 2 + 4$

$a_1=-3$, $a_2=2$, $a_3=4$ $Op_1=*$, $op_2=+$

Initialization
$M[1,1]=m[1,1]=a_1=-3$
$M[2,2]=m[2,2]=a_2=2$
$M[3,3]=m[3,3]=a_3=4$

Example

Given: -3 * 2 + 4

Recall M[i,j] is the maximum value obtainable from E(i,j)

M[1,2]=Eval$_{max}$ [1,2,1]= max{M[1,1]*M[2,2],
M[1,1]*m[2,2],
m[1,1]*M[2,2],
m[1,1]*m[2,2],}
= - 3 * 2=- 6

M[2,3]= 2+4=6

Example

Given: -3 * 2 + 4

Recall m[i,j] is the minimum value obtainable from E(i,j)

Similarly,

$m[1,2]=Eval_{min}[1,2,1]= \min\{M[1,1]*M[2,2],$
$M[1,1]*m[2,2],$
$m[1,1]*M[2,2],$
$m[1,1]*m[2,2],\}$
$= - 3 * 2=- 6$

m[2,3]= 2+4=6

Example

Given: -3 * 2 + 4

M[1,3] =max{ Eval_{max} [1,3,1], Eval_{max} [1,3,2] }

=max{ (- 3 * (2 + 4))),((-3 * 2) + 4) }

=max{ - 18 , - 2 }

= - 2

We don’t need to calculate m[1,3].. ☺

Pseudocode

```
1.  Given $n$
2.  for $i \leftarrow 1$ to $n$        // initialization: $O(n)$ time
3.      do $M[i, i] \leftarrow a_i$, $m[i,i] \leftarrow a_i$
4.  for $L \leftarrow 2$ to $n$              // $L$ = length of sub-expression
5.      do for $i \leftarrow 1$ to $n - L+1$
6.              do $j \leftarrow i + L - 1$
7.          $M[i, j] \leftarrow -\infty$, $m[i, j] \leftarrow \infty$
8.                  for $k \leftarrow i$ to $j - 1$
9.                      do $qM \leftarrow \text{Eval}_{\max}[i,j,k]$, $qm \leftarrow \text{Eval}_{\min}[i,j,k]$
10.                         if $qM > M[i, j]$
11.                         then $M[i, j] \leftarrow qM$
12.             if $qm < m[i, j]$
13.                         then $m[i, j] \leftarrow qm$
14. return $M[1,n]$
```

Runtime

- M[1,n] is the required value.
- O(n^2) sub-problems as there are O(n^2) pairs of (i,j).
- Each sub-problem takes O(n) time.

- Runtime: **O(n^3).**

- We can also see that there are 3 nested loops and each can iterate at most n times, so the total running time is **O(n^3).**

Practice Problems (Coin Change)

- Given coin denominations, devise a method to pay an amount to a customer using the fewest possible number of coins.
- Input
  - An array **denominations**[1..n] containing the **n** coin denominations $d_1,\ldots,d_n$ in increasing order
  - The amount **M** that you need to pay
- Output
  - The optimal (minimum) number of coins needed to make change for M

Practice Problems (Coin Change)

Eg:
Input : denominations {5, 10, 30}, M=25
Output: 3

Is there any particular case when we can solve this problem using a greedy algorithm?

Practice Problems (Longest Palindromic Subsequence)

A **subsequence** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

Design an algorithm that takes a sequence and returns the length of the longest palindromic subsequence.

- Input
  - A sequence of characters **x[1...n]**
- Output
  - The length of the longest palindromic subsequence

Practice Problems (Longest Palindromic Subsequence)

Eg:
Input : s=“AABCDEBAZ”

Some of the longest palindromic sub-sequences are ABCBA, ABEBA and ABDBA

Output: 5

How do you find one possible longest palindromic sub-sequence..??

Hint: what is the LCS of s and reverse(s)?

Thank You..!!