Analysis of Algorithms

V. Adamchik

Lecture 6

Fall 2023

CSCI 570

University of Southern California

Dynamic Programming

Reading: chapter 6

Exam - I

Date: Friday Oct. 6

Time: starts at 5pm

Locations: multiple room

Practice Exam: posted

TA Review: next week

Closed book and notes.
No internet searching.
No talking to each other (chat, phone, messenger).
One page cheat sheet.

REVIEW QUESTIONS

1. (T/F) For a divide-and-conquer algorithm, it is possible that the dividing step takes asymptotically longer time than the combining step.
2. (T/F) A divide-and-conquer algorithm acting on an input size of n can have a lower bound less than $\Theta(n \log n)$.
3. (T/F) There exist some problems that can be efficiently solved by a divide-and-conquer algorithm but cannot be solved by a greedy algorithm.
4. (T/F) It is possible for a divide-and-conquer algorithm to have an exponential runtime.
5. (T/F) A divide-and-conquer algorithm is always recursive.
6. (T/F) The master theorem can be applied to the following recurrence: $T(n) = 1.2\,T(n/2) + n$.
7. (T/F) The master theorem can be applied to the following recurrence: $T(n) = 9\,T(n/3) - n^2 \log n + n$.
8. (T/F) Karatsuba's algorithm reduces the number of multiplications from four to three.
9. (T/F) The runtime complexity of mergesort can be asymptotically improved by recursively splitting an array into three parts (rather than into two parts).

10. (**T**/**F**) Two $n \times n$ matrices of integers are multiplied in $\Theta(n^2)$ time.

11. (**Fill in the blank**) Let A, B be two 2×2 matrices that are multiplied using the standard multiplication method and Strassen's method.

    a. Number of multiplications in the standard method: 8

    b. Number of additions in the standard method: 4

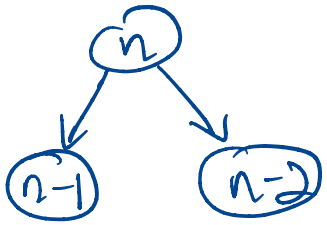
    c. Number of multiplications using Strassen's method: 7

    d. Number of additions using Strassen's method: 18

12. (**Fill in the blank**) The space complexity of Strassen's algorithm is: $\Theta(n^2)$.

Fibonacci Numbers

Fibonacci number F_n is defined as the sum of two previous Fibonacci numbers:



$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = F_1 = 1$$

Design a divide & conquer algorithm to compute Fibonacci numbers. What is its runtime complexity?

Runtime ?

PTO

$T(n)$ → Computing n^{th} Fibonacci Number

$T(n) = T(n-1) + T(n-2) + O(n)$

How many bits in n^{th} fibonacci number.

$$\log(F_n) = \log\theta(\psi^n) = \theta(n \log \psi)$$
$$= \theta(n)$$

ψ = golden ratio

Solve for $T(n)$

$T(n) \leq 2T(n-1) + O(n)$

$\Rightarrow T(n)$ is exponential

$\approx 2^n$

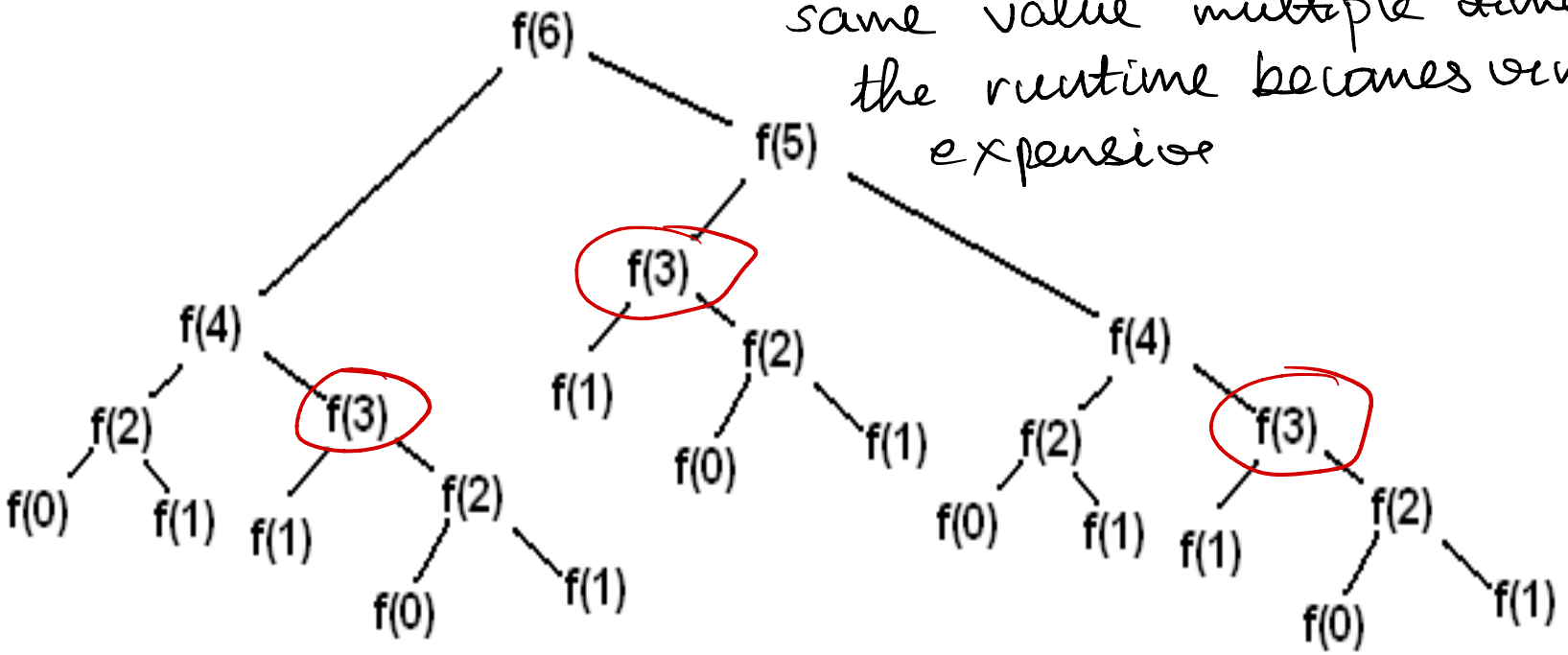
At each iteration we reduce the value of n by 1 & multiply by 2

The text in yellow is the explanation as to how we got $O(n)$ and not $O(\log n)$

Overlapping Subproblems

Problem of using D&C for the fibonacci number

Since we compute the same value multiple times the runtime becomes very expensive



STORING THESE VALUES WILL SAVE COMPUTE - MEMOIZATION

Fibonacci Numbers: $F_n = F_{n-1} + F_{n-2}$

Memoization

```
int table [50];  //initialize to zero
table[0] = table[1] = 1;

int fib(int n)
{
        if (table[n] != 0) return table[n];
        else
        table[n] = fib(n-1) + fib(n-2);

        return table[n];
}
```

main step of memoization

⇒ O(1)

O(1)

Runtime complexity? $T(n) = T(n-1) + T(n-2) + O(n)$

$T(n) = T(n-1) + O(n) = O(n^2)$

Tabulation

To make the previous update more efficient, we use this method to remove recursion.

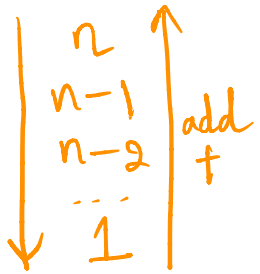
```
int  table [n];

void fib(int n)
{
        table[0] = table[1] = 1;
        for(int k = 2; k < n; k++)
            table[k] = table[k-1] + table[k-2];

        return;
}
```

Two Approaches

```
int table [n];
table[0] = table[1] = 1;
int fib(int n)
{
  if (table[n] != 0)
    return table[n];
  else
    table[n] = fib(n-1) + fib(n-2);
  return table[n];
}
```

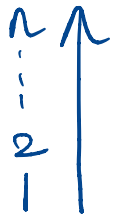


Memoization:
a top-down approach.

```
int table [n];

int[] fib(int n)
{
  table[0] = table[1] = 1;
  for(int k = 2; k < n; k++)
    table[k]=table[k-1]+table[k-2];

  return table;
}
```



Tabulation:
a bottom-up approach.

Dynamic Programming

General approach: in order to solve a larger problem, we solve smaller subproblems and store their values in a table.

DP is applicable when the subproblems are greatly overlap.
DP is not D&C. Compare with Mergesort.

DP is not greedy either. DP tries every choice before solving the problem. It is much more expensive than greedy.

DP is slower than greedy because it solves all the subproblems to conclude with a final solution.

DP can be implemented by means of memoization or tabulation.

Dynamic Programming

Optimal substructure means that the solution can be obtained by the combination of optimal solutions to its subproblems. Such optimal substructures are usually described recursively. ⇒ Unlike greedy we do not need to prove the optimality because it is guaranteed that an optimal solution exists since all subproblems are solved.

Overlapping subproblems means that the space of subproblems must be small, so an algorithm solving the problem should solve the same subproblems over and over again.

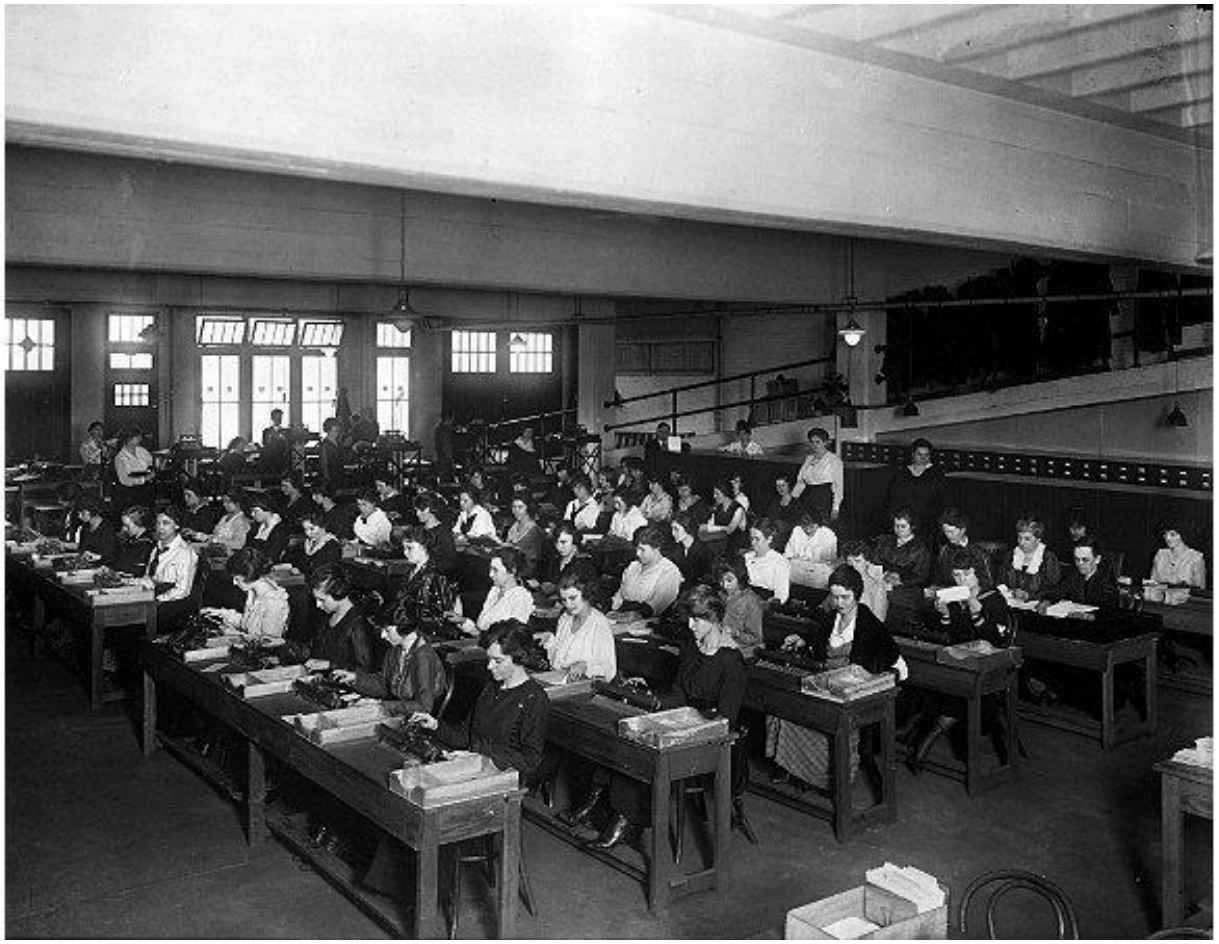
Dynamic Programming

The term dynamic programming was originally used in the 1950s by Richard Bellman.

The term computer (dated from 1613) meant a person performing mathematical calculations.

In the 30-50s those early computers were mostly *women* who used painstaking calculations on paper and later punch cards.

The earliest human computers



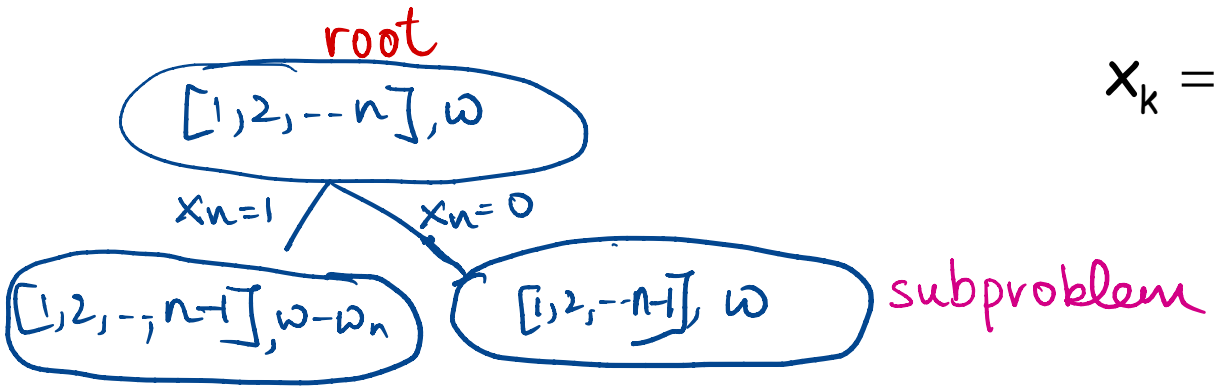
Who put a man to the moon?

0-1 Knapsack Problem

Given a set of n unbreakable *unique* items, each with a weight w_k and a value v_k, determine the subset of items such that the total weight is less or equal to a given knapsack capacity W and the total value is as large as possible.

- Fractional knapsack – greedy approach
- 0-1 knapsack, brute force – $O(2^n)$

Decision Tree



$$x_k = \begin{cases} 1, \text{item k selected} \\ 0, \text{item k not selected} \end{cases}$$

OPT[K, X]

K = # of items available, $0 \le K \le n$

X = is a capacity, $0 \le X \le W$

Subproblems

Let $OPT[k, x]$ be the maximum value achievable using a knapsack of capacity x with k items.

Our choices:

1. $x_k = 1$: $OPT[k, x] = v_k + OPT[k-1, x-w_k]$

2. $x_k = 0$: $OPT[k, x] = OPT[k-1, x]$

Recurrence Formula

table

$OPT[K,X] = MAX[V_K + OPT[K-1, X-\omega_K], OPT[K-1, X]]$

$OPT[K-1, X-\omega_K]$: O(1) ?

$OPT[K-1, X]$: O(1) table lookup

Base cases

$OPT[0,X] = 0$

$OPT[K,0] = 0$

$OPT[K,X] = OPT[K-1,X]$, if $\omega_K > X$

Tracing the Algorithm

n = 4, W = 5

(weight, value) = (2,3), (3,4), (4,5), (5,6)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
1,2	0	0	3	4	4	3+4
1,2,3	0	0	3	4	5	7
1,2,3,4	0	0	3	4	5	7

knapsack capacity

items

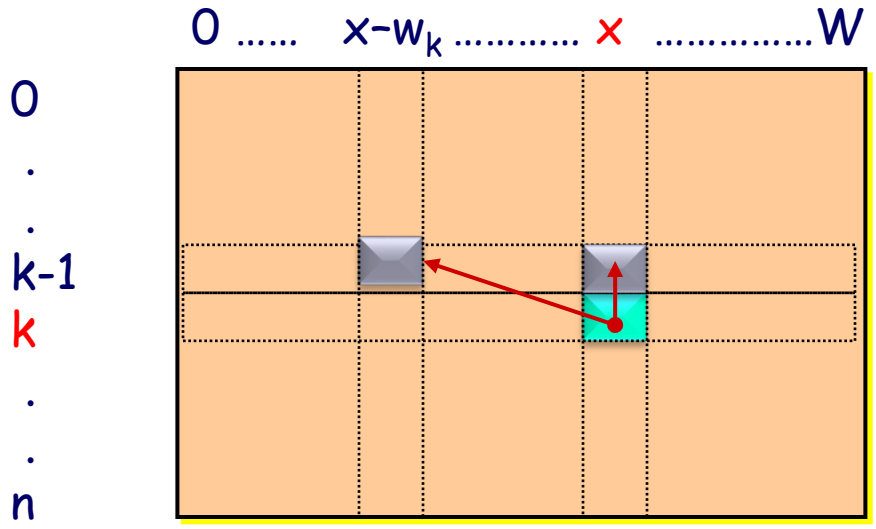
OPT[n, W]

Pseudo-code

Input

```
int knapsack(int W, int w[], int v[], int n) {
   int Opt[n+1][W+1];
   for (k = 0; k <= n; k++) {
      for (x = 0; x <= W; x++) {
         if (k==0 || x==0) Opt[k][x] = 0;
         if (w[x] > x) Opt[k][x] = Opt[k-1][x];
         else
           Opt[k][x] = max( v[k] + Opt[k-1][x - w[k]],
                            Opt[k-1][x] );
      }
   }
   return Opt[n][W];
}
```

Complexity



$O(n \cdot W \cdot 1)$

Runtime Complexity? table size times the work you do at each cell. $O(nW)$

Space Complexity?

$O(nW)$

Pseudo-Polynomial Runtime

exponential

Definition. A numeric algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input but is exponential in the length of the input.

$$V = \max(v_1, v_2, \ldots v_n)$$

0-1 Knapsack is pseudo-polynomial algorithm, $T(n) = \Theta(n \cdot W)$

Runtime complexity must be a function of input size

Input size = $O(\log W + n \cdot \log W + n \cdot \log V + \log N)$

Runtime = $O(n \cdot W)$

exp

Input size = $O(n \cdot \log W)$

Actual Reentime : $O(n \cdot 2^{\text{input size of } W})$

How would you find the actual items?

The table built in the algorithm does not show the optimal items, but only the maximum value. How do we find which items give us that optimal value?

→ If arr[i][j] is got from arr[i-1][j] it is not picked, continue to go up.

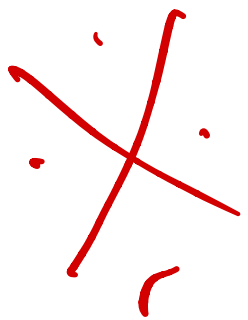
n = 4, W = 5

(weight, value) = (2, 3), (3, 4), (4, 5), (5, 6)

→ Else: move the pointer from current to [i-1][j-w_K]

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

DP Approach



solve using the following four steps:

1. Define (in plain English) subproblems to be solved.
2. Write the recurrence relation for subproblems.
3. Write pseudo-code to compute the optimal value.
4. Compute the runtime of the above DP algorithm in terms of the input size.

Discussion Problem 1

You are to compute the minimum number of coins needed to make change for a given amount m. Assume that we have an unlimited supply of coins. All denominations d_k are sorted in ascending order:

$$1 = d_1 < d_2 < \ldots < d_n$$

Step 1:

Let OPT[k, x] be the minimum number of coins to represent x ($0 \le x \le m$) using first k ($1 \le k \le n$) denominations.

Step 2:

$$OPT[k,X] = \underset{O(1)}{MIN}\left(\overbrace{OPT[k-1,X]}^{O(1)}, \overbrace{OPT[k,X-d_k]+1}^{O(1)}\right)$$

Base cases

$$OPT[1,X] = X$$

$$OPT[k,0] = 0$$

Step 3:

```
Opt[i][0] = 1
for (i = 0 ; i < coins.length ; i++)
    { for (j = 0; j <= amount ; j++
        { if (coins[i] > j) : a[i][j] = a[i-1][j];
```

else

{ a[i][j] = a[i-1][j] + a[i][j-coins[i]]

}

Step 4: Runtime : $O(n \cdot m)$

Is it polynomial? NO

Same steps as knapsack

Longest Common Subsequence

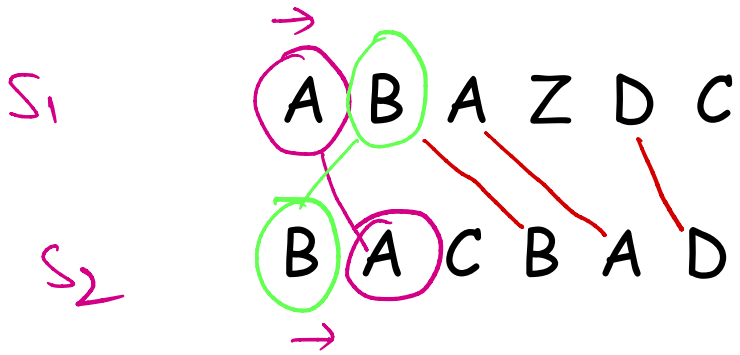
We are given string S_1 of length n, and string S_2 of length m.

Our goal is to produce their longest common subsequence.

A subsequence is a subset of elements in the sequence taken in order (with strictly increasing indexes.) Or you may think as removing some characters from one string to get another.

Note, a subsequence is not a substring.

Intuition



① prefix $S_1[0, 1, \ldots, i]$ → A, AB, ABA, ...

② suffix $S_1[i, \ldots, n]$ → C, DC, ZDC, ...

③ $S_1[i, \ldots, j]$

Subproblems

Let $LCS[i,j]$ be the max length of the LCS $S_1[0,\ldots i]$ & $S_2[0,\ldots j]$ (prefix technique is used here)

Choices ⟹

① $S_1[i] = S_2[j]$

$$LCS[i,j] = 1 + LCS[i-1, j-1]$$

② $S_1[i] \neq S_2[j]$

$$LCS[i,j] = \max[LCS[i-1,j], LCS[i,j-1]]$$

combine the above cases

Recurrence

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1], & \text{if } S_1[i] = S_2[j] \\ MAX\begin{bmatrix} LCS[i-1, j], \\ LCS[i, j-1] \end{bmatrix}, & \text{if } S_1[i] \neq S_2[j] \end{cases}$$

Base cases (empty string)

$$LCS[0, j] = LCS[i, 0] = 0$$

Runtime : $O(n \cdot m)$

is it polynomial ? YES!!

Example

S = ABAZDC
T = BACBAD

		B	A	C	B	A	D
	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1
B	0	1	1	1	2	2	2
A	0	1	2	2	2	3	3
Z	0	1	2	2	2	3	3
D	0	1	2	2	2	3	4
C	0	1	2	3	3	3	4

Pseudo-code

```
int LCS(char[] S1, int n, char[] S2, int m)
{
int table[n+1, m+1];
table[0…n, 0] = table[0, 0…m] = 0; //init
for(i = 1; i <= n; i++)
  for(j = 1; j <= m; j++)
      if (S1[i] == S2[j])  table[i, j] = 1 + table[i-1, j-1]
      else
      table[i, j] = max(table[i, j-1], table[i-1, j]);

return table[n, m];
}
```

How much space do we need?

		B	A	C	B	A	D
	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1
B	0	1	1	1	2	2	2
A	0	1	2	2	2	3	3
Z	0	1	2	2	2	3	3
D	0	1	2	2	2	3	4
C	0	1	2	3	3	3	4

} 2 Rows

How do we find the common sequence?

	0	0	0	0	0	0	0
	0	0	1	1	1	1	1
	0	1	1	1	2	2	2
	0	1	2	2	2	3	3
	0	1	2	2	2	3	3
	0	1	2	2	2	3	4
	0	1	2	3	3	3	4

Discussion Problem 2

A subsequence is palindromic if it reads the same left and right. Devise a DP algorithm that takes a string and returns the length of the longest palindromic subsequence (not necessarily contiguous) .

For example, the string

QRAECCETCAURP

has several palindromic subsequences, RACECAR is one of them.

Given a string of size n

① How many substrings? $O(n^2)$

② How many subsequences $O(2^n)$

Step1: (Since this problem is asking for the palindrome we opt for choice ③ from above)

Let OPT[i,j] be the longest palindrome.

$[s_0, s_1, \ldots [s_i, \ldots, s_j] \ldots s_n]$

Step 2:

Case 1: $s[i] == s[j]$

$OPT[i,j] = OPT[i+1, j-1] + 2$ ← because 2 common characters

Case 2: $s[i] \neq s[j]$

$OPT[i,j] = MAX[OPT[i+1, j], OPT[i, j-1]]$

Base cases

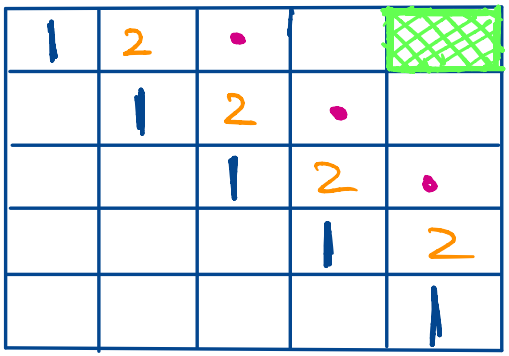
$OPT[i, i] = 1$

$OPT[i, j] = 2$, if $s[i] = s[j]$ and $j = i+1$

$\Rightarrow$ example
"AA"
$i=0, j=1$
$i=1, j=0$

Runtime : $O(n^2)$

Is it polynomial? YES



Can we use LCS to solve this problem?

Ans: Yes we can, the input will be as follows:

$LCS[s_1, reverse(s_1)]$