

# Analysis of Algorithms

V. Adamchik

CSCI 570

Lecture 7

University of Southern California

## Dynamic Programming

Reading: chapter 6

# Exam - I

**Date:** Thursday March 11

**Time:** starts at 5pm Pacific time

**Locations:** online, DEN Quiz

**Practice:** posted

**TA Review:** March 9 and March 10

Open book and notes

Use scratch paper

No internet searching

No talking to each other (chat, phone, messenger)

# 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  $\Theta(n \log n)$ . *binary search*
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. *today's lecture*
5. (T/F) A divide-and-conquer algorithm is always recursive.
6. (T/F) The master theorem can be applied to the following recurrence:  
 $T(n) = \underline{1.2} T(n/2) + n$ .
7. (T/F) The master theorem can be applied to the following recurrence:  
 $T(n) = 9 T(n/3) \left( - n^2 \log n + n. \right) \xrightarrow{n \rightarrow \infty} \text{negative}$
8. (T/F) Karatsuba's algorithm reduces the number of multiplications from four to three.  *$n \rightarrow \infty$*
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  $2 \times 2$  matrices that are multiplied using the standard multiplication method and Strassen's method.
- Number of multiplications in the standard method: 8
  - Number of additions in the standard method: 4
  - Number of multiplications using Strassen's method: 7
  - Number of additions using Strassen's method: 18
12. (Fill in the blank) The space complexity of Strassen's algorithm is:  $O(n^2)$ .

$$\begin{matrix}
 & A & & B & & C \\
 & \begin{pmatrix} a_{11} & \vdots & \dots \\ \vdots & \ddots & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} & & \begin{pmatrix} b_{11} & \vdots & \dots \\ \vdots & \ddots & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} & & \begin{pmatrix} s_{11}, s_{12} & \vdots & \dots \\ \vdots & \ddots & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}
 \end{matrix}$$

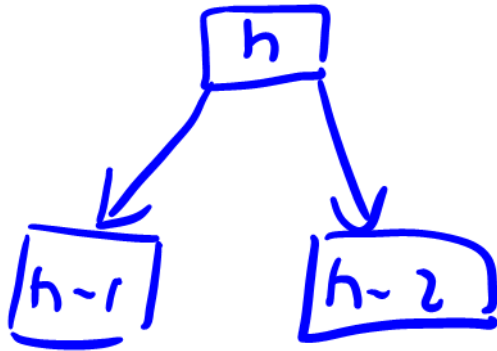
$\rightarrow s_1 = (\dots) - (\dots)$   
 $s_2 = \dots$   
 $\vdots$   
 $s_7 = \dots$

extra space 7+2 matrices

$$S(n) = S\left(\frac{n}{2}\right) + 9 \cdot \left(\frac{n}{2}\right)^2$$

# 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?

$T(n)$  - computing  $n$ -th Fib. number

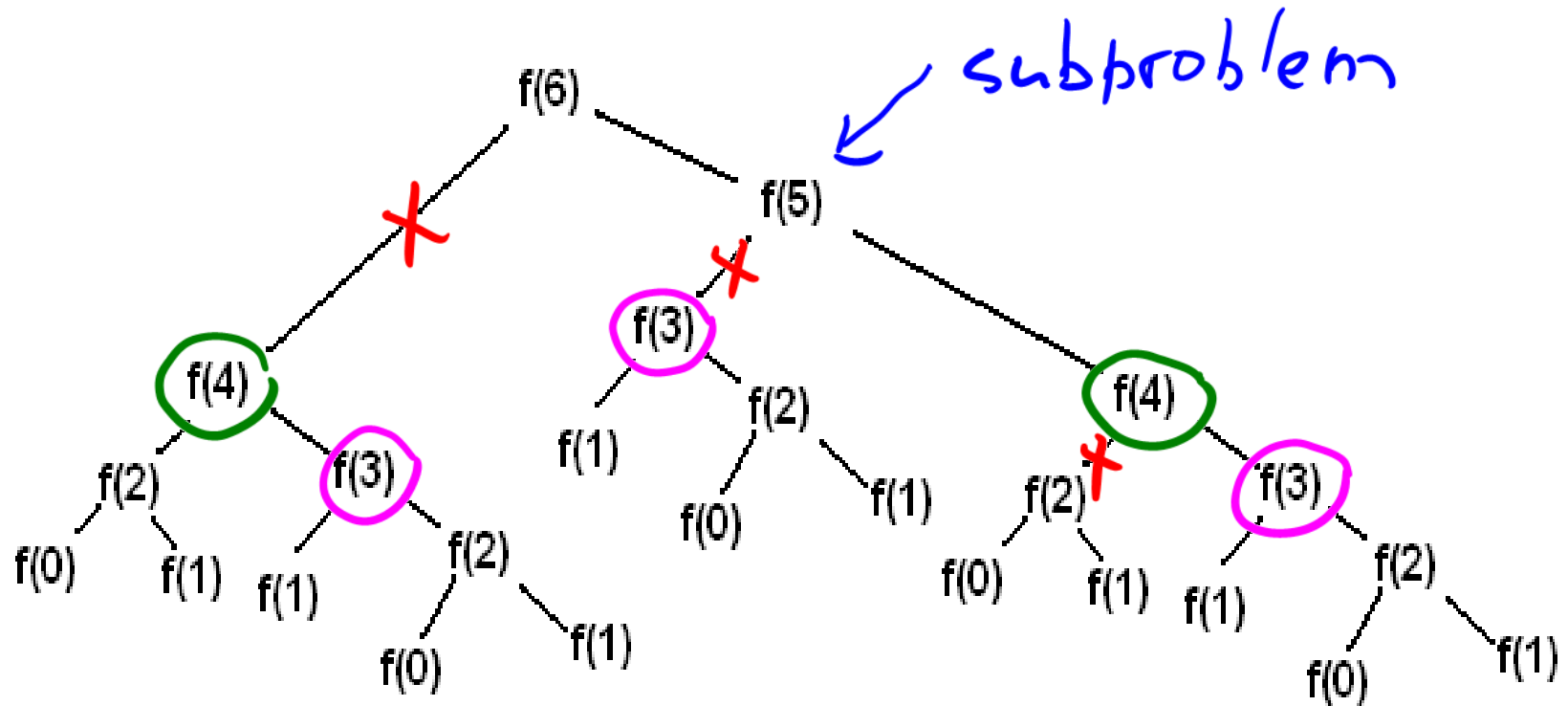
$$T(n) \approx T(n-1) + T(n-2) + \underline{O(n)} \text{ why?}$$

How many bits in  $n$ -th  $F_n$ ?

$$\log F_n = \log \Theta(\varphi^n) = \Theta(n \log \varphi)$$

$$\varphi - \text{golden ratio} \approx \Theta(n)$$

# Overlapping Subproblems

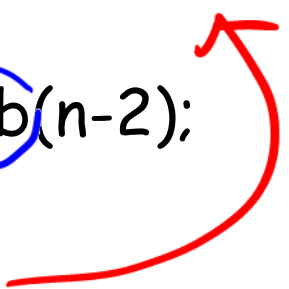


Fibonacci Numbers:  $F_n = F_{n-1} + F_{n-2}$

# 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];  
}
```



$$T(n) = T(n-1) + \cancel{T(n-2)} + O(n)$$

$\Theta(1)$

Runtime complexity?

$$\rightarrow T(n) \approx T(n-1) + \Theta(n), \quad T(n) \approx \Theta(n^2)$$



# Tabulation

```
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];
```

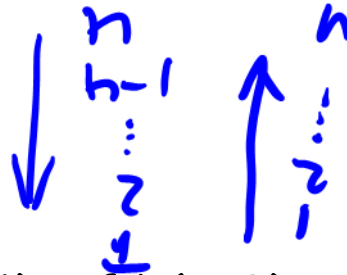
```
    return;
```

```
}
```

# 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.

cs570 way

# 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. Compare with Mergesort.

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

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.

*must*

*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.

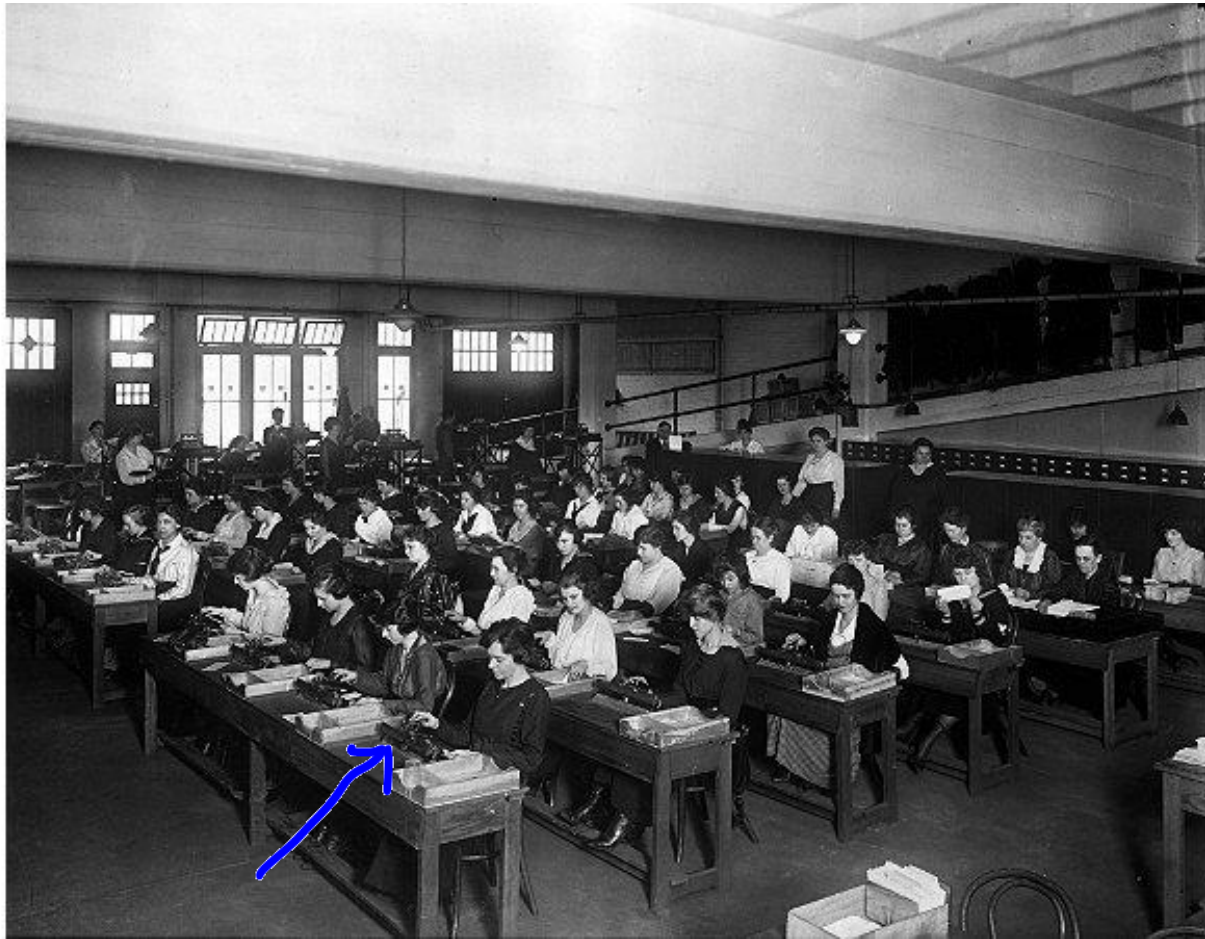
# Dynamic [Programming]

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 – by greedy algo  
Brute-force, runtime  $O(2^n)$



root

# Decision Tree



$x_h = 1$

$x_h = 0$



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

each vertex  
is a subproblem

SOLUTION:  
 $\text{OPT}[h, W]$

$\text{OPT}[k, x]$

$k$  is # of items,  
 $x$  is a capacity,

$$0 \leq k \leq h$$
$$0 \leq x \leq W$$



## Subproblems

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

Our choices:

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

# Recurrence Formula

table

$$OPT[k, x] = \underline{MAX} \left( \underbrace{OPT[k-1, x]}_{\substack{O(1) \\ \text{look up}}}, \underbrace{V_k + OPT[k-1, x - w_k]}_{\substack{< 0 \\ O(1)}} \right)$$

Base cases:

$$OPT[k, x] = 0, \text{ if } k = 0 \text{ or } x = 0$$

$$OPT[k, x] = OPT[k-1, x], \text{ if } w_k > x$$

# Tracing the Algorithm

$opt[k, x]$

$n = 4, W = 5$

$k=1$

(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	?7
1,2,3	0	0	3	4	?5	7
1,2,3,4	0	0	3	4	5	7

x  
← knapsack

x  
↑  
items

$opt[n, W]$

$$w_k < W$$

## Pseudo-code

INPUT

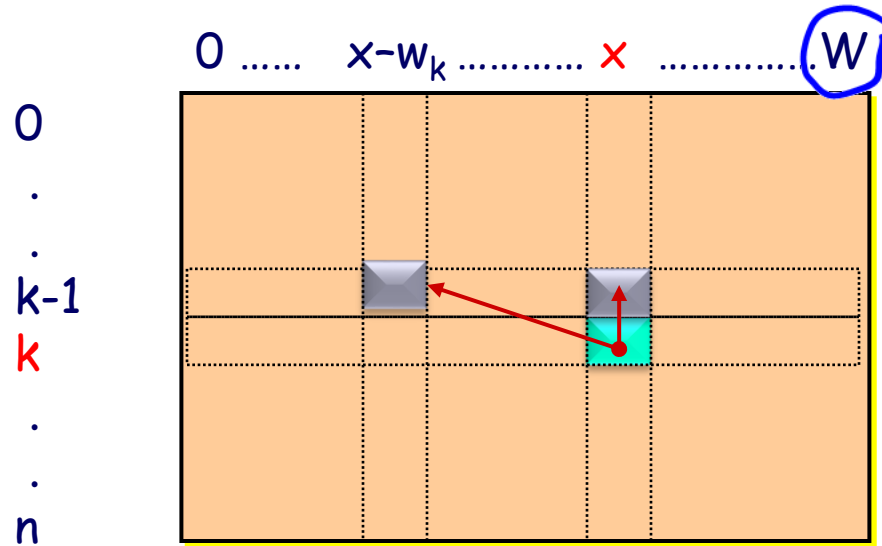
```
int knapsack(int W, int w[], int v[], int n) {  
    int Opt[n+1][W+1];  
    for (k = 0; k <= n; k++) {  
        for (x = 0; x <= W; x++) {  
            if (k==0 || x==0) Opt[k][x] = 0;  
            if (w[x] > x) Opt[k][x] = Opt[k-1][x];  
            else  
                Opt[k][x] = max( v[k]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

*solution*

# Complexity

$DP[k, x]$

binary



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

Runtime Complexity? *table size times the work per each cell*

Space Complexity?

$O(n \cdot W)$

# Pseudo-Polynomial Runtime

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. ??  $V = \max(v_1, v_2, \dots, v_n)$

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

Input size:  $O(\log W + n \cdot \log W + \cancel{n \cdot \log V} + \log n)$

Runtime:  $O(n \cdot W)$  exponential

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

Actual Runtime  $O(n \cdot 2^{\text{input size of } W})$   
in the number of bits

## Discussion Problem 2

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?

$n = 4, W = 5$   
(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
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

max value

# 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.  
*tabulation*
4. Compute the runtime of the above DP algorithm in terms of the input size.



# Discussion Problem 3

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:

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

Step 1.

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

Step 2. Recurrence for  $OPT[k, x]$

$$OPT[k, x] = \min \left( OPT[k-1, x], 1 + OPT[k, x - d_k] \right)$$

Base cases:

$$OPT[1, x] = x$$

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

Step 3. Pseudocode  $\rightarrow$  DIY

Step 4. Runtime - ?

$O(n \cdot m)$   $\rightarrow$  is it polynomial?  
NO

DNA

## 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.

Unix, diff

# Intuition



TOIA SPACE

$$O(n^4)$$

$$A B (A Z D) C \leftarrow s_1(i_1, \dots, j_1)$$

$$B A C (B A D) \leftarrow s_2(i_2, \dots, j_2)$$

no two lines cross



(1)  $s_1[0 \dots i]$  prefix

(2)  $s_1[i \dots n]$

~~(3)  $s_1[i \dots j]$~~

A, AB, APA, ...

C, DC, ZDC, ...

## Subproblems

Let  $LCS[i, j]$  be the max length of the LCS of  $s_1[0, \dots, i]$  and  $s_2[0, \dots, j]$   
 $0 \leq i \leq n$   $0 \leq j \leq m$

Choices:

- 1)  $s_1[i] = s_2[j]$  (last characters)  
 $LCS(i, j) = 1 + LCS(\underline{i-1}, \underline{j-1})$
- 2)  $s_1[i] \neq s_2[j]$   
 $LCS(i, j) = \text{MAX}(\underset{\substack{\text{look up} \\ O(1)}}{LCS(i-1, j)}, \underset{O(1)}{LCS(i, j-1)})$

Recurrence

combine those 2 cases:  $\mathcal{O}(1)$

Base cases:

$$\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$$

Runtime?  $\mathcal{O}(n \cdot m \cdot 1)$

is it polynomial? Yes!!

Space:  $\mathcal{O}(n \cdot m)$

$LCS[i, j]$

Example:  $S = \text{ABAZDC}$   
 $T = \text{BACBAD}$

		B	A	C	B	A	D	← S
	0	0	0	0	0	0	0	
A	0	? 0	? 1	1	1	1	1	
B	0	? 1	1	1	? 2	2	2	BACB AB
A	0					? 3		BACBA ABA
Z	0							
D	0							
C	0							

↑  
T

**4** answer

# 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]);
```

*diagonal*

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

```
}
```

*solution*



# How much space do we need?

BACBA

A ↗

		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

$O(n)$

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

Backtracking

common  
chars

# Discussion Problem 4

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?  $O(2^n)$

Step 1. Subproblems.

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

Step 2. Recurrence.

2 cases

case 1.  $s[i] = s[j] \Rightarrow OPT[i, j] = 2 + OPT[i+1, j-1]$

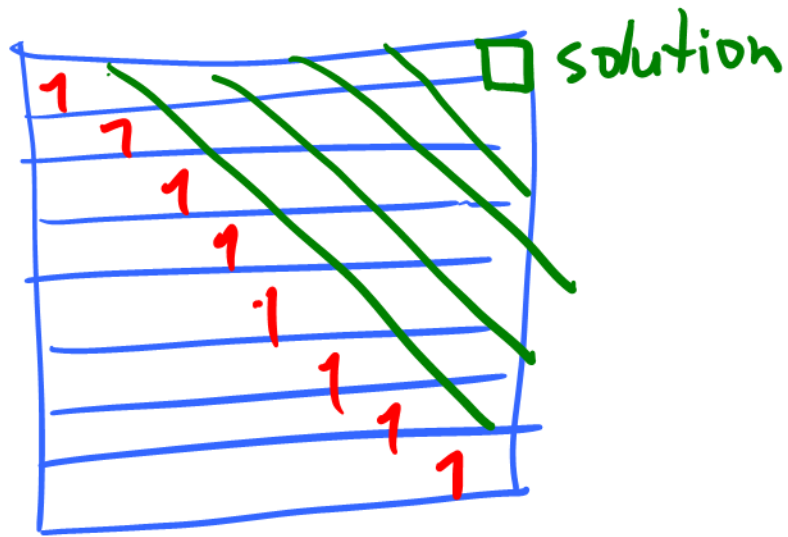
case 2.  $s[i] \neq s[j] \Rightarrow OPT[i, j] = \max(OPT[i+1, j], OPT[i, j-1])$

Base cases

$OPT[i, i] = 1$  / string of one char)

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

example  
"AA"  
 $i: 0 \rightarrow 1$   
 $j: 1 \rightarrow 0$



$DP[i, h]$

Runtime:  $O(n^2)$

Question: Can we use LCS to solve this problem?

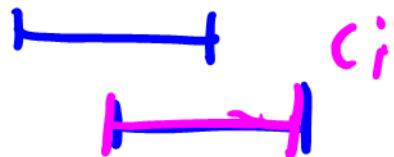
$LCS(str, reverse\ str)$

## Discussion Problem 5

You are to plan the spring 2022 schedule of classes. Suppose that you can sign up for as many classes as you want, and you'll have infinite amount of energy to handle all the classes, but you cannot have any time conflict between the lectures. Also assume that the problem reduces to planning your schedule of *one particular day*. Thus, consider one particular day of the week and all the classes happening on that day:  $c_1, \dots, c_n$ . Associated with each class  $c_i$  is a start time  $s_i$  and a finish time  $f_i$  and you also assign a score  $v_i$  to that class based on your interests and your program requirement. Assume  $s_i < f_i$ . You would like to choose a set of courses  $C$  for that day to maximize the total score. Devise an algorithm for planning your schedule.

INPUT: array of classes

$c_1, c_2, \dots, c_{i-1}, c_i, c_{i+1}, \dots, c_n$



$O(n \log n)$

Sort classes by finish-time.

start with the rightmost class


step 1. Let  $OPT[j]$  be the optimal for  $c_1, c_2, \dots, c_j$

step 2.

$$OPT[j] = \max(OPT[j-1], V_j + OPT[p(j)])$$

$OPT[0] = 0$

$p(j) = \max_i (i \mid f_i < s_j)$

Runtime :  $O(h \cdot h)$   linear search

Space :  $O(h)$

binary search for  $P_j$ )

$O(h \cdot \log h)$