Analysis of Algorithms

V. Adamchik

CSCI 570

Lecture 6

University of Southern California

Fall 2023

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 $\Theta(n \log n)$. BS
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: $T(n) = 1.2\,T(n/2) + n$.
7. (T/F) The master theorem can be applied to the following recurrence: $T(n) = 9\,T(n/3) - n^2 \log n + n$. < 0
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.

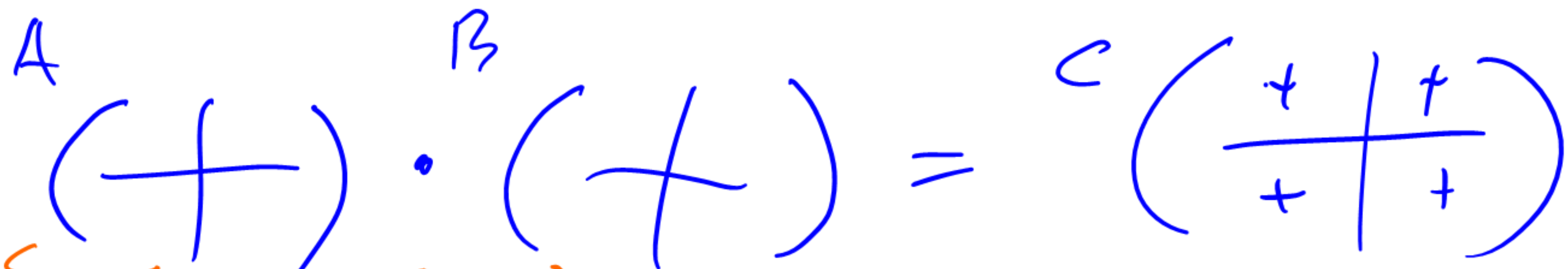
    a. Number of multiplications in the standard method: 8

    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: $\Theta(n^2)$.



$S_1 = (\ =\ ) + (\ =\ )$

$S_2 =$

$\vdots$

$S_7 =$

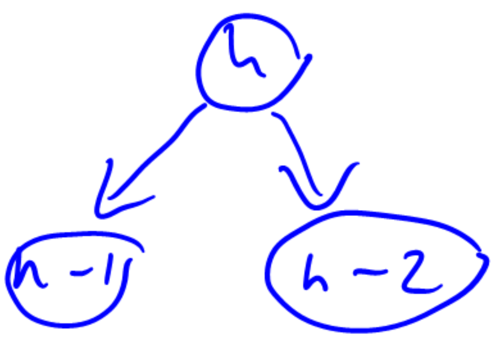
Extra space

$7 + 2 = 9$

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

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

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

How many bits in n-th Fib. num.

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

$$= \Theta(n)$$

φ = Golden Ratio

Solve for $T(n)$

$$T(n) \leq 2T(n-1) + O(n) \Rightarrow \text{exponential}$$

$= 2^n$

$T(n)$

Memoization

$T(n) = O(n^2)$

```
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];
}
```

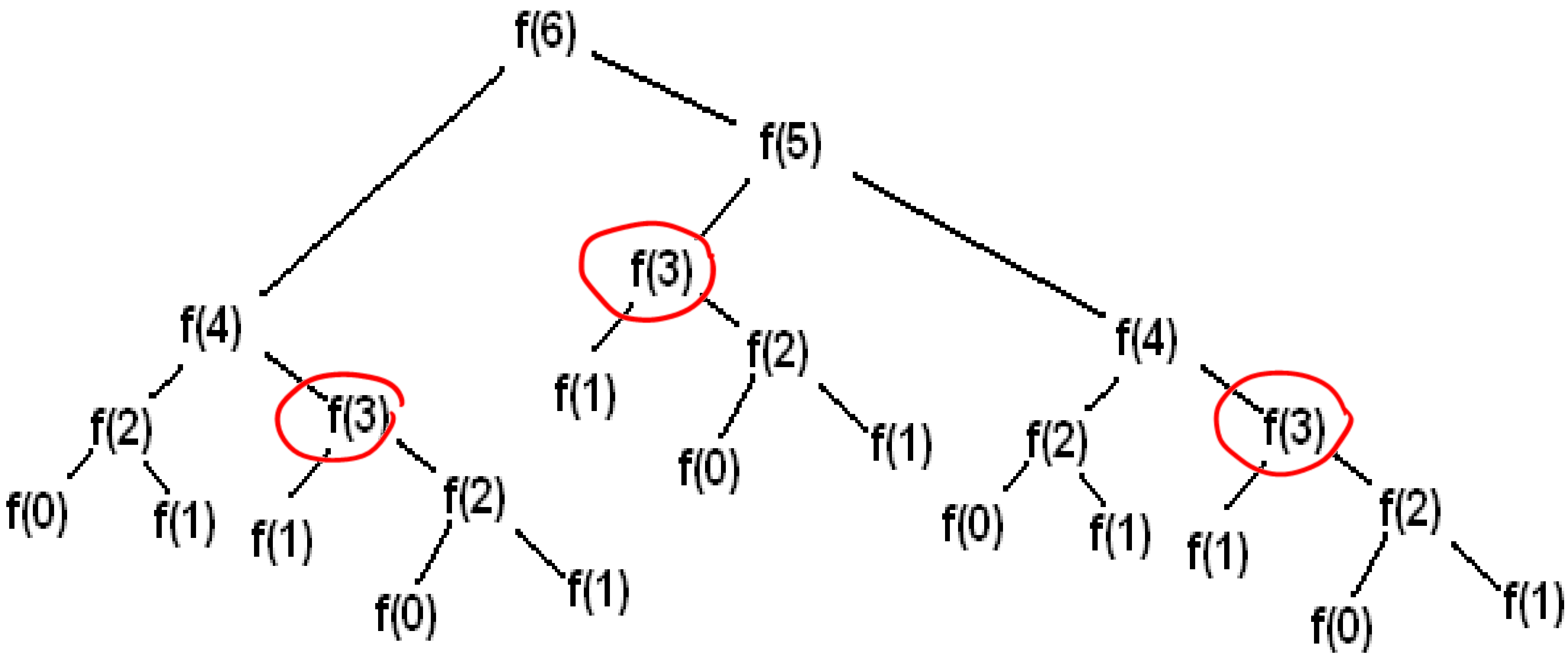
$O(1)$

$O(1)$

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

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

Overlapping Subproblems



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

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];
}
```

n
n-1
n-2
...
1

f

```
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;
}
```

n
⋮
2
1

Memoization:
a top-down approach.

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.
DP is not D&C. Compare with Mergesort.

DP is not greedy either. DP tries <u>every</u> 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.

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.

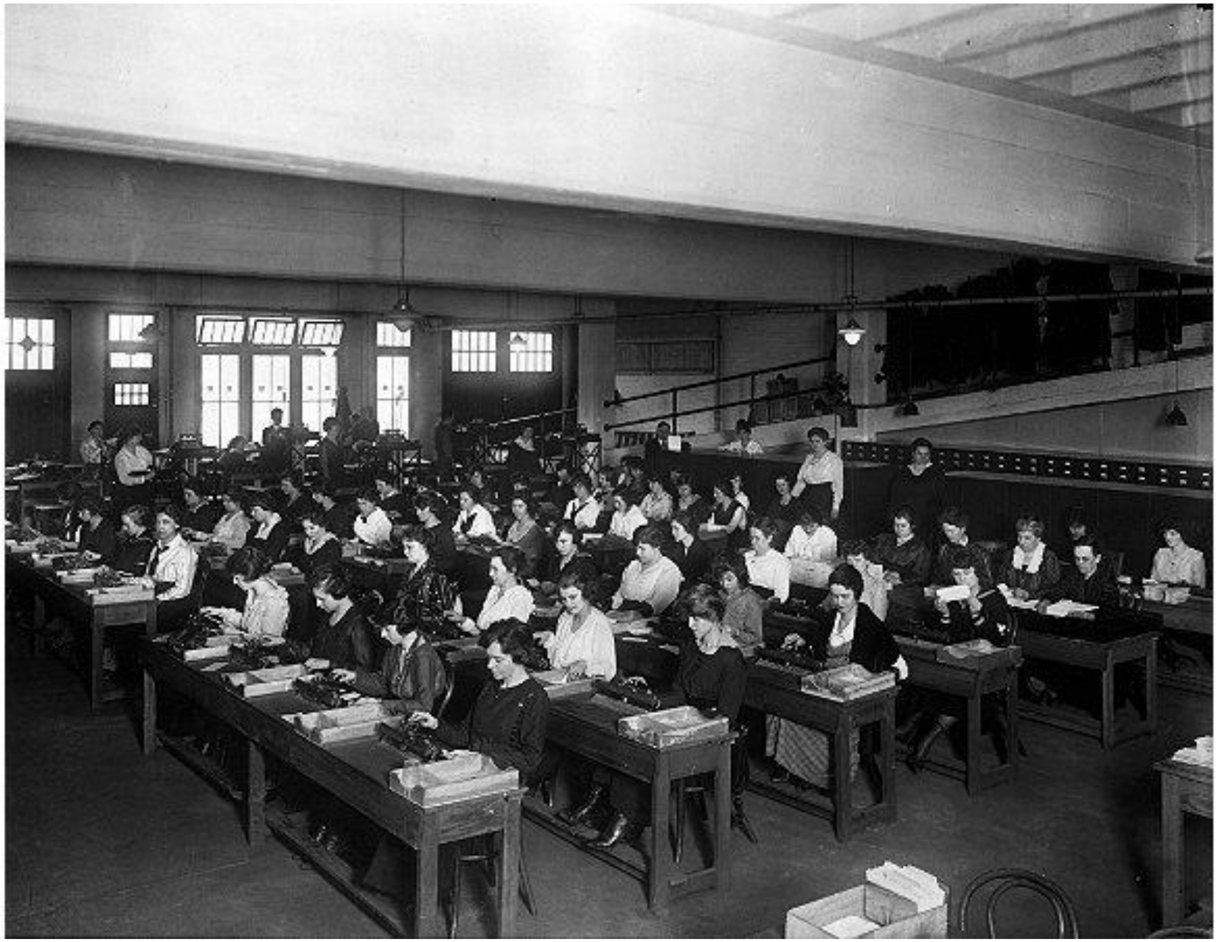
Dynamic Programming

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?

Break.

0-1 Knapsack Problem

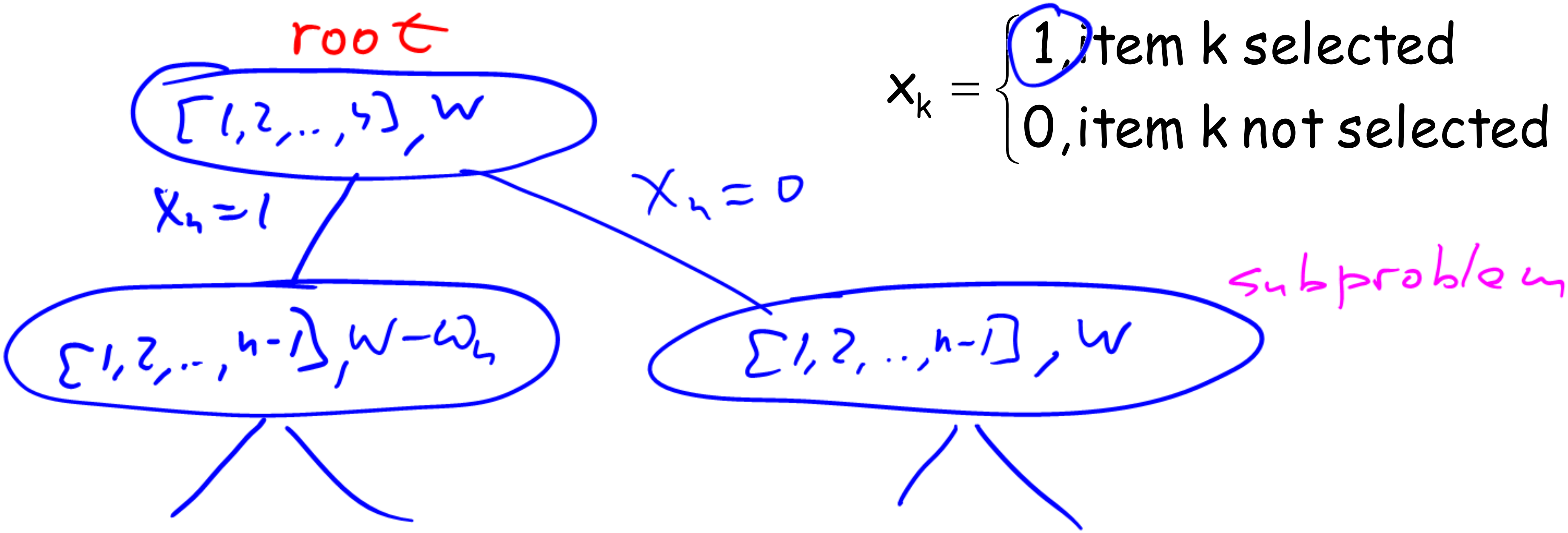
Given a set of n unbreakable *unique* items, each with a weight w_k and a value v_k, determine the subset of items such that the total weight is less or equal to a given knapsack capacity W and the total value is as large as possible.

Fractional knapsack — greedy

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 max 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

$O(1)$

table

$$OPT[K,X] = MAX\begin{cases} V_K + OPT[K-1,\ X-W_K], \\ OPT[K-1,X] \end{cases}$$

?

$O(1)$ table lookup

Base cases

$OPT[0,X] = 0$

$OPT[K,0] = 0$

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

Tracing the Algorithm

n = 4, W = 5

K=1 K=2

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

	0	w=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 OPT[n,W]

knapsack capacity

items

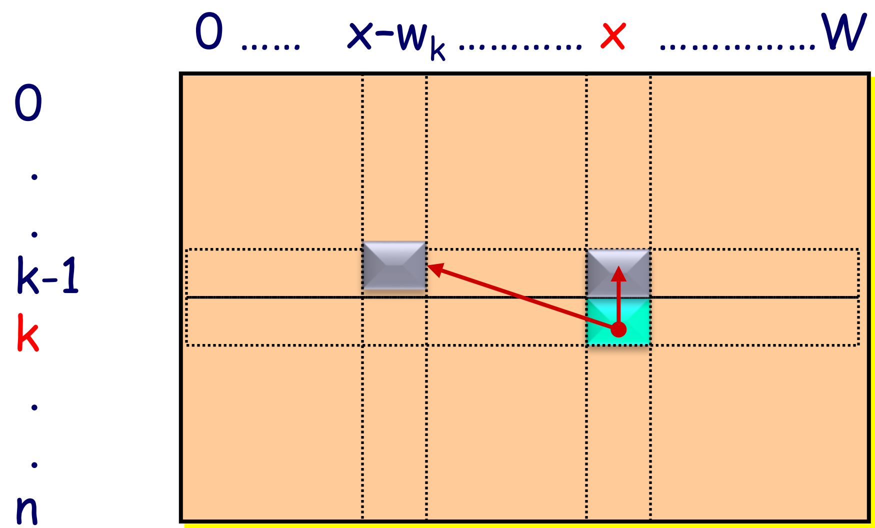
Pseudo-code

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;
             K
         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];
}
```

!!

Complexity



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

Runtime Complexity? table size times the work you do at each cell

Space Complexity?
$O(nW)$

Pseudo-Polynomial Runtime

exponential

<u>Definition</u>. 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, $T(n) = \Theta(n \cdot W)$

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

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

Inputsize: $O(n \cdot \log W)$ input size of W

Actual Runtime: $O(n \cdot 2 \qquad)$

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?

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

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.

$40 = 20 + 20$

$40 = 25 + 10 + 5$

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:

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

step 1

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

Step 2

$$OPT[K,X] = \underbrace{MIN}_{O(1)}\left(\overbrace{OPT[\underbrace{K-1}, X]}^{O(1)},\ \underbrace{OPT[K, X - \underbrace{d_K}] + 1}_{O(1)}\right)$$

Base cases:

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

$$OPT[K, 0] = 0$$

Steps 3 DIY

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

not an input size

is it polynomial? NO

Break

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

S_1 A B A Z D C

S_2 B A C B A D

① prefix $s_1[0, 1, \ldots, i] \to A, AB, ABA, \ldots$

② suffix $s_2[i, \ldots, n] \to C, DC, ZDC, \ldots$

③ $s_1[i, \ldots, j]$

Subproblems

Let $LCS[i,j]$ be the max length of the LCS of $S_1[0,\dots i]$ and $S_2[0\dots j]$

choices (last characters)

1) $S_1[i] = S_2[j]$

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

2) $S_1[i] \neq S_2[j]$

$$LCS[i,j] = MAX(LCS[i-1, j], LCS[i, j-1])$$

Recurrence

Combine those 2 cases

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1], & \text{if} \ldots \\ MAX\left(LCS[i-1, j],\ LCS[i, j-1]\right) & \text{if} \ldots \end{cases}$$

Base cases:

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

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						
Z	0						
D	0						
C	0						$\boxed{4}$

← S

AB – BACB

answer

T

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? $O(2^n)$

Step 1

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

$OPT[i,j] = OPT[i+1, j-1] + 2$

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

$$OPT[i,j] = MAX\left[\begin{array}{l} OPT[i+1, j], \\ OPT[i, j-1] \end{array}\right]$$

Base cases

$OPT[i,i] = 1$

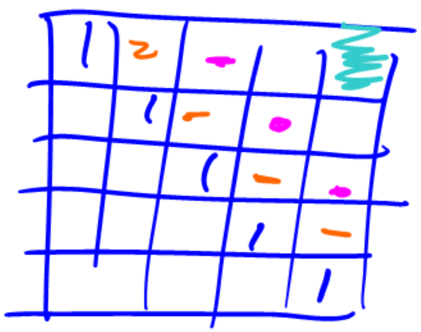
$OPT[i,j] = 2$, if $s[i] = s[j]$ and $j = i+1$

example

"AA"

$i=0, j=1$

$i=1, j=0$



final answer $OPT[0,n]$

Runtime: $O(n^2)$

Question: Can we use LCS to solve this problem? LCS[str, reverse]