

Dynamic programming

Dynamic programming is an algorithm design technique for solving problems that have:

- an optimal substructure property (recursion)
- overlapping subproblems

Idea: Do not repeatedly solve the same subproblems, instead solve them only once and store the solutions in a dynamic programming table.

Example: Fibonacci numbers $0, 1, 1, 2, 3, 5, 8, 13, 21, \dots, n$

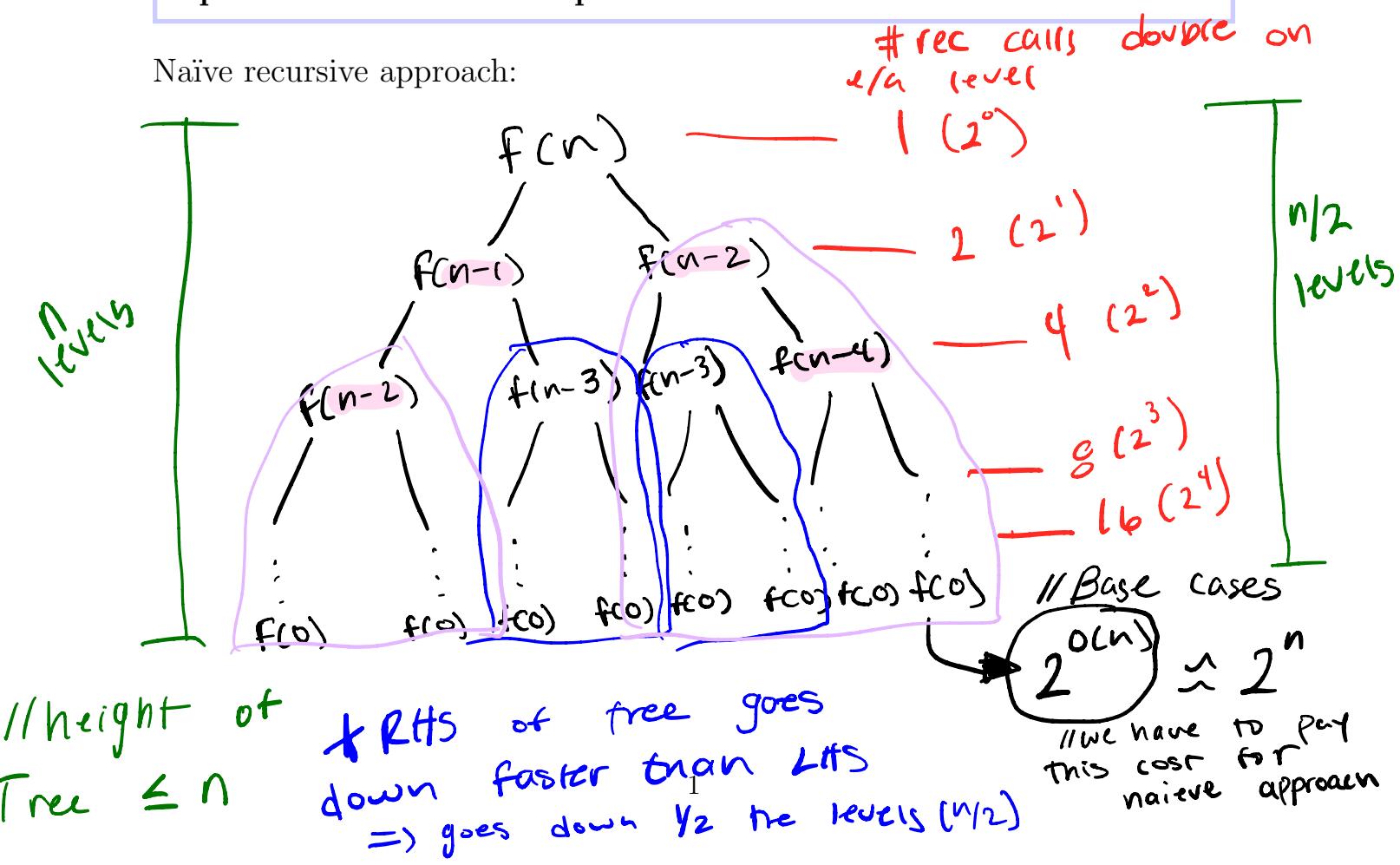
Base case: $F(0) = 0$ $F(1) = 1$

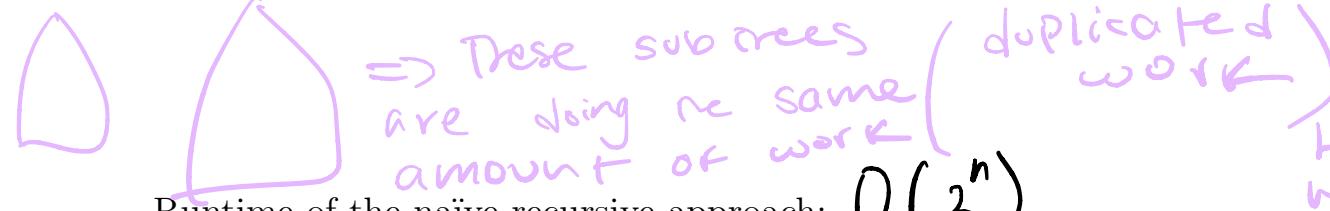
Recursive case: $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$ // 2 recursive calls

Dynamic Programming Hallmark #1:

An optimal solution to the problem depends on one or more optimal solutions to subproblems.

Naïve recursive approach:





duplicated work

Runtime of the naïve recursive approach: $O(2^n)$

Tells us we can do better

The number of distinct Fibonacci subproblems is only n .

$0, 1, 1, 2, 3, \dots, n$
n+1 #'s

Dynamic Programming Hallmark #2:

A naïve recursive solution has a few distinct subproblems which are solved repeatedly.

There are two variants of dynamic programming:

- Bottom-up dynamic programming (starts w/ smaller problems working up to bigger ones)
- Memoization (also called “top-down dynamic programming”) (starts w/ big problem seeing how it connects to smaller problems)

Bottom-up dynamic programming:

Store DP-table and fill bottom-up:

0	1	2	3	4	5	6	7	n
0	1	1	2	3	5	8	13	...

~ doesn't use recursion

Algorithm 1 int fibBottomUpDP(int n)

```

1: Create int array  $F[0 \dots n]$ 
2:  $F[0] = 0$ ; } fill base cases
3:  $F[1] = 1$ ; } cases
4: for  $i = 2; i \leq n; i++$  do } 2, 3, 4, 5, ..., n
5:    $F[i] = F[i - 1] + F[i - 2]$ ; }  $O(n)$ 
6: end for
7: return  $F[n]$ ;
```

Runtime: $\Theta(n)$

\Rightarrow Using an array to store calculated fib. numbers

\Rightarrow reduced runtime but we need slightly more memory, as we have an array of n size

Memoization:

Use a recursive algorithm. After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Algorithm 2 int fibMemoization(int n)

```

1: Create new array F of size n and initialize each element  $F[i] = \text{null}$ 
2: fibMemoizationRec( $n, F$ );
3: return  $F[n]$ ;

```

↳ helper function

Algorithm 3 int fibMemoizationRec(int n, int[] F)

```

1: if  $F[n] = \text{null}$  then
2:   if  $n = 0$  then
3:      $F[n] = 0$ ;
4:   else if  $n = 1$  then
5:      $F[n] = 1$ ;
6:   else
7:      $F[n] = \text{fibMemoizationRec}(n-1, F) + \text{fibMemoizationRec}(n-2, F)$ ;
8:   end if
9: end if
10: return  $F[n]$ ;

```

// How many rec. calls needed?

=> $n+1$ values, each makes 2 rec. calls beside base cases

=> 0, 1, 1, 2, 3, 5, 8, ...

$2 + 2(n-1) = 2n$ steps
 $\Theta(n)$

Time: $\Theta(n)$

Space: $\Theta(n)$ for Array to store saved values

+ $\Theta(n)$ for recursive calls
 $\Theta(n)$

Is this a useful algorithm design technique? Yes, dynamic prog. is

=> As always these algorithms serve certain purposes. BOTTOM-UP 3 very efficient in comparison to the naive approach

is a better tool than memoization for this problem.

From Wikipedia: https://en.wikipedia.org/wiki/Dynamic_programming

- Dijkstra's algorithm for finding single source shortest paths.
- The Needleman–Wunsch algorithm and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Floyd's all-pairs shortest path algorithm
- Recurrent solutions to lattice models for protein-DNA binding
- Optimizing the order for chain matrix multiplication
- Kadane's algorithm for the maximum subarray problem
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models, and particularly in part of speech tagging)
- The Earley algorithm (a type of chart parser)
- Pseudo-polynomial time algorithms for the subset sum, knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted
- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content-aware image resizing)
- Some approximate solution methods for the linear search problem
- Optimization of electric generation expansion plans in the Wein Automatic System Planning (WASP) package
- ...and more!

Greedy Algorithms

Making a locally optimal solution at every decision point will yield a globally optimal solution. = Focus on each subproblem

Greedy Hallmark #1:

An optimal solution to the problem depends on *exactly one* subproblem's solution.

Example: Make change using fewest number of coins (for the coin denominations: 1, 5, 10, 25)

Greedy strategy: [Choose the largest possible coin.] Repeat this until we reach the desired total.

Try for 89 cents: \Rightarrow 89 — 25 cents

$$= 64 - 25 \text{ cents}$$

= 39 — 25 cents

= 14 — 10 cents

$$= \frac{4 \text{ cents (1 cent)}}{8 \text{ coins total}}$$

Claim: This is least amount of coins possible for 89 cents

\Rightarrow Is this correct in
every case?

We have to prove that a greedy solution is optimal!

(Otherwise it is just a heuristic)

Correctness proof:

Let $n = 25Q + 10D + 5N + P$ be the optimal solution for n cents.

Thus, $C(n) = \underline{Q + D + N + P} = \# \text{ coins used}$

- $P < 5$

Suppose $P \geq 5$

\Rightarrow If $P \geq 5$, we could replace
5 pennies for 1 nickel for a
better solution.

- $5N + P < 10$ // money in nickles & pennies < 10

Suppose $5N + P \geq 10$

$\Rightarrow P \leq 4$,

\Rightarrow Thus at least 2 nickles,
which can be replaced w/ 1
dime

- $10D + \underline{5N + P} < 25$ // money in dimes, nickles, pennies < 25

Suppose $10D + 5N + P \geq 25$ \leftarrow
// 2 ways this can happen

case 1: suppose we had 1 nickel
 \Rightarrow Then we could replace nickel
w/ 2 dimes w/ 1 quarter

case 2: suppose we had 0 nickles // $10D + \underline{5N + P} < 25$

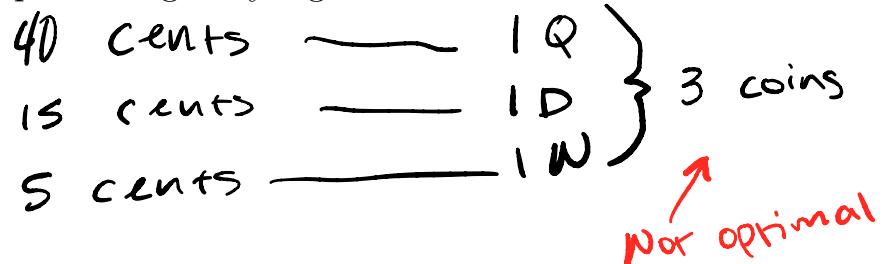
\Rightarrow Then we could replace
2 dimes w/ 1 quarter
& 1 nickel

Does this greedy algorithm work for any set of coin denominations?

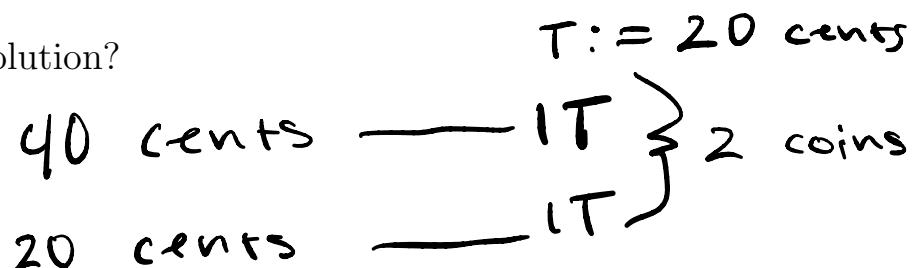
Try adding a 20 cent coin denomination:

1, 5, 10, 20, 25

How many coins will the previous greedy algorithm use to form $n = 40$ cents?



What is the optimal solution?



Return to Dynamic Programming

Let's find a dynamic programming solution for the making change problem. Specifically, suppose we have the following k coin denominations:

d_1, d_2, \dots, d_k] of k coins

$C(n)$ = minimum number of coins necessary to make change for n cents. // $C(27) = 3$ coins

Base cases:

$$C(d_1) = 1 \quad C(d_2) = 1 \quad \dots \quad C(d_k) = 1 \Rightarrow // C(1) = 1, C(5) = 1, C(25) = 1 \\ \text{Set } C(n) = \infty \quad \text{if } n < 0 \quad // C(-4) = \infty$$

Recursive case:

$$C(n) = \min_{i=1}^k [C(n - d_i) + 1] \quad // \text{Try every denomination } + 1 \text{ for coin being used}$$

For example:

Let $d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 20, d_5 = 25$

(case where dynamic prog. works)

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
C(i):	1	2	3	4	1	2	3	4	5	1	2	3	4	5			

$\boxed{1+1=2}$ $\boxed{3+1=4}$ \uparrow $10 + \underset{\text{(current)}}{1 \text{ coin}} = 15$

Examples:

$$C(1) = 1 \text{ (base case)}$$

$$\begin{aligned}
 & C(2) = \min(1+C(2-1), 1+C(2-5), 1+C(2-10), 1+C(2-20), 1+C(2-25)) \\
 & . = \min(1+C(1), 1+C(-3), 1+C(-8), 1+C(-18), 1+C(-23)) \\
 & . = \min(1+1, 1+\infty, 1+\infty, 1+\infty, 1+\infty) \\
 & = \min(2) = 2 \text{ coins}
 \end{aligned}$$

$$\begin{aligned}
 & C(6) = \min(1+C(5), 1+C(1), 1+C(-4), 1+C(-14), 1+C(-19)) \\
 & . = \min(1+1, 1+1, 1+\infty, \dots, 1+\infty) \\
 & = \min(2, 2, \dots, \infty) \\
 & = 2 \text{ coins}
 \end{aligned}$$

$$\begin{aligned}
 & C(15) = \min(1+C(14), 1+C(10), 1+C(5), 1+C(-5), 1+C(-10)) \\
 & . = \min(1+5, 1+1, 1+1, \dots, 1+\infty) \\
 & = \min(2) = 2 \text{ coins}
 \end{aligned}$$

\therefore Table above helps with
saving work

Bottom up dynamic programming algorithm:

Algorithm 4 int MinCoinBottomUpDP(int n , int[] d)

```

1: //Let  $k$  denote the number of denominations in array  $d$            //  $c = [1, 1, 1, 1, 1, \dots k]$ 
2: //Base Cases
3: for  $i = 1; i \leq k; i++$  do
4:    $C[d[i]] = 1;$            ] Fill c array
5: end for
6:
7: //Recursive Cases
8: for  $i = 1; i \leq n; i++$  do - $O(n)$ 
9:    $min = \infty;$            ] impossible signal
10:  if  $i$  is not a base case then
11:    for  $j = 1; j \leq k; j++$  do - $O(k)$ 
12:      if  $d[j] < i$  and  $C[i - d[j]] < min$  then
13:         $min = C[i - d[j]];$            ] not negative
14:      end if
15:    end for                   ] better than
16:    end if                     ] best so far?
17:     $C[i] = min + 1;$ 
18:  end for
19: return c;  $C[n];$ 
```

Time: $O(n \cdot k)$

Slightly simpler alternative using a base case of 0:

Algorithm 5 int MinCoinBottomUpDP(int n , int[] d)

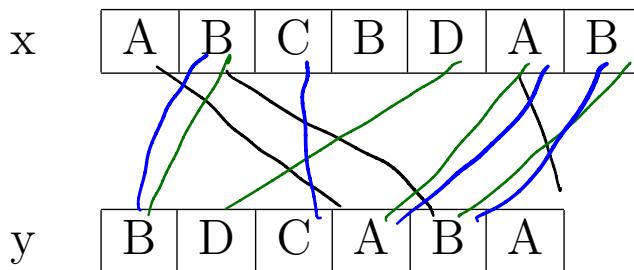
```

1:  $C[0] = 0;$            ] removed base cases
2: for  $i = 1; i \leq n; i++$  do
3:    $min = \infty;$            ] here, but same as
4:   for  $j = 1; j \leq k; j++$  do
5:     if  $d[j] < i$  and  $C[i - d[j]] < min$  then
6:        $min = C[i - d[j]];$            ] above
7:     end if
8:   end for
9:    $C[i] = min + 1;$ 
10: end for
11: return c;  $C[n];$ 
```

What is the runtime for the algorithm? $O(n \cdot k)$

Example DP Problem: Longest Common Subsequence (LCS)

Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$, find a longest subsequence common to them both.



//After picking element
you must move forward

Brute-force LCS algorithm: We want to show 4 is LCS

Check every subsequence of $x[1 \dots m]$ to see if it is also a subsequence of $y[1 \dots n]$.

=> check every subsequence

of x , then see if subsequence

of y

=> $x = \boxed{1 \dots m}$ //every element
is either a
subsequence
or not (y)
 $\Rightarrow x = \begin{matrix} & & & & & m \\ y_0 & y_0 & y_0 & y_0 & y_0 & y_0 \\ 2 & 2 & 2 & 2 & 2 & 2 \end{matrix}$
 $= 2^m$ subsequences of X

Idea for a better algorithm:

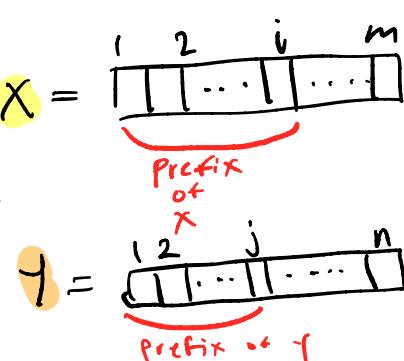
- Look at the length of a longest common subsequence.
- Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$. = length of string

Strategy: Consider prefixes of x and y .

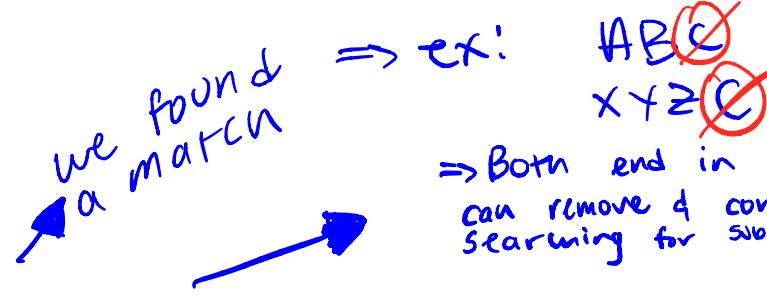
- Define $c[i, j] = |LCS(x[1 \dots i], y[1 \dots j])|$.
- Then, $c[m, n] = |LCS(x, y)|$

When $i, j = m, n$
then we found our
answer, which return $(c[m, n])$

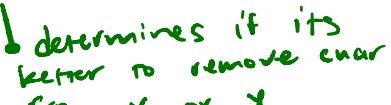


Recursive formulation

Theorem.


 $\Rightarrow \text{Both end in } C, \text{ we can remove it and continue searching for subsequences}$

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } X[i] == Y[j] \\ \max(c[i-1, j], c[i, j-1]) & \text{otherwise} \end{cases}$$


 determines if its better to remove char from X or Y

Proof. Case $x[i] = y[j]$

$$\text{if } X[i] == Y[j]$$

other wise
 $(\text{characters of } X \neq Y)$ ∵ 2 cases
 \Rightarrow thus we should remove 1 or the 2
 \Rightarrow remove X or
 $\text{characters } (i \text{ or } j) \Rightarrow \text{remove } Y$

Suppose $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$.

Therefore, $c[i, j] = |z| = |LCS(x[1 \dots i], y[1 \dots j])|$.

It must be the case that $z[k] = x[i] = y[j]$

Thus, $z[1 \dots k-1]$ is common subsequence of $x[1 \dots i-1]$ and $y[1 \dots j-1]$

We want to show $z[1 \dots k-1]$ is $LCS(x[1 \dots i-1], y[1 \dots j-1])$.

Suppose w is a larger common subsequence of $LCS(x[1 \dots i-1], y[1 \dots j-1])$.

That is to say $|w| > k-1$.

Then concatenate $z[k]$ onto the end of w . Clearly, $|w+z[k]| > k$.

This contradicts the assumption that $c[i, j] = k$. Therefore, no such w exists.

Naïve recursive approach based on the previous recurrence:

Algorithm 6 int LCS(String x , String y , int i , int j)

```

1: if  $i = 0$  or  $j = 0$  then → NO matching (chars) left
2:   return 0;
3: end if
4: if  $x[i] = y[j]$  then
5:   return LCS( $x, y, i - 1, j - 1$ ) + 1;
6: else
7:   return Max{LCS( $x, y, i - 1, j$ ), LCS( $x, y, i, j - 1$ )};
8: end if

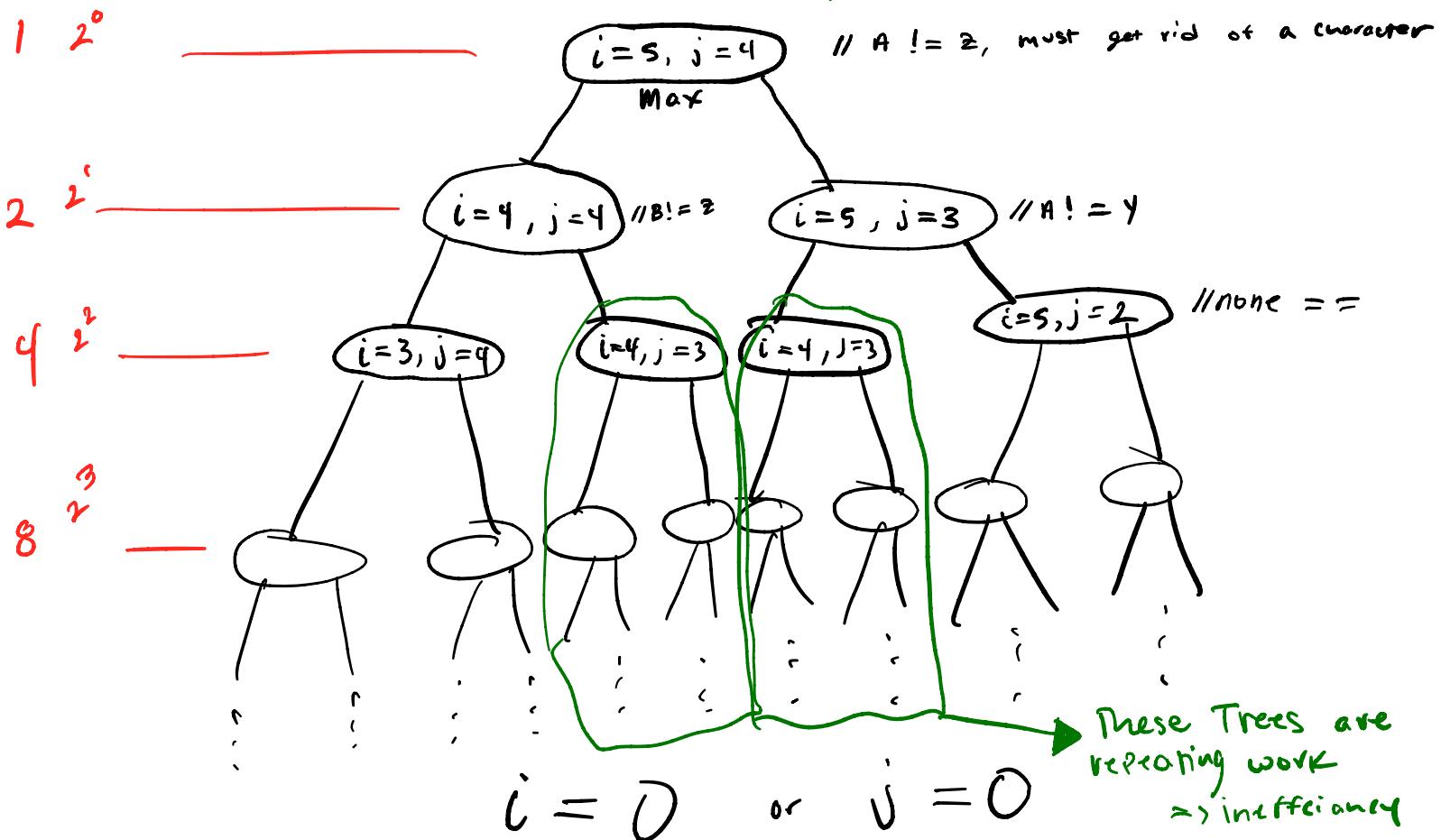
```

∴ Initial call
 $\Rightarrow \text{LCS}[x, y, m, n]$
 Then works its way down

Worst case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Runtime: $2^{n+m} = \text{len of array}$

Recursion tree for $x = EDCBA$, $y = WXYZ$, $i = 5$, $j = 4$



The number of distinct LCS subproblems for two strings of lengths m and n is only mn . DP runtime should be $O(m \cdot n)$

Thus, we should try to solve this using dynamic programming.

Ex: let $x = BAB$
 $y = AB$

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Algorithm 7 int LCSMemoization(String x , String y , int m , int n)

- 1: Create new 2d array c of size m by n .
 - 2: Initialize each element $c[i, j] = \text{null}$
 - 3: LCSMemoizationRec(x, y, m, n, c);
 - 4: return $c[m, n]$;
-

Table
store
use
solutions

Algorithm 8 int LCSMemoizationRec(String x , String y , int i , int j , int[] c)

- 1: if $c[i, j] = \text{null}$ then

Base cases
Recursive calls
 - 2: if $i = 0$ then

$c[i, j] = 0$
 - 3: else if $j = 0$ then

$c[i, j] = 0$
 - 4: else if $x[i] = y[j]$ then

$c[i, j] = \text{LCSMemoizationRec}(x, y, i - 1, j - 1, c) + 1$
 - 5: else

$c[i, j] = \text{Max}\{\text{LCSMemoizationRec}(x, y, i - 1, j, c),$
 $\text{LCSMemoizationRec}(x, y, i, j - 1, c)\}$
 - 6: end if
 - 7: end if
 - 8: return $c[i, j]$;
-

\Rightarrow Every # in table is computed at most 1 time

Simple LCS example:

4 possibilities

3

	$i=0$ no char left	$j=1$	$j=2$	$j=3$
$i=1$ A	0	0	0	0
$i=2$ B	0	1	1	2
	$AB \leftrightarrow B$	$AB \leftrightarrow BA$	$AB \leftrightarrow BAB$	

Annotations in the table:

- Top row: $i=0$ no char left, $j=0, 1, 2, 3$
- Left column: $i=1$ A, $i=2$ B
- Cells:
 - $(i=0, j=0)$: 0
 - $(i=0, j=1)$: 0
 - $(i=0, j=2)$: 0
 - $(i=0, j=3)$: 0
 - $(i=1, j=0)$: 0
 - $(i=1, j=1)$: 0 (labeled $\text{Max} = 1$)
 - $(i=1, j=2)$: 1 (labeled $\text{Max} = 1$)
 - $(i=1, j=3)$: 1
 - $(i=2, j=0)$: 0
 - $(i=2, j=1)$: 1
 - $(i=2, j=2)$: 1 (labeled $\text{Max} = 1$)
 - $(i=2, j=3)$: 2 (labeled $\text{Max} = 2$)
- Arrows and annotations:
 - From $(i=1, j=1)$ to $(i=0, j=0)$: $\text{Max} = 1$
 - From $(i=1, j=2)$ to $(i=0, j=0)$: $\text{Max} = 1$
 - From $(i=2, j=1)$ to $(i=1, j=0)$: $\text{Max} = 1$
 - From $(i=2, j=2)$ to $(i=1, j=0)$: $\text{Max} = 1$
 - From $(i=2, j=3)$ to $(i=1, j=0)$: $\text{Max} = 2$

\Rightarrow claim from start of lecture these strings will return 4 \therefore if matching characters -i, -j, +1 otherwise: max of neighbors

Longer LCS example:

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	" "	A	B	C	B	D	A	B
i=1	B	0 0	1	1	1	1	1	1
i=2	D	0 0	1	1	1	2	2	2
i=3	C	0 0	1	2	2	2	2	2
i=4	A	0 1	1	2	2	2	3	3
i=5	B	0 1	1	2	3	3	3	4
i=6	A	0 1	2	2	3	3	4	4

\Rightarrow Appears of length of LCS = 4

\Rightarrow Drew arrows to display Traceback

\hookrightarrow shows what substring is matched

1.E BDA B using traceback Arrows

• BOTTOM-UP was used here (fully fills table) // memoization starts at 4 and then solves problems. \Rightarrow It would skip areas to not do extra work

Our solution to LCS can be extended to solve a variety of problems which require measuring the similarity of two strings:

- Spell-checking
- Optical character recognition
- Comparing DNA sequences
- etc.

As table grows, region we want to calculate excludes stuff in corners

\Rightarrow Memoization chooses to fill in only relevant parts of table

