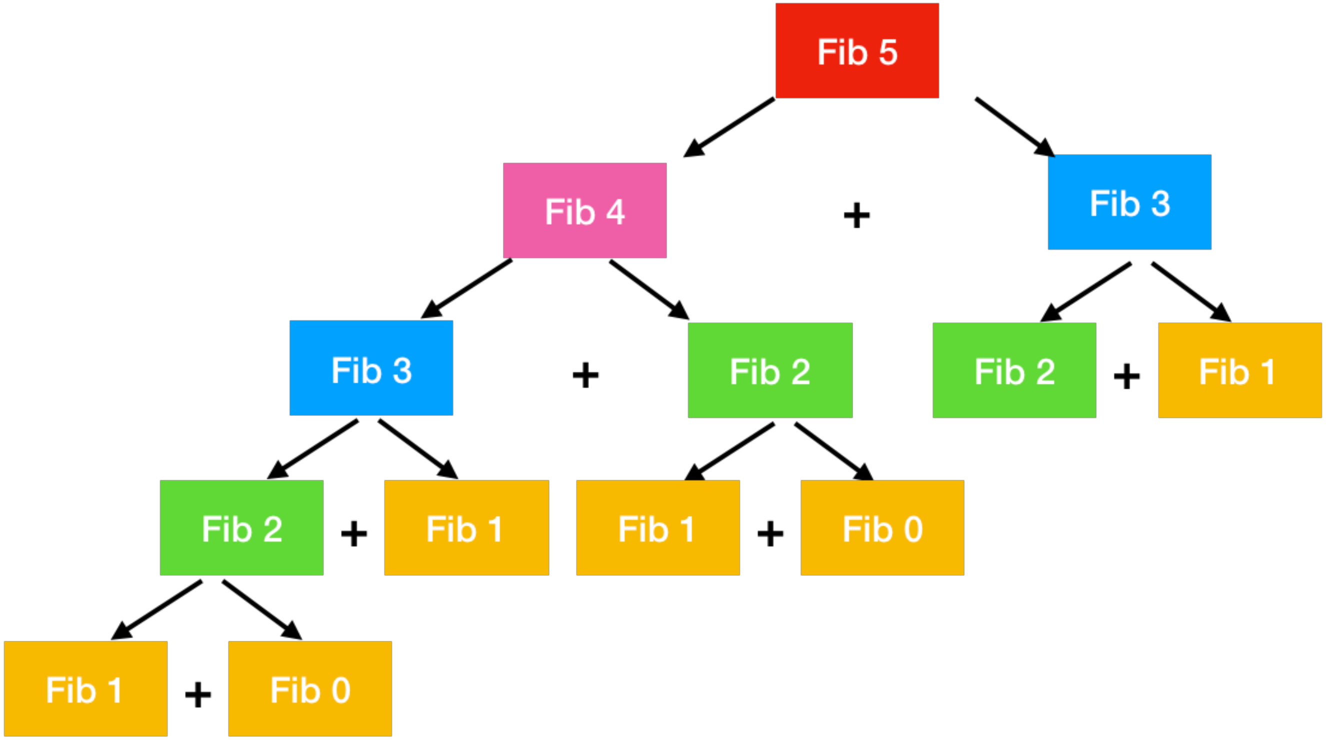
Dynamic Programming



NYIT CSCI-651

Complexity Recap

– constant:	$O(1)$	
– logarithmic:	$O(\log n)$	($\log_k n$, $\log n^2 \in O(\log n)$)
– poly-log:	$O(\log^k n)$	(k is a constant >1)
– linear:	$O(n)$	
– (log-linear):	$O(n \log n)$	(usually called "n log n")
– (superlinear):	$O(n^{1+c})$	(c is a constant, $0 < c < 1$)
– quadratic:	$O(n^2)$	
– cubic:	$O(n^3)$	
– polynomial:	$O(n^k)$	(k is a constant) "tractable"
– exponential:	$O(c^n)$	(c is a constant > 1) "intractable"

What is Dynamic Programming?

- DP is an algorithm technique to solve optimization problems
  - Such problems can have many possible solutions.
  - Each solution has a value, and
  - We wish to find a solution with the optimal (minimum or maximum) value.
- Mainly used when solution to a problem can be viewed as the result of a sequence of decisions

What is Dynamic Programming?

- Careful brute-force (checking all possibilities) that can give us polynomial time
  - Not always possible to achieve polynomial. But when it is, it is very useful

DP = subproblems + reuse

DP Methods

- **Top-Down with Memoization**
  - Solve problem naturally with recursion
  - Modified: Save the results of each problem in a data structure (Array or Hash Table)
  - Always checks to see if it has solved a subproblem. If yes, returns the saved value. If no, computes the value in the normal manner
  - **The recursive procedure has been memoized; it "remembers" what results it has computed previously**
- **Bottom-Up**
  - Sort the subproblems by size
  - Solve the smaller subproblems first
  - When solving a subproblem, we would have already solved all the smaller subproblems it depends on

The two methods yield algorithms with the same asymptotic running time in most cases.

Fibonacci (updated)

- Remember Fibonacci :

  0, 1, 1, 2, 3, 5, 8, 13, ….

- $f_0 = f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$, for $n > 1$.

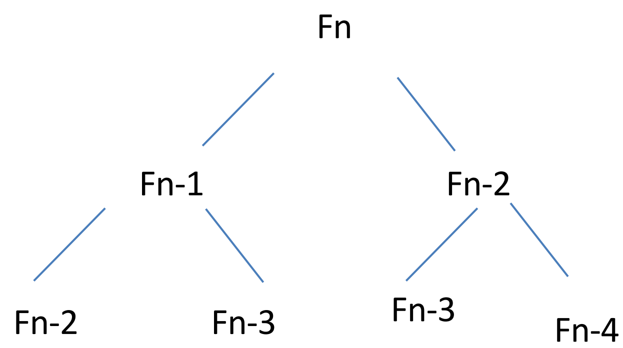
- Can be computed easily using recursion!

Fibonacci – naïve (updated)

```
fib_naive(n):
    if n <= 1:
        return n
    return fib_naive (n-1) + fib_naive (n-2)
```

- This has exponential complexity

Fibonacci : DP - Memoization



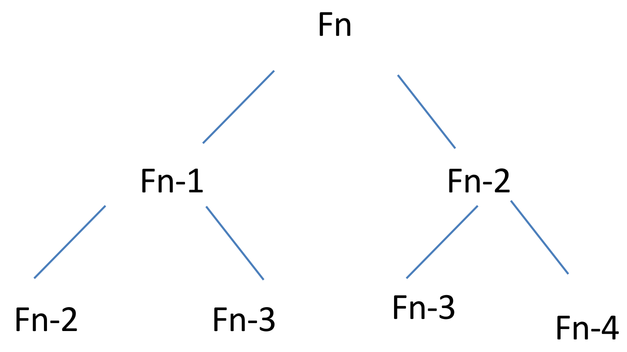
- Here, Subproblems overlap , that is, subproblems share subsubproblems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

As you can see there are repetitions, which we can memorize

Memorandum: To be remembered

Fibonacci : DP - Memoization

- You can think of two ways of calling fib(k)
  - The first time it is called, it uses recursion
  - The second time, it uses memoization



Fibonacci : DP – Memoization (updated)

```
memo={}
fib_top_down(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    f = fib_top_down(n-1) + fib_top_down(n-2)
    memo[n] = f
    return memo[n]
```

Fibonacci : DP - Memoization

- What is the time complexity of calling memoized values?
  - O(1)

- How many times do we call non-memoized fib?
  - n
  - But at each call the cost is constant

- Overall : O(n)

DP - Memoization

- DP:
  Memoize (remember)
  and reuse these solutions to sub-problems that help solve the problem

- Therefore: DP ~ recursion + memoization

- Therefore:
  running time of DP ~
  number of subproblems x time spent per subproblem

- Ignore recursive calls when you are computing time spent per subproblem

Fibonacci : DP - Bottom-up (updated)

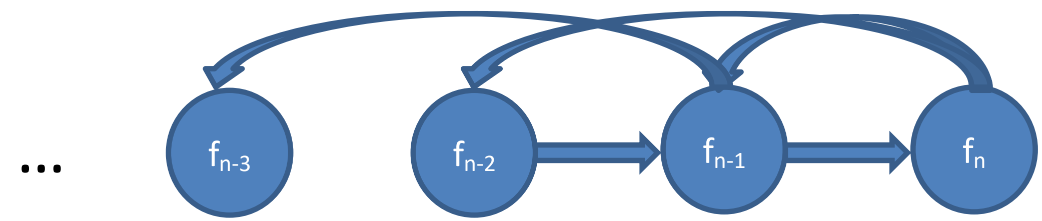
```
fib_bottom_up(n):
    if n <= 1:
        return n
    fib = [0] * (n + 1)
    fib[1] = 1
    for i in range(2, n + 1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

- Same thing happening as before, in the same order. We are using a loop instead of recursion

The bottom-up approach has much better constant factors, since it has less overhead for procedure calls.

DP - Bottom-up

- Exactly the same running time as memoization
- What we are doing is: Topological sort of subproblem dependency DAG
  - In case of fibonacci:



- Above is topologically sorted from left to right (f_1 to f_{n-1}) in order

Recap

- Memoization and bottom-up are two perspectives to look at the problem

- Memoization and bottom-up have the same running time

- With bottom-up we usually save space too
  - We only keep what is useful

Algorithm Strategies - Review

- Dynamic programming algorithms:
  - Choice is made based on evaluation of all possible results
  - Time and space complexity are usually higher
- Greedy algorithms:
  - Choice is made based on locally optimal solution
  - Usually faster, but may not result in globally optimal solution
- Divide and conquer algorithms:
  - Choice of input division is made based on assumption that merging result of sub-problems is optimal

Some problems that can be effectively solved with DP:

- Sequence alignment and bioinformatics algorithms
- Longest common subsequence
- Subset sum
- 0/1 knapsack problem
- Chain matrix multiplication
- Rod cutting problem
- All-pairs shortest path
- ….

Sequence alignment

- Compare two strings to see if they are similar
  - We need to define similarity
  - Very useful in many applications
  - Comparing DNA sequences, articles, source code, etc.

  - Examples: Edit Distance, Longest Common Subsequence problem (LCS)

Hamming Distance

- Hamming distance is the number of positions where two strings of equal length have different characters.
- Example: The strings

0010100111010

0110110110010

have a Hamming distance of 3.

- In simple terms, Hamming distance is the minimum number of *substitutions* required to transform one string into another.

- While Hamming distance has the advantage of simple O(n) calculation, its real world uses are somewhat limited.

- In applications such as spell checkers and bioinformatics (DNA sequences) we are often concerned with two additional types of errors, *insertions* and *deletions*.

- The edit distance between two strings is the minimum number of single character insertions, deletions, and substitutions needed to change one string into the other.

- The edit and Hamming distances can be quite different. The following two strings have a Hamming distance of 16 but an edit distance of 2:

0101010101010101

1010101010101010

- The first string can be converted to the second by deleting the leading 0, then inserting a trailing 0, i.e., two edits.

Edit Distance

- How many edits are needed to exactly match the Target with the Pattern
- Target: TCGACGTCA
- Pattern: TGACGTGC

Edit Distance

- How many edits are needed to exactly match the Target with the Pattern
- Target: TCGACGT CA
- Pattern: T GACGTGC
- Three:
  - By Deleting C and A from the target, and by Deleting G from the Pattern

Edit Distance

- Some applications are:
  - Approximate String Matching
  - Spell checking
  - Web search for finding similar word variations
  - DNA sequence comparison
  - Pattern Recognition

The Edit Distance Problem

- Solutions:
  - Brute Force – $O(K^N)$
  - Greedy – No Optimal Algorithms yet
  - Divide & Conquer – None discovered yet
  - Dynamic Programming – $O(N^2)$

- What is the edit distance between the words *impartial* and *parallel*? We will assume that inserts, deletes, and substitutions have a cost of 1, while a copy costs 0. (Notice that deletes move to the next letter in the "from" word and inserts the next letter in the "to" word while copies and subs move to the next letter in both words. This will be important when we construct our array.)

Sent (from)	i	m	p	a	r	t	i	a	l
Received (to)	p	a	r	a	l	l	e	l	

delete delete copy copy copy sub sub sub insert copy

Optimal substructure: The optimal solution contains within it subsolutions, i.e, optimal solutions to subproblems

Overlapping subsolutions: The subsolutions overlap and would be computed over and over again by a brute-force algorithm.

For edit distance:

Subproblem: edit distance of two prefixes

Overlap: most distances of prefixes are needed 3 times (when moving right, diagonally, down in the matrix)

- The previous words are at an edit distance of 6. However, the number of changes to covert one string to another is not always obvious. What is the edit distance of these two strings?

S_1 = Parallel algorithms confuse me.

S_2 = All alligators consume meat.

- Consider the following naïve algorithm for finding the edit distance between S_1 and S_2.

1. Try every possible single move (deletion, insertion or substitution) on both strings to see if they are ever equal.
2. If this does not work, try every possible combination of 2 moves.
3. If this does not work, try every possible combination of 3 moves.
4. Etc.

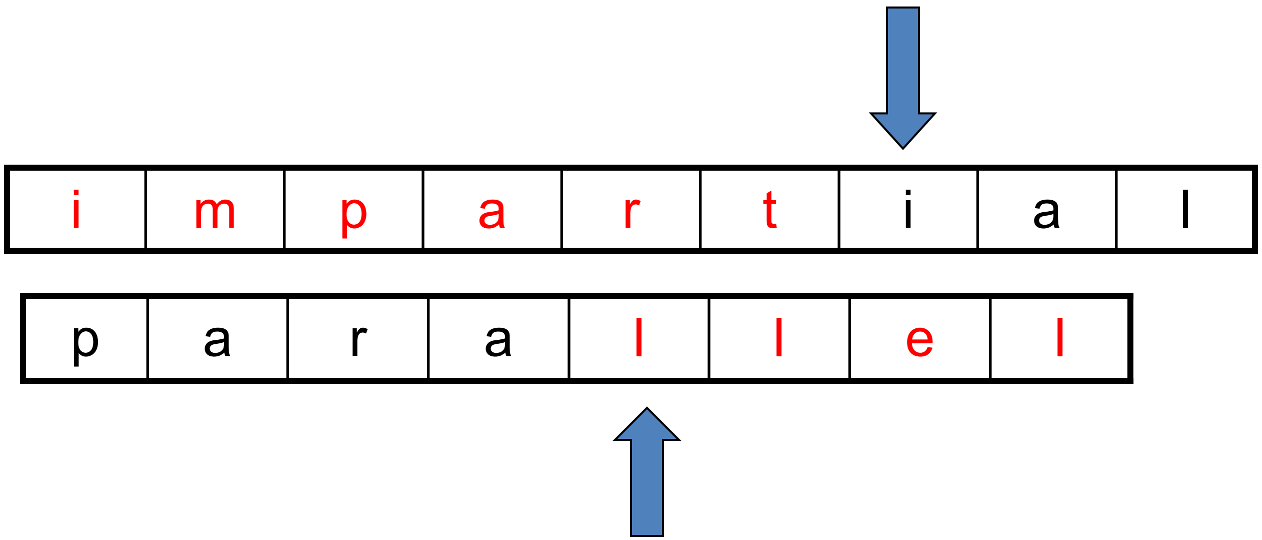
- The above algorithm would not only be difficult to implement, it is exponential complexity.

- Let’s check to see if this problem is a candidate for a solution by dynamic programming: must have optimal substructure.

- As with most dynamic programming problems we will prove optimal substructure by contradiction. Assume a non optimal substructure and show that this results in a contradiction.

- Assume the sequence of inserts, deletes, copies and subs from the previous example is an optimal solution. Let us stop this sequence at some arbitrary point (marked with red font) and look at the resulting sub problem.

- delete delete copy copy copy sub sub sub insert copy
- At this point in the sequence the problem looks like this and has a cost so far of 3:



- Assume there is a lower cost (say 2) sequence of moves that will bring us to the same point. If this were true we could just continue from this point with sub sub insert copy for a total cost of 5. But this contradicts our original assumption that 6 was the optimal cost. Therefore, there is no lower cost sequence of moves, and the substructure is optimal.

- We will represent the series of possible sub problems by an array. The rows represent the *from* string and the columns the *to* string. Here are the first few rows of the array for the previous problem:

to

from

		p	a	r	a	l	l	e	l
i		x							
m									
p			y						
a				z					

- Every element of the array will hold the minimum cost for that sub problem. For example, x holds the minimum cost to convert **i** to **p**. y holds the minimum cost to convert **imp** to **pa**. z holds the minimum cost to convert **impa** to **par**.

- We build the array from the top-left to the bottom-right. At each step we have three of four choices – insert, delete, substitute, or copy. We chose the move that gives the lowest total cost.

- **Deletes** move to the next letter in the “from” word (**down**)
- **Inserts** move to the next letter in the “to” word (**right**)
- **Subs** and **copies** move to the next letter in both words (**diagonally down**).

to

from

		p	a	r	a
	0	1	2	3	4
i	1	1	2	3	4
m	2	2	2	3	4
p	3	2	3	3	4
a	4	3	?		

to

from

		p	a	r	a
	0	1	2	3	4
i	1	1	2	3	4
m	2	2	2	3	4
p	3	2	3	3	4
a	4	3	?		

- We could move down with a delete for a total cost of 3 + 1 = 4.
- We could move across with an insert for a total cost of 3 + 1 = 4.
- We could move diagonally with a copy for a total cost of 2 + 0 = 2. This is the best move.

- Every element of the array will hold the optimum value for that sub problem. It should also hold the move that gave us this value to assist with a trace back (if required).

		p	a	r	a
	0	ins 1	ins 2	ins 3	ins 4
i	del 1	sub 1			
m	del 2				
p	del 3				
a	del 4				

- Let $E(i,j)$ be the edit cost for the sub problem of row i and column j. Let D be the cost of a delete, S be the cost of a substitution, C be the cost of a copy (usually zero) and I be the cost of an insert. Often D, S, I are all one, but they need not be. The recursive formula for the edit distance problem is:

$$E(i, 0) = i * D$$

$$E(0, j) = j * I$$

$$E(i, j) = \min \begin{cases} E(i-1, j) + D \\ E(i, j-1) + I \\ E(i-1, j-1) + S \text{ where } string_1[i] \neq string_2[j] \\ E(i-1, j-1) + C \text{ where } string_1[i] = string_2[j] \end{cases}$$

- Let's fill in the matrix for the words *impartial* and *parallel*.

We will assume that inserts, deletes, and substitutes all have a cost of 1, while a copy costs zero.

The first row is easy, since the only way to convert a null string to a string is with a series of inserts.

The first column is also easy, since the only way to convert a string to a null string is with a series of deletes.

		p	a	r	a	l	l	e	l
	0	1 ins	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
i	1 del								
m	2 del								
p	3 del								
a	4 del								
r	5 del								
t	6 del								
i	7 del								
a	8 del								
l	9 del								

Now we have a choice of:

Moving down with a delete for a total cost of 2.

Moving across with an insert for a total cost of 2.

Moving diagonally with a substitute for a total cost of 1.

		p	a	r	a	l	l	e	l
	0	1 ins	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
i	1 del								
m	2 del								
p	3 del								
a	4 del								
r	5 del								
t	6 del								
i	7 del								
a	8 del								
l	9 del								

Clearly, sub is the best choice. Continue filling this row with inserts.

Note: There may not always be one single optimum choice. For example, most of the second row could have been filled with substitutes for the same cost as inserts.

		p	a	r	a	l	l	e	l
	0	1 ins	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
i	1 del	1 sub							
m	2 del								
p	3 del								
a	4 del								
r	5 del								
t	6 del								
i	7 del								
a	8 del								
l	9 del								

Our choices here are:

Delete for a total cost of 3.

Insert for a total cost of 4.

Copy for a total cost of 2.

Notice that a copy is only possible when $string_1[i] = string_2[j]$.

		p	a	r	a	l	l	e	l
	0	1 ins	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
i	1 del	1 sub	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
m	2 del	2 sub	2 sub	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
p	3 del								
a	4 del								
r	5 del								
t	6 del								
i	7 del								
a	8 del								
l	9 del								

This is the completed array. Now we trace back to get our sequence of operations.

		p	a	r	a	l	l	e	l
	0	1 ins	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
i	1 del	1 sub	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
m	2 del	2 sub	2 sub	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
p	3 del	2 copy	3 ins	3 sub	4 ins	5 ins	6 ins	7 ins	8 ins
a	4 del	3 del	2 copy	3 ins	3 copy	4 ins	5 ins	6 ins	7 ins
r	5 del	4 del	3 del	2 copy	3 ins	4 ins	5 ins	6 ins	7 ins
t	6 del	5 del	4 del	3 del	3 sub	4 ins	5 ins	6 ins	7 ins
i	7 del	6 del	5 del	4 del	4 del	4 sub	5 ins	6 ins	7 ins
a	8 del	7 del	6 del	5 del	4 copy	5 del	5 sub	6 sub	7 ins
l	9 del	8 del	7 del	6 del	5 del	4 copy	5 copy	6 ins	6 copy

The optimal sequence is:

Del Del Copy Copy

Copy Sub Sub Ins

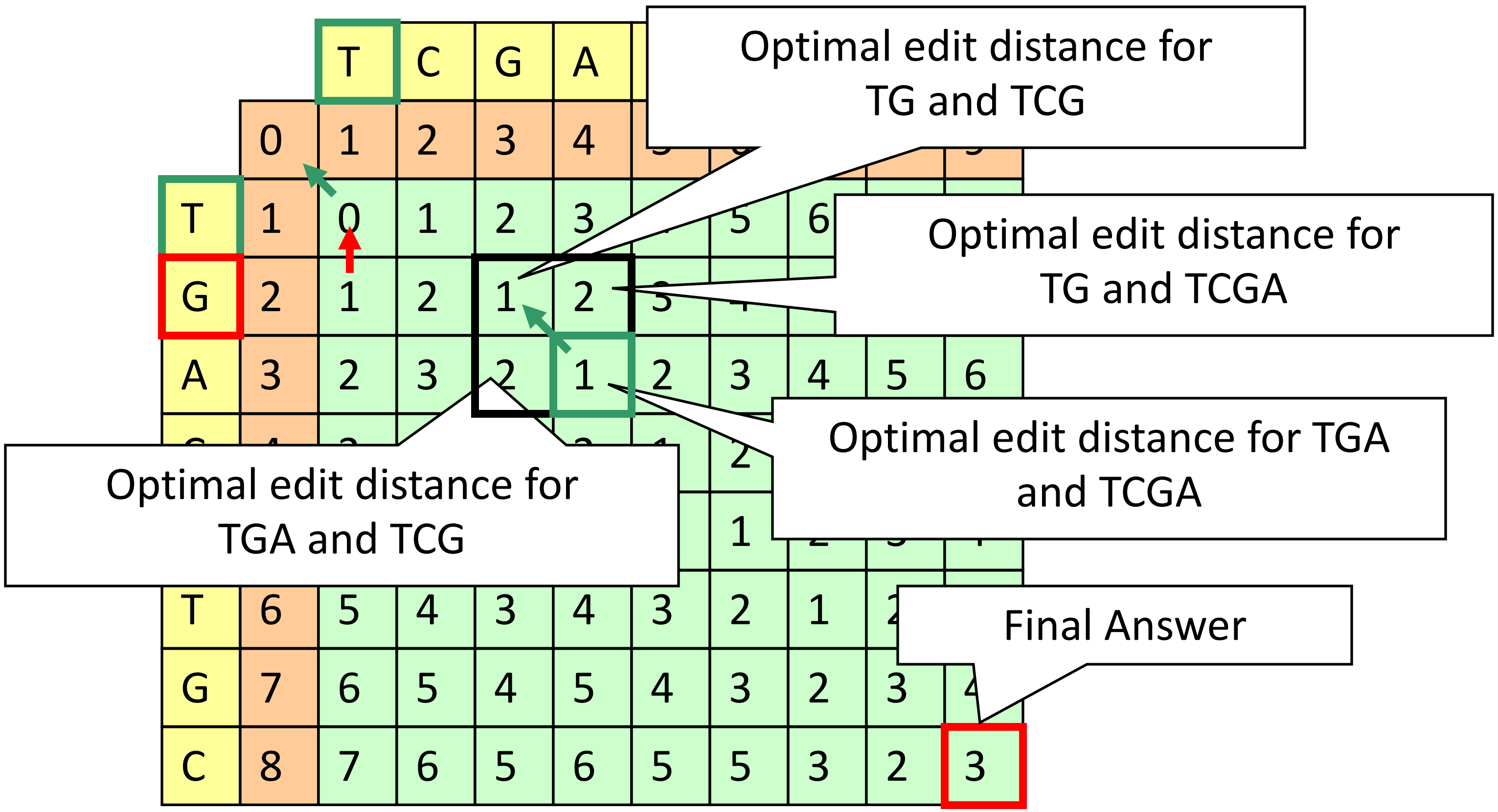
Sub Copy

Notice that this is not exactly the same as the previous sequence but it has the same cost.

This algorithm runs in O(mn) time where m is the length of $string_1$ and n is the length of $string_2$.

		p	a	r	a	l	l	e	l
	0	1 ins	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
i	1 del	1 sub	2 ins	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
m	2 del	2 sub	2 sub	3 ins	4 ins	5 ins	6 ins	7 ins	8 ins
p	3 del	2 copy	3 ins	3 sub	4 ins	5 ins	6 ins	7 ins	8 ins
a	4 del	3 del	2 copy	3 ins	3 copy	4 ins	5 ins	6 ins	7 ins
r	5 del	4 del	3 del	2 copy	3 ins	4 ins	5 ins	6 ins	7 ins
t	6 del	5 del	4 del	3 del	3 sub	4 ins	5 ins	6 ins	7 ins
i	7 del	6 del	5 del	4 del	4 del	4 sub	5 ins	6 ins	7 ins
a	8 del	7 del	6 del	5 del	4 copy	5 del	5 sub	6 sub	7 ins
l	9 del	8 del	7 del	6 del	5 del	4 copy	5 copy	6 ins	6 copy

Edit Distance – Dynamic Programming



Longest Common Subsequence

- Given a sequence $X =< x_1, x_2, \dots, x_m >$, a sequence $Z =< z_1, z_2, \dots, z_m >$ is a subsequence of X if there exists a strictly increasing sequence $< i_1, i_2, \dots, i_k >$ of indices of X such that for all j $= 1,2, \dots, k$, we have $x_{i_j} = z_j$

Example: $X =< A, \boldsymbol{B}, \boldsymbol{C}, B, \boldsymbol{D}, A, \boldsymbol{B} >$ and $Z =< B, C, D, B >$
Indices are $< 2,3,5,7 >$

Common Subsequence

- Given two sequences X and Y, we say that Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.
- Example: x = {A B C B D A B }, y = {B D C A B A}, then {B C} and {A A} are both common subsequences of x and y

Longest Common Subsequence

- In the longest-common-subsequence problem, we are given two sequences
  $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$
  We wish to find a maximum-length common subsequence of X and Y .

- In LCS problem, we are looking for the longest subsequence that is common between two sequences.

Longest Common Subsequence

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

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

$\mathrm{BCBA} = \mathrm{LCS}(x, y)$

functional notation,
but not a function

LCS: Brute Force

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

Analysis

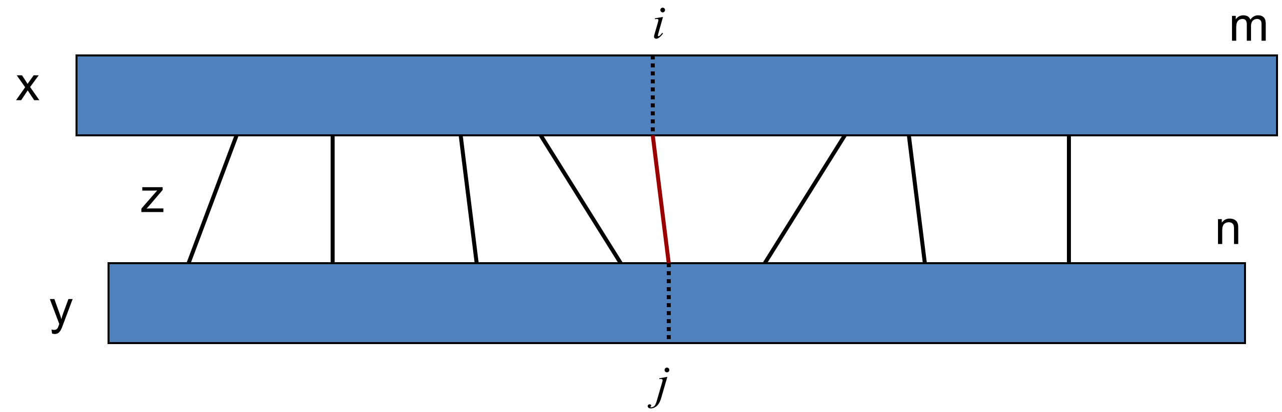
- Checking $= O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$

$= \text{exponential time.}$

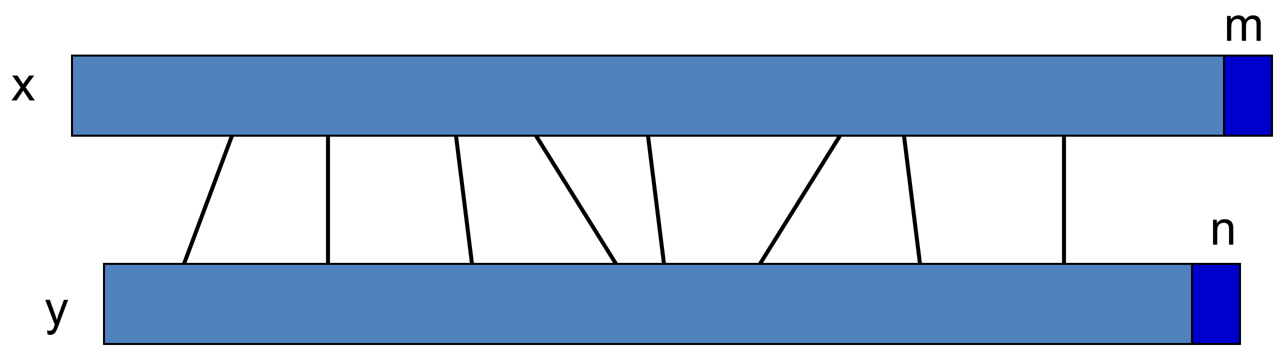
Optimal substructure

- Notice that the LCS problem has *optimal substructure*: parts of the final solution are solutions of subproblems.
  - If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.



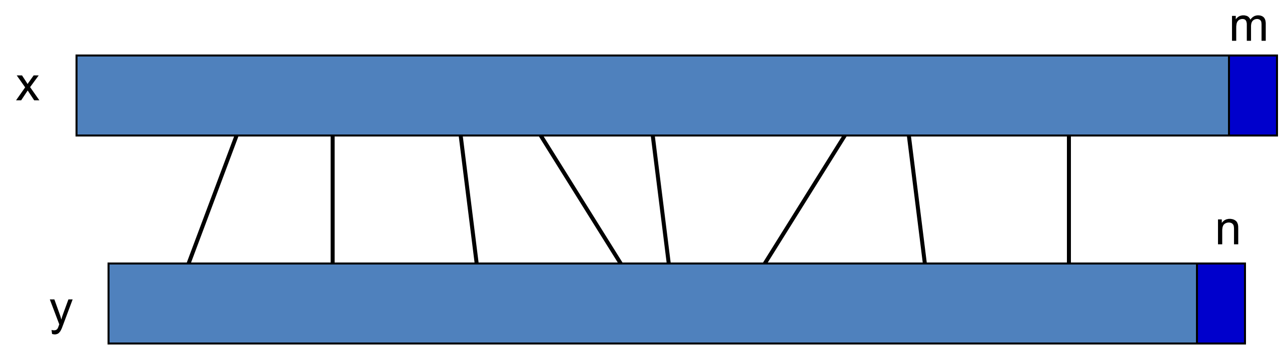
- Subproblems: "find LCS of pairs of *prefixes* of x and y"

Recursive thinking



- Case 1: $x[m]=y[n]$. There is **an** optimal LCS that matches $x[m]$ with $y[n]$. → Find out LCS $(x[1..m-1], y[1..n-1])$
- Case 2: $x[m]\neq y[n]$. At most one of them is in LCS
  - Case 2.1: $x[m]$ *not in LCS* → Find out LCS $(x[1..m-1], y[1..n])$
  - Case 2.2: $y[n]$ *not in LCS* → Find out LCS $(x[1..m], y[1..n-1])$

Recursive thinking



- Case 1: $x[m]=y[n]$
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n-1]) \,||\, x[m]$

Reduce both sequences by 1 char

concatenate

- Case 2: $x[m] \neq y[n]$
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n])$ or

    $LCS(x[1..m], y[1..n-1])$, whichever is longer

Reduce either sequence by 1 char

LCS: Recursive Formulation

Simplification:

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

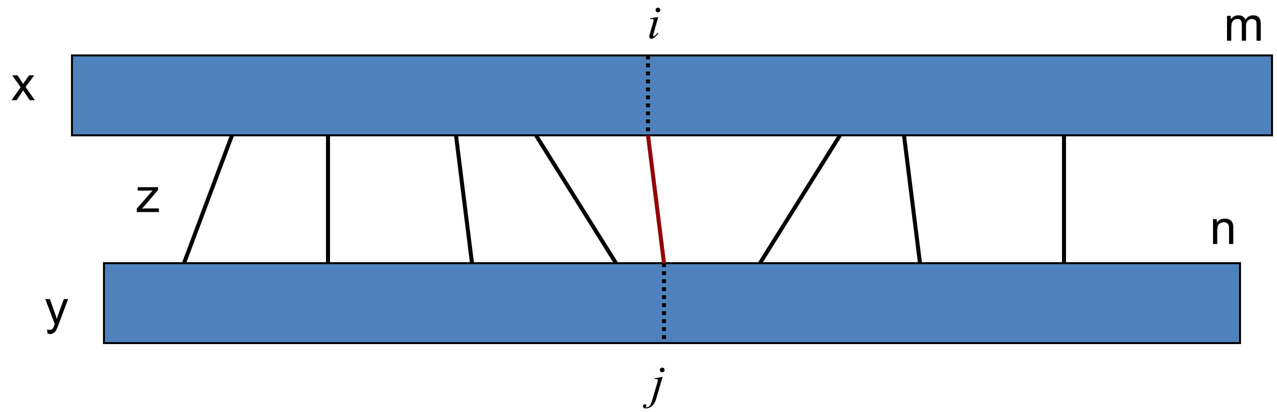
Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.

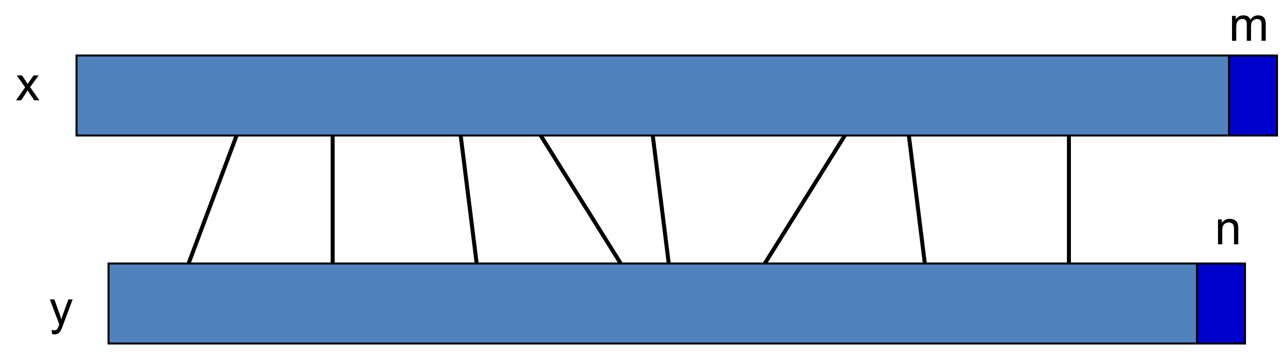
- Define $c[i, j] = |\text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

LCS: Recursive Formulation

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$



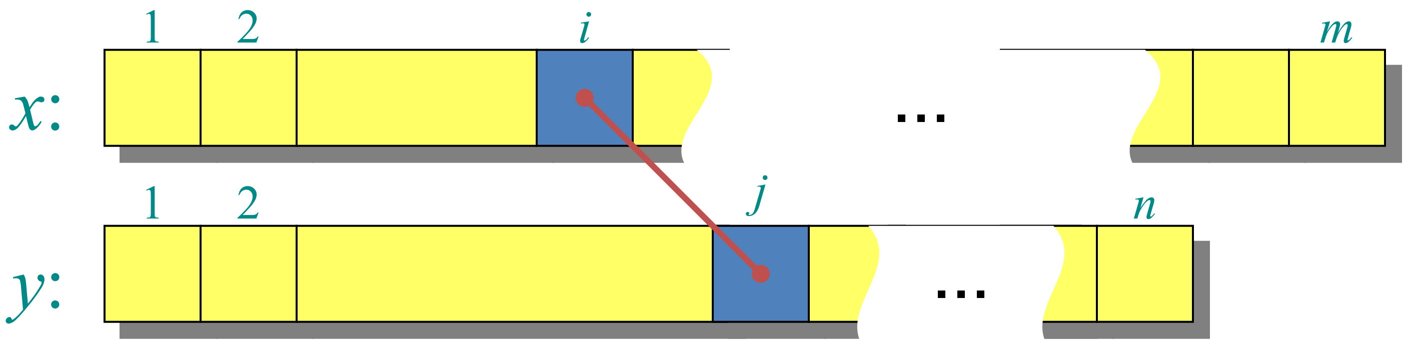
Finding length of LCS



- Let *c[i, j]* be the length of LCS(*x[1..i], y[1..j]*)
  => c[m, n] is the length of LCS(*x, y*)
- If $x[m] = y[n]$
  $c[m, n] = c[m\text{-}1, n\text{-}1] + 1$
- If $x[m] \mathrel{!=} y[n]$
  $c[m, n] = max\ \{ c[m\text{-}1, n], c[m, n\text{-}1] \}$

Generalize: recursive formulation

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$



DP Optimal Substructure

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

LCS: Recursive Formulation

$$
\begin{array}{l}
\mathrm{LCS}(x, y, i, j) \quad \text{// ignoring base cases} \\
\quad \textbf{if } x[i] = y[j] \\
\qquad \textbf{then } c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1 \\
\qquad \textbf{else } c[i, j] \leftarrow \max\{\mathrm{LCS}(x, y, i-1, j), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathrm{LCS}(x, y, i, j-1)\} \\
\quad \textbf{return } c[i, j]
\end{array}
$$

LCS: Recursive Formulation

$\text{LCS}(x, y, i, j)$ // ignoring base cases

if $x[i] = y[j]$

then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j),$

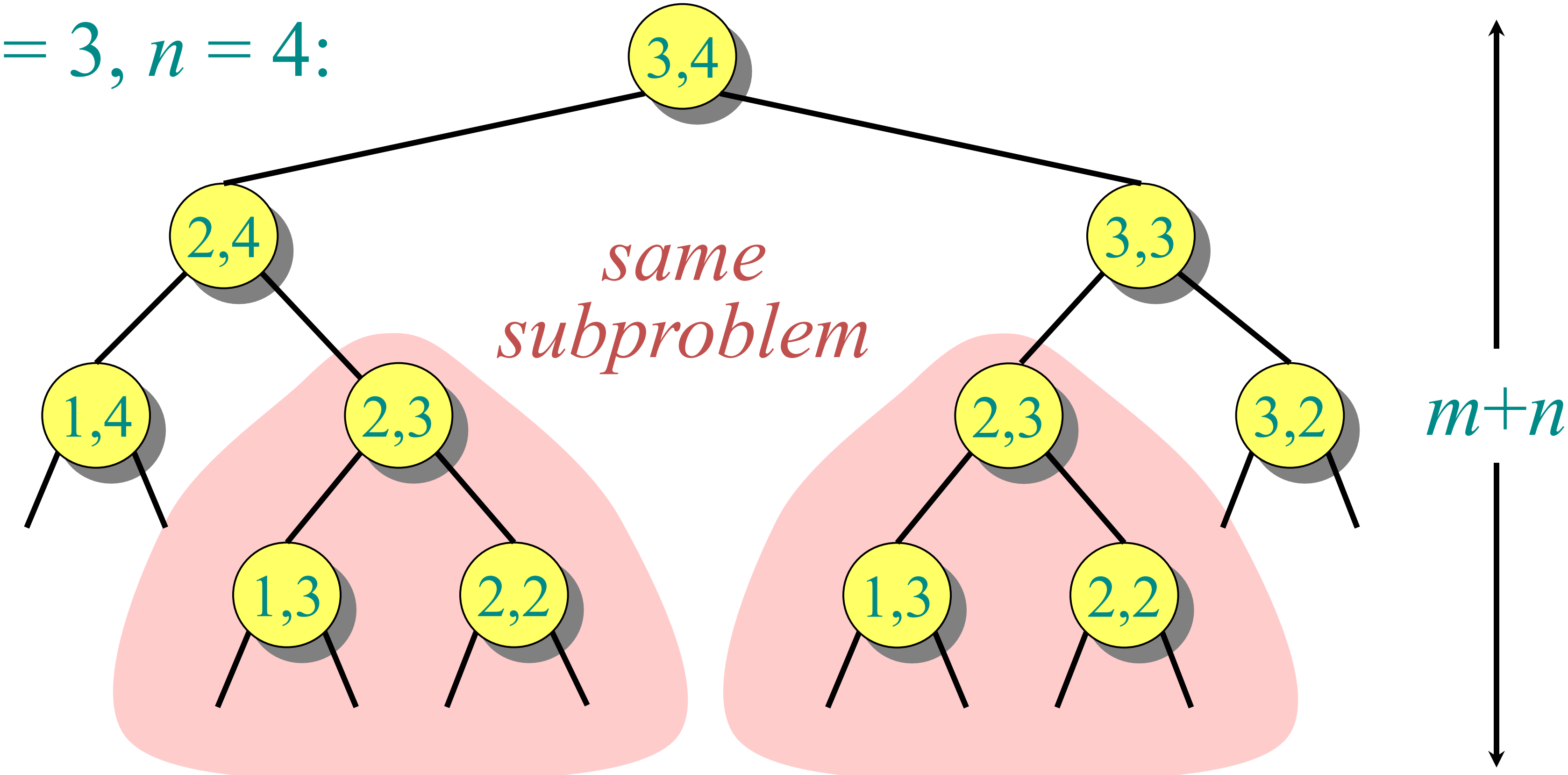
$\text{LCS}(x, y, i, j-1)\}$

return $c[i, j]$

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

Recursion tree

$m = 3,\ n = 4$:



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

DP Hallmarks: Overlapping Subproblems

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

DP Solution: Top-Down Memoization

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

$$
\left.\begin{array}{l}
\mathrm{LCS}(x, y, i, j) \\
\quad \textbf{if } c[i, j] = \mathrm{NIL} \\
\qquad \textbf{then if } x[i] = y[j] \\
\qquad\quad \textbf{then } c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1 \\
\qquad\quad \textbf{else } c[i, j] \leftarrow \max\{\mathrm{LCS}(x, y, i-1, j), \\
\qquad\qquad\qquad\qquad\qquad \mathrm{LCS}(x, y, i, j-1)\}
\end{array}\right\} \textit{same as before}
$$

- Initialization of c[i,j] values to NIL needs to be performed before calling LCS()

DP Solution: Top-Down Memoization

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

LCS(x, y, i, j)
    if $c[i, j] = \text{NIL}$
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$
            else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

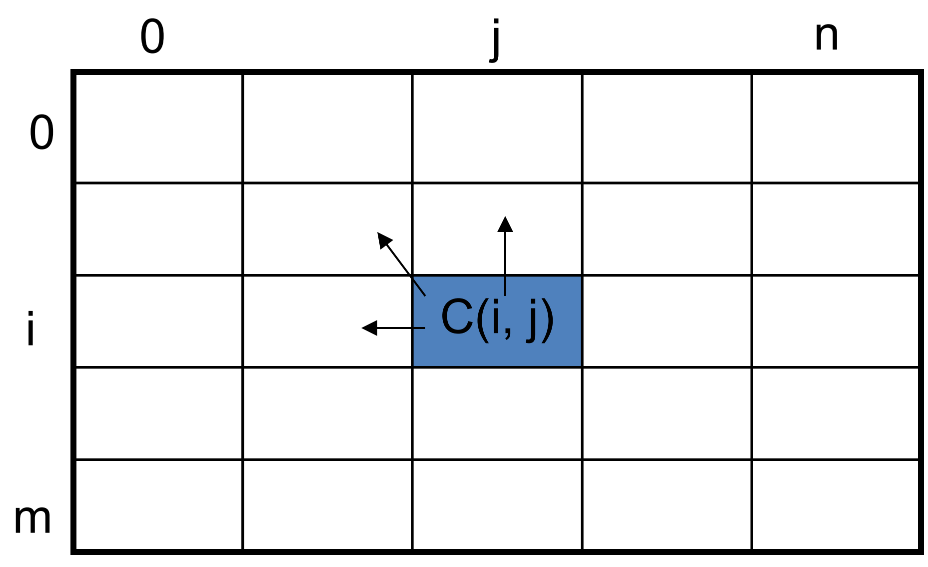
same as before

Time $= \Theta(mn) =$ constant work per table entry.
Space $= \Theta(mn)$.

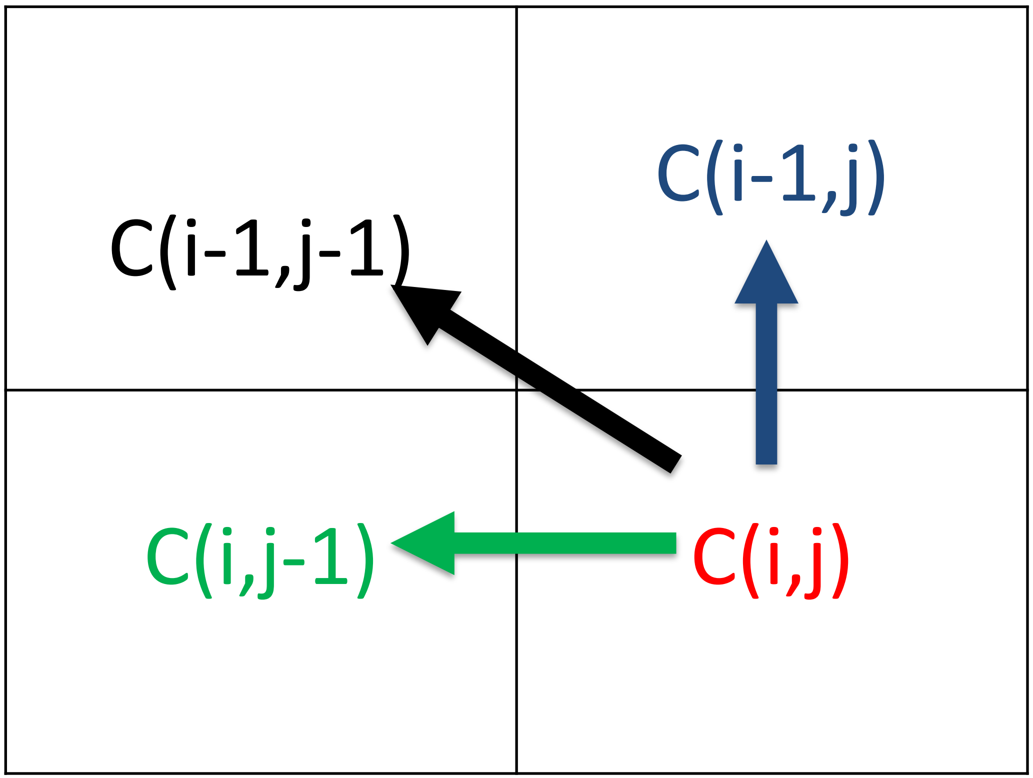
DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: *m * n*

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$



Longest Common Subsequence



if $x_i == y_j$
    $c[i,j] = c[i-1, j-1] + 1$
    $b[i,j] =$ "$\nwarrow$"
elseif $c[i-1,j] \geq c[i,j-1]$
    $c[i,j] = c[i-1,j]$
    $b[i,j] =$ "$\uparrow$"
else $c[i,j] = c[i,j-1]$
    $b[i,j] =$ "$\leftarrow$"

DP Algorithm

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m  c[i,0] = 0    // special case: Y[0]
4. for j = 1 to n   c[0,j] = 0    // special case: X[0]
5. for i = 1 to m             // for all X[i]
6.   for j = 1 to n              // for all Y[j]
7.       if ( X[i] == Y[j])
8.           c[i,j] = c[i-1,j-1] + 1
9.       else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the LCS of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]							
1	**A**							
2	**B**							
3	**C**							
4	**B**							

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for $i = 1$ to m $\quad c[i,0] = 0$
for $j = 1$ to n $\quad c[0,j] = 0$

LCS Example (2)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**	**0**				
2	B	**0**					
3	C	**0**					
4	B	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (3)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)

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

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Y[j]		B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (8)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (9)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Y[j]		B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (11)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (12)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1				

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (13)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if ($X_i == Y_j$)
  $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

if ($X_i == Y_j$)
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- The algorithm just found the *length* of LCS, but not LCS itself.
- How to find the actual LCS?
- For each c[i,j] we know how it was acquired:

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

- A match happens only when the first equation is taken
- So we can start from *c[m,n]* and go backwards, remember *x[i]* whenever *c[i,j] = c[i-1, j-1]+1*.

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finding LCS

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Time for trace back: $O(m+n)$.

i	X[i] \ j	0	1	2	3	4	5
	Y[j]		**B**	**D**	**C**	**A**	**B**
0		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)

DP Solution: Bottom-up

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

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

DP Solution: Bottom-up

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

DP Solution: Bottom-up

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Subset Sum

- For positive integers $w_1, \ldots, w_n$, find $X \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in X} w_i = t$.
- For $i = 1, \ldots, n$ and $j = 1, \ldots, t$ define $A[i, j]$ to be true iff $\exists X \subseteq \{1, \ldots, i\}$ s.t. $\sum_{e \in X} w_e = j$. So we want $A[n, t]$.
- Instance {1 2 3 4 5 6 7 8 9 10 11 12}, t= 50
- True/false: A[0,0],A[2,0],A[3,5],A[6,12],A[4,20],A[12,50]?

A[0,0],A[2,0],A[3,5],A[6,12],A[4,20],A[12,50]?

$$A[i,j] = \begin{cases} \textbf{false} & \text{if } i = 0, j > 0, \\ \textbf{true} & \text{if } i = 0, j = 0, \\ A[i-1, j] & \text{if } i > 0,\ j < w_i, \\ A[i-1, j] \vee A[i-1, j - w_i] & \text{otherwise.} \end{cases}$$

	0	1	…	n
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
…				
t				?

Algorithm $\mathtt{sss}(w_1, \ldots, w_n, t)$

Output: Whether there exists $X \subseteq \{1, \ldots, n\}$ with $\sum_{e \in X} w_e = t$.

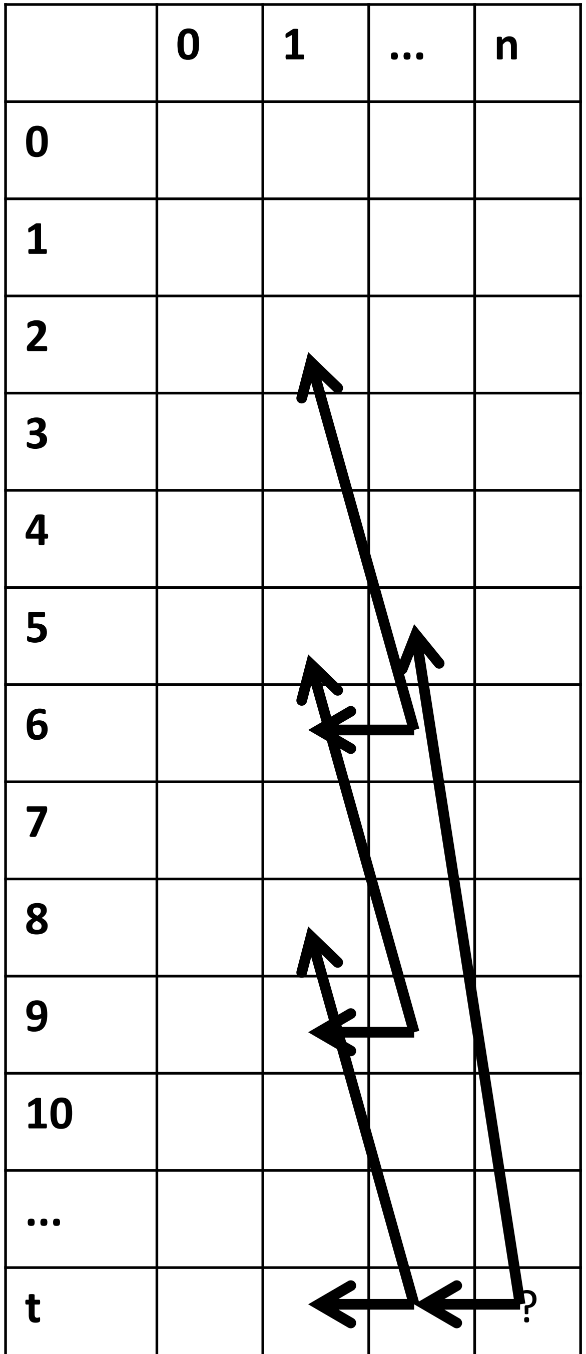
$A[0,0] \leftarrow$ **true**
for $j = 1$ **to** t **do**
    $A[0,j] \leftarrow$ **false**
for $i = 1$ **to** n **do**
    for $j = 1$ **to** t **do**
        if $j < w_i$ **then**
            $A[i,j] \leftarrow A[i-1,j]$
        else
            $A[i,j] \leftarrow A[i-1,j] \vee A[i-1,j-w_i]$
return $A[n,t]$

	0	1	...	n
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
...				
t				?

Algorithm sss$(w_1, \ldots, w_n, t)$

Output: Whether there exists $X \subseteq \{1, \ldots, n\}$ with $\sum_{e \in X} w_e = t$.

```
A[0,0] ← true
for j = 1 to t do
    A[0,j] ← false
for i = 1 to n do
    for j = 1 to t do
        if  j < w_i then
            A[i,j] ← A[i−1,j]
        else
            A[i,j] ← A[i−1,j] ∨ A[i−1,j−w_i]
return A[n,t]
```



Algorithm $\mathrm{sss}(w_1, \ldots, w_n, t)$

Output: Whether there exists $X \subseteq \{1, \ldots, n\}$ with $\sum_{e \in X} w_e = t$.

$A[0,0] \leftarrow$ **true**
for $j = 1$ **to** t **do**
  $A[0,j] \leftarrow$ **false**
for $i = 1$ **to** n **do**
  for $j = 1$ **to** t **do**
    if $j < w_i$ **then**
      $A[i,j] \leftarrow A[i-1,j]$
    else
      $A[i,j] \leftarrow A[i-1,j] \vee A[i-1,j-w_i]$
return $A[n,t]$

- $O(nt)$ time
- Not polynomial: t can be exponentially large since it is represented in binary!

0/1 Knapsack Problem

- Knapsack problem.
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - Goal: fill knapsack so as to maximize total value.

- Ex: { 3, 4 } has value 40.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Greedy: repeatedly add item with maximum ratio v_i / w_i.
- Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming

- Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

  – Case 1: OPT does not select item i.
    - OPT selects best of { 1, 2, …, i-1 } using weight limit w

  – Case 2: OPT selects item i.
    - new weight limit = w – w_i
    - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\quad v_i + OPT(i-1, w - w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

- Knapsack. Fill up an n-by-W array.

```
Input: n, W, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\ \ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

- Q. Knapsack algorithm computes optimal value. What if we want the solution itself?

- A. Do some post-processing
  - Start from the last item i=n, and w=W, Considering M[n, W]:
  - If **`M[i, w] = M[i-1, w]`, `then OPT does not include item i.`**
    - **`We move to consider item M[i-1, w]`**
  - **If $M[i, w] = v_i + M[i-1, w-w_i]$, then OPT includes current item i. → output item i**
    - **We move to consider item $M[i-1, w-w_i]$**

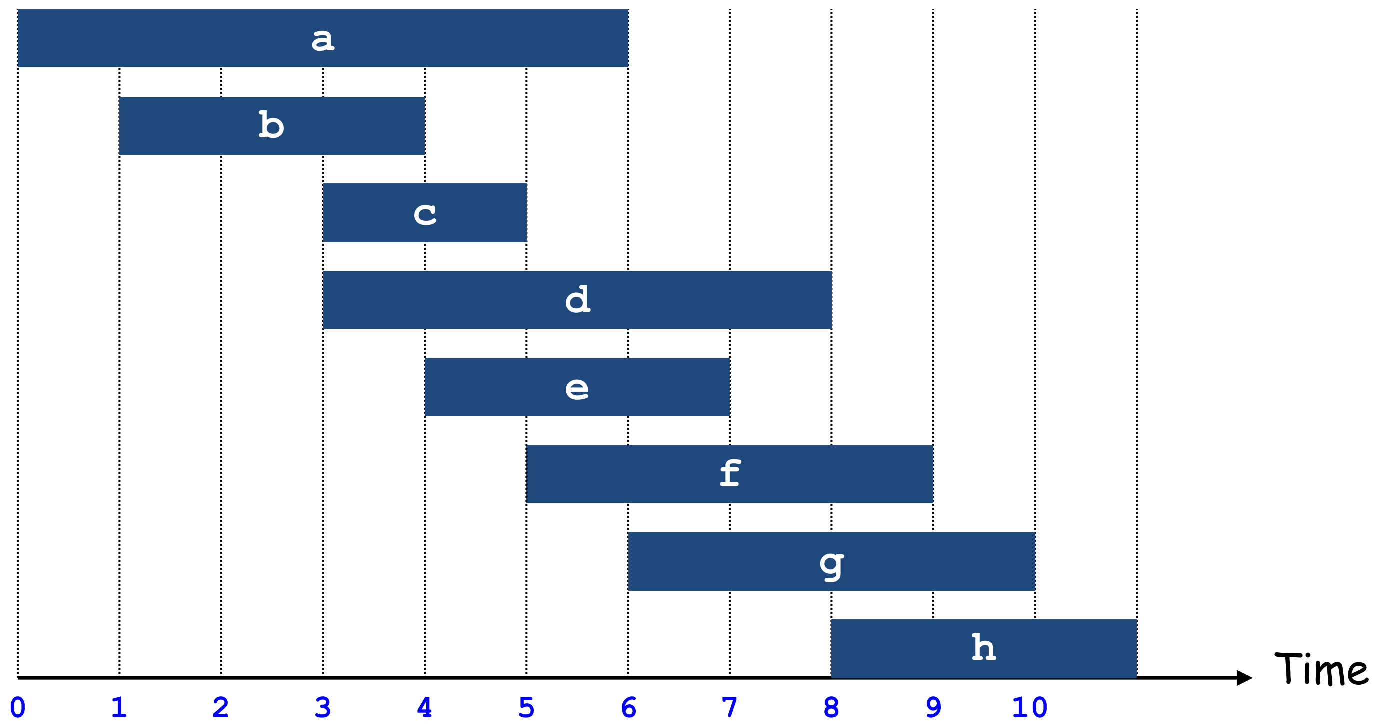
  - # of recursive calls $\leq$ n $\Rightarrow$ O(n).

Knapsack Problem: Running Time

- Running time is $\Theta(n\ W)$.
  - That is NOT polynomial in input size!
  - It is called "Pseudo-polynomial"!
  - Decision version of Knapsack is NP-complete. We will talk about that later in this course.

Weighted Interval Scheduling

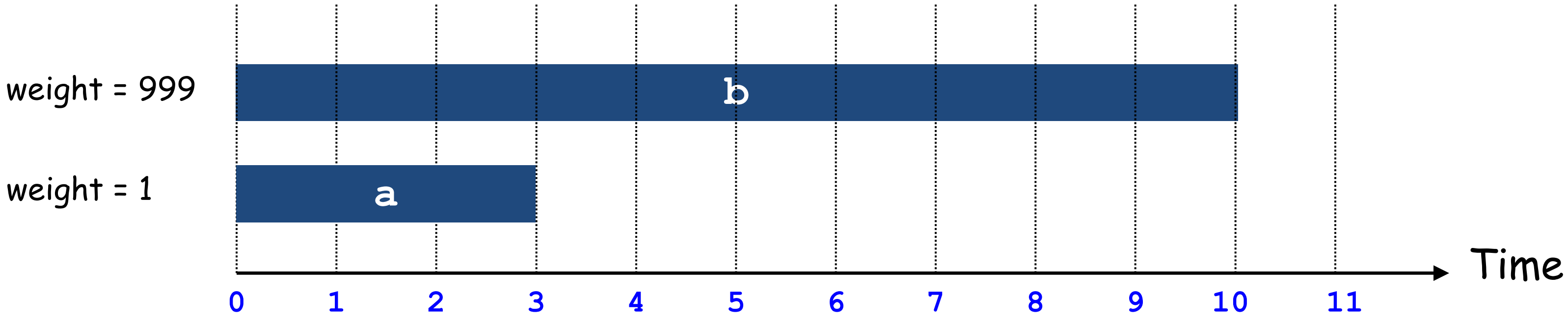
- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs are compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



Unweighted Interval Scheduling Review

- Recall. Greedy algorithm works if all weights are 1.
  - Consider jobs in ascending order of finish time.
  - Add job to subset if it is compatible with previously chosen jobs.

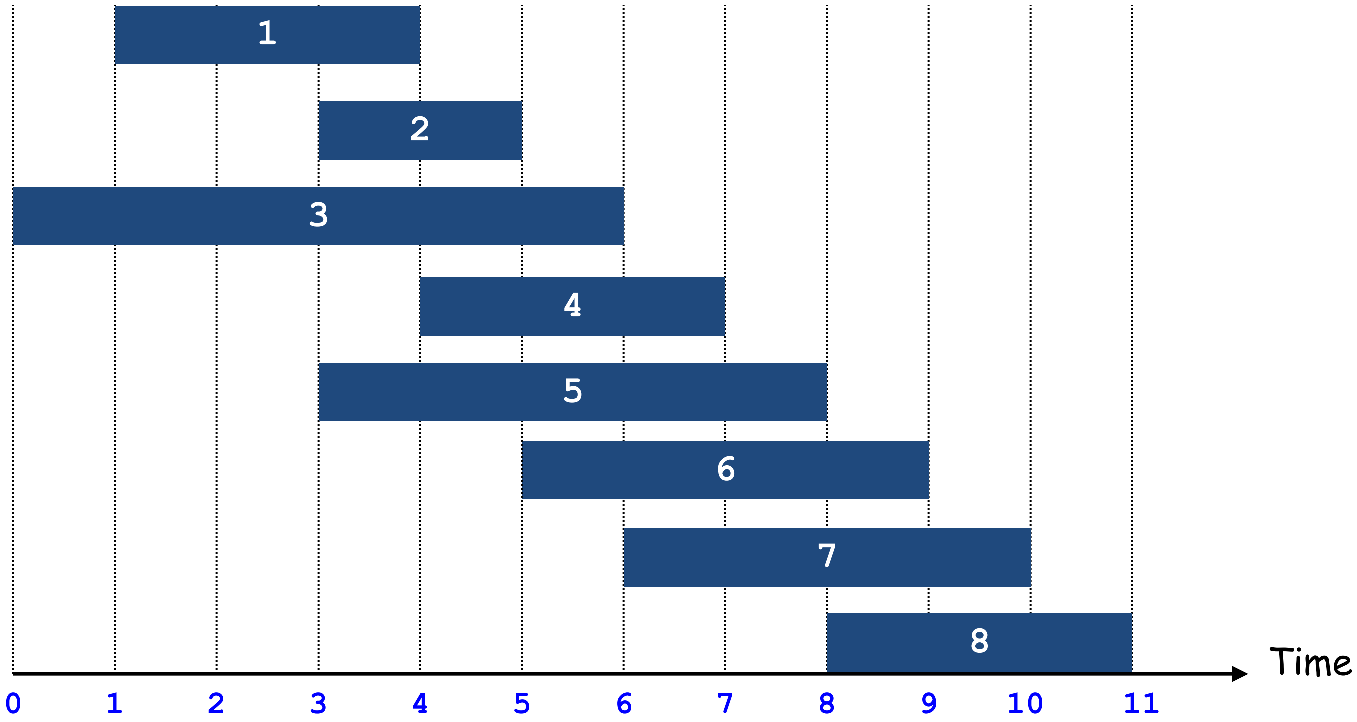
- Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



DP: Binary Choice

- Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

  - Case 1: OPT selects job j.
    - collect profit v_j
    - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)
  - Case 2: OPT does not select job j.
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\ \{\ v_j + OPT(p(j)),\ OPT(j-1)\ \} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling

- Brute force algorithm: Recursive algorithm is exponential because of redundant sub-problems

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Memoization

- Memoization. Store results of each sub-problem in a cache and lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)
                          global array

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

- Claim. Memoized version of algorithm takes O(n log n) time.
  - Sort by finish time: O(n log n).
  - Computing p(·) : O(n log n) via sorting by start time.

  - `M-Compute-Opt(j)`: each invocation takes O(1) time and either
    - (i) returns an existing value `M[j]`
    - (ii) fills in one new entry `M[j]` and makes two recursive calls

  - Progress measure Φ = # nonempty entries of `M[]`.
    - initially $\Phi = 0$, throughout $\Phi \leq n$.
    - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.

  - Overall running time of `M-Compute-Opt(n)` is O(n). ▪

- **Remark. O(n) if jobs are pre-sorted by start and finish times.**

•Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

•A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

– # of recursive calls $\leq$ n $\Rightarrow$ O(n).

Weighted Interval Scheduling: Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

DP Paradigm Review

- Overlapping sub-problem = sub-problem whose results can be reused several times.
- DP: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.