CS 381 – Spring 2019

Week 6

Dynamic programming (DP)

Break a problem into a series of overlapping subproblems, and use the corresponding recurrence to build up solutions to larger and larger subproblems.

- Overlapping Subproblems

- Optimal Substructure (Optimality Conditions)

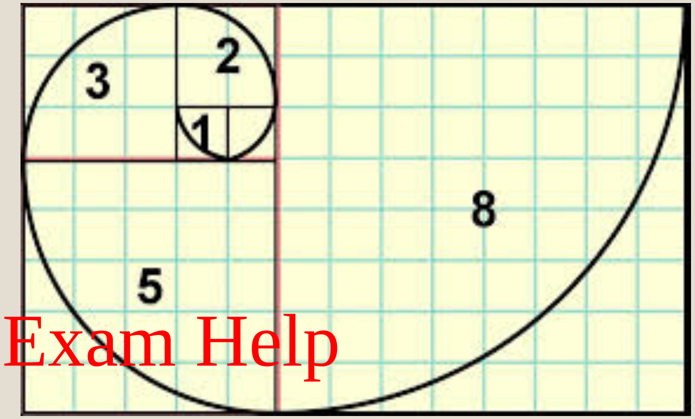
  An optimal solution to a problem (instance) contains optimal solutions to subproblems.

DP typically solves optimization problems by combining optimum solutions for subproblems.

Steps taken when designing a DP algorithm

1. Characterize the structure of an optimal solution
2. **Recursively define the value of an optimal solution in terms of optimum subsolutions**
3. Compute the subsolution entries (never re-compute).
4. Construct an optimal solution from the computed entries and other information.

Fibonacci Sequence



- $F_0 = 0$, $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

- 0, 1, 1, 2, 3, 5, 8, 13, 21, …

- The most natural approach is Divide-and-Conquer
- How efficient is a D&Q algorithm?

$$2T(n-2) + c \leq T(n) = T(n-1) + T(n-2) + c \leq 2T(n-1) + c$$

- Why is it exponential? Is there a better Solution?

Review: Fibonacci numbers F(n) = F(n-1) + F(n-2)

Recursion:

```
rec-fib(n):
    base cases…
    return rec-fib(n-1)+rec-fib(n-2)
```

DP Top-down (Memorization):


```
mem-fib(n):
    initialize array M[1..n]
    if M[n] == NIL
       M[n] = mem-fib(n-1)+mem-fib(n-2)
    return M[n]
```

DP Bottom-up:

```
dp-fib(n):
    initialize array M[1..n]
    for I = 3 to n
        M[i] = M[i-1]+M[i-2]
    return M[n]
```

Problem 1: Non Adjacent Selection (NAS)

S is an array of size **n** (positive integers in arbitrary order)

Select entries in S so that

i. the sum of the selected entries is a maximum

ii. no two selected entries are adjacent in array S

Examples

[14, 6, 33, 1, 2, 8]

[1, 4, 5, 4]

[15, 14, 10, 17, 10]

Recurrence for an Efficient DP Algorithm

$OPT(n) = \max\{OPT(n-1), OPT(n-2) + S[n]\}$

$OPT[1] = S[1]$

$OPT[2] = \max\{OPT(1), S[2]\}$

$OPT[k] = \max\{OPT(k-1), OPT(k-2) + S[k]\}$, $3 \le k \le n$

Compute entries of array OPT in O(n) time in one left to right scan (at position k, look at k-1 and k-2)

$S = [14, 6, 8, 9, 7, 2]$

Start at n scanning left and determine elements in set T

T={}; k=n

while k ≥ 1

if OPT[k-1] ≥ OPT[k-2] + S[k]

then k = k-1 // S[k] is not selected

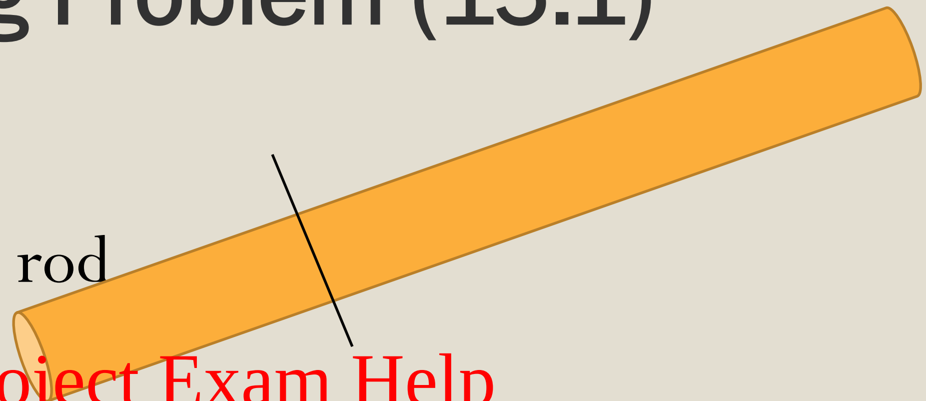
else add index k to set T; k=k-2

Return T

Generating the elements in the solution costs O(n) time

Note: Revisit the O(n) time iterative solution to maximum subarray problem (it is DP)

Problem 2: Rod Cutting Problem (15.1)



- Input is
  - **n**, the length of a steel rod
  - an array **p** of size n

The rod is cut into shorter rods.

- A rod of length k is sold for profit p[k], 1≤k ≤n.

Cut the rod into pieces that maximize the total profit

- No cuts can be undone
- Making a cut is "free"

Example

n=5

	1	2	3	4	5
p	3	5	10	12	14

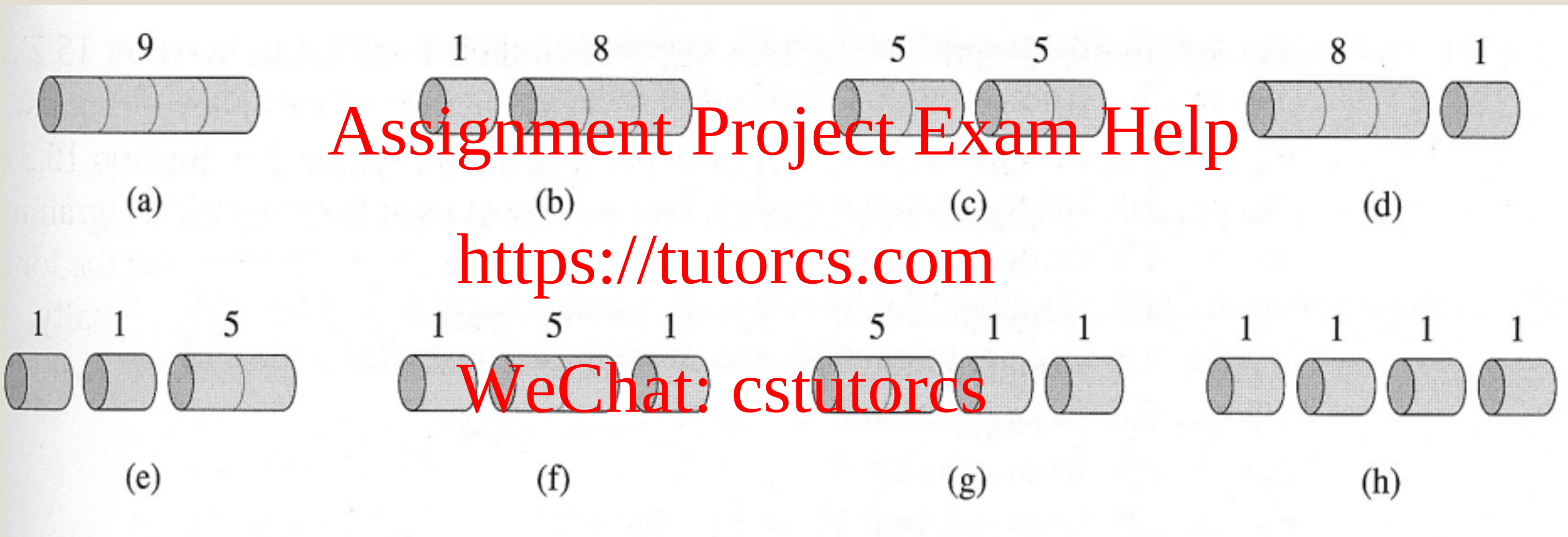
Making no cut has a profit of 14

Making one cut creating pieces of length 1 and 4

- profit of 3 + 3 +10 = 16

Profit of 16 is possible

There are 2^{n-1} ways to cut a rod of length n.



$n = 4$

$p = [\,1, 5, 8, 9]$

Can we use DP?

Does the Principle of Optimality hold?

Can we use DP?

Does the Principle of Optimality hold?

Assume we make an optimal cut creating one piece of length k and one of length n-k.

Then, both pieces are cut in an optimal way. Why?

Otherwise we don't have an optimal solution.

Can we use DP?

Does the Principle of Optimality hold?

Assume we make an optimal cut creating one piece of length k and one of length n-k.

Then, both pieces are cut in an optimal way. Why? Otherwise we don't have an optimal solution.

How about overlapping subproblems?

How to use DP?

- If we make an optimal cut creating a piece of length k and one of length n-k, both pieces are cut in a optimal way.

- Let opt(n) be the profit of an optimal solution for a rod of length **n**. Then,

$$\mathbf{opt(n) = max\ \{p[n],\ opt(1)+ opt(n\text{-}1),}$$
$$\mathbf{opt(2)+opt(n\text{-}2),}$$
$$\mathbf{\ldots}$$
$$\mathbf{opt(n\text{-}2)+opt(2),}$$
$$\mathbf{opt(n\text{-}1)+opt(1)\}}$$

Another way to look at the cuts …

$$\mathbf{opt(n) = max}_{1 \le i \le n} \{\mathbf{p[i] + opt(n\text{-}i)}\}$$

Another way to look at the cuts …

$$\mathbf{opt(n) = max}_{1 \le i \le n} \{\mathbf{p[i] + opt(n\text{-}i)}\}$$

If a piece of length i is the leftmost piece cut from the rod, it generates a profit of p[i].

The remaining rod of length n-i is cut in an optimal way maximizing the profit.

New recurrence:

$$\mathbf{opt(j) = max}_{1 \le i \le j} \{\mathbf{p[i] + opt(j\text{-}i)}\} \textbf{ for } \mathbf{1 \le j \le n}$$

$$r(j) = \max_{1 \le i \le j} \{p[i] + r(j-i)\} \text{ for } 1 \le j \le n \quad (r \text{ stands for opt})$$

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```


Total time is $O(n^2)$ and space is $O(n)$.

See page 366 for more details.

$r(j) = \max_{1\le i\le j} \{p[i] + r(j-i)\}$ for $1\le j\le n$ (r stands for opt)

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```


Profit of a piece of length i

Optimum solution for a rod of length j-i

Total time is $O(n^2)$ and space is $O(n)$.

See page 366 for more details.

Example

n=5

	1	2	3	4	5
p	**3**	**5**	**10**	**12**	**14**
opt (r)	**3**	**6**	**10**	**13**	**16**

How to record where the cuts are made?

Use an arrays to record which index **k** resulted in the maximum for opt(j)

- Needs some adjusting of indices to generate cut positions

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0 . . n] and s[0 . . n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```



```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n − s[n]
```

Dynamic Programming Problems

1) Non-Adjacent Selection
2) Rod Cutting
3) Weighted Selection

4) Longest Common Subsequence
5) Sequence Alignment
6) Matrix Chain Multiplication
7) 0/1 Knapsack
8) Coins in a Line

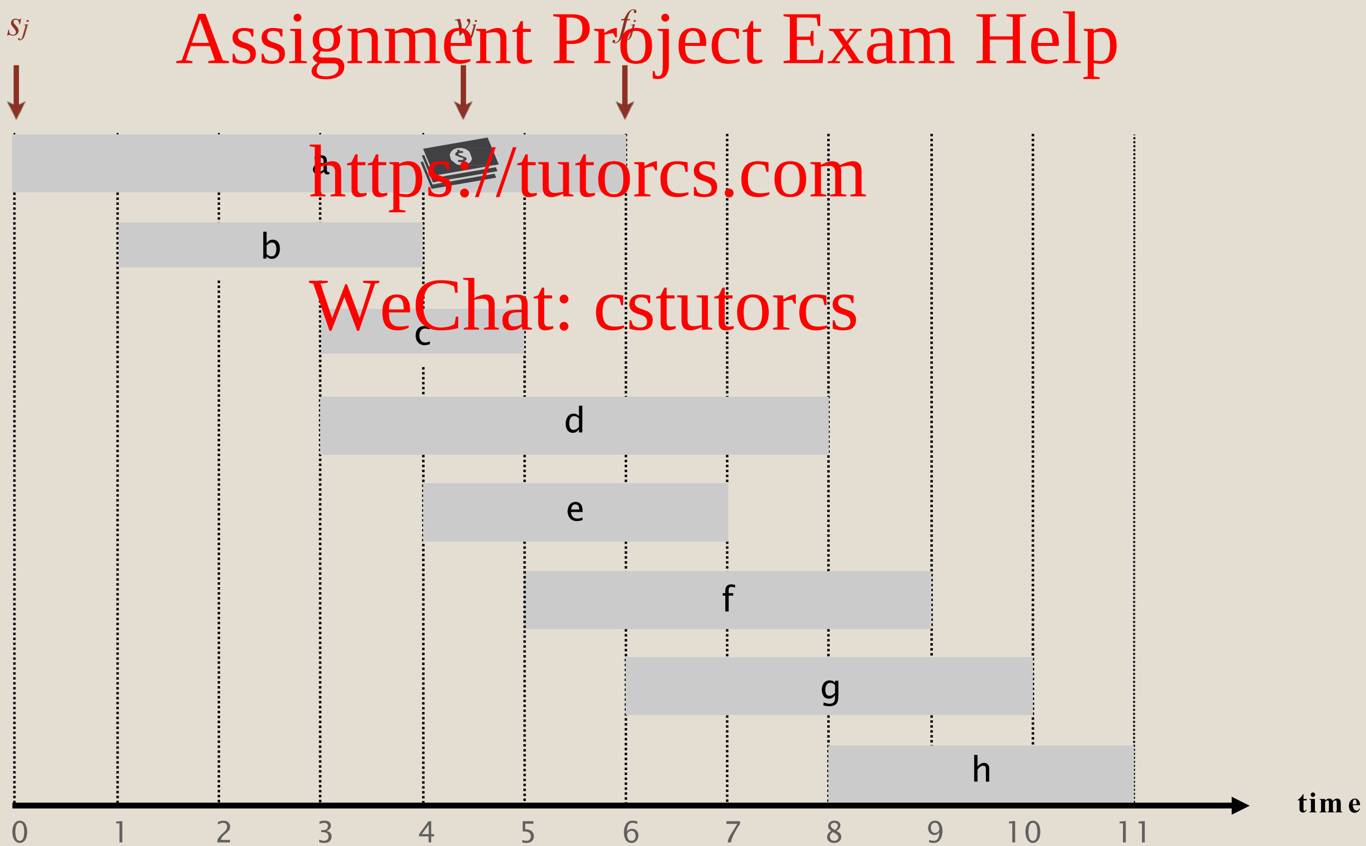
Steps taken when designing a DP algorithm

1. Characterize the structure of an optimal solution
2. **Recursively define the value of an optimal solution in terms of optimum subsolutions**
3. Compute the subsolution entries (never re-compute).
4. Construct an optimal solution from the computed entries and other information.

Problem 3: 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.

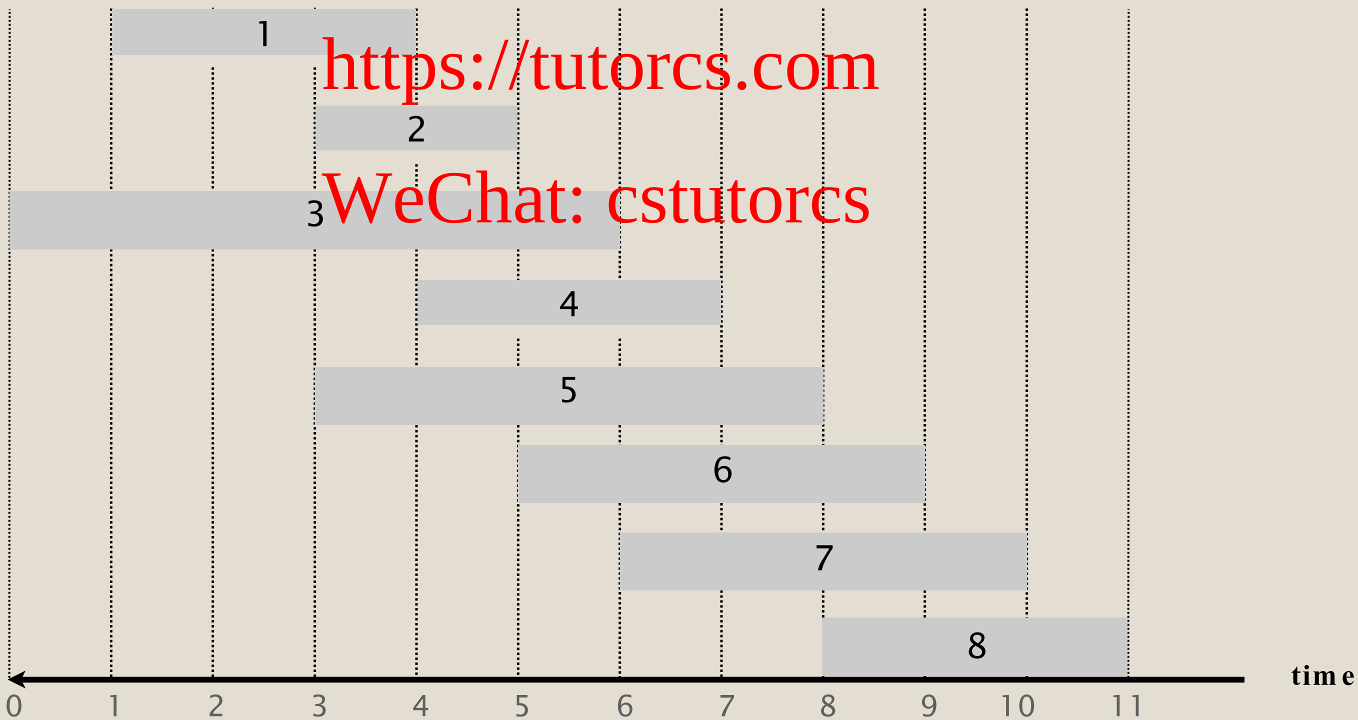


Weighted interval scheduling

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



Dynamic programming: Binary Choice

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

Goal. $OPT(n)$ = value of optimal solution to the original problem.

Case 1. $OPT(j)$ 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 $1, 2, ..., p(j)$.

optimal substructure property
(proof via exchange argument)

Case 2. $OPT(j)$ does not select job j.

- Must include optimal solution to problem consisting of remaining jobs $1, 2, ..., j - 1$.

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

Weighted interval scheduling: brute force

BRUTE-FORCE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF $j = 0$

    RETURN 0.

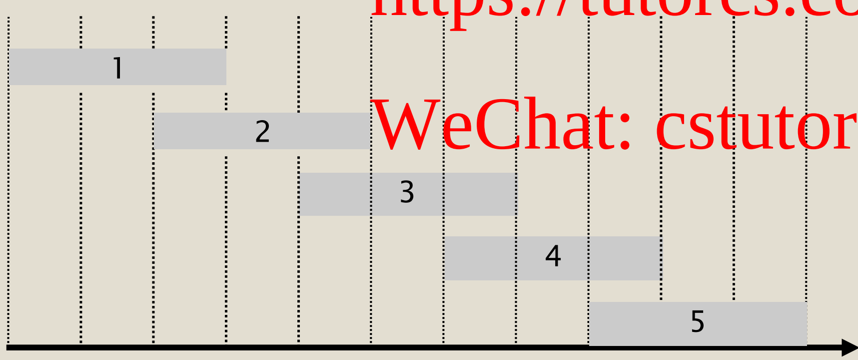
ELSE

    RETURN max { v_j + COMPUTE-OPT($p[j]$), COMPUTE-OPT($j - 1$) }.

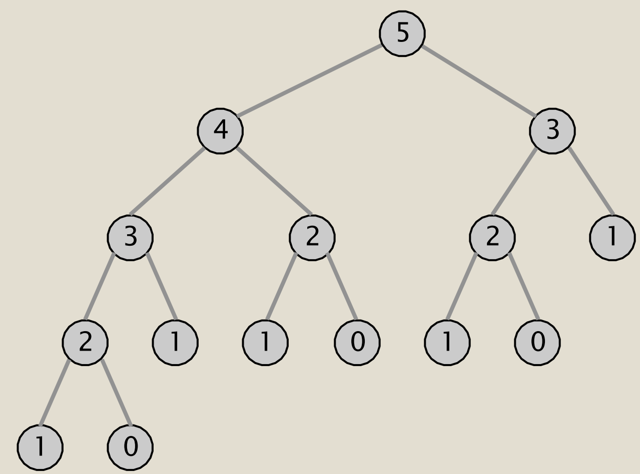
Weighted interval scheduling: Brute Force

Observation. Recursive algorithm is spectacularly slow because of overlapping subproblems $\Rightarrow$ exponential-time algorithm.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



recursion tree

Top-down dynamic programming (memoization). Cache result of each subproblem; lookup as needed.

TOP-DOWN $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0.$ ← global array M[]

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF $M[j]$ = *uninitialized*

$M[j] \leftarrow \max\{ v_j +$ M-COMPUTE-OPT$(p[j])$, M-COMPUTE-OPT$(j-1)\}$.

RETURN $M[j]$.

Weighted interval scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Compute the vector $p[\,]$: $O(n \log n)$.

- 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 among $M[1..n]$.
  - 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 presorted by start and finish times.

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass by calling FIND-SOLUTION(n).

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass by calling FIND-SOLUTION(n).

```
FIND-SOLUTION(j)
____________________________________________
IF j = 0
   RETURN ∅
ELSE IF (v_j + M[p[j]] > M[j–1])
   RETURN {j} ∪ FIND-SOLUTION(p[j]).
ELSE
   RETURN FIND-SOLUTION(j–1).
```


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

Weighted interval scheduling: bottom-up dynamic programming

Bottom-up dynamic programming. Unwind recursion.

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \le f_2 \le \ldots \le f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

previously computed values

FOR $j = 1$ TO n

$\quad M[j] \leftarrow \max \{ v_j + M[p[j]],\ M[j-1] \}$.

Running time. The bottom-up version takes $O(n \log n)$ time.

Weighted Interval Scheduling

Weighted interval scheduling DP algorithm has *O(n log n)* running time:

- Sort by finish time: *O(n log n)* time

- Computing **P** : *O(n log n)* time.

- Compute **M** entries: *O(n)*
- Backtrack for finding intervals to select: *O(n)*

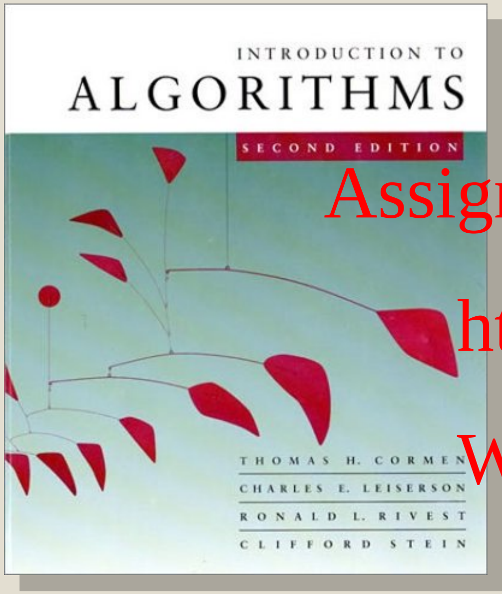
DP: Problems on strings

String problems have numerous applications; an important application area is computational biology/bioinformatics.

- the alphabet is generally small

- strings can be very long and there may be noise in the string (approximate string matching)

- algorithms need to be fast



Dynamic Programming

- Longest Common Subsequence
- Optimal substructure
- Overlapping subproblems
- Sequence alignment (Edit Dist.)

Based on slides by Erik D. Demaine, Charles E. Leiserson and Kevin Wayne

Longest Common Subsequence (LCS)

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

Dynamic programming

Example: ***Longest Common Subsequence (LCS)***

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

"a" *not* "the"

Dynamic programming

Example: ***Longest Common Subsequence (LCS)***

- 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

Dynamic programming

Example: ***Longest Common Subsequence (LCS)***

- 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

$\left.\right\}$ BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

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

Brute-force LCS algorithm

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 $= O(n2^m)$

$= $ exponential time.

Towards a better algorithm

Simplification:

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

Towards a better algorithm

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 $|s|$.

Towards a better algorithm

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

Recursive formulation

Theorem.

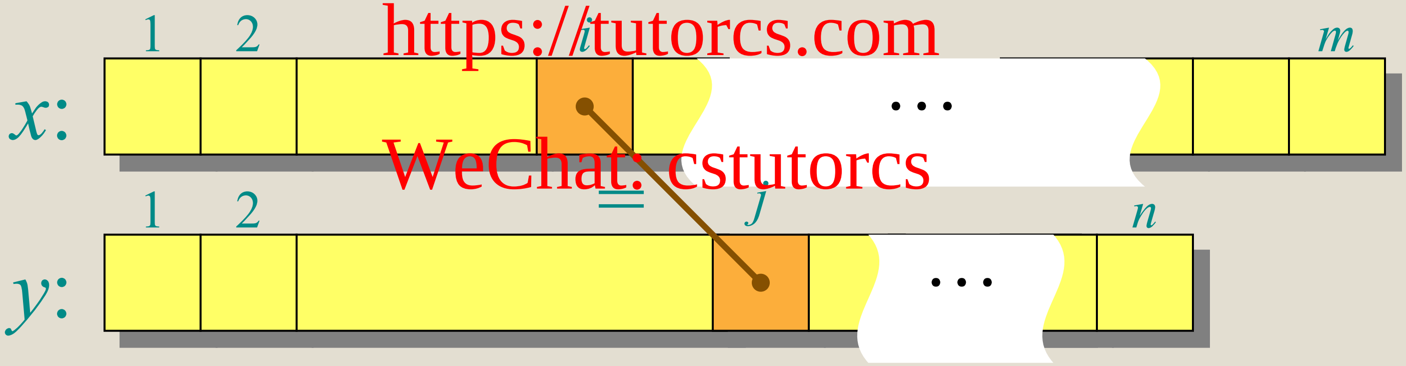
$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Recursive formulation

Theorem.

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

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

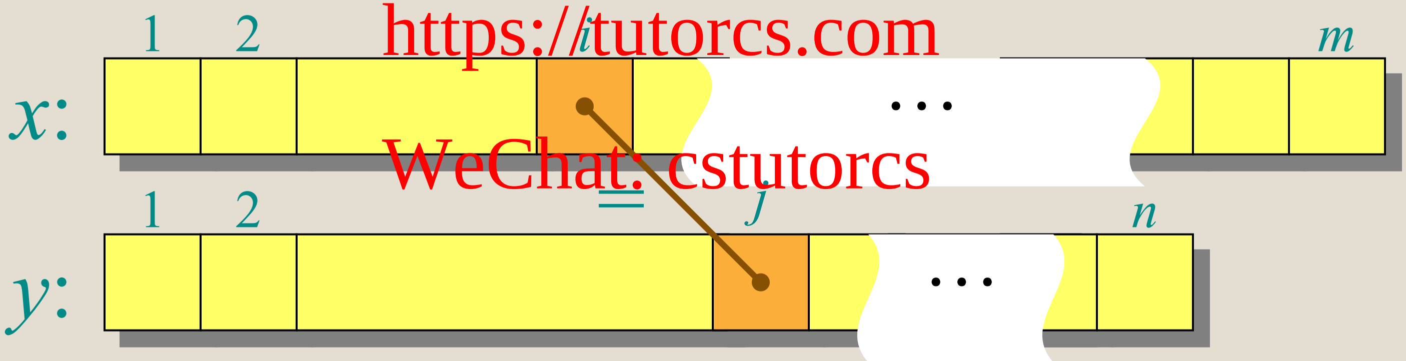


Recursive formulation

Theorem.

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

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



Let $z[1 \,.\,.\, k] = \mathrm{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.

Proof (continued)

Claim: $z[1 \,.\,.\, k-1] = \text{LCS}(x[1 \,.\,.\, i-1], y[1 \,.\,.\, j-1])$.

Suppose w is a longer CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1 \,.\,.\, i]$ and $y[1 \,.\,.\, j]$ with $|w \,||\, z[k]| > k$. Contradiction, proving the claim.

Proof (continued)

Claim: $z[1 \,.\,.\, k-1] = \text{LCS}(x[1 \,.\,.\, i-1], y[1 \,.\,.\, j-1])$.

Suppose w is a longer CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,\|\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1 \,.\,.\, i]$ and $y[1 \,.\,.\, j]$ with $|w \,\|\, z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k-1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar. □

Dynamic-programming hallmark #1

Optimal substructure

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

Dynamic-programming hallmark #1

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.

Recursive algorithm for LCS

LCS(x, y, i, j)
if $x[i] = y[j]$
then *return* LCS$(x, y, i-1, j-1) + 1$
else *return* $\max\{$LCS$(x, y, i-1, j)$,
LCS$(x, y, i, j-1)\}$

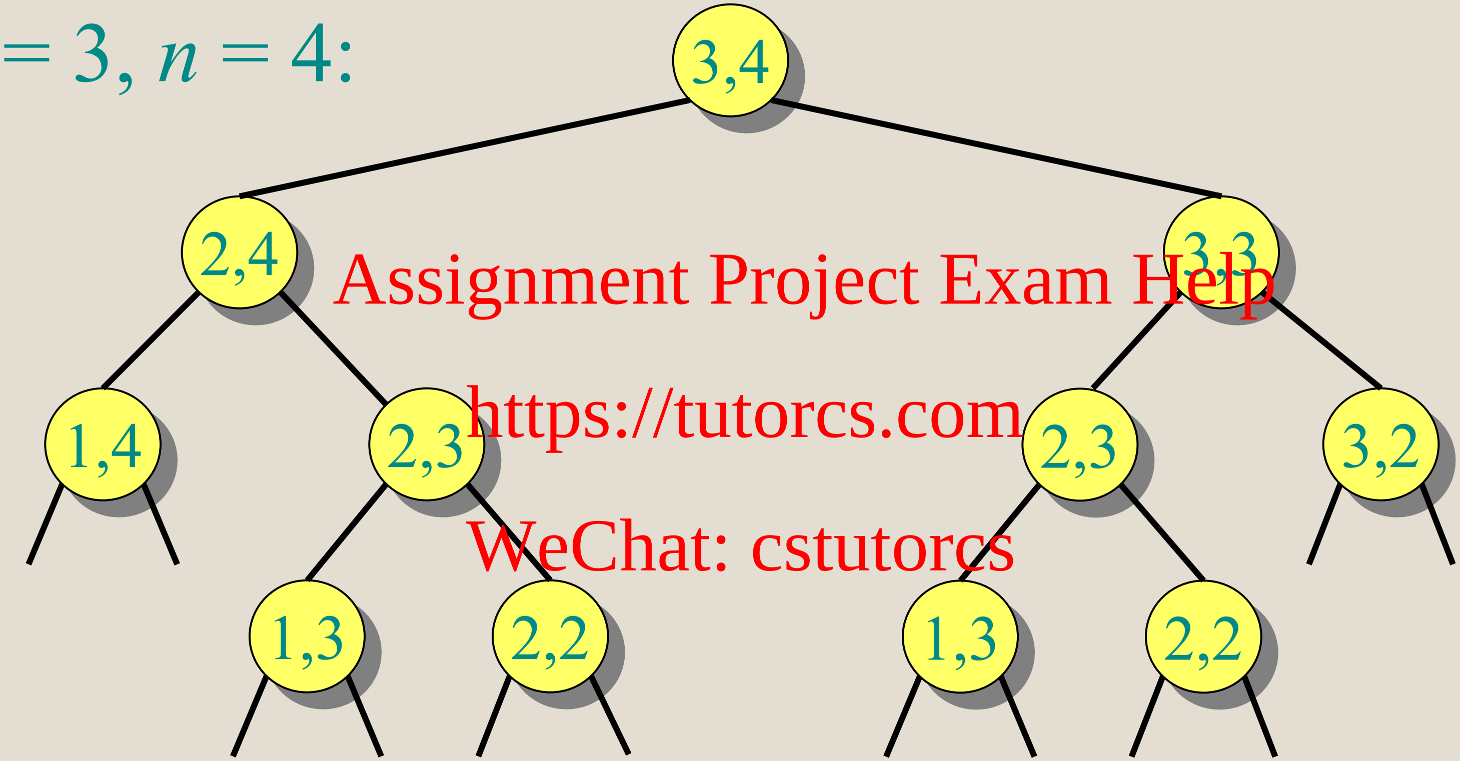
Recursive algorithm for LCS

LCS(x, y, i, j)
    if $x[i] = y[j]$
        then *return* LCS$(x, y, i-1, j-1) + 1$
        else *return* $\max\{$LCS$(x, y, i-1, j)$,
                        LCS$(x, y, i, j-1)\}$

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

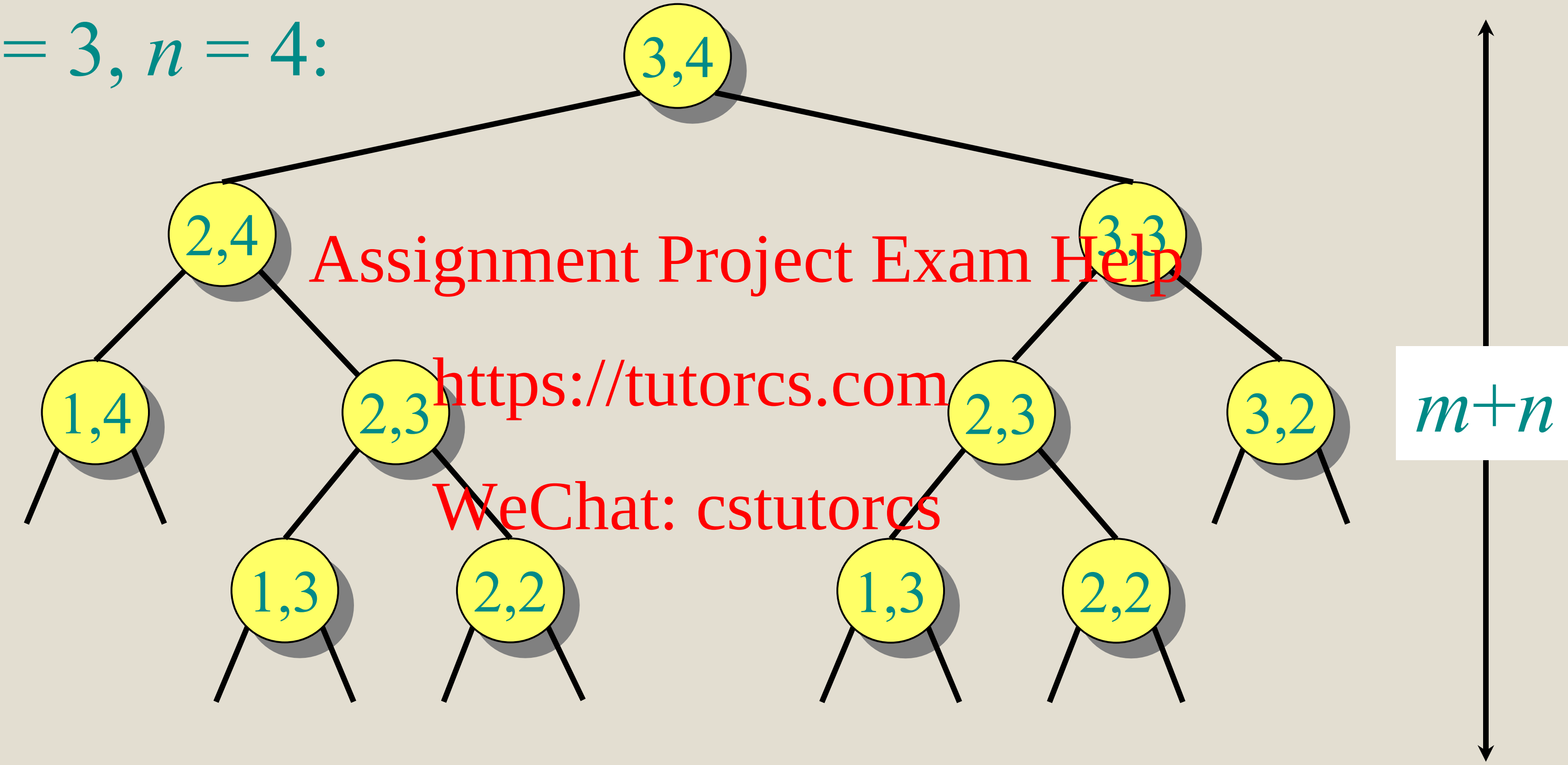
Recursion tree

$m = 3, n = 4$:



Recursion tree

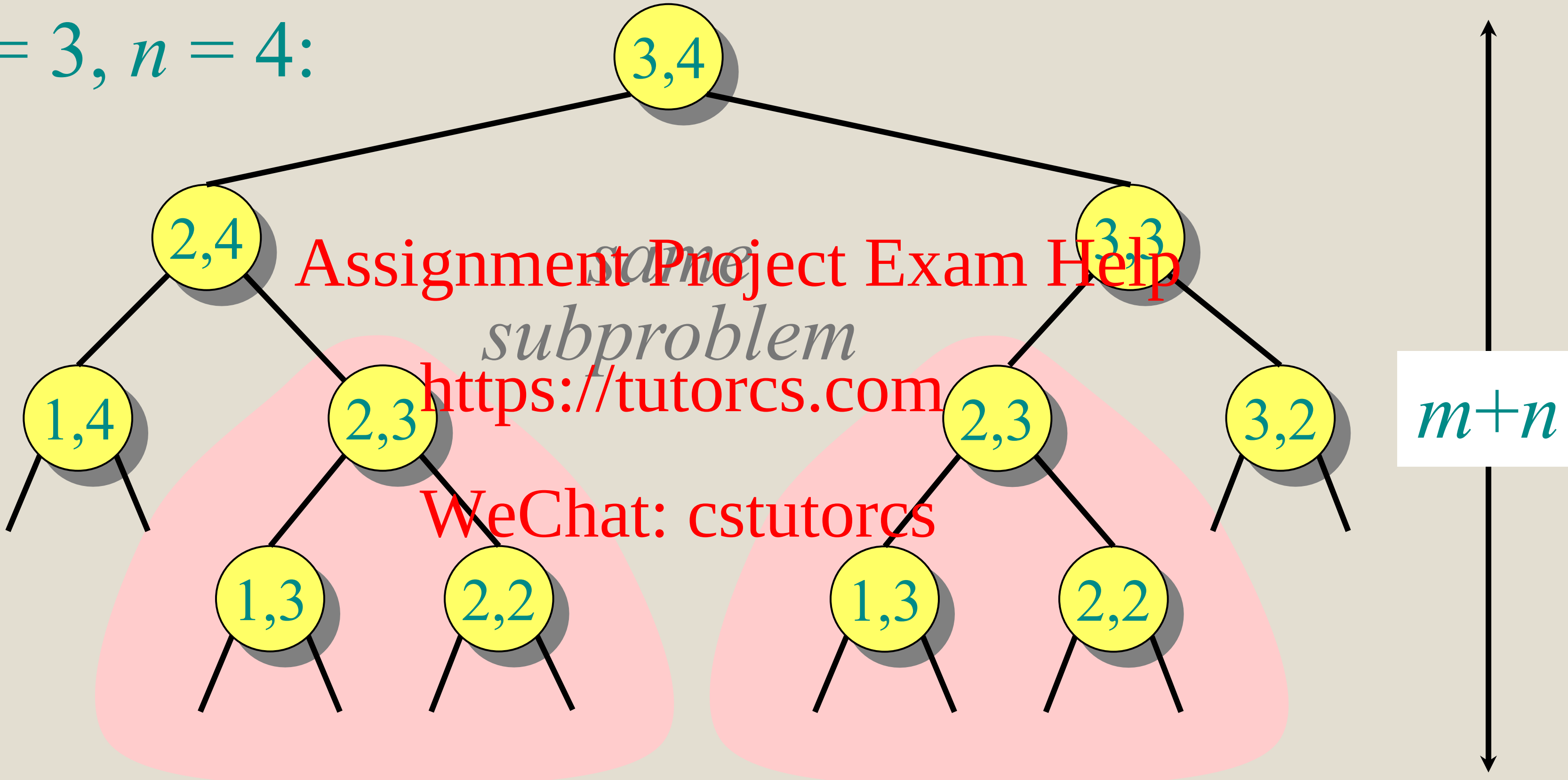
$m = 3, n = 4$:



Height $= m + n \Rightarrow$ potentially work exponential.

Recursion tree

$m = 3, n = 4$:



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