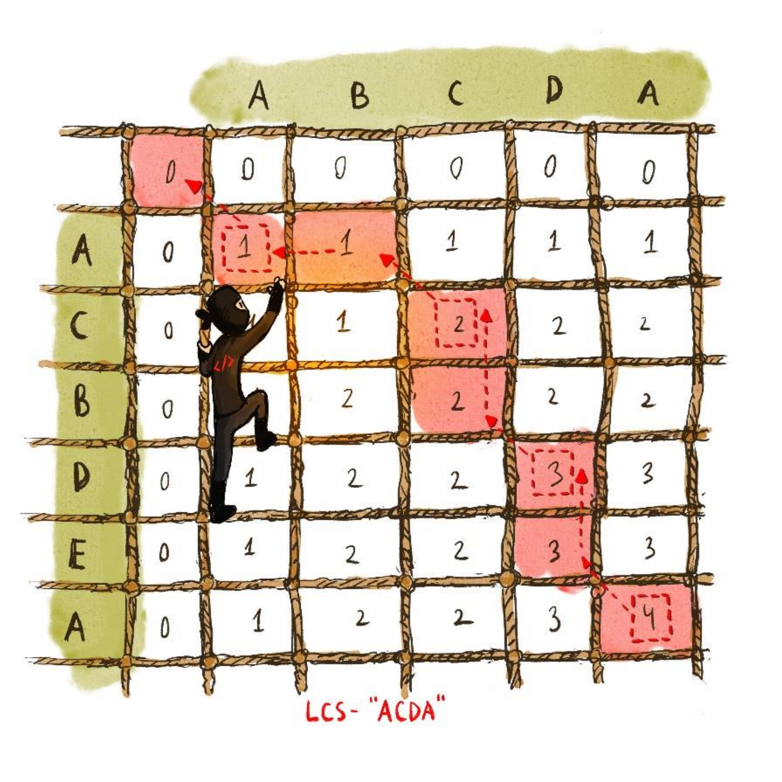
CS141: Intermediate Data Structures and Algorithms



Dynamic Programming

Yan Gu

Class announcement

- **Quiz 2 is in the next lecture (will talk about the details at the end of this lecture)**

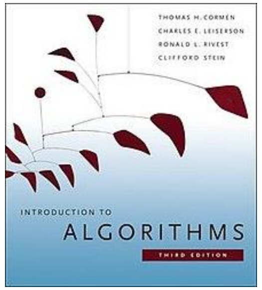
- **Midterm: 6:30-8:30pm Oct 28, MSE 116**
  - Must be in-person (paper-based) unless there is an unresolvable conflict
  - If so, please let me know before Oct 21 (the next lecture) and we will review the case
  - Total score: 115 (basic) + 13 (bonus)
  - Your final score will be: min(your-basic-score,100) + your-bonus-score
  - We will give you some mock problems later this week
  - You can bring 2 pieces of double-sided letter-size paper as your cheat sheets

- **All materials are covered in the slides and homework problems**
  - Don’t forget that CS 141 is still the hardest course you are going to take at UCR
  - Consider it as a basketball training camp

Knapsack problem

- Your little brother is attending university this year
- Unfortunately, he did not get an offer from UCR, and he has to go to the east coast, and needs to take a flight

$50, 1lb



$70, 5lb

$1500, 8lb

$80, 2lb

A naïve algorithm

Item 1: 5 lb, $150
Item 2: 4 lb, $100
Item 3: 2 lb, $10

suitcase(8):
Case 1: first put item 1,
total value = suitcase(3) + 150
Case 2: first put item 2,
total value = suitcase(4) + 100
Case 3: first put item 3,
total value = suitcase(6) + 10
Best = max of the above three

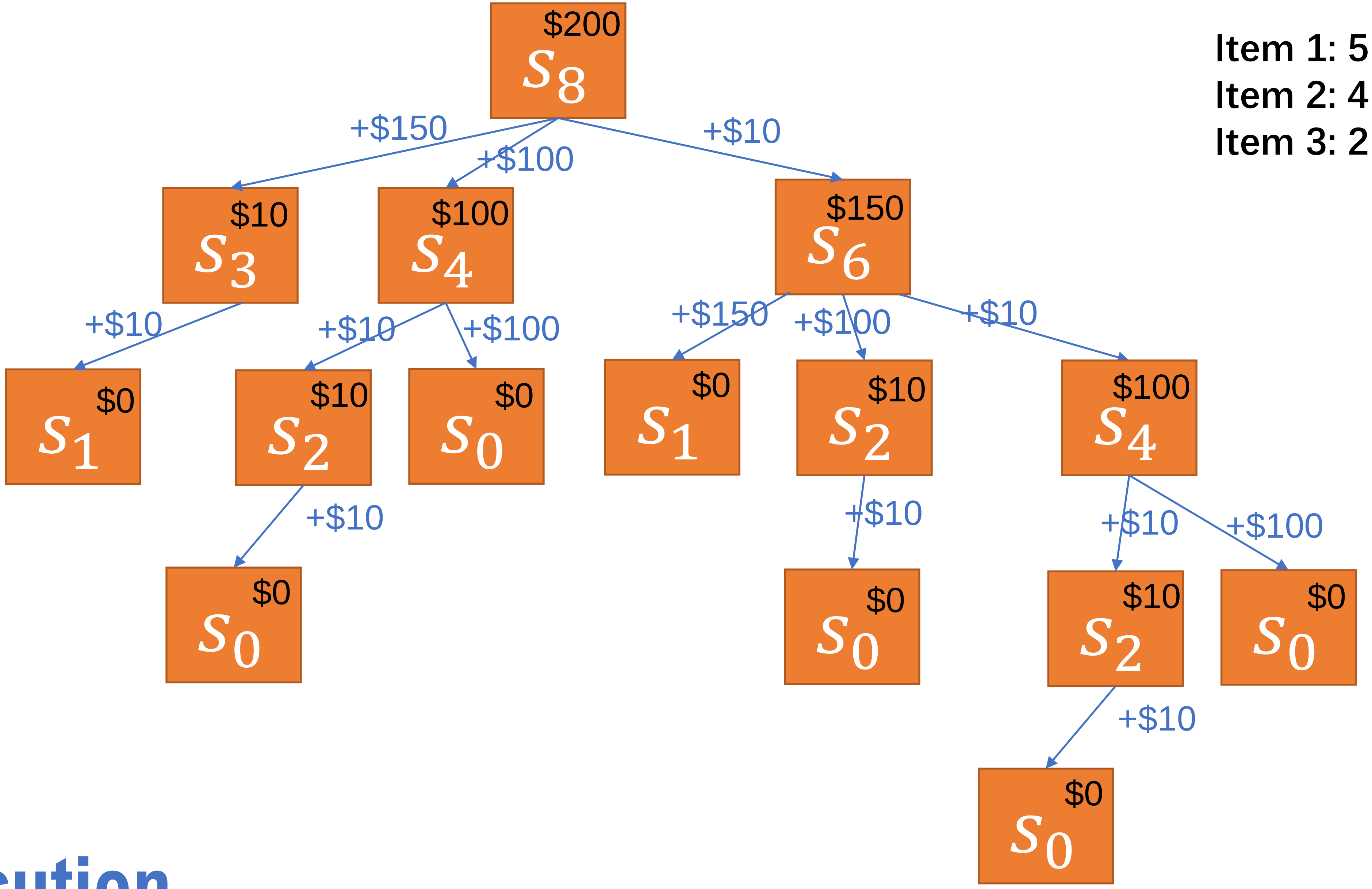
```
int suitcase(int leftWeight) {
    int curBest = 0;
    foreach item of (weight, value)
        if (leftWeight >= weight)
            curBest = max(curBest, suitcase(leftWeight - weight) + value);
    return curBest;
}
```

Recursive call

```
answer = suitcase(8);
```

This algorithm takes exponential time, and only works for very small instances

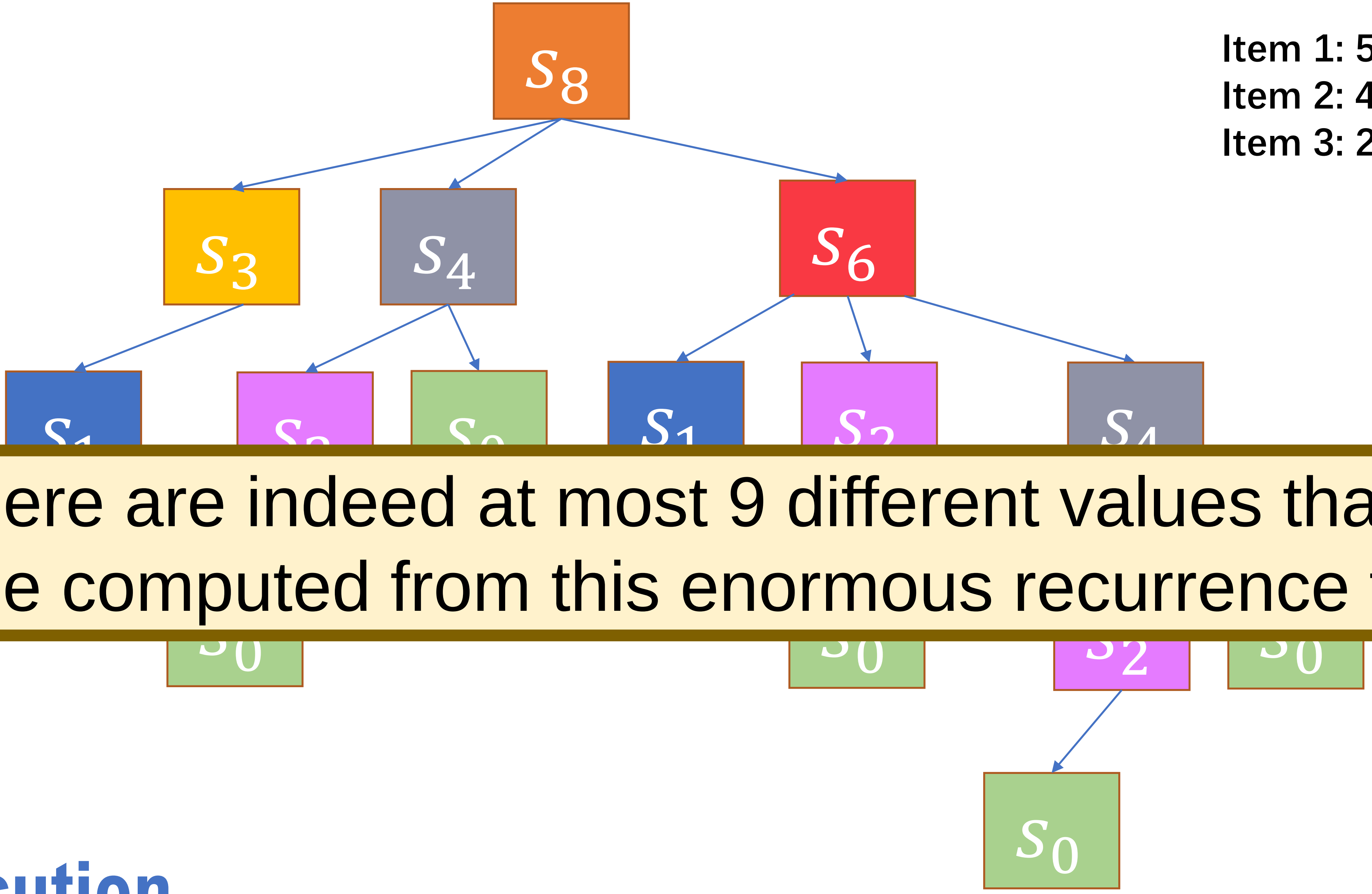
Item 1: 5 lb, $150
Item 2: 4 lb, $100
Item 3: 2 lb, $10



Execution Recurrence Tree

Execution Recurrence Tree

Item 1: 5 lb, $150
Item 2: 4 lb, $100
Item 3: 2 lb, $10

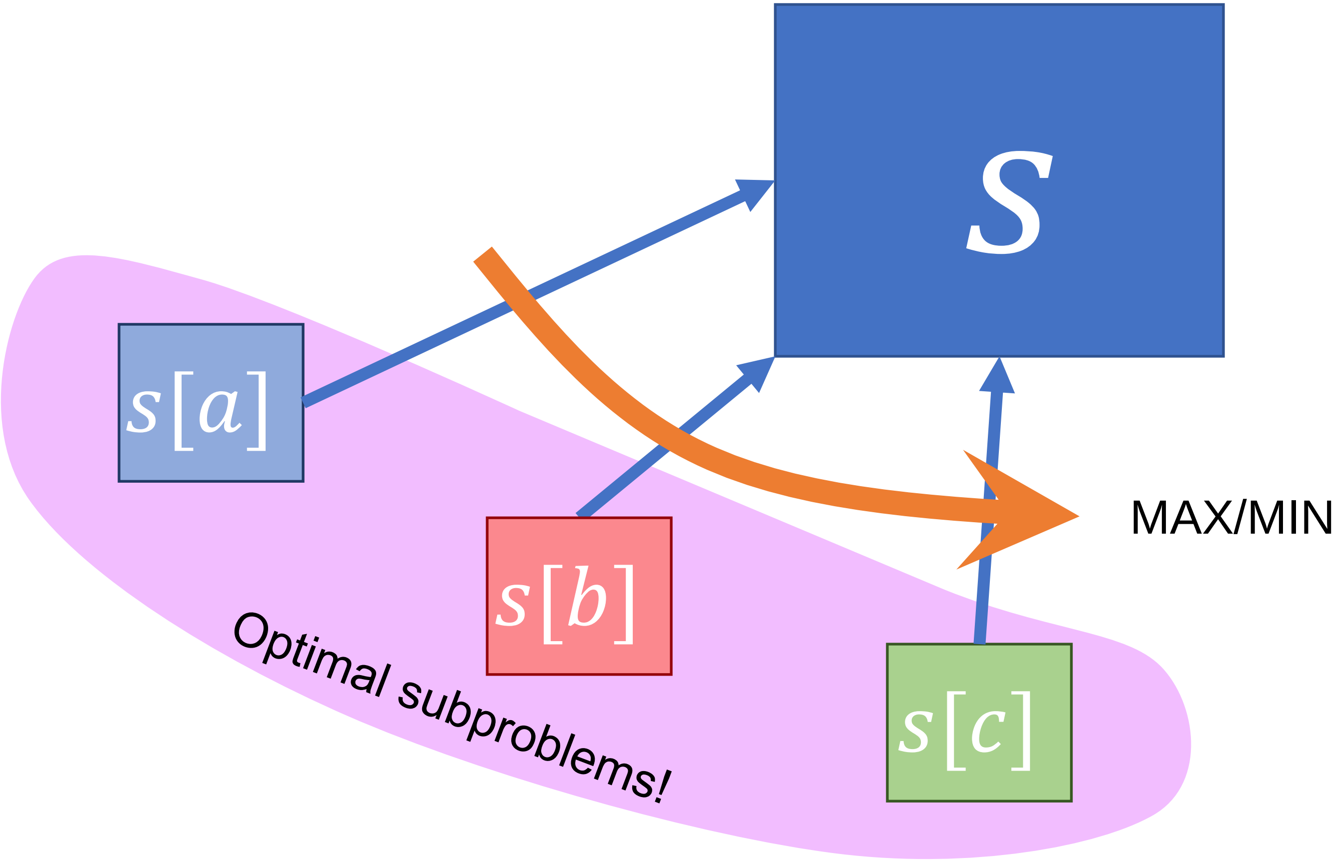


There are indeed at most 9 different values that can be computed from this enormous recurrence tree

What is dynamic programming?

$$s[i] = \max \begin{cases} 0 \\ \max\limits_{(w_j, v_j) \text{ is an item}} \{s[i - w_j] + v_j\} \mid i > w_j \end{cases}$$

Use an array!
Memorization!

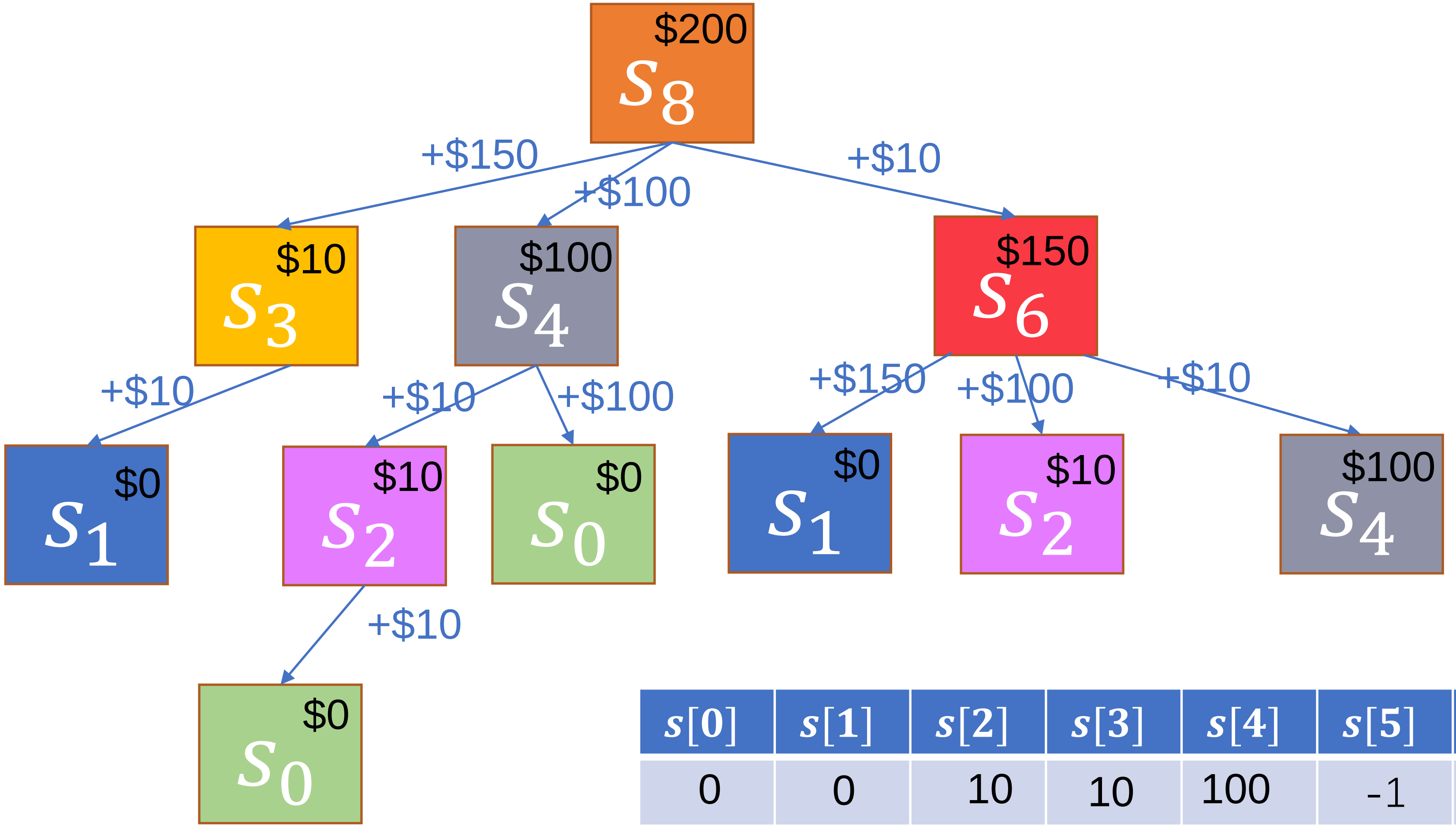


A DP algorithm

```
int suitcase(int leftWeight) {
    if (ans[leftWeight] != -1) return ans[leftWeight];
    int curBest = 0;
    foreach item (weight, value)
        if (leftWeight >= weight)
            curBest = max(curBest, suitcase(leftWeight - Weight) + value);
    return ans[leftWeight] = curBest;
}

int ans[0..8] = {-1, … , -1};
answer = suitcase(8);
```

Item 1: 5 lb, \$150
Item 2: 4 lb, \$100
Item 3: 2 lb, \$10



$s[0]$	$s[1]$	$s[2]$	$s[3]$	$s[4]$	$s[5]$	$s[6]$	$s[7]$	$s[8]$
0	0	10	10	100	-1	150	-1	200

$\Theta(nk)$ cost
where n is #items
k is weight limit

Execution Recurrence Tree

What is dynamic programming?

- Optimal substructure (**states**)
  - What defines a subproblem?
    - weight limit
  - What should be memorized as the index/value of your array?
    - The best value of a given weight limit
- The **decisions**
  - What are the possible “last move”?
    - Put in item 1, 2, 3, …
  - Take a min or max?
- **Boundary**
  - What are the base cases?
    - $s[0] = 0$ (no weight => no value)
- **Recurrence**
  - Compute current state from previous states

$$s[i] = \max \begin{cases} 0 \\ \max\limits_{(w_j, v_j) \text{ is an item}} \{s[i - w_j] + v_j\} \mid i > w_j \end{cases}$$

Why the solution doesn't work for 0/1 knapsack?

- What is the "optimal subproblem"?
- After we choose item j, is the leftover problem "best value of weight limit $k - w_j$"?

- No! It's "best value of weight limit $k - w_j$ and we cannot use item j again"!

- How can we change the state to accommodate this?

- The subproblem we use must not contain item j!
- Add another dimension!

What is the optimal substructure for the new problem?

- **Let $s[i, j]$ be the optimal value for total weight j using only the first i items**
- **How to calculate $s[i, j]$? There are two options:**
  - Use the item i (value of i + best solution of weight limit $j - w_i$ using first i-1 items)

    $s[i-1, j-w_i] + v_i$

  - Do not use item i (best solution of weight limit j using first i-1 items)

    $s[i-1, j]$

  - We added a dimension ("first i items"): the current "stage"
  - The subproblem does not contain item i!

What is the optimal substructure for the new problem?

- **Let $s[i,j]$ be the optimal value for total weight j using only the first i items**
- **What do we need for s[i,j]? What happens if we know s[i-1, j]?**
  - We already know the best value for each weight limit when we have the first i-1 items!
  - Just add the i-th item to see if it changes anything!
- **How to calculate $s[i,j]$? There are two options (take max):**
  - Use the item i (value of i + best solution of weight limit $j - w_i$ using first i-1 items)

    $$s[i-1, j-w_i] + v_i$$
  - Do not use item i (best solution of weight limit j using first i-1 items)

    $$s[i-1, j]$$
- **We added a dimension ("first i items"): the current "stage"**
  - We are putting items one by one (so no duplicates!)
  - The subproblem does not contain item i!

Recurrence for 0/1 knapsack

- **The recurrence:**

$$s[i,j] = \max \begin{cases} s[i-1,j] & \\ s[i-1,j-w_i] + v_i & j \geq w_i \end{cases}$$

- **The boundary:** $s[0,j] = 0$

The DP implementation

```
int suitcase(int i, int j) {
    if (ans[i][j] != -1) return ans[i][j];
    if (i == 0) return 0;
    int best = suitcase(i-1, j);
    if (j >= weight[i])
        best = max(best, suitcase(i-1, j-weight[i])+value[i]);
    return ans[i][j] = best;
}

int ans[n][k] = {-1, … , -1};
answer = suitcase(n, k);
```

A non-recursive implementation

```
int ans[0][i] = {0, … , 0};
for i = 1 to n do
    for i = 0 to k do {
        ans[i][j] = ans[i-1][j];
        if (i >= weight[j])
            ans[i][j] = max(ans[i][j], ans[i-1][j-weight[i]]+value[i]);
    }
return ans[n][k];
```

- Generally, be careful to use the non-recursive implementation — easy to err if a state that should be computed is actually not

So easy!

- Conversation between a mom and her four-year-old kid:
- - What is 1+1+1+1+1+1+1+1?
- - (Thought for a while) 8!
- - What is 1+1+1+1+1+1+1+1+1?
- - (Immediately) 9!
- - How can you do that so fast?
- - Because I know 1+1+1+1+1+1+1+1 is 8!
- - That’s **memorization**. Congratulations, **you understand dynamic programming now**!

What is dynamic programming?

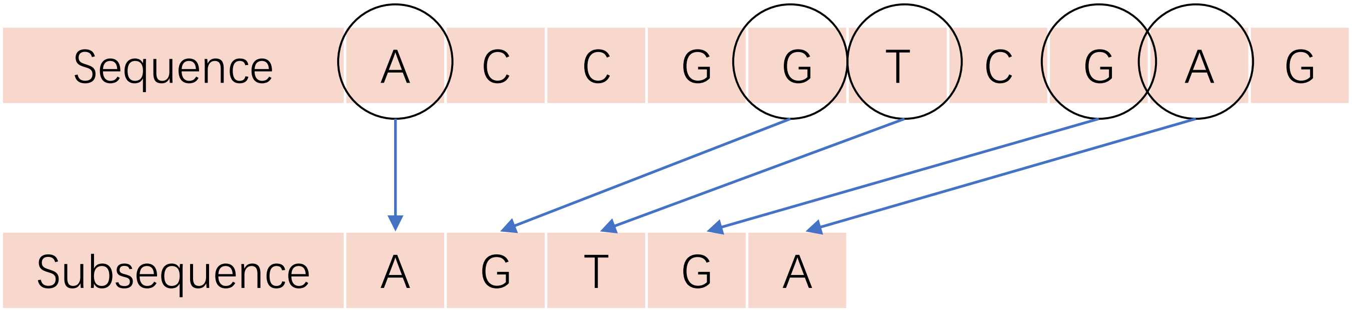
- Optimal substructure (**states**)
  - What defines a subproblem?
    - First i items and weight limit j
  - What should be memorized as the index/value of your array?
    - The best value of a given weight limit and first i items

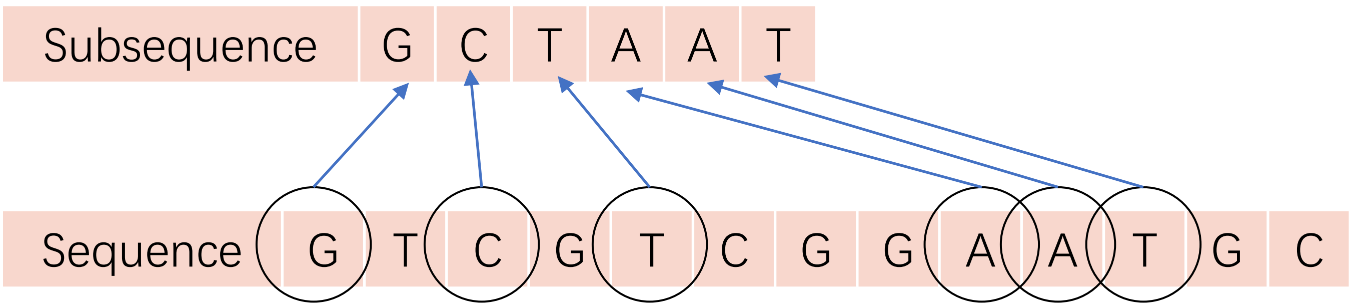
$$s[i,j] = \max\begin{cases} s[i-1,j] & \\ s[i-1,j-w_i]+v_i & j \geq w_i \end{cases}$$

- The **decisions**
  - What are the possible "last move"?
    - Put in item i or not?
  - Take max?
- **Boundary**
  - What are the base cases?
    - $s[0,j]=0$ (no item => no value)
- **Recurrence**
  - Compute current state from previous states

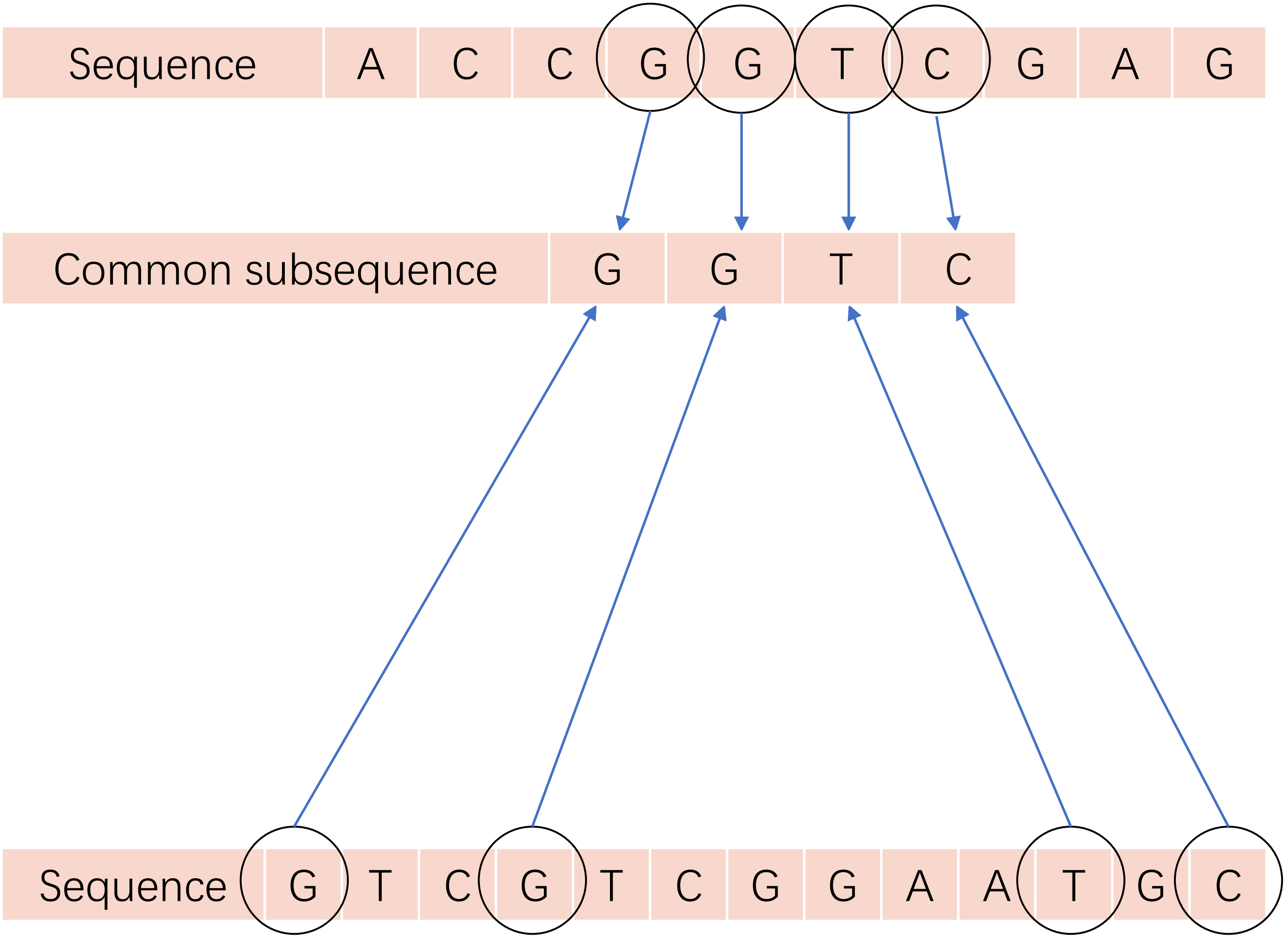
Longest Common Subsequence (LCS)

Definitions

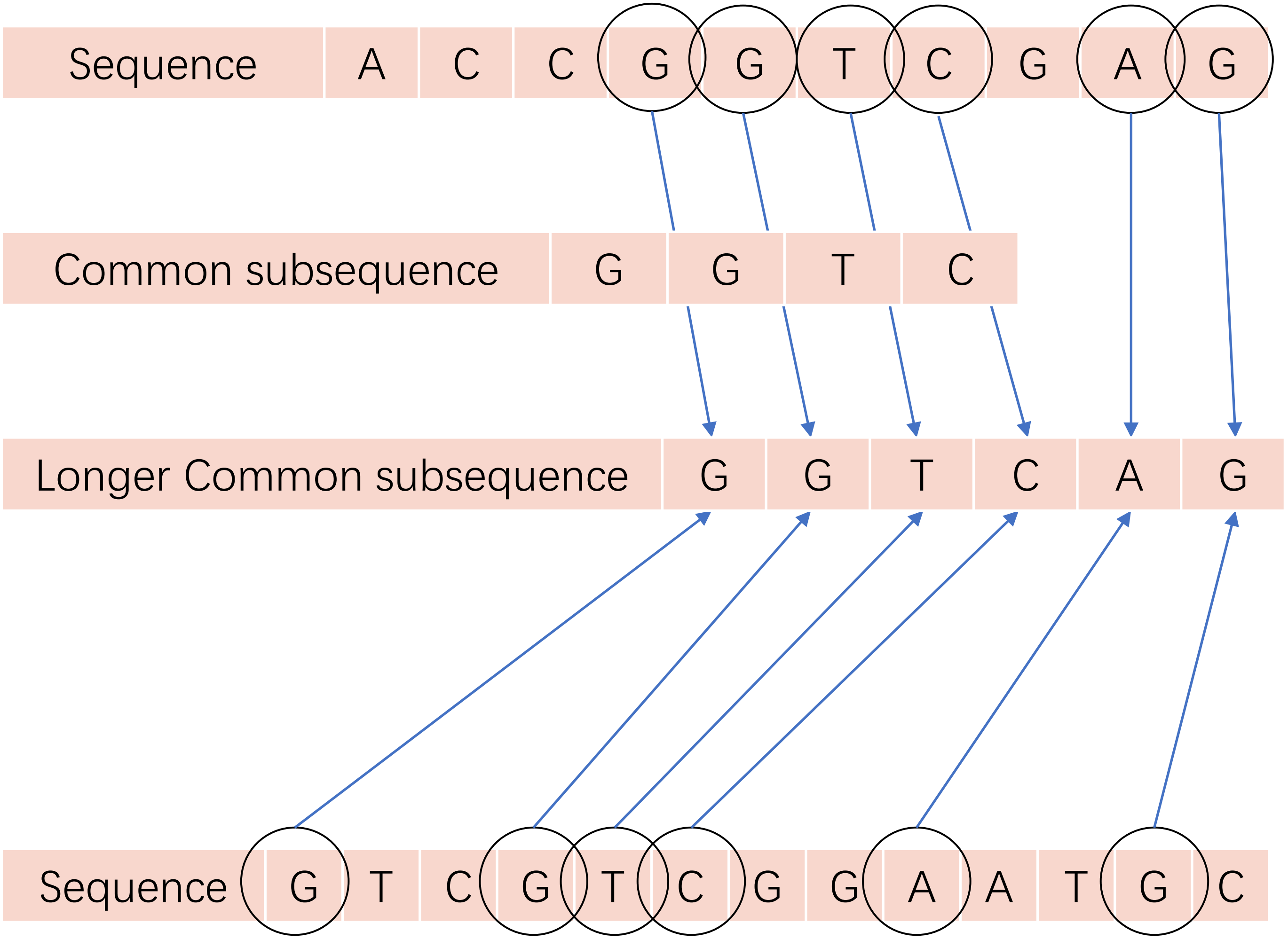




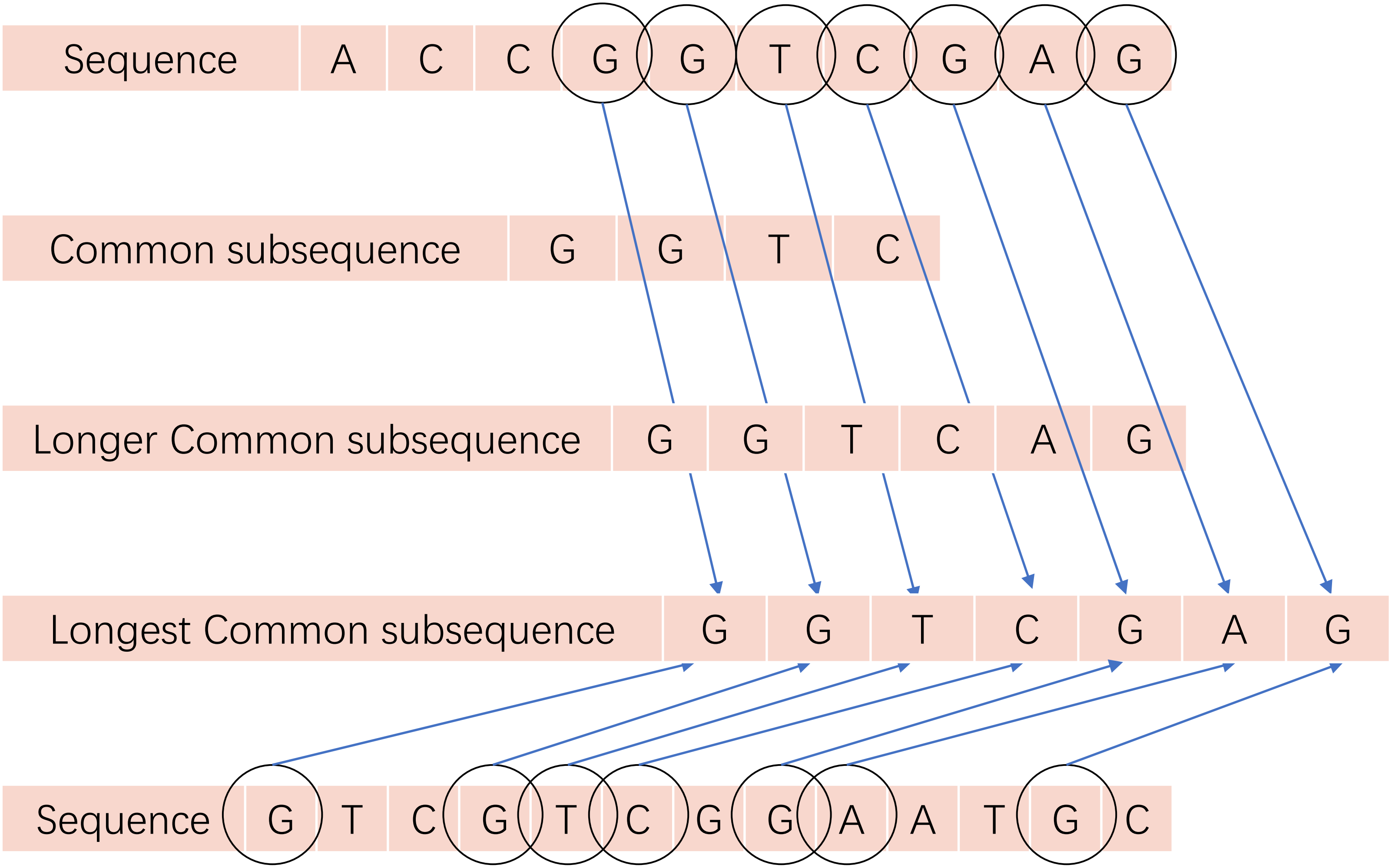
Definitions



Definitions



Definitions



Problem Definition

- **Input: two sequences X and Y**
- **We say that a sequence Z is a common subsequence of X and Y if it is a subsequence of both X and Y**
  - For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of X and Y; not a longest one though
- **The problem is to find a longest common subsequence Z of X and Y**
  - For the previous example, the longest common subsequence is $Z = \langle B, C, B, A \rangle$
- **What are the "subproblems" here?**
- **What is the possible "last move"?**
  - For ABCBDAB and BDCABA, we want to know, how should we deal with the last element X ('B') and Y ('A'), respectively

Consider the last characters of two input sequences X and Y

LCS

- **Let's consider the subproblem of LCS of $X[1..i]$ and $Y[1..j]$**
- **Let's compare the last character $X[i]$ and $Y[j]$**
  - So the subproblems rely on smaller prefixes
- **What if $X[i] = Y[j]$?**
  - ABCBDA and BDCABA
- **What if $X[i] \neq Y[j]$?**
  - ABCBDAB and BDCABA
- **What else do we need?**

Solution for LCS

- **Use $s[i, j]$ to denote the LCS of**
  - The first i characters in X
  - And
  - The first j characters in Y

- **If we want to compute $s[i, j]$, what do we need?**

LCS

- **if $X[i] = Y[j] = c$**
  - The last character of LCS of $X[1..i]$ and $Y[1..j]$ must be c (why?)
  - Then we just need to find the LCS of X[1..i-1] and Y[1..j-1] and add c at the end
  - s[i, j] = s[i-1, j-1] + 1

Index :	1	2	3	4	
X =	A	B	C	B	
Y =	B	D	C	A	B

LCS of "**ABCB**" and "**BDCAB**" must be:
(the LCS of "**ABC**" and "**BDCA**") + "**B**"

s[4, 5] = s[3, 4] + 1

Recursive Algorithm

- **if $X[i] \neq Y[j]$**
  - Three choices: keep $X[i]$ as the last one, $Y[i]$ as the last one, or discard both $X[i]$ and $Y[j]$
  - return MAX(s[i-1, j], s[i, j-1])

Index :	1	2	3		
X =	A	B	C		
Y =	B	D	C	A	B

LCS of “**ABC**” and “**BDCAB**” can be:

the LCS of “**AB**” and “**BDCAB**”

the LCS of “**ABC**” and “**BDCA**”

the LCS of “**AB**” and “**BDCA**” (included above)

s[3, 5] = max(s[2, 5], s[3, 4])

LCS

- Let $s[i, j]$ be the LCS of $X[1..i]$ and $Y[1..j]$

- $s[i, j] = \begin{cases} s[i-1, j-1] + 1 & : X[i] = Y[j] \\ max(s[i-1, j], s[i, j-1]) & : X[i] \neq Y[j] \end{cases}$

- $s[i, 0] = 0,\ s[0, j] = 0$

Naïve recursive Algorithm

- int LCS(i, j):
    - if i == 0 or j == 0 return 0
    - if X[i] == Y[j]
        - return LCS(i-1, j-1) + 1
    - if X[i] != Y[j]
        - return max(LCS(i, j-1), LCS(i-1, j))

- ans = LCS(n, m)

Recursive Algorithm

- int LCS(i, j):
  - if s[i,j] != -1 then return s[i,j]
  - if i == 0 or j == 0 return s[i,j] = 0
  - if X[i] == Y[j]
    - return s[i,j] = LCS(i-1, j-1) + 1
  - if X[i] != Y[j]
    - return s[i,j] = max(LCS(i, j-1), LCS(i-1, j))

- ans = LCS(n, m)

Bottom-up Algorithm

$$s[i,j] = \begin{cases} s[i-1,j-1]+1 & x_i = y_j \\ max\{s[i-1,j], s[i,j-1]\} & x_i \neq y_j \end{cases}$$

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

Bottom-up Algorithm

$$s[i,j]=\begin{cases} s[i-1,j-1]+1 & x_i=y_j \\ max\{s[i-1,j],s[i,j-1]\} & x_i\neq y_j \end{cases}$$

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

Bottom-up Algorithm

$$s[i,j] = \begin{cases} s[i-1,j-1]+1 & x_i = y_j \\ max\{s[i-1,j], s[i,j-1]\} & x_i \neq y_j \end{cases}$$

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

Bottom-up Algorithm

$$s[i,j] = \begin{cases} s[i-1,j-1]+1 & x_i = y_j \\ max\{s[i-1,j], s[i,j-1]\} & x_i \neq y_j \end{cases}$$

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

Bottom-up Algorithm

$$s[i,j] = \begin{cases} s[i-1,j-1]+1 & x_i = y_j \\ max\{s[i-1,j], s[i,j-1]\} & x_i \neq y_j \end{cases}$$

i \ j	0	B 1	D 2	C 3	A 4	B 5	A 6
0	0	0	0	0	0	0	0
A 1	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↑ 1
B 2	0	↖ 1	← 1	← 1	↑ 1	↖ 2	2
C 3	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B 4	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D 5	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A 6	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B 7	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Bottom-up Algorithm

$$s[i,j] = \begin{cases} s[i-1, j-1] + 1 & x_i = y_j \\ max\{s[i-1,j], s[i,j-1]\} & x_i \neq y_j \end{cases}$$

i \ j	0	B 1	D 2	C 3	A 4	B 5	A 6
0	0	0	0	0	0	0	0
A 1	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↑ 1
B 2	0	↖ 1	← 1	← 1	↑ 1	↖ 2	2
C 3	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B 4	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D 5	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A 6	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B 7	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Construction Algorithm

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new
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
```

Print-LCS

```
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)
```

States and decision

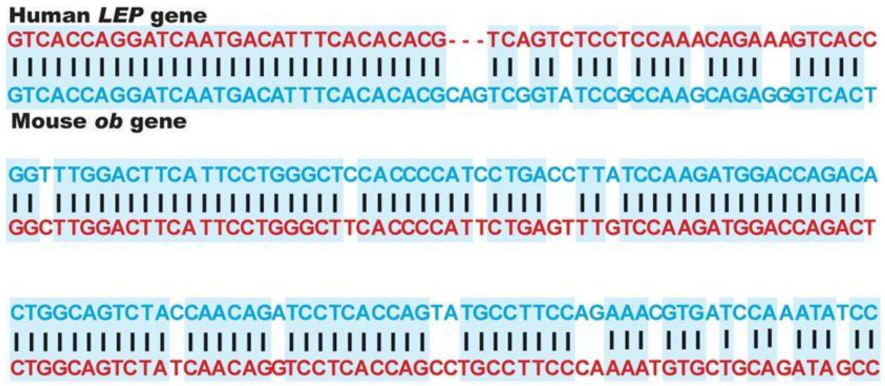
- **states**
  - What defines a subproblem?
    - The first i characters in X and first j characters in Y
  - What should be memorized as the index/value of your array?
    - The LCS of X[1..i] and Y[1..j] – We'll use them later!
- **decisions**
  - What are the possible "last move"?
    - Match X[i] and/or Y[j]
    - If X[i]=Y[j], use it as the last character
    - If X[i] != Y[j], drop X[i], or Y[j]
  - Take max
- **Boundary**
  - What are the base cases?
    - $s[0, i] = 0, s[i, 0] = 0$ (when one string is empty, LCS=0)

Edit Distance

Minimum Edit Distance

- **How to measure the similarity of words or strings?**
- **Auto corrections: “rationg” -> {“rating”, “ration”}**
- **Alignment of DNA sequences**
- **How many edits we need (at least) to transform a sequence X to Y?**
  - Insertion
  - Deletion
  - Replace
- **rationg -> rating**
  - Delete o, edit distance 1
- **rationg -> action**
  - Delete r, add c, delete g
  - Edit distance 3

An Example of DNA sequence alignment



Adapted from Klug p. 384

Determine the matching score.

Recurrence of Edit Distance

- **Similar to LCS, consider the cost to transform $X[1..i]$ to $Y[1..j]$**
- **Look at the last character $X[i]$ and $Y[j]$**
- **What happens if $X[i] = Y[j]$?**

Index :	1	2	3	4	
X =	A	B	C	B	
Y =	B	D	C	A	B

  - Keep $X[i]$ and $Y[j]$ – no edit needed
  - Need to transform ABC to BDCA
  - →s[i-1,j-1]

Recurrence of Edit Distance

- **Similar to LCS, consider the cost to transform $X[1..i]$ to $Y[1..j]$**
- **Look at the last character $X[i]$ and $Y[j]$**
- **What happens if $X[i] \neq Y[j]$?**

Index :	1	2	3		
X =	A	B	C		
Y =	B	D	C	A	B

- **Delete C**. Cost = (cost of transforming AB => BDCAB) + 1 → s[i-1, j] + 1
- **Adding B**. Cost = (cost of transforming ABC => BDCA) + 1 → s[i, j-1] + 1
- **Editing C to B**. Cost = (cost of transforming AB => BDCA) + 1 → s[i-1, j-1] + 1
- Use the min of the above three!

Recurrence Relation

- $s[i,j]$: The cost of transforming $X[1..i]$ to $Y[1..j]$

$$s[i,j] = \begin{cases} \max\{i,j\} & ; i = 0 \lor j = 0 \\ s[i-1,j-1] & ; i > 0 \land j > 0 \land x_i = y_j \\ \min\begin{Bmatrix} s[i,j-1]+1 \\ s[i-1,j]+1 \\ s[i-1,j-1]+1 \end{Bmatrix} & ; i > 0 \land j > 0 \land x_i \neq x_j \end{cases}$$

States and decision

- **states**
  - What defines a subproblem?
    - The edit distance between first i characters in X and first j characters in Y
  - What should be memorized as the index/value of your array?
    - The edit distance between X[1..i] and Y[1..j] – We'll use them later!
- **decisions**
  - What are the possible "last move"?
    - Make the last character match!
    - If X[i]=Y[j], no edit needed
    - If X[i] != Y[j], we can either delete X[i], or insert Y[j], or change X[i] to Y[j]
  - Take min
- **Boundary**
  - What are the base cases?
    - $s[0, i] = i, s[i, 0] = i$ (when one string is empty, need i insertion/deletions)

Dynamic Programming (DP)

- **Recall the concept of "dynamic programming" (DP)**
  - DP is not an algorithm, but an algorithm design idea (methodology)
  - DP works on problems with optimal substructure

- **A DP recurrence of the states, decisions, with boundary cases**

- **We can convert a DP recurrence to a DP algorithm**
  - Recursive implementation: straightforward
  - Non-recursive implementation: faster, and easy to be optimized

- **To design a DP algorithm, we usually consider some "prefix" of the problem, and work on the last element (so the subproblem is still a prefix)**

Summary of today's lecture

- **Longest common sequence and edit distance**
  - Two closely related problems and very similar DP solutions
- **Usually, algorithms in this flavor are programmed directly using nested for-loops, but can also be implemented using recursion**
- **The only way to understand DP is by practice**
  - Two similar problem in the Prog HW 4 for you to practice
  - Prog HW 4 is also for dynamic programming (8 prog problems in total, 2 written)
- **More examples for DP in the next lecture**

Quiz 2

- **Next class: Thursday**
  - Session 1: 2:00-2:20pm
  - Session 2: 6:30-6:50pm
- **You can take the quiz for either session 1 or 2, or BOTH**
  - Score = max of your score
- **We encourage you to take it in the classroom**
  - A bonus question (3 points)
  - 33/30 points