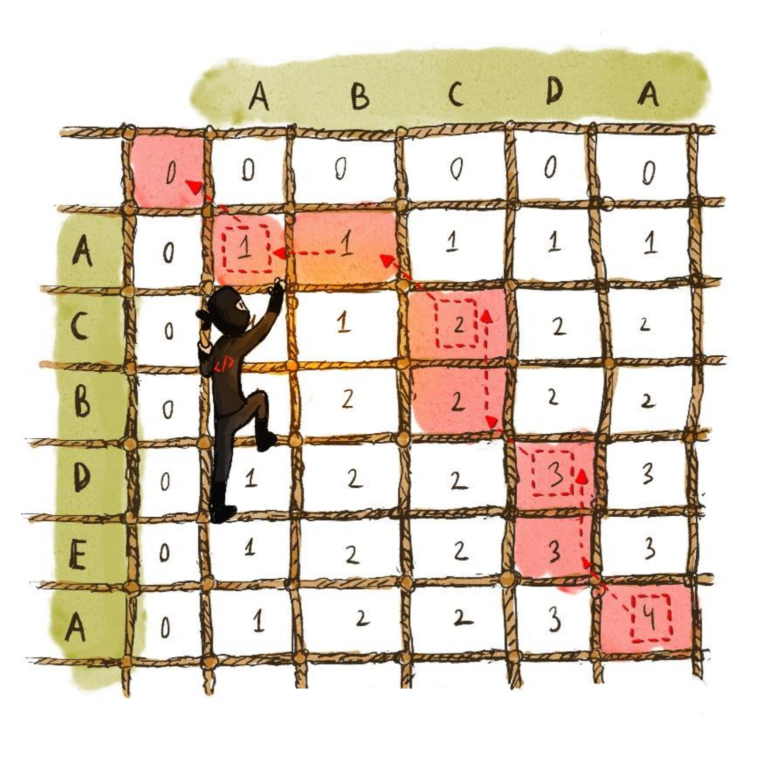
CS218: Design And Analysis Of Algorithms



Dynamic Programming

Yan Gu

Unlimited knapsack problem

- A knapsack of weight limit W
- n items with value v_i and weight w_i
- How to use the knapsack to take the maximum total value?

Value = 150

\$5

\$4 Value = 100

\$2 Value = 10

budget = 8 dollars

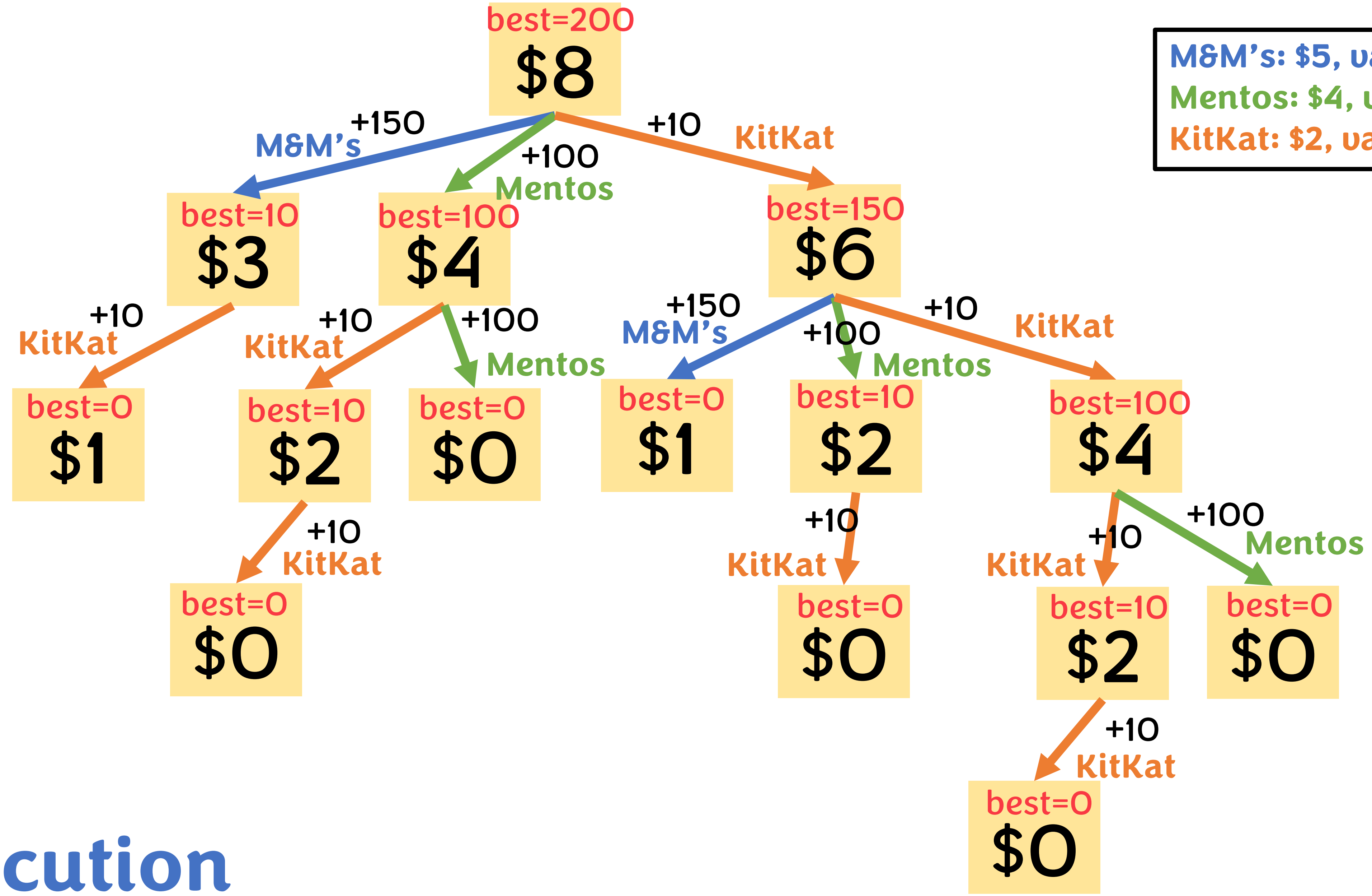
A naïve algorithm

```
int candy(int budget) {
    int best = 0;
    foreach item of (price, value)
        if (budget >= price) {
            current = candy(budget - price) + value;
            best = max(best, current); }
    return best;
}

answer = candy(8);
```

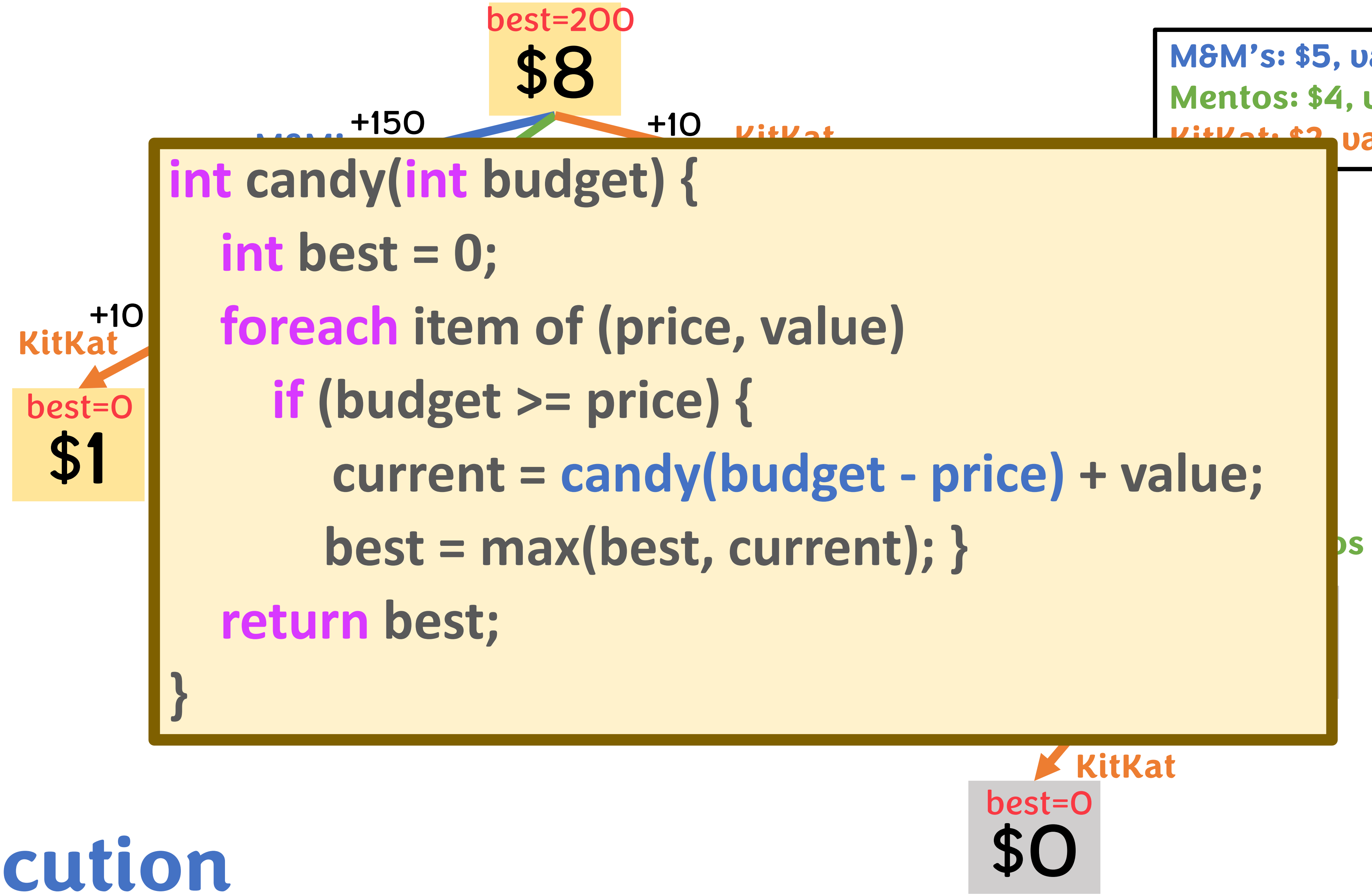
Recursive call
(optimal substructure)

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



M&M's: $5, value = 150
Mentos: $4, value = 100
KitKat: $2, value = 10

Execution Recurrence Tree

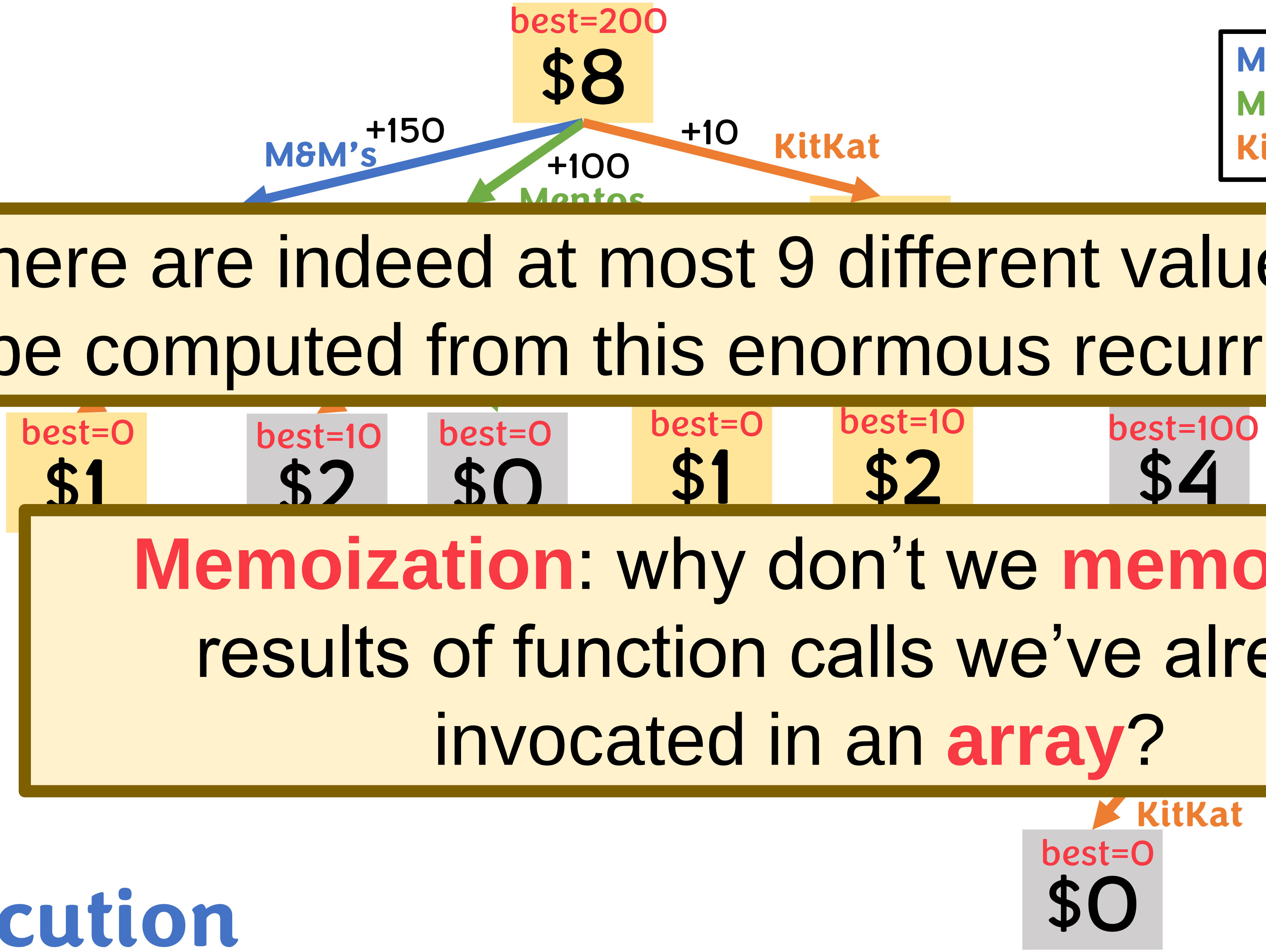


M&M’s: $5, value = 150
Mentos: $4, value = 100
KitKat: $2, value = 10

```
int candy(int budget) {
   int best = 0;
   foreach item of (price, value)
      if (budget >= price) {
         current = candy(budget - price) + value;
         best = max(best, current); }
   return best;
}
```

Execution
Recurrence Tree

M&M's: $5, value = 150
Mentos: $4, value = 100
KitKat: $2, value = 10



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

Memoization: why don't we **memoize** all results of function calls we've already invocated in an **array**?

Execution Recurrence Tree

A naïve algorithm

```
int candy(int budget) {
   int best = 0;
   foreach item (price, value)
      if (budget >= price) {
         current = candy(budget - price) + value;
         best = max(best, current); }
   return best;
}

answer = candy(8);
```

Recursive call
(optimal substructure)

A DP algorithm

```
int candy(int budget) {
  int best = 0;
  foreach item (price, value)
    if (budget >= price) {
      current = candy(budget - price) + value;
      best = max(best, current); }
  return best;
}
```

Recursive call
(optimal substructure)

```
int s[0..8] = {-1, … , -1};  // Initialize as -1, indicating “not computed”
answer = candy(8);
```

A DP algorithm

```
int candy(int budget) {
    if (s[budget] != -1) return s[budget];   // if already computed, directly return
    int best = 0;
    foreach item (price, value)
        if (budget >= price) {
            current = candy(budget - price) + value;
            best = max(best, current); }
    return best;
}

int s[0..8] = {-1, … , -1};  // Initialize as -1, indicating "not computed"
answer = candy(8);
```

Recursive call
(optimal substructure)

A DP algorithm

```
int candy(int budget) {
    if (s[budget] != -1) return s[budget];   // if already computed, directly return
    int best = 0;
    foreach item (price, value)
        if (budget >= price) {
            current = candy(budget - price) + value;
            best = max(best, current); }
    s[budget] = best; // memoize the current return value
    return best;
}

int s[0..8] = {-1, … , -1};  // Initialize as -1, indicating “not computed”
answer = candy(8);
```

Recursive call
(optimal substructure)

A DP algorithm

```
int candy(int budget) {
    if (s[budget] != -1) return s[budget];   // if already computed, directly return
    int best = 0;
    foreach item (price, value)
        if (budget >= price) {
            current = candy(budget - price) + value;
            best = max(best, current); }
    s[budget] = best; // memoize the current return value
    return best;
}

int s[0..8] = {-1, … , -1};  // Initialize as -1, indicating "not computed"
answer = candy(8);
```

Recursive call
(optimal substructure)

But if calculated before, it will directly find the answer!

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 **memoization**. Congratulations, **you understand dynamic programming now**!

Memoization and dynamic programming

- **(the previous setting is not exactly DP since that's not optimization problem, but the idea for memoization is the same!)**
- **Store your previous result in an array**
- **When you need it, directly lookup**
  - So that you don't need to calculate one subproblem multiple times
- **We use "state" to call the identifier for us to find what to lookup (the subproblem)**
  - "the current budget/weight"
- **Usually it's the index of the array**

Recursive Solution

- **Define $s[i]$ as the maximum value you can get for a total weight of i**
- **We can express $s[i]$ as the following recurrence:**

The best value with $\boldsymbol{i - w_j}$ weight

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

Trying all possible items

- **$\boldsymbol{s[0] = 0}$, Final answer is $s[W]$**
- **Time complexity: $\boldsymbol{O(Wn)}$**
  - $\boldsymbol{W}$ = weight budget, $\boldsymbol{n}$ = #items

Optimal Substructure

- **The correctness of this problem also relies on optimal substructure:**

- **To achieve the optimal solution for capacity i (the value of $s[i]$)**
- **If we want to try item j**
- **The rest $i - w_j$ space must use the optimal solution for capacity $i - w_j$ (so we lookup the "tabular" to use $s[i - w_j]$)**
  - If $s[i - w_j]$ is ready, use it. Otherwise, compute it and memorize it!

What is the difference between greedy and DP?

- **Greedy = greedy choice + optimal substructure**
- **DP = optimal substructure + Try all possible choices**

- **Both of them contain "optimal substructure" – we need to find best solution for subproblems**
  - Greedy: the choice is a fixed one: your greedy choice
  - [one choice, one subproblem => recursively or iteratively]
  - DP: We don't know which choice is the best. So try all of them, compare, and keep the best one
  - [More choices, more subproblems, may overlap!... => memorization, avoid redundant work]

Memorization and dynamic programming

- **Store your previous result in an array**
- **When you need it, directly lookup**
  - So that you don’t need to calculate one subproblem multiple times
- **We use “state” to call the identifier for us to find what to lookup (the subproblem)**
  - “the current budget/weight”
- **Usually it’s the index of the array**

Is the previous algorithm perfect?
What if each item can be used only once? (0/1 knapsack)

```
int candy(int budget) {
    if (s[budget] != -1) return s[budget];  // if already computed, directly return
    int best = 0;
    foreach item (price, value)
        if (budget >= price) {
            current = candy(budget - price) + value;
            best = max(best, current);       }
    return s[budget] = best; // memorize the current return value
}

int s[5] = {-1, … , -1};
answer = candy(5);
```

What if we have used this already?!

When one recursive call is trying a possible item, it has no idea whether this item has been used before or not…
This information is not contained in its "state"

Is the previous algorithm perfect?
What if each item can be used only once? (0/1 knapsack)

- **Is $s[i]$ sufficient for the new problem (still have optimal substructure)?**
  - No!! We do not know whether the optimal arrangement for weight i uses item j or not
  - If our decision is "add j", our subproblem is "best value with budget s-w[j] without using item j"
  - How can we guarantee that the subproblem exclude item j?
- **What can we do?**
- **Add another dimension! Memoize more!**

Is the previous algorithm perfect?
What if each item can be used only once? (0/1 knapsack)

- **Let $s[i, j]$ be the optimal value**
  - Only use the first i items
  - Given weight budget j

How to calculate $s[i, j]$? There are two options:

- **Use the item i → $v_i + s[i - 1, j - w_i]$**
  - So we get v_i value from item i
  - For the rest, we can use the first $i - 1$ items to fill in weight $j - w_i$
- **Do not use item i → $s[i - 1, j]$**
  - Without the i-th item, we are just using the first $i - 1$ items to fill in weight j
- **Compare the two decisions and choose the better one**

Recurrence of 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:** $s[i,0] = 0, s[0,j] = 0$

$\boldsymbol{s[i,j]}$ = the optimal value

- Only use the first i items
- Given weight budget j

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

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8
i=0	0	0	0	0	0	0	0	0	0
			+150						
i=1	0	0	0	0	0	150	150	150	150
			+100				+100		
i=2	0	0	0	0	100	150	150	150	150
		+10		+10			+10		+10
i=3	0	0	10	10	100	150	150	160	160

1

Value = 150

$5

2

$4 Value = 100

3

$2 Value = 10

The DP implementation

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

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

A non-recursive implementation

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

- Generally, you need to be careful when using the non-recursive implementation — when computing a state, all the other states it depends on must be ready

An even simpler implementation – 0/1 knapsack

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

- We only need to store a 1D array

The simpler implementation for unlimited knapsack

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

- We only need to store a 1D array
- You can try to figure out why these simpler versions work.

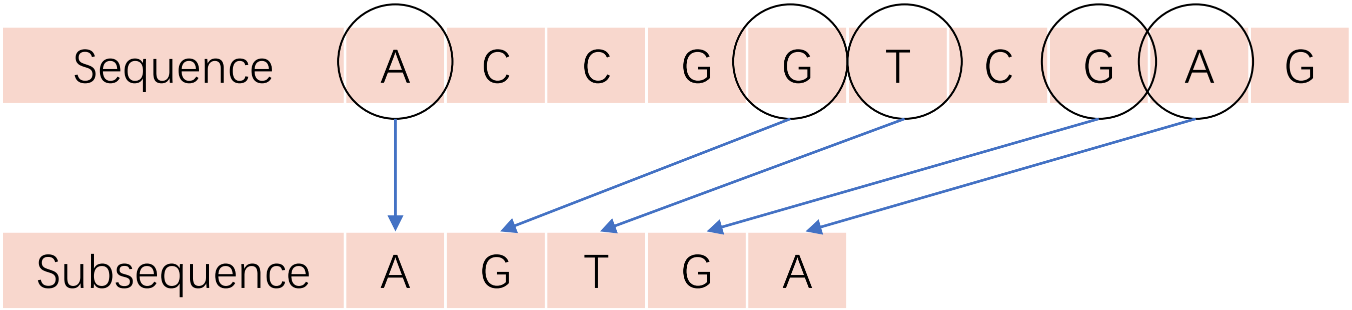
What is dynamic programming?

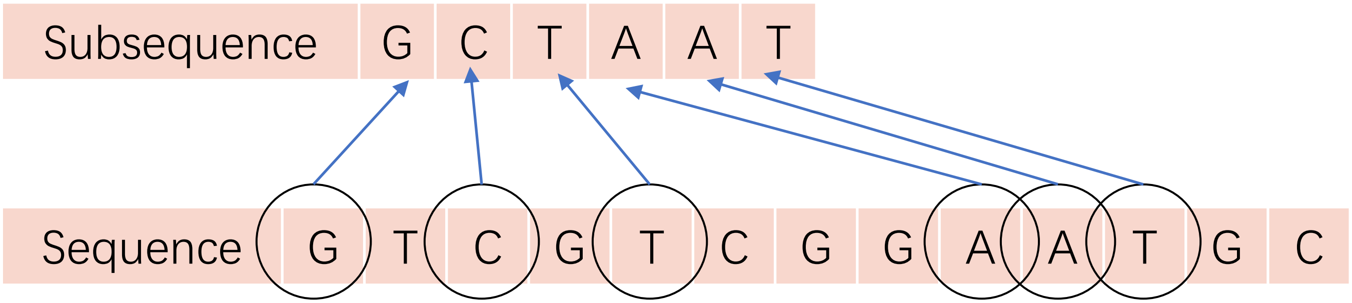
- Optimal substructure (**states**)
  - What defines a **subproblem**?
    - First i items and weight limit j
  - What should be **memorized** as the index/value of your array? What will you look up for later computations?
    - The best value of a given weight limit and first i items
- The **decisions**
  - What are the possible "first/last move"?
    - Put in item i or not?
  - Take max for all decisions
- **Boundary**
  - What are the base cases?
    - $s[0, j] = 0$ (no item => no value)
- **Recurrence**
  - Compute current state from previous states

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

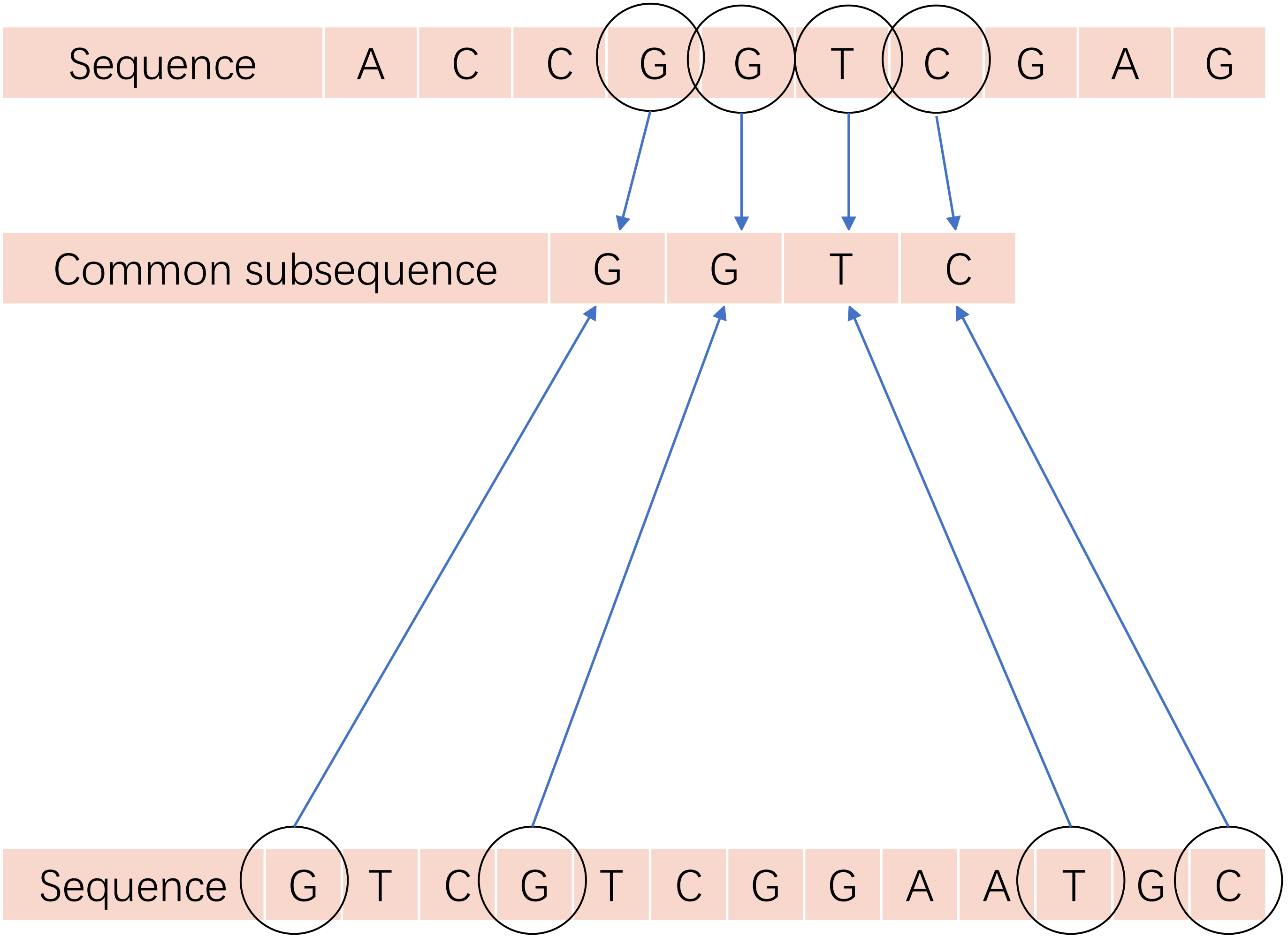
Longest Common Subsequence (LCS)

Definitions

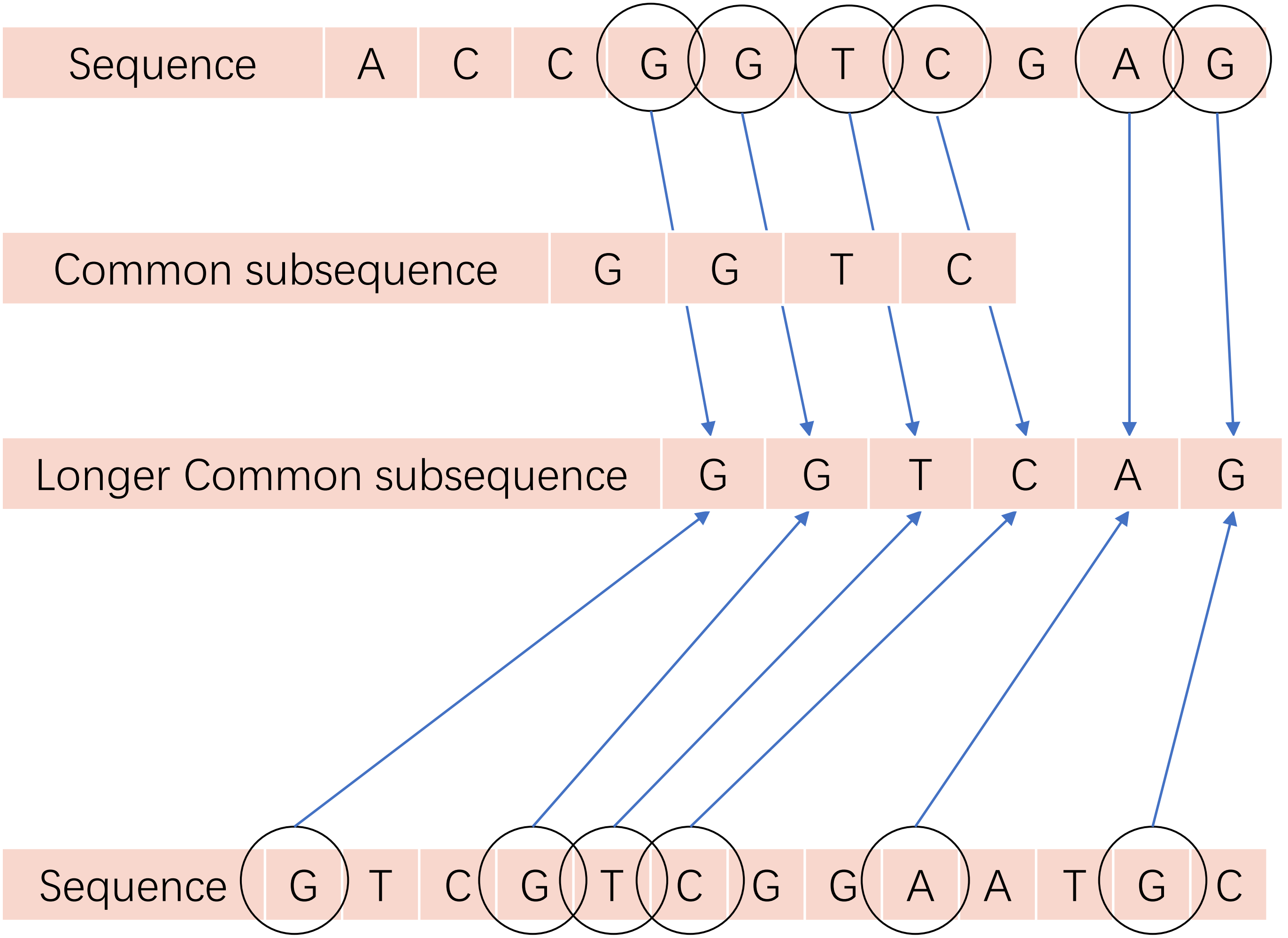




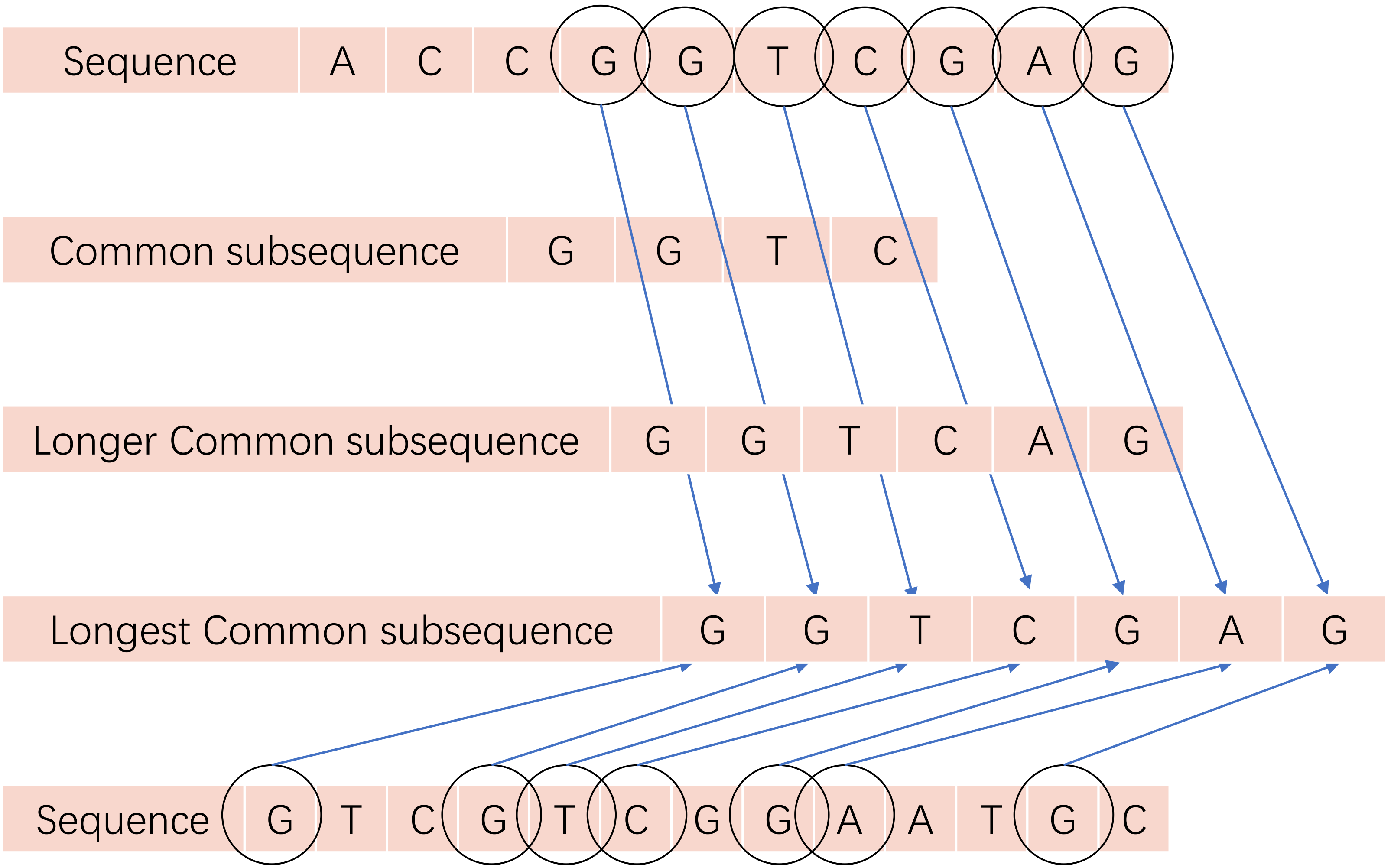
Definitions



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**
  - $X = \langle A, B, C, B, D, A, B \rangle$ 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 the 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 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?**
- **Let $s[i, j]$ be the length of LCS of $X[1..i]$ and $Y[1..j]$**

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		B	D	C	A	B	A
	0	1	2	3	4	5	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

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

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	← 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

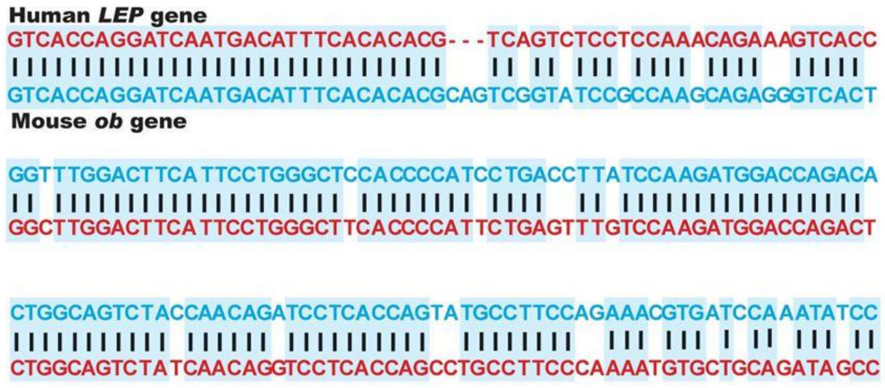
- **states**
  - What defines a subproblem?
    - The first i characters in X and first j characters in Y
  - What should be memorized as the index/value of your array?
    - The LCS of X[1..i] and Y[1..j] – We'll use them later!
- **decisions**
  - What are the possible "last move"?
    - Match X[i] 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
					↑

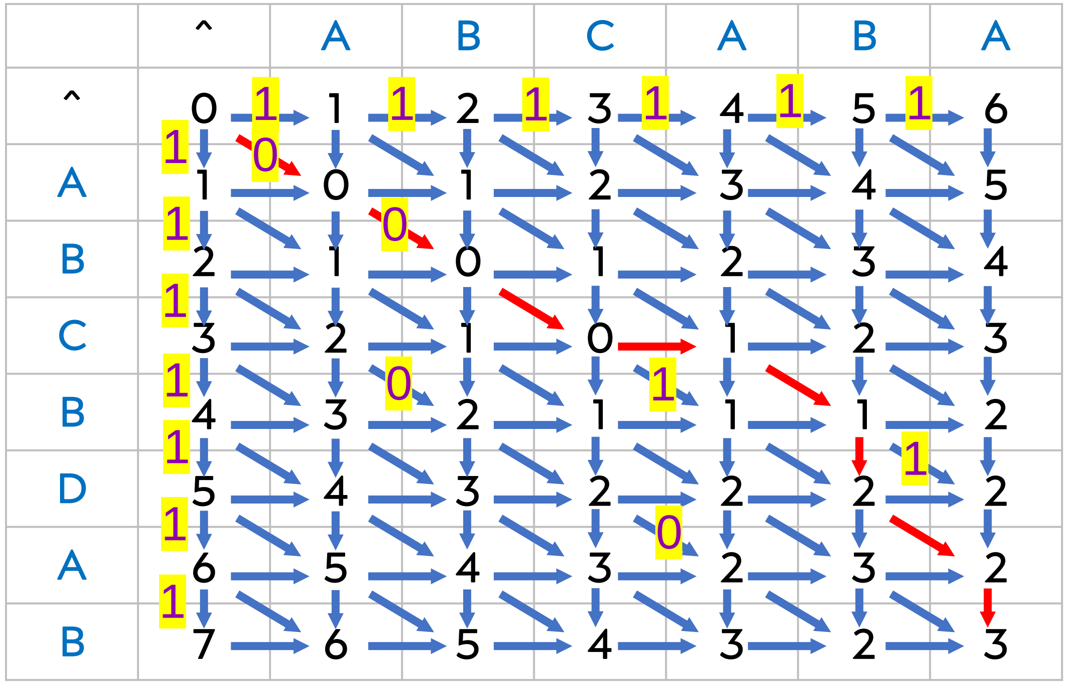
- **Delete C**. Cost = (cost of transforming AB => BDCAB) + 1 → s[i-1, j] + 1
- **Adding B**. Cost = (cost of transforming ABC => BDCA) + 1 → s[i, j-1] + 1
- **Editing C to B**. Cost = (cost of transforming AB => BDCA) + 1 → s[i-1, j-1] + 1
- Use the min of the above three!

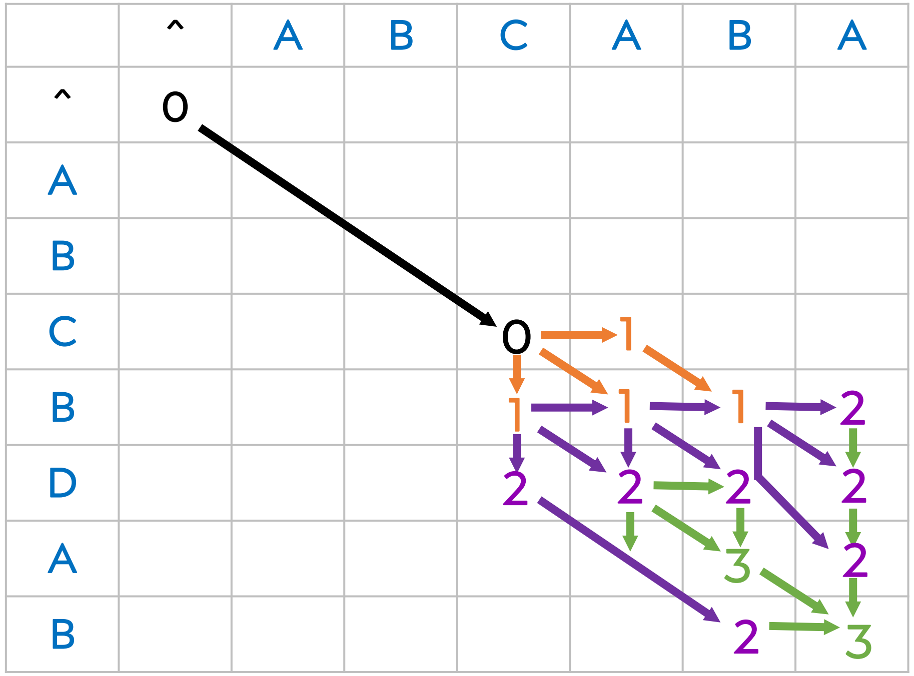
Recurrence Relation

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

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

Edit Distance and BFS





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

What is the time complexity of this algorithm?

Summary for Dynamic Programming

Dynamic Programming (DP)

- **Looks hard ☹ it usually takes a long time for you to understand it**
- **But once you understand it, you suddenly know how to solve a huge class of problems!**
  - E.g., LCS and edit distance are very similar, all knapsack problems are very similar, …
- **We will summarize again at the end of all four lectures on DP**
- **And you'll find out they are easy: usually correctness is straightforward**
  - For all states, we compute the solution based on enumerating all possibilities

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, 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

Dynamic programming and memoization

- **For a given subproblem (uniquely identified by a state), after we calculate the result, memorize it!**
- **Usually just use an array with indexes as your states**
  - E.g., for knapsack, once you know the highest value using 6lb weight, you can memorize it (s[6]), and don't need to compute it again
  - So for 8lb, if we decide to choose a 2lb item, the best result must be highest value using 6lb + value of that 2lb-item

DP on trees

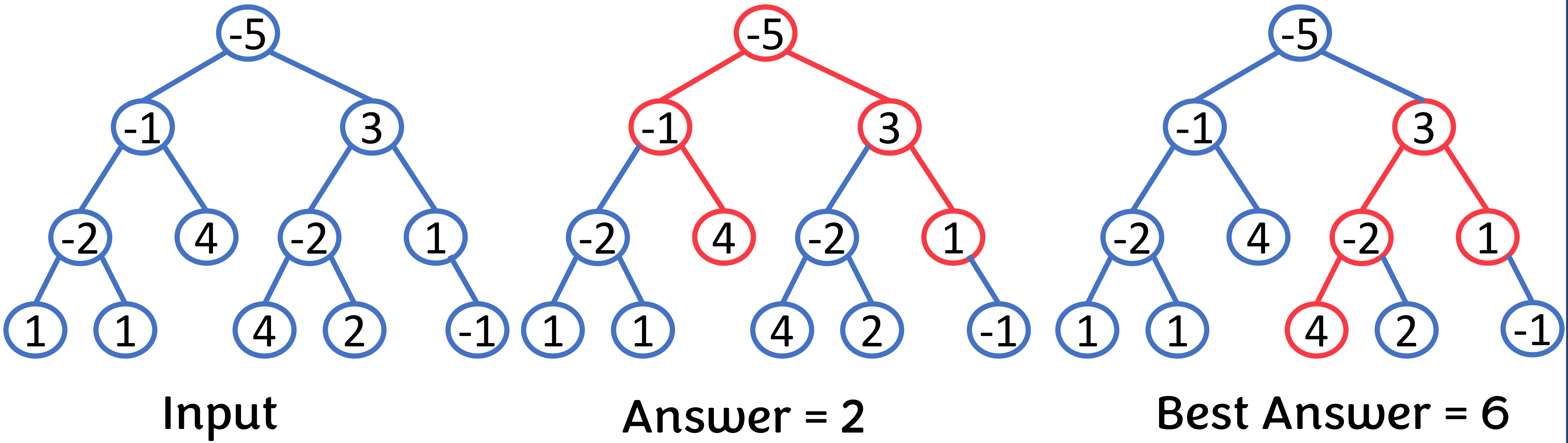
Sometimes we need to deal with a tree structure using dynamic programming

- Well, it's still dynamic programming, but we can use some small tricks for this special case
- Recall that in the previous class, we said that the "dependency" between states cannot form cycles
- Tree structure is totally fine!

- Usually we can start from the top (root) of the tree
- Usually the state of a node can depend on all its children

Recall the interview problem in the first class...

- Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

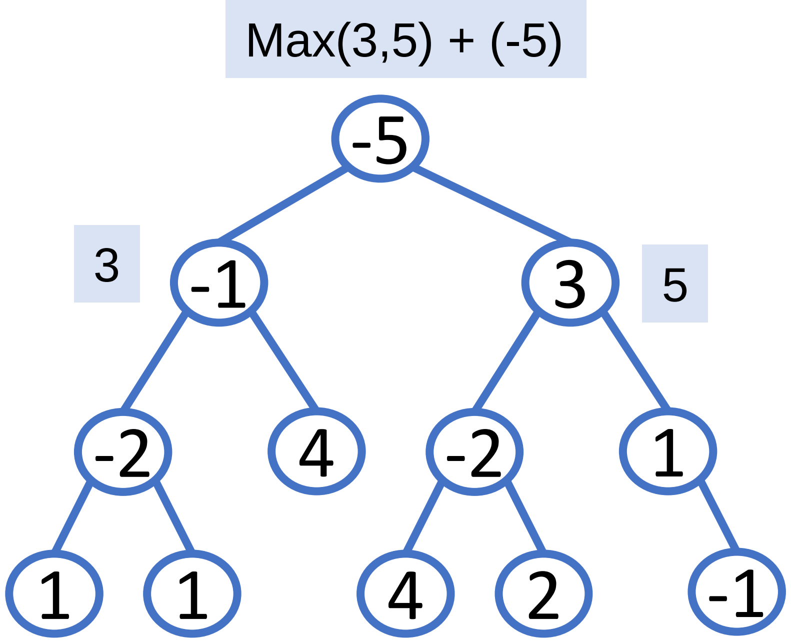


How can we design the state?

- Instead of directly working on the final output, let's define the state as something else…
- Observe: A path first goes up then down
- f[i] = the largest path sum with node i as the topmost node!

- Let j and k be i'th two children
- f[i] = max(f[j] + w[i], f[k] + w[i])

- Is it correct?

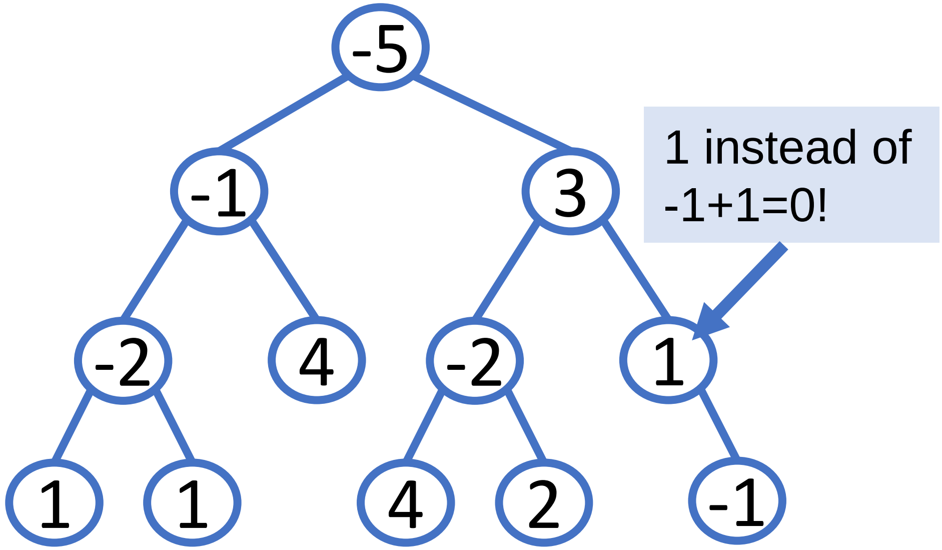


How can we design the state?

- Instead of directly working on the final output, let's define the state as something else…
- Observe: A path first goes up then down
- f[i] = the largest path sum with node i as the topmost node!

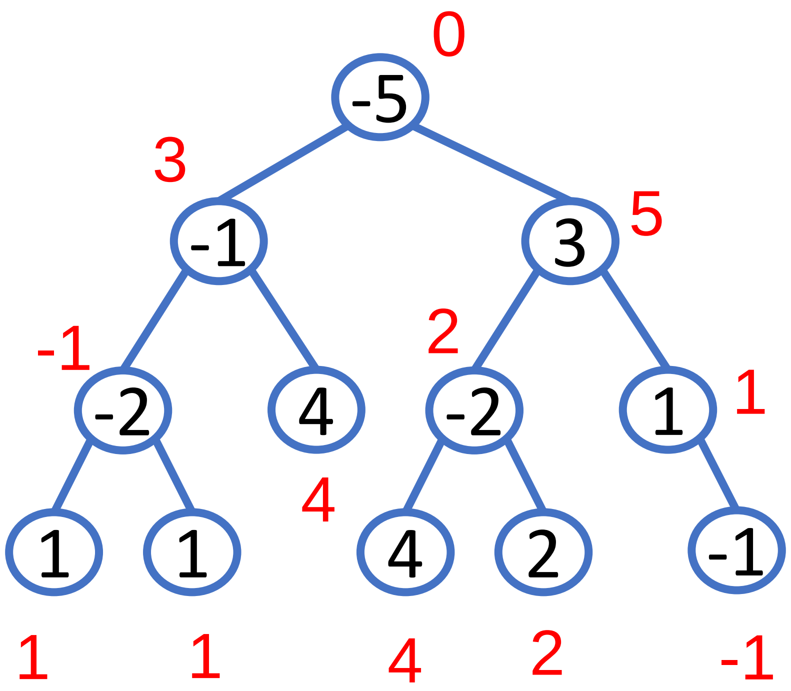
- Let j and k be i'th two children
- f[i] = max(f[j] + w[i], f[k] + w[i], **w[i]**)

- Must consider all cases:
  the path can be just i!



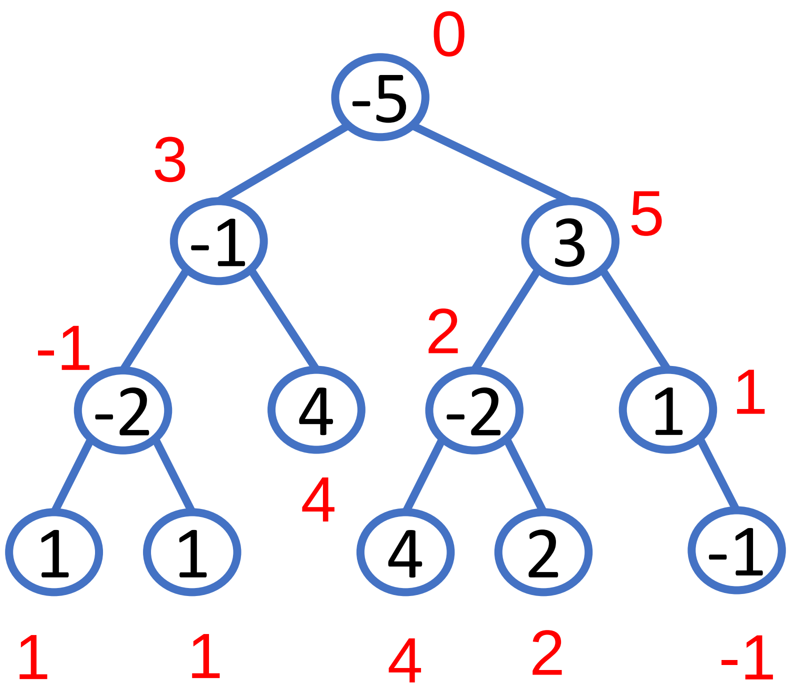
How can we design the state?

- f[i] = max(f[j] + w[i], f[k] + w[i], **w[i]**)



How can we design the state?

- With f[i], we can enumerate all nodes as the "shallowest" node




```
ans = -infty
foreach tree node i {
  let j and k be its two children;
  ans = max(ans, f[j]+f[k]+w[i]);
}
Output ans
```

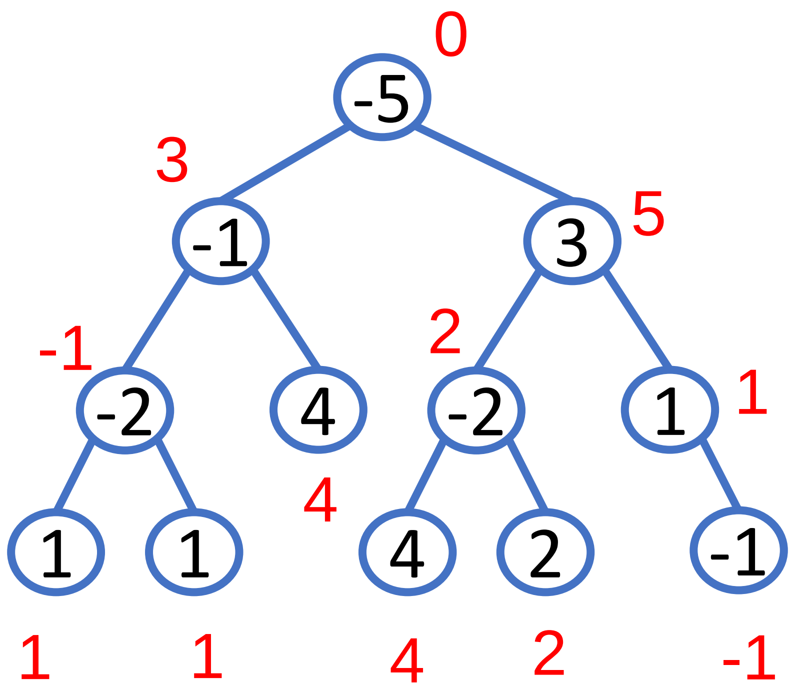
Is this correct?

Let j and k be the two children of i, the best path across node i is:

f[j] + f[k] + w[i]

How can we design the state?

- **Again, consider all cases! Maybe it only contains one side of the branch!**




```
ans = -infty
foreach tree node i {
  let j and k be its two children;
  ans = ……
}
Output ans
```

A simpler solution: allow f[i] to be max(f[i], 0) (you can think about how to do this, and potential issues of doing this)

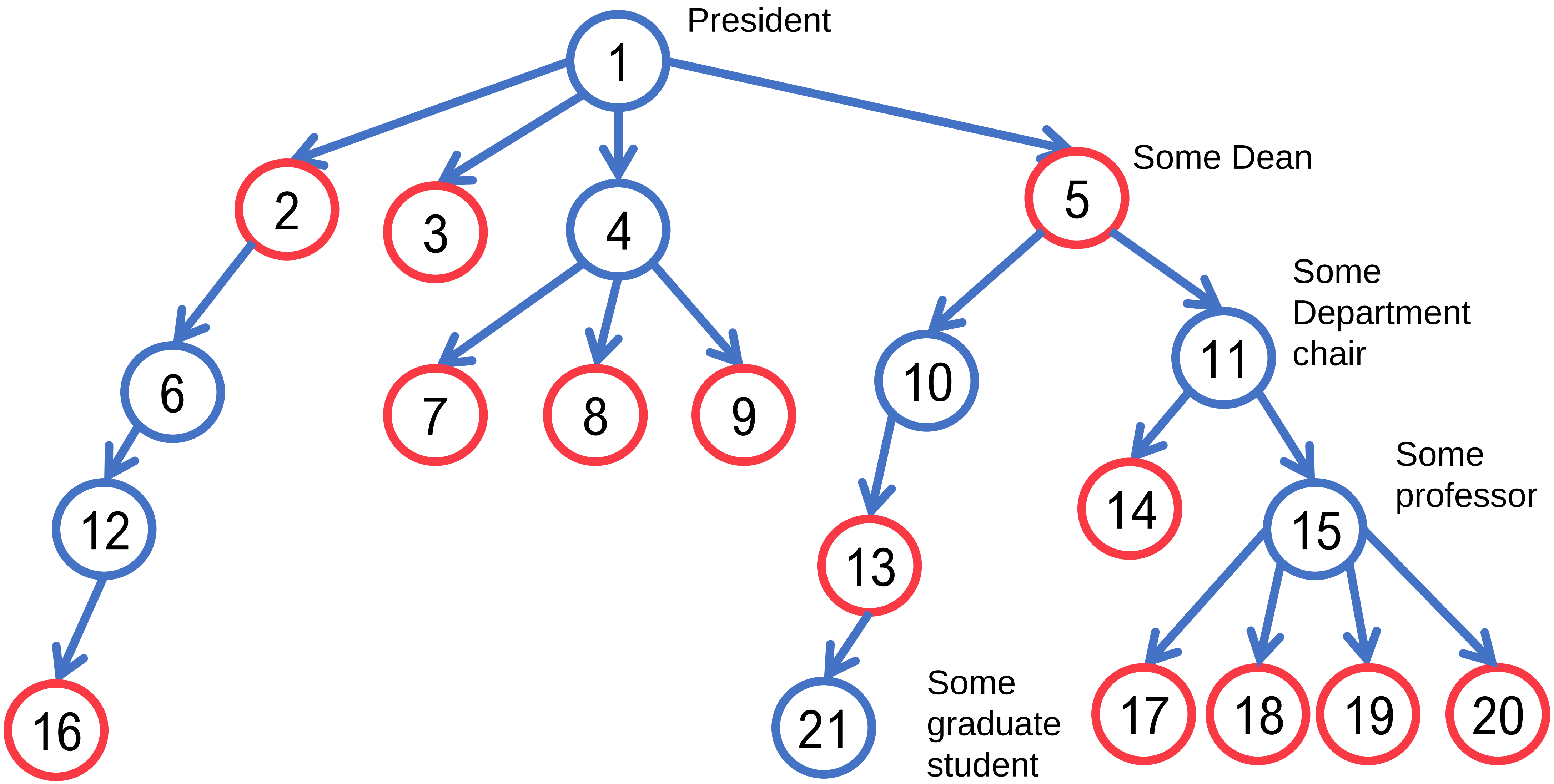
Let j and k be the two children of i, the best path across node i is:

Max(f[j] + f[k] + w[i], w[i], w[i]+f[j], w[i]+f[k])

Example: no-boss party

- In UCR, every employee has one direct boss
- All employees can be represented as a tree structure: every employee is represented as a tree node, and its parent is his/her direct boss
- Now we want to invite a subset of the employees to a party, but no one wants to join the party with his/her direct boss
- What is the maximum number of participants we can invite to the same party?

Example: no-boss party



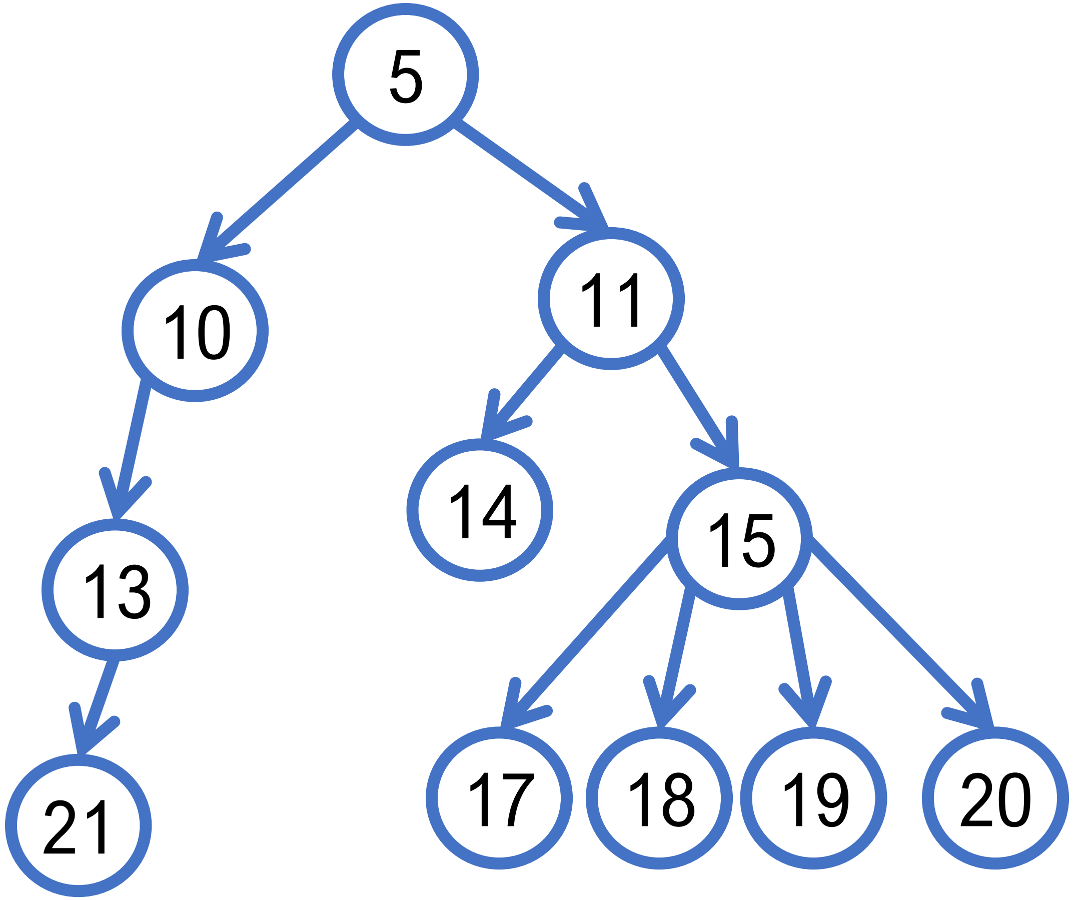
No-boss party

- **We can use $f[i]$ to denote the largest number of nodes we can choose from i's subtree**
  - f[i] should be computed using all f[j] for all its children j

- **But how can we make sure a node is never selected with its parent?**

- **Add! Another! Dimension!**

$f[5] = ?$
It should be computed from $f[10]$ and $f[11]$

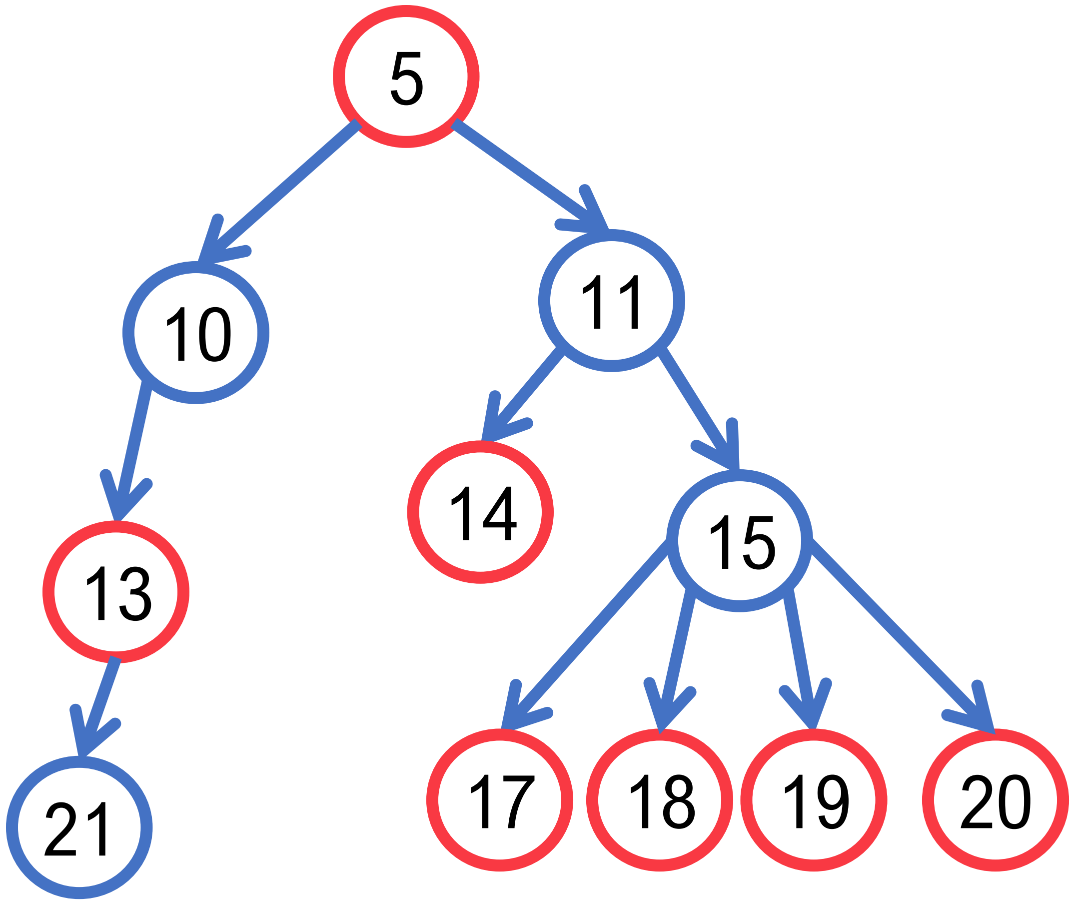


No-boss party

$$f[5,1] = 1 + f[10,0] + f[11,0]$$

- $f[i, 0]$ = the maximum number of people we can invite, if we don't invite i
- $f[i, 1]$ = the maximum number of people we can invite, if we invite i

- $f[i, 1] = 1 + \sum_{j \in child(i)} f[j, 0]$
  - If we invite i, we cannot invite any of its children
  - For each subtree j, the best solution is of course the maximum number of participants in j's subtree without j

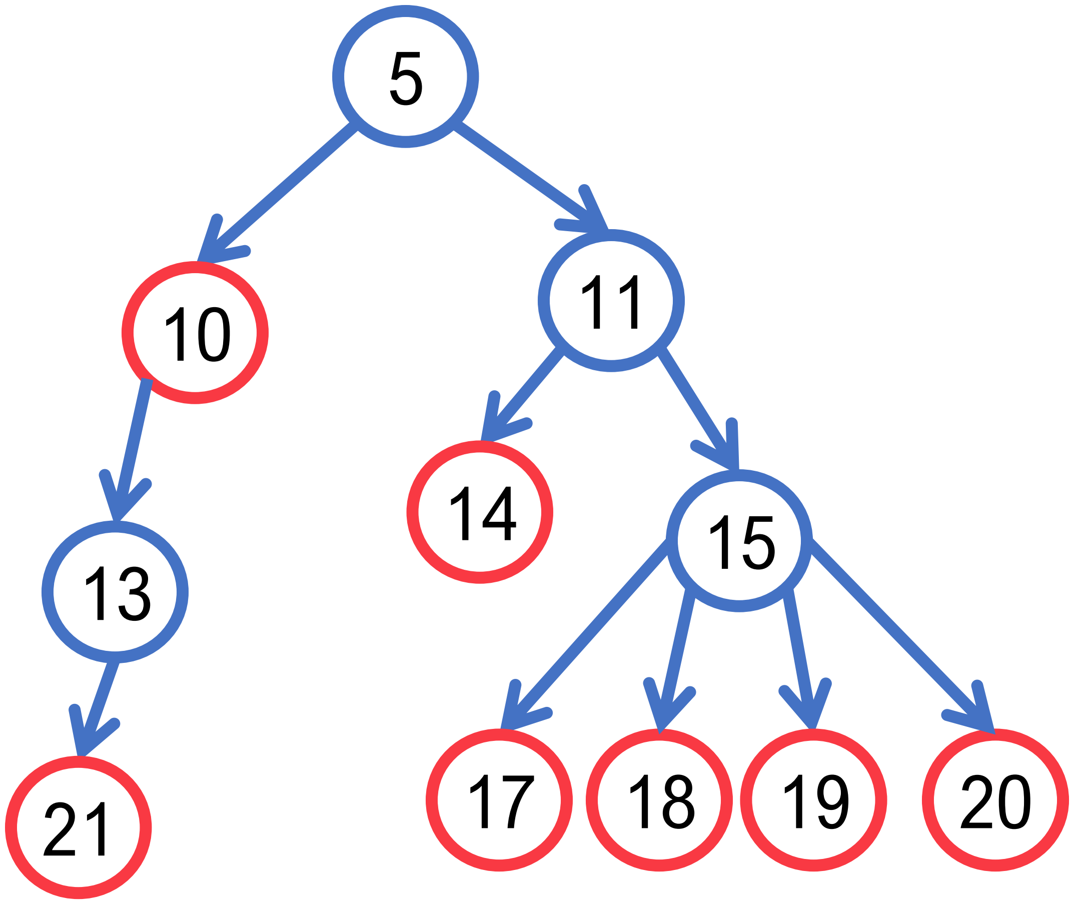


No-boss party

- $f[i,0]$ = the maximum number of people we can invite, if we don't invite i
- $f[i,1]$ = the maximum number of people we can invite, if we invite I

- $f[i,0]$ = $\sum_{j \in child(i)} \max(f[j,0], f[j,1])$
  - If we don't invite i, we can either invite its children or not
  - For each subtree j, the best solution is of course the better solution between if we invite j or not

$$f[5,0] = \max(f[10,0] + f[10,1]) + \max(f[11,0], f[11,1])$$



No-boss party: algorithm

- $f[i,1] = 1 + \sum_{j \in child(i)} f[j,0]$
- $f[i,0] = \sum_{j \in child(i)} \max(f[j,0], f[j,1])$
- **Base case:** $f[i,0] = 0$ **and** $f[i,1] = 1$
- **An easy way: memorization**
  - Start from the root, traverse the tree until the leaves

- **A non-recursively way: decide the order based on the height**
  - First compute the f[ ] value for all leaves (height 1)
  - Then all nodes with height 2
  - Then height 3
  - …

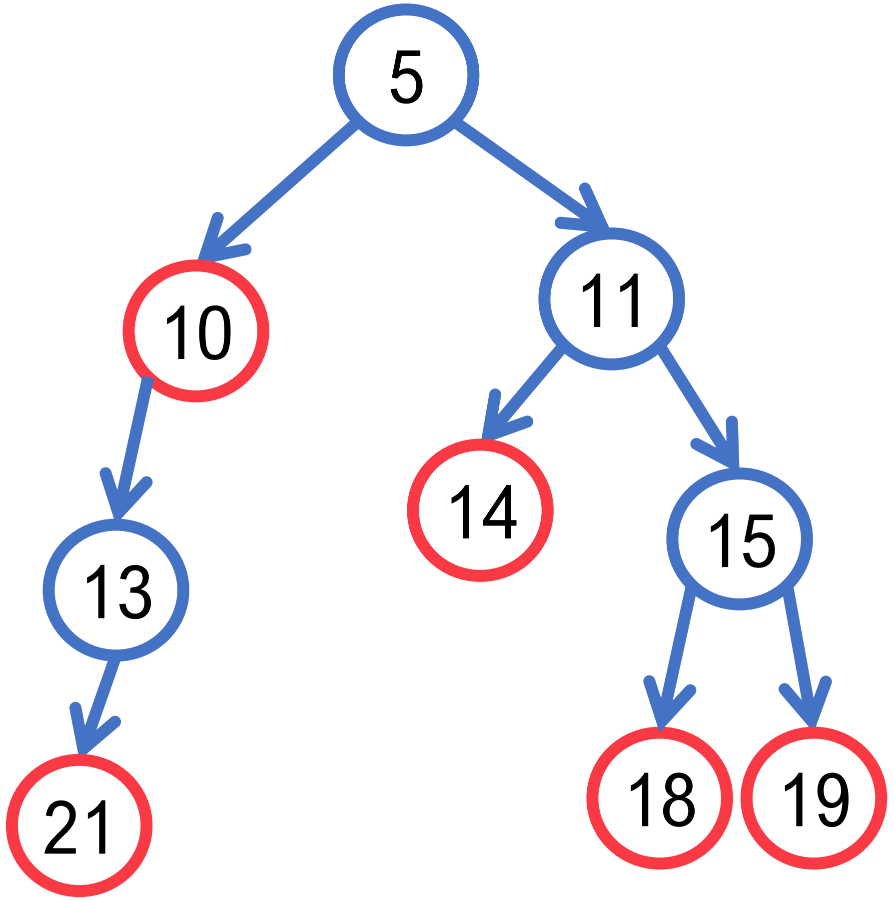
No-boss party: other variants

- If each node has a value $v[i]$, we want to maximize total value of selected people
- $f[i,0]$ is max value of i's subtree with i, and $f[i,1]$ is max value of i's subtree without i
- $f[i,1] = \boldsymbol{v[i]} + \sum_{j \in child(i)} f[j,0]$
- $f[i,0] = \sum_{j \in child(i)} \max(f[j,0], f[j,1])$
- Base case: $f[i,0] = 0$ and $\boldsymbol{f[i,1] = v[i]}$

No-boss party: other variants

$$f[5,k,1] = v[5] + \max_{k_1+k_2=k-1} f[10,k_1,0] + f[11,k_2,0]$$

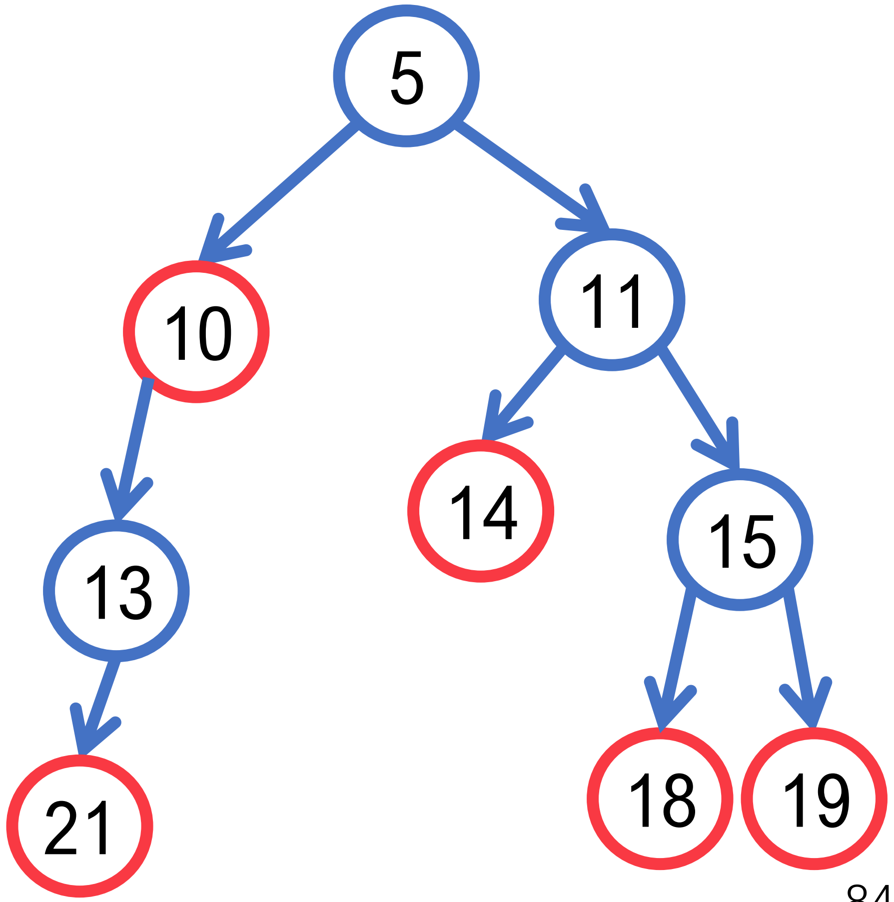
- If we can **only choose m people**
- If each node has a value $v[i]$, we want to maximize total value of selected people
- $f[i,k,1/0]$ is the max value of i's subtree if we select k people with/without selecting i
- $f[i,k,1] = v[i] +$ (select k-1 people from all its subtrees, but not choosing its children), i.e., transit from $f[j,*,0]$



No-boss party: other variants

$$f[5,k,0] = \max_{k_1+k_2=k}(\max(f[10,k_1,0], f[10,k_1,1]) + \max(f[11,k_2,0], f[11,k_2,1]))$$

- If we can **only choose m people**
- If each node has a value $v[i]$, we want to maximize total value of selected people
- $f[i,k,0]$ = select k people from all its subtrees
- How to compute "select k people of all its subtrees"?
  - This is a knapsack problem!
  - Try to figure out the details: see the homework problem (that's a must-have-a-boss party)



DP for trees

- **Usually we can start from the top (root) of the tree**
- **Usually the state of a node can depend on all its children**

- **Sometimes we can use another dimension for some additional state**
  - f[i, 0/1] for the i's subtree with choosing/not choosing the current subtree root
  - f[i, k] for the i's subtree with choosing k elements in this subtree