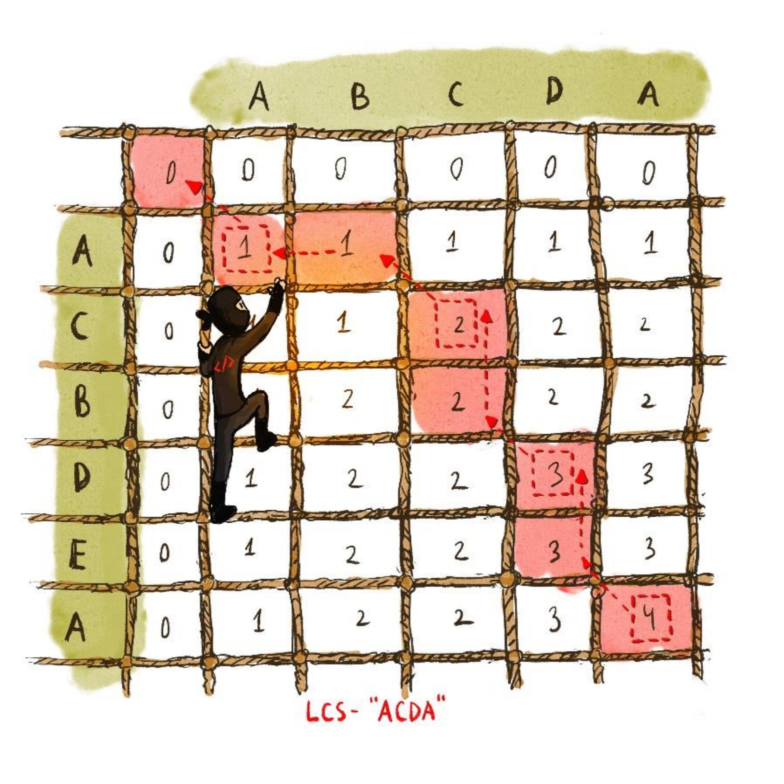


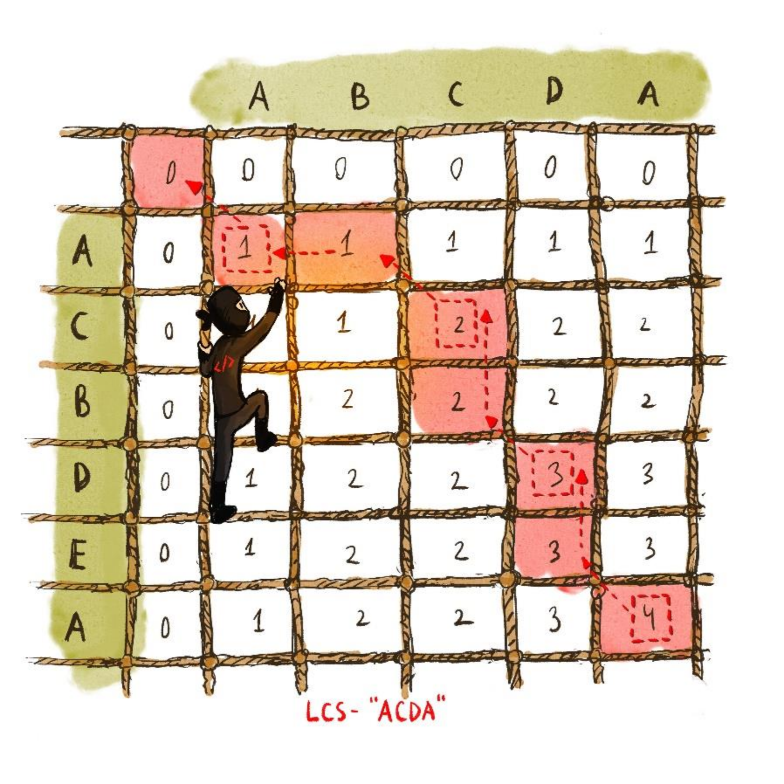
Dynamic Programming

Yan Gu

Class announcement

- **We do not know your emergency situations (e.g., an active COVID case), so if you think you need to let us know, please inform us via an email, on campuswire, or on a private message on the slack channel**

- **Midterm: 6:30-8:30pm Oct 28, MSE 116**
- **Similarly, if you have a conflict, please inform us via email/campuswire/slack since we need to your information to discuss more details**
- **Basically if you need to start a bit earlier or later (like between 6-7pm), that's easy to handle (but you need to let us know)**
- **It will be harder if you want to do it in a non-overlapping time: we will review it in a case-by-case manner**
    - It's definitely resolvable, but I don't recommend that personally



Dynamic Programming

Yan Gu

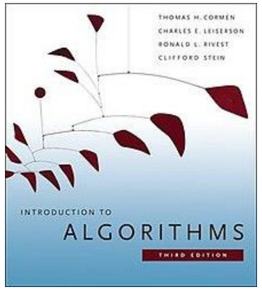
What cannot be solved by greedy strategies?

- Different candies have different "values" (say, how much you like them)
- With a fixed budget of S dollars, how to maximize the total value?
- No known greedy algorithm can solve this ☹
- A lot of variants: 0/1 knapsack, unlimited knapsack, k-knapsack, …

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 simplified case: unlimited knapsack

- Overall weight limit: 8 lb, we can take an unlimited number of each item
- Item 1: 5 lb, $150
- Item 2: 4 lb, $100
- Item 3: 2 lb, $10

- Expensive first: Item 1 + Item 3, value: $160
- Lightest first: Item 3 * 4, value $40
- Optimal: Item 2 * 2, value: $200

- Greedy strategy does not provide the optimal solution
- A naïve solution? Try all possibilities!

A naïve algorithm

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

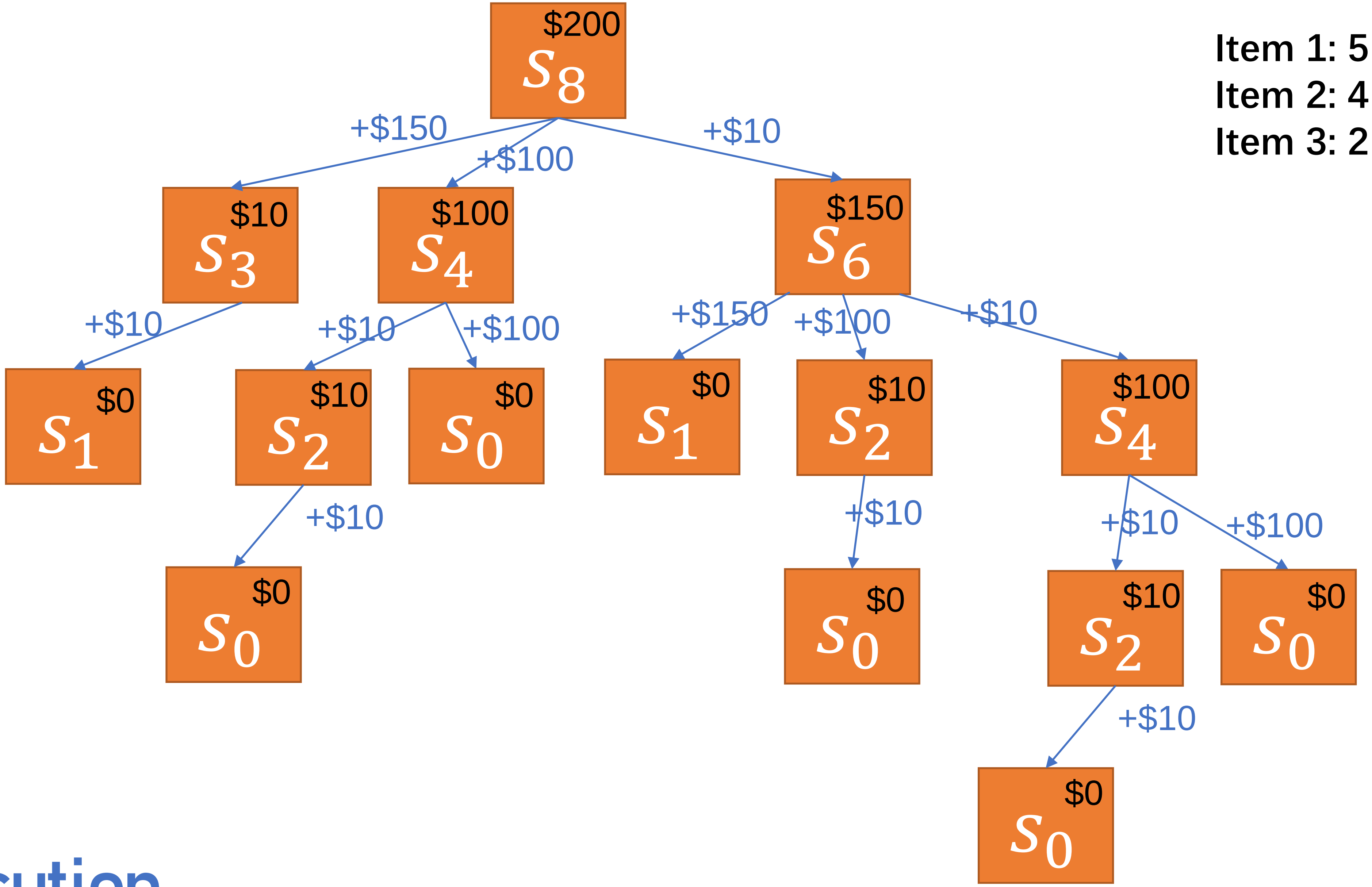
suitcase(8):
Case 1: first put item 1,
total value = suitcase(3) + 150
Case 2: first put item 2,
total value = suitcase(4) + 100
Case 3: first put item 3,
total value = suitcase(6) + 10
Best = max of the above three

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

answer = suitcase(8);
```

Recursive call

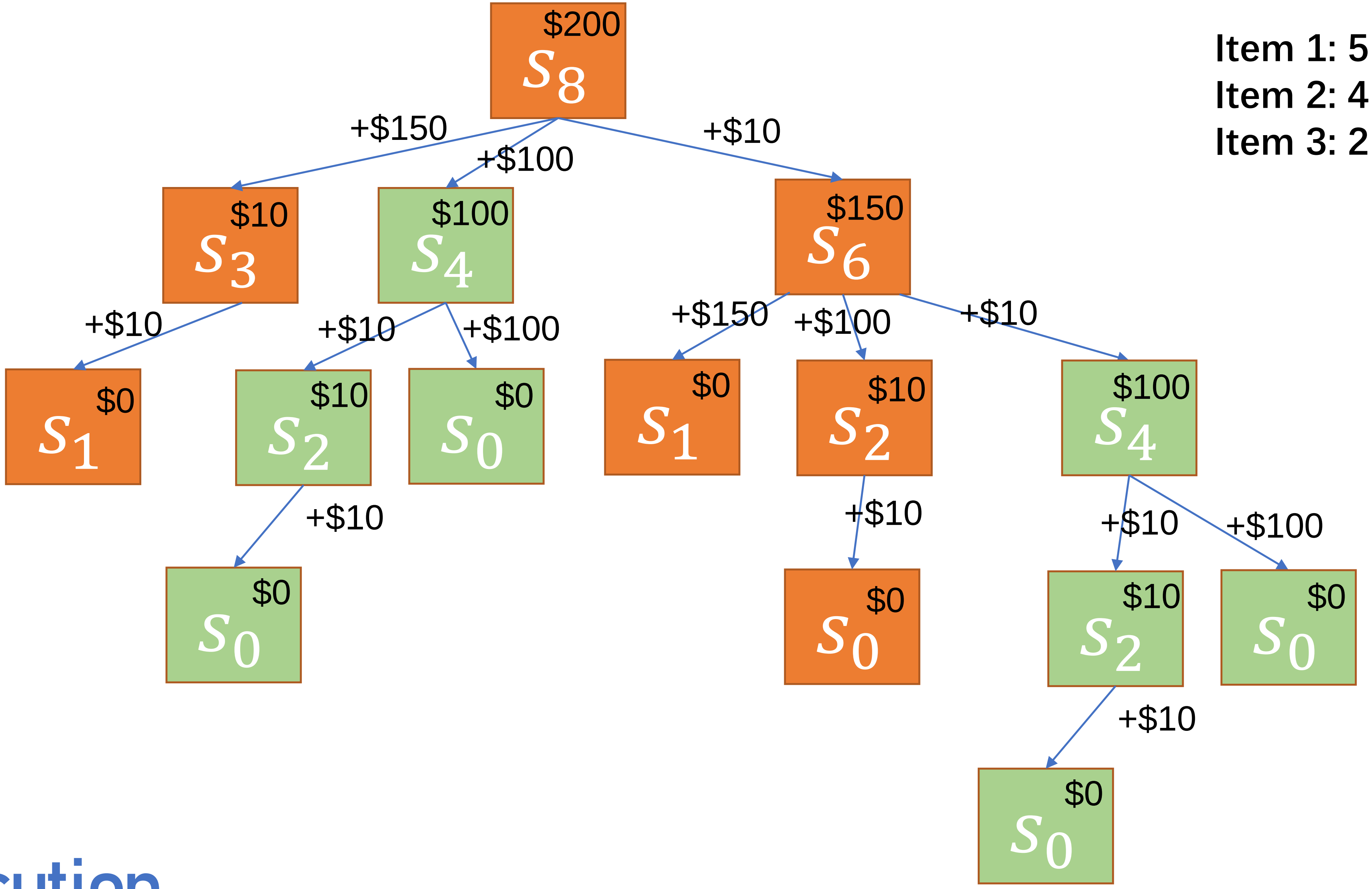
This algorithm takes exponential time, and only works for very small instances



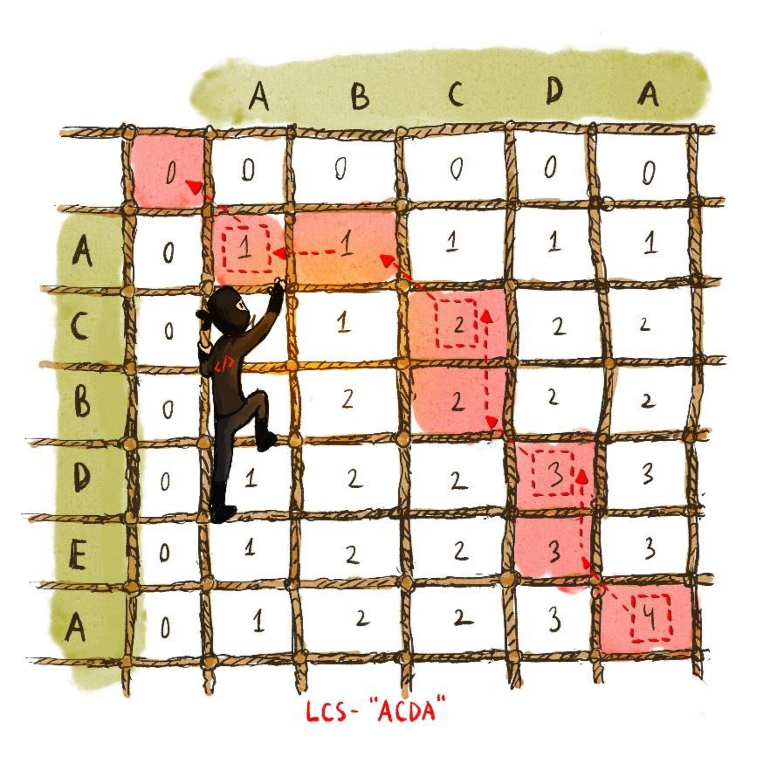
Item 1: 5 lb, $150
Item 2: 4 lb, $100
Item 3: 2 lb, $10

Execution Recurrence Tree

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



Execution Recurrence Tree



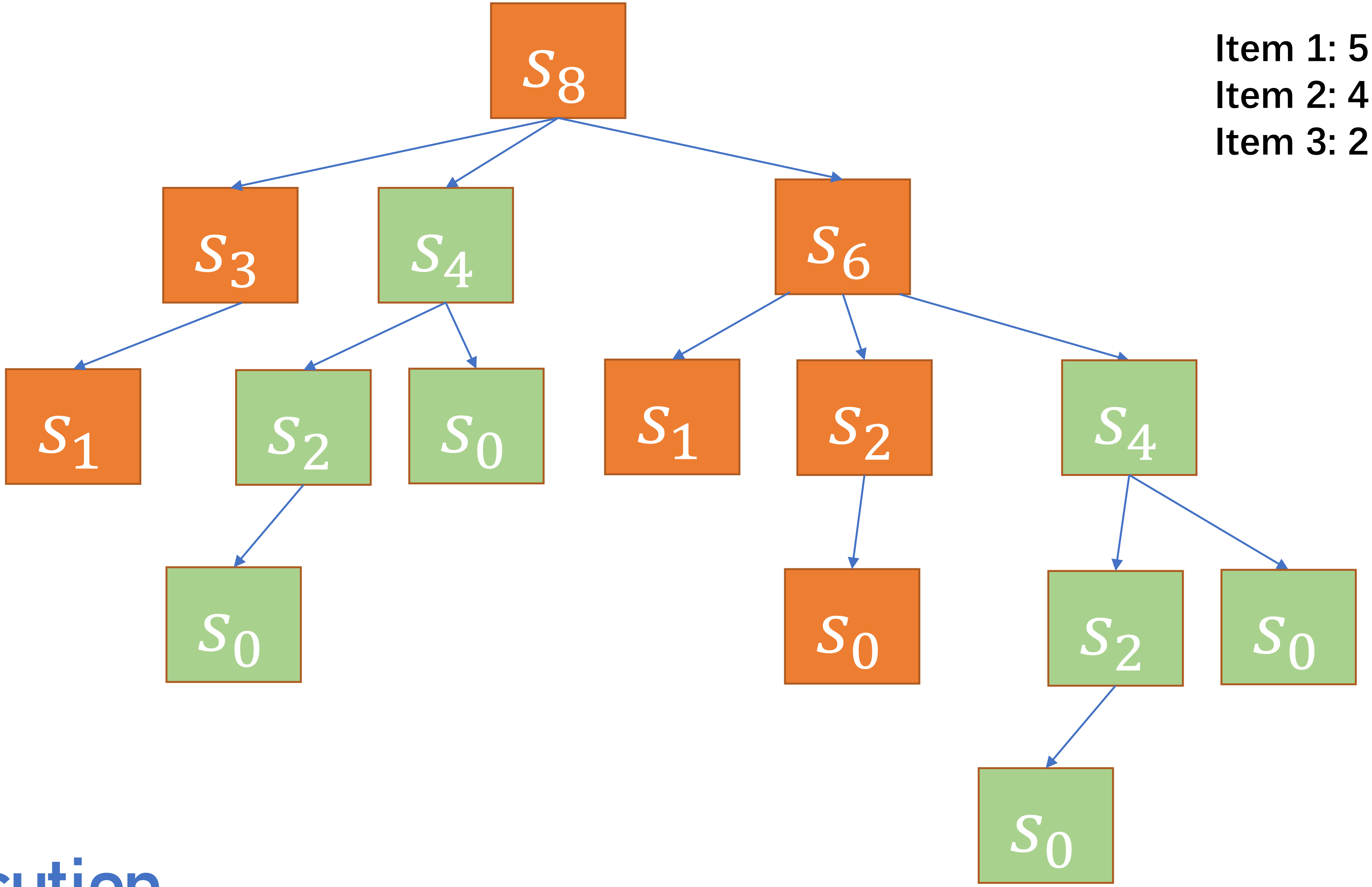
Next 3.5 lectures:

Dynamic Programming

Programming?

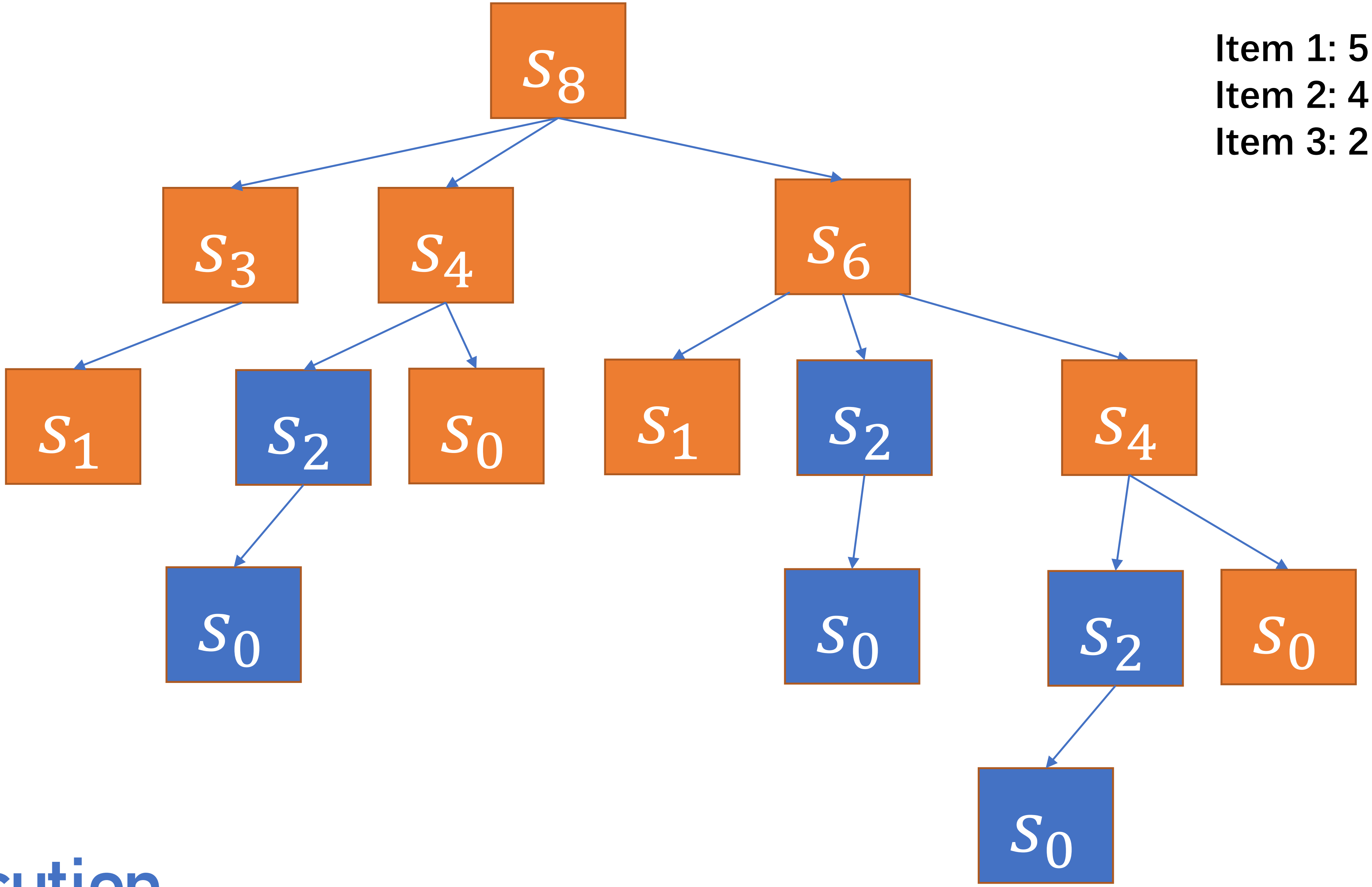
- **Program (noun) \ ˈprō-ˌgram , -grəm **
  - a sequence of coded instructions that can be inserted into a mechanism (such as a computer)
- **Programming (noun) \ ˈprō-ˌgra-miŋ , -grə-**
  - a plan of action to accomplish a specified end

- **In dynamic programming, or linear programming, the word programming means a “tabular solution method”**
  - In fact, the concept of dynamic programming was proposed before computers, and was a subarea of operating research
  - Without a computer and memory, you have to write down the intermediate results on a piece of paper, and in a table



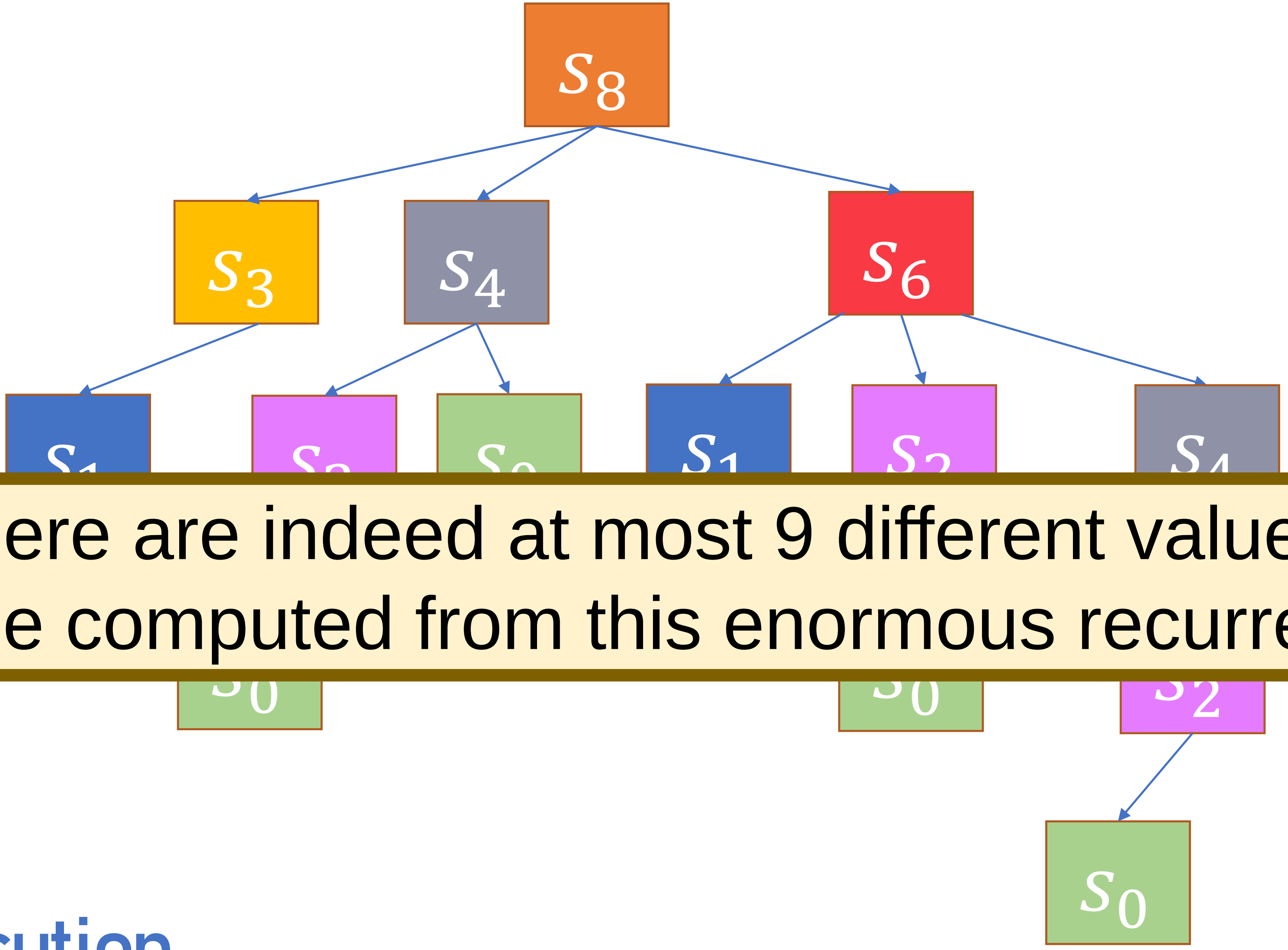
Item 1: 5 lb, $150
Item 2: 4 lb, $100
Item 3: 2 lb, $10

Execution Recurrence Tree



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

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

Execution Recurrence Tree

A naïve algorithm

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

answer = suitcase(8);
```

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);
```

A naïve algorithm

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

answer = suitcase(8);
```

$$s_i = \max \begin{cases} 0 \\ \max\limits_{(w_j, v_j) \text{ is an item}} \left\{ s_{i-w_j} + v_j \right\} \mid i \geq w_j \end{cases}$$

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 weight $\boldsymbol{i - w_j}$

$$s_i = \max \begin{cases} 0 \\ \max\limits_{(w_j, v_j) \text{ is an item}} \left\{ s_{i-w_j} + v_j \right\} \mid i \geq w_j \end{cases}$$

Trying all possible items

- **Final answer is s_k ($\boldsymbol{k}$ is the weight limit)**

Optimal Substructure

- **The problem is to find the optimal value s_i for a total weight of i**
- **What is the best solution with item j?**

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

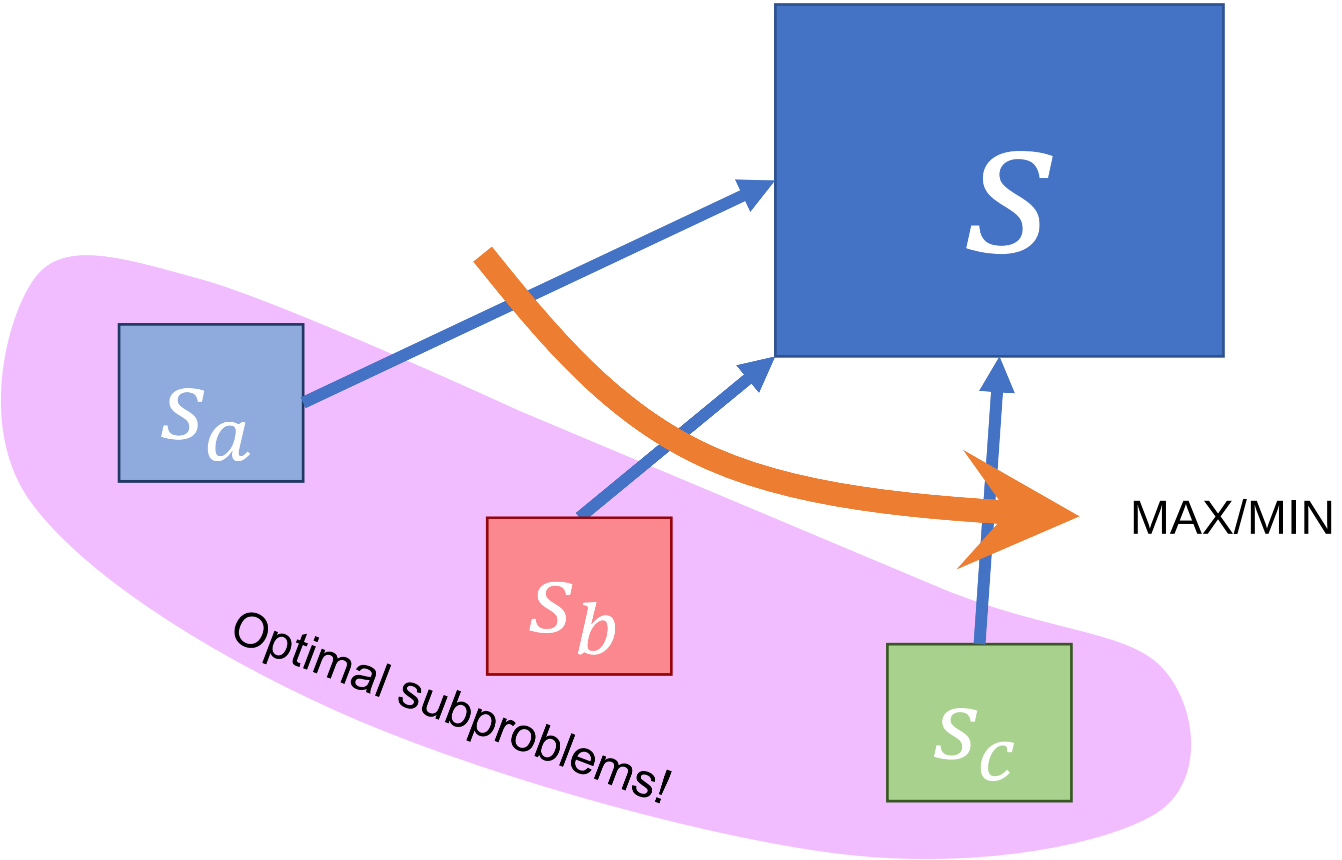
- **The rest must be the optimal solution when we have weight limit $i - w_j$!**
- **The same problem with new weight limit $i - w_j$!**
- **That's s_{i-w_j}! (optimal substructure again!)**

What is dynamic programming?

$$s_i = \max \begin{cases} 0 \\ \max\limits_{(w_j, v_j)\text{ is an item}} \left\{ s_{i-w_j} + v_j \right\} \mid i > w_j \end{cases}$$

The goal is to avoid computing any s_i multiple times!

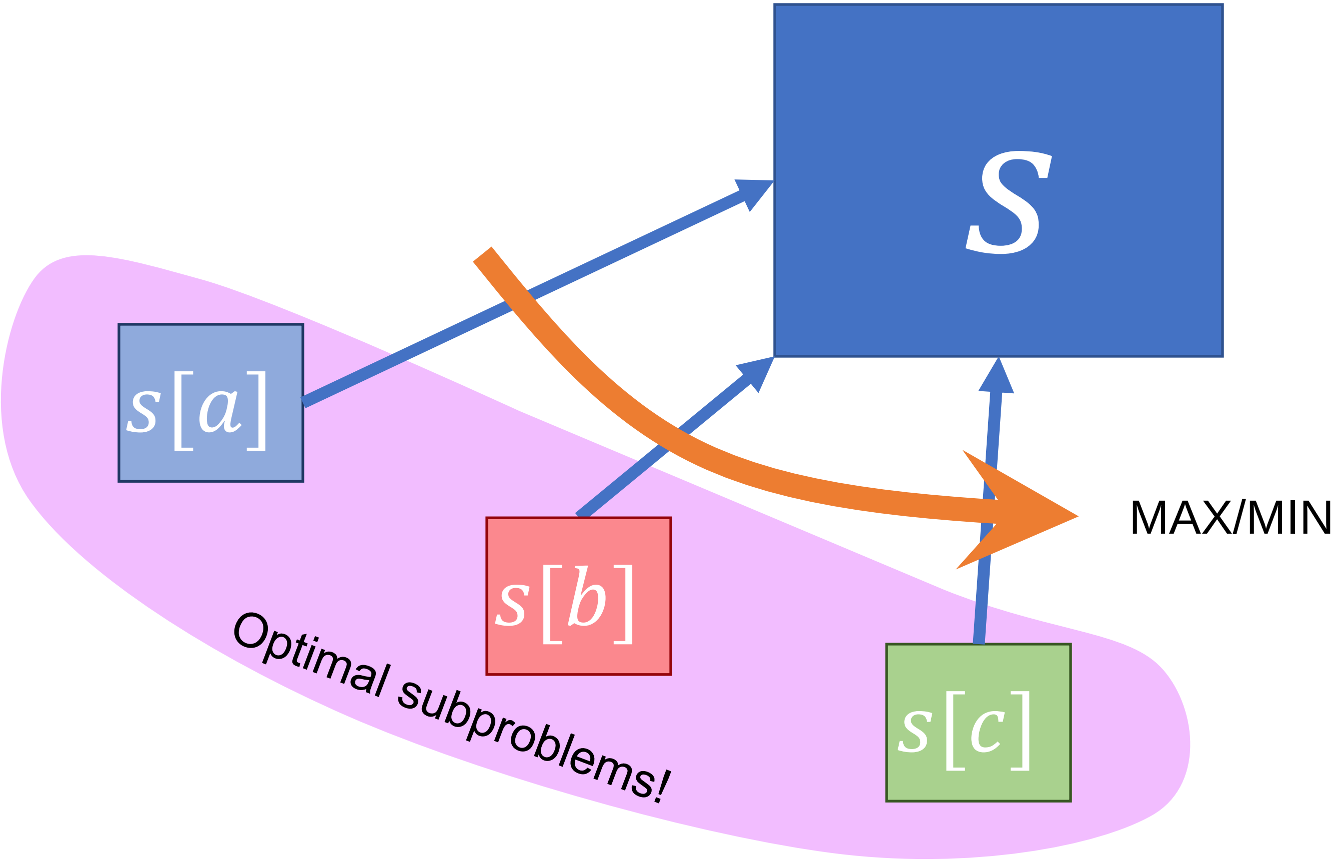
How can we “memorize” the values of s_{i-w_j}?



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!

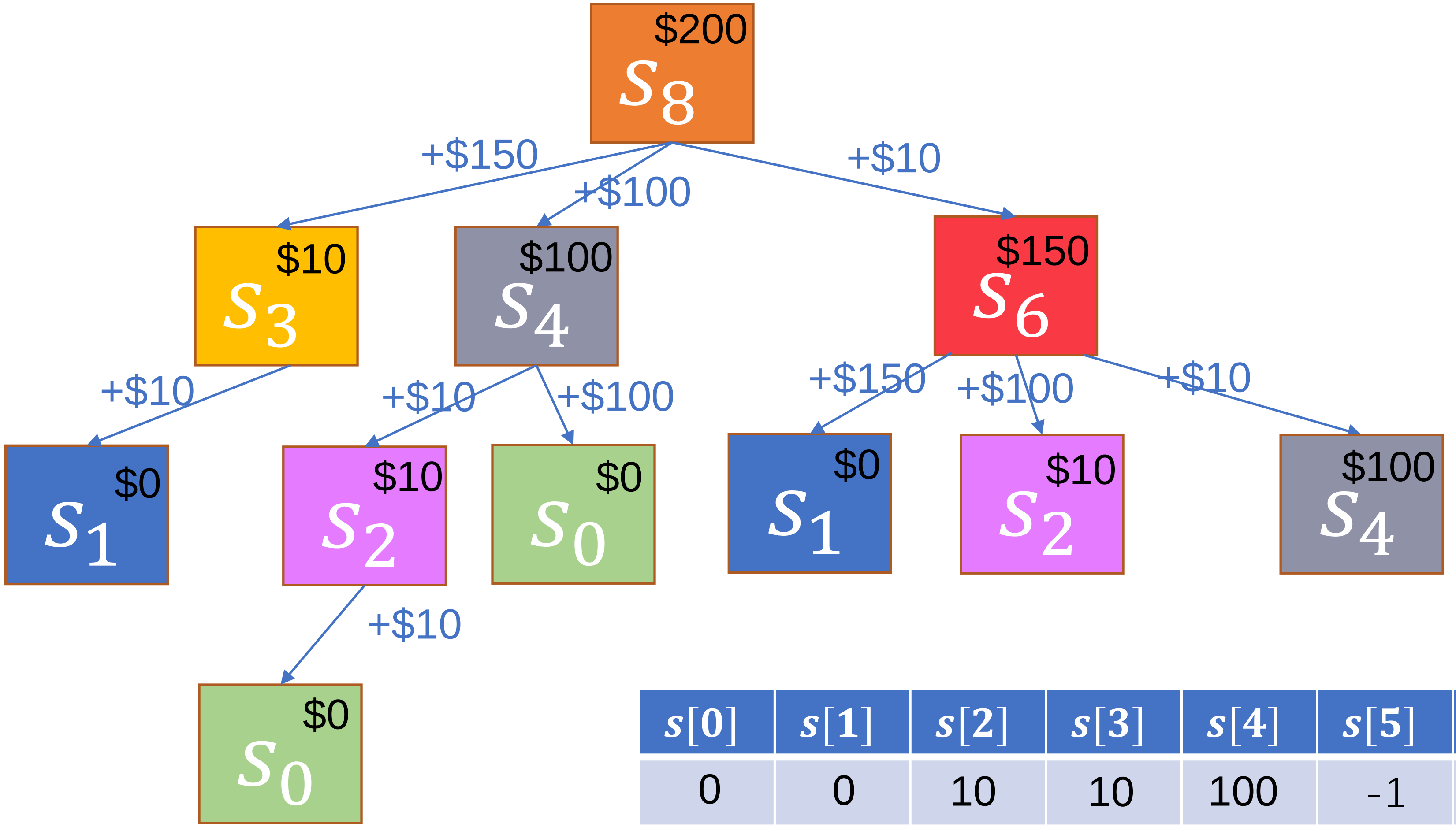


A DP algorithm

```
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

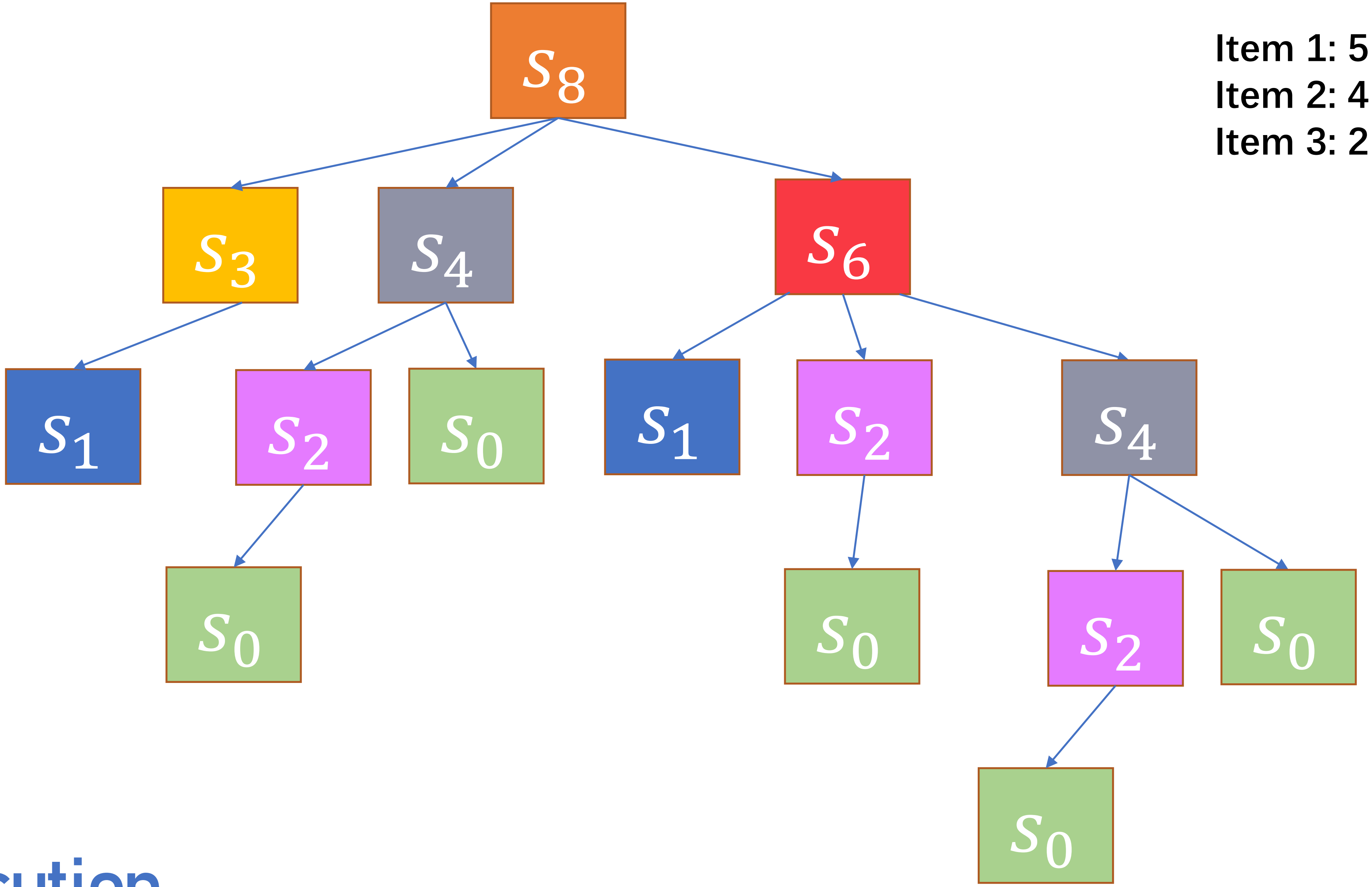
$\Theta(nk)$ cost

where n is #items

k is weight limit

Execution Recurrence Tree

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

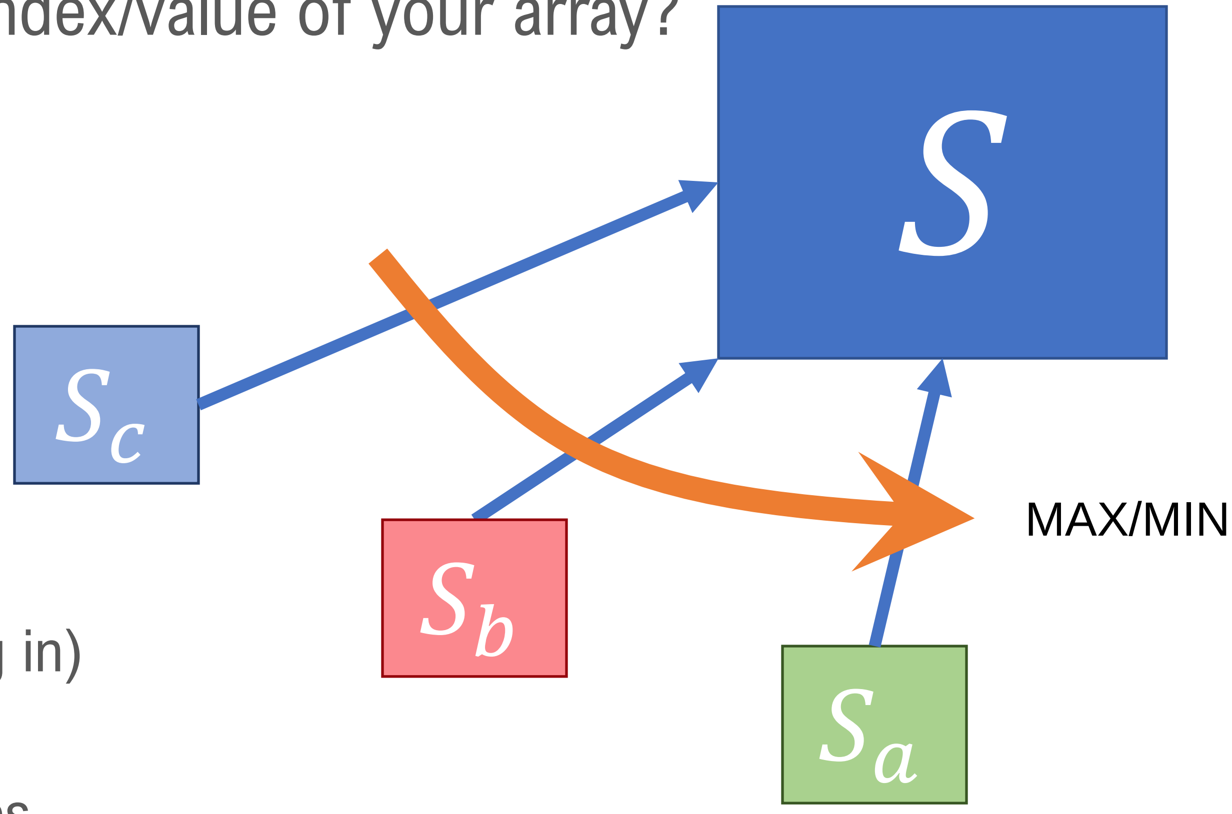


Execution Recurrence Tree

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?
    - $s[i]$ is at least 0 (when we put nothing in)
- **Recurrence**
  - Compute current state from previous states

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

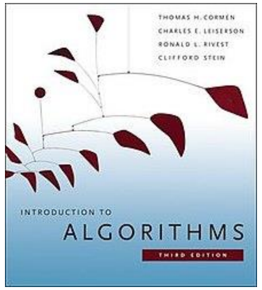


Is the previous solution perfect?

```
int suitcase(int leftWeight) {
    if (ans[leftWeight] != -1) return ans[leftWeight];
    int curBest = 0;
    foreach item of (weight, value)
        if (leftWeight >= weight)
            curBest = max(curBest, suitcase(leftWeight -
            Weight) + value);
    return ans[leftWeight] = curBest;
}
int ans[8] = {-1, … , -1};
answer = suitcase(8);
```

$50, 1lb

$1500, 12lb



$70, 5lb

$80, 2lb

Is this solution still correct if we only allow to use an item once?

```
int suitcase(int leftWeight) {
    if (ans[leftWeight] != -1) return ans[leftWeight];
    int curBest = 0;
    foreach item of (weight, value)
        if (leftWeight >= weight)
            curBest = max(curBest, suitcase(leftWeight -
            Weight) + value);
    return ans[leftWeight] = curBest;
}
int ans[8] = {-1, … , -1};
answer = suitcase(8);
```

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

    $$s[i - w_j, j - 1] + v_j$$

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

    $$s[i, j - 1]$$

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

Recurrence for 0/1 knapsack

- **The recurrence:**

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

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

The DP implementation

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

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

A non-recursive implementation

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

An even simpler implementation

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

- We only need to store a 1D array

The simpler implementation for the first problem

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

- We only need to store a 1D array
- As the bonus question in the assignment, you show that these two implementations are correct

The famous knapsack problem

- Put some items in a weight-limited container under some certain rules, and maximize the total value
- Believe it or not, suitcases (especially with telescopic handle and wheels) are pretty new: history of suitcases
- The first problem is referred to as the unbounded knapsack problem (UKP)
- The second problem is referred to as the 0-1 knapsack problem
- Multiple knapsack problem: item j has x_j copies
- Combinations of the previous three
- Each item has more than one dimension (e.g., both weight and size)

Conclusion for today's lecture

- **We introduce 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 (possibly with stages), 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

Conclusion for today's lecture

- **The best way to improve the understanding of DP is by practicing**
  - A few more examples in the next two lectures
  - We gave you 11 problems in the assignments (5 mandatory and 6 bonus), please try as many as possible
  - I do not use the examples in CLRS, and you can read Chapter 15 in CLRS with new examples, and work on the additional questions in the book