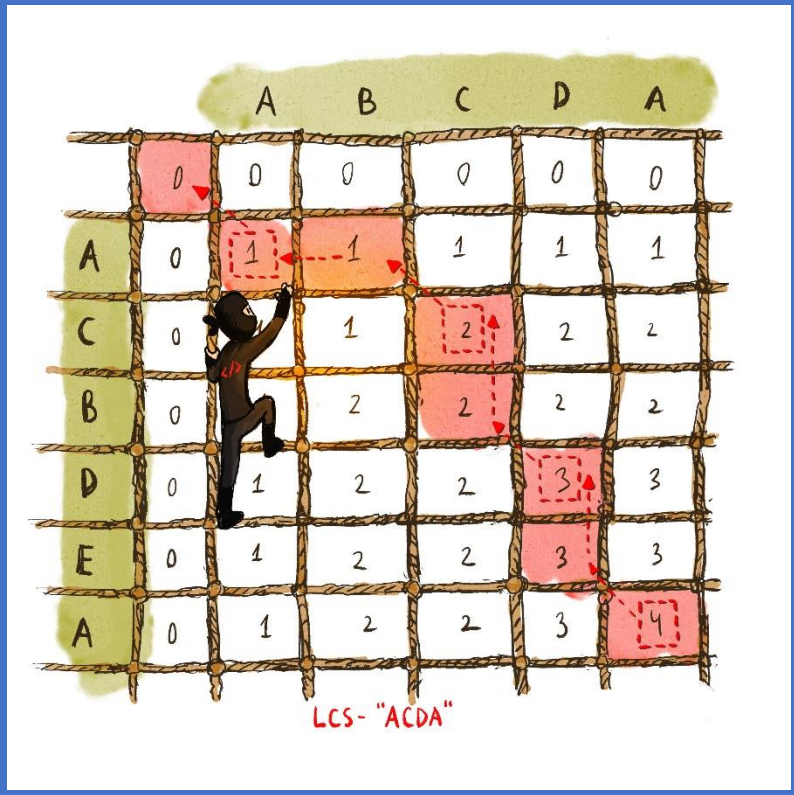
CS 141: Design And Analysis Of Algorithms

Dynamic Programming



Yan Gu

So easy!

- Conversation between a mon 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!
- - Congratulations, you understand dynamic programming now!

So easy!

```
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][k];
```

- We only need to store a 1D array

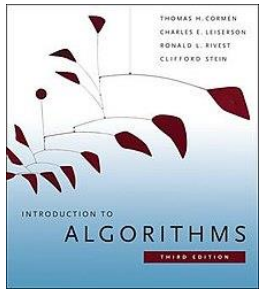
Not so easy?

- **Hard to get the idea by yourself**
  - I personally solved over 100 DP problems before I figured out what DP was
- **However, with appropriate training, most of you can understand it**
  - You need to solve about 20-30 high-quality problems
  - In my 141, we gave out about 12-15 problems, which is not quite sufficient, but about 1/4 of the students end up with having a good understanding
  - We have 4 problems so far (P2 in training 1, P1 in training 2, P1 and P2 in training 3)
  - I attached our 141 homework for DP, which contains 3 more problems
  - Read CLRS and the competition programming handbook for more details
  - Come to talk to us for any questions
- **I have trained about 1000 students so far**

A motivating example: unbounded knapsack

- New quarter starts! You need to bring your luggage back to Riverside
- You need to take a flight, so you can only take one suitcase with limited weight

$50, 1lb



$70, 5lb

$1500, 8lb

$80, 2lb

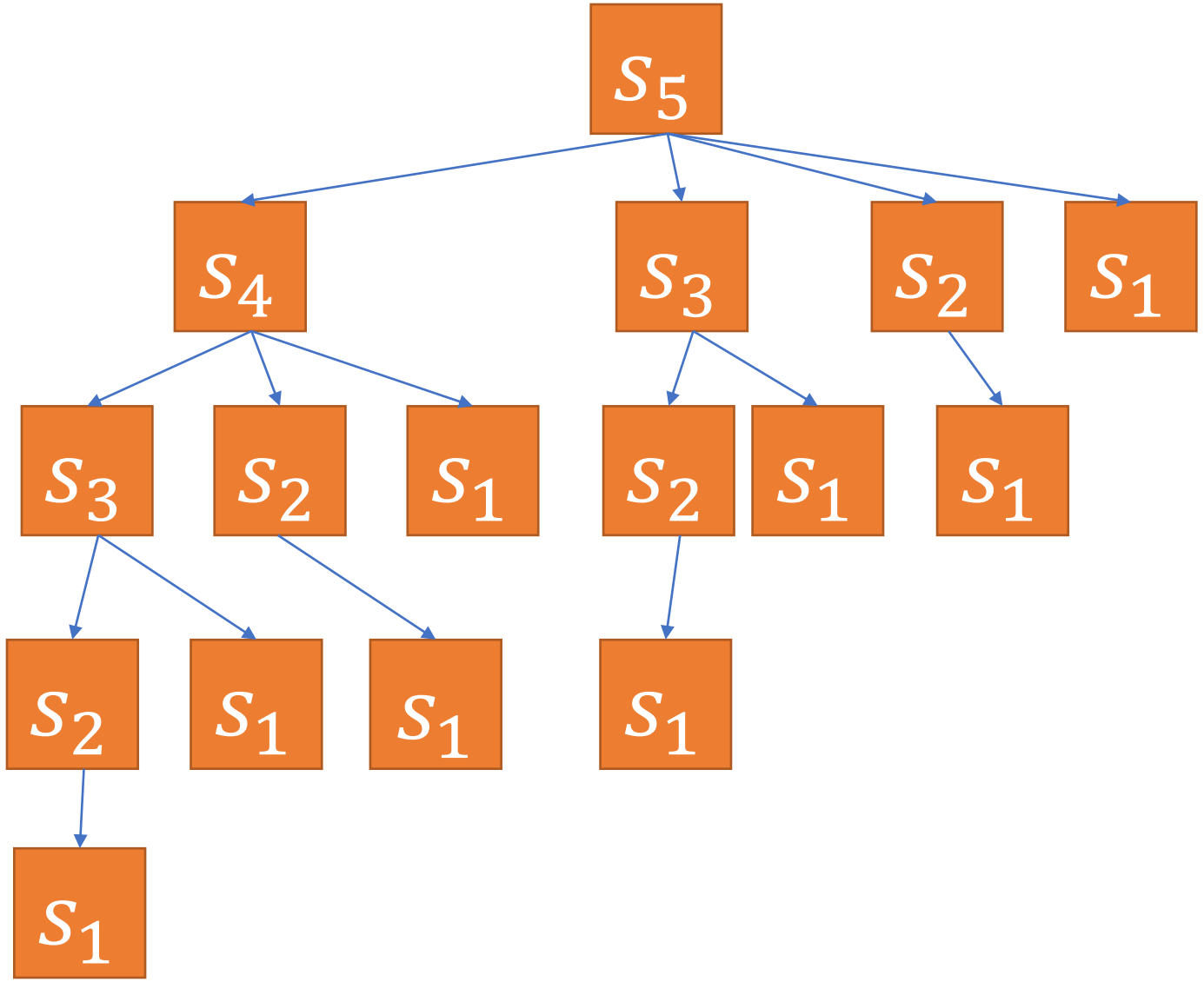
A naïve algorithm

```
//best solution using curWeight capacity
int suitcase(int curWeight) {
  int best = 0;
  foreach item (weight, value)
    if (curWeight >= weight) {    // if we have enough space, try to choose this item
      newVal = suitcase(curWeight - weight) + value
      best = max(best, newVal); //update best value if we get a better solution
    }
  return best;
}

answer = suitcase(5);
```

Exponential time!!

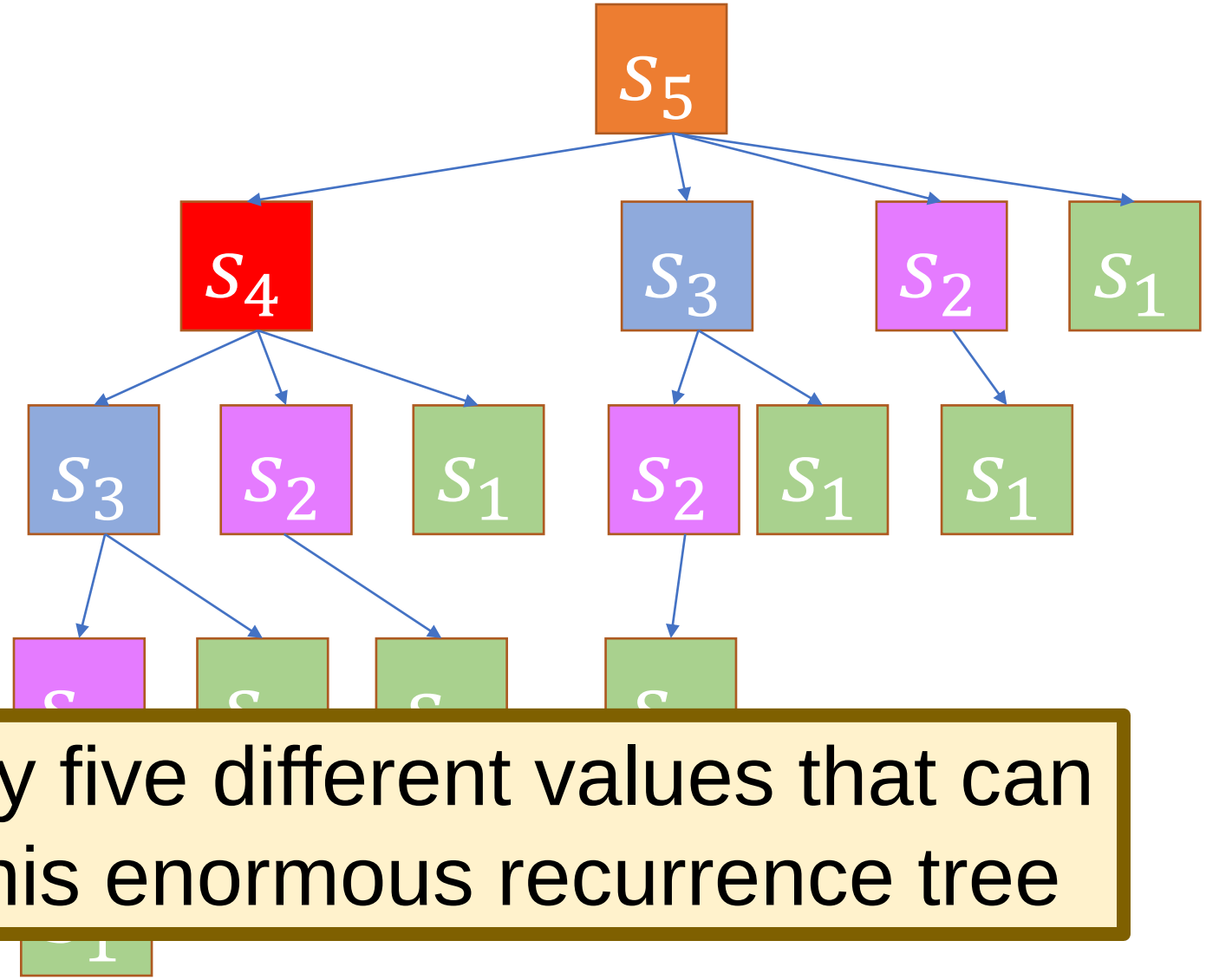
Knapsack problem: native solution

- **Essentially, we are searching all possibilities**
  - Too expensive!
- **Some sub-problems are calculated multiple times?**



Knapsack problem: native solution

- **Essentially, we are searching all possibilities**
  - Too expensive!
- **Some sub-problems are calculated multiple times?**
- **But for each possibility, we can just compute it once!**



There are indeed only five different values that can be computed from this enormous recurrence tree

A naïve algorithm

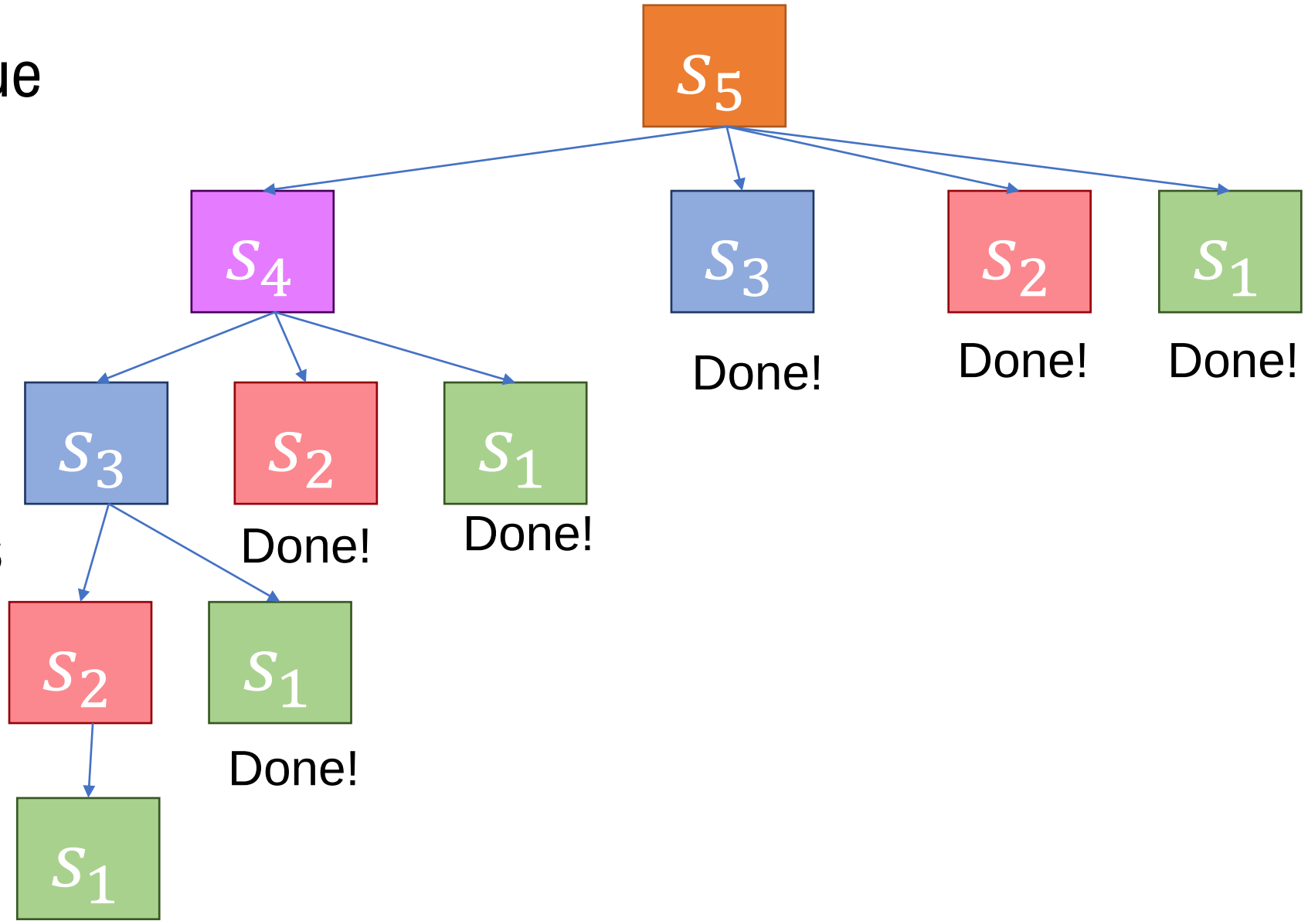
```
int suitcase(int curWeight) {
  int best = 0;
  foreach item (weight, value)
    if (curWeight >= weight) {
      newVal = suitcase(curWeight - weight) + value
      best = max(best, newVal);
    }
  return best;
}

answer = suitcase(5);
```

A DP algorithm

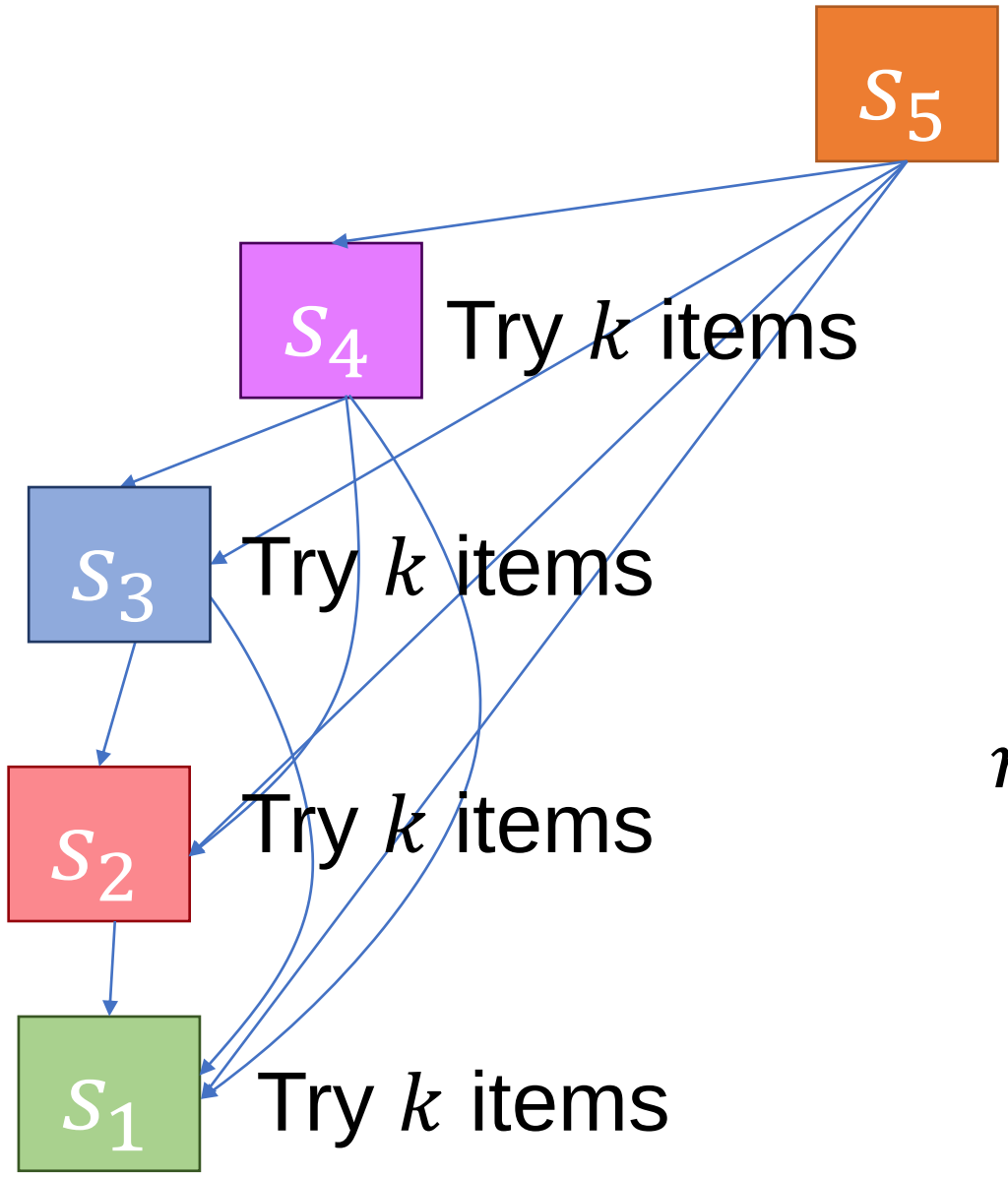
```
int suitcase(int curWeight) {
    // if already computed, directly return
    if (ans[curWeight] != -1) return ans[curWeight];
    int best = 0;
    foreach item (weight, value)
        if (curWeight >= weight) {
            newVal = suitcase(curWeight - weight) + value
            best = max(best, newVal);
        }
    return ans[curWeight] = best; // memorize the current return value
}

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

For k items and capacity n

- Each capacity value needs to be computed
  - $O(n)$ of them
- For each of them, enumerate k possible elements
- **$O(nk)$ cost**



Execution Recurrence Tree



$\Theta(nk)$ cost

n is total weight,

k is #items

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 $i - w_j$ weight

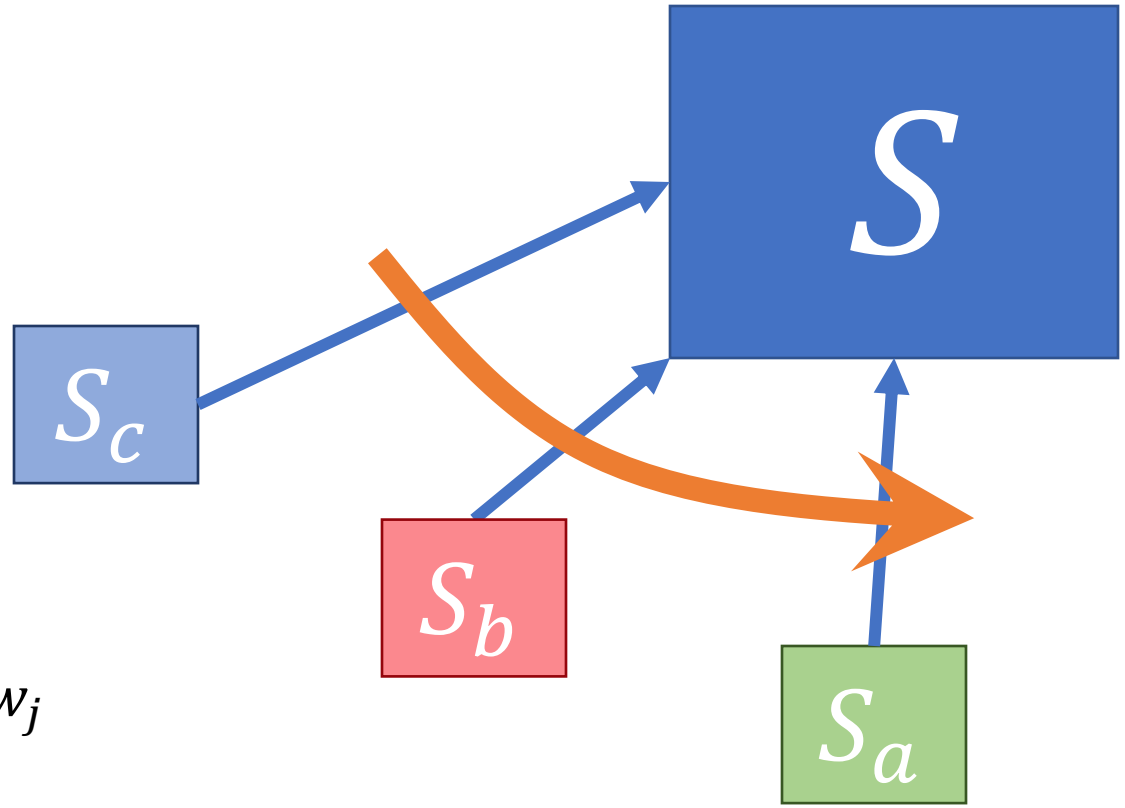
$$s[i] = \max \begin{cases} 0 \\ \max\limits_{(w[j],v[j]) \text{ is an item}} \{s[i - w[j]] + v[j]\} : \ i \geq w[j] \end{cases}$$

Trying all possible items

- **$s[0] = 0$, Final answer is $s[n]$**

What is dynamic programming?

- The key components in a DP algorithm
  - Optimal substructure (**states**)
  - The **decisions**
  - **Boundary**
- All can be presented in a **recurrence**



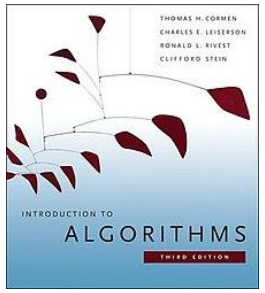
$$s_i = \max \begin{cases} 0 \\ \max\limits_{(w_j, v_j) \text{ is an item}} \{s_{i-w_j} + v_j\} \mid i > w_j \end{cases}$$

Is the previous solution perfect?

```
int suitcase(int leftWeight) {
    if (ans[leftWeight] != -1) return ans[leftWeight];
    int curBest = 0;
    foreach item (weight, value)
        if (leftWeight >= weight)
            curBest = max(curBest, suitcase(leftWeight -
            Weight) + value);
    return ans[leftWeight] = curBest;
}
int ans[50] = {-1, … , -1};
answer = suitcase(50);
```

$50, 1lb

$1500, 12lb



$70, 5lb

$80, 2lb

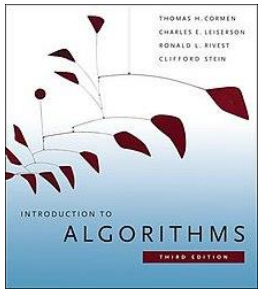
Is this solution still correct if we only allow to use an item once?


$50, 1lb


$50, 1lb


$50, 1lb


$1500, 12lb



$70, 5lb


$80, 2lb

```
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[50] = {-1, … , -1};
answer = suitcase(50);
```

What is the optimal substructure for the new problem?

- **Is s_i sufficient for the new problem?**
  - No!! We do not know if the optimal arrangement for weight i use item j or not
- **What can we do?**
- **Add another dimension!**

Is the previous algorithm perfect? What if each item can be used only once?

- **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, so the best solution is $s[i - w[j], j - 1] + v[j]$**
  - Add value of $v[j]$, the rest must be "best solution using the first $j - 1$ items for weight $i - w[j]$"

- **Do not use item j, so the best solution is $s[i, j - 1]$**
  - Not using j, then we just "use the first $j - 1$ items for weight i", and we want the best result

Another way to state the relationship of $s[i][j]$

- **The recurrence:**

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

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

- **The recurrence cannot be circular**
  - You can not have **states** a, b, and c that computing a relies on b, b on c, and c on a

The DP implementation

```
int suitcase(int i, int j) {
    if (ans[i][j] != -1) return ans[i][j];
    if (i==0 or 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, 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

```
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][k];
```

- 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][k];
```

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

The famous knapsack problem

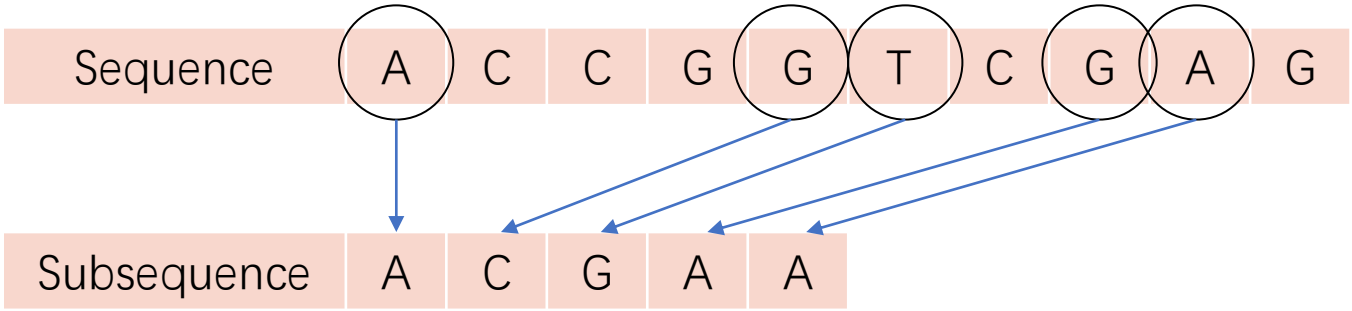
- **The first problem is referred to as the unbounded knapsack problem (UKP)**
- **The second problem is referred to as the 0-1 knapsack problem**
- **Other variants:**
  - Multiple knapsack problem: item j can be used x_j times
  - Some of them cannot be added together
  - Each item has more than one dimension (e.g., both weight and volume)
  - Dependencies between items (Problem C)

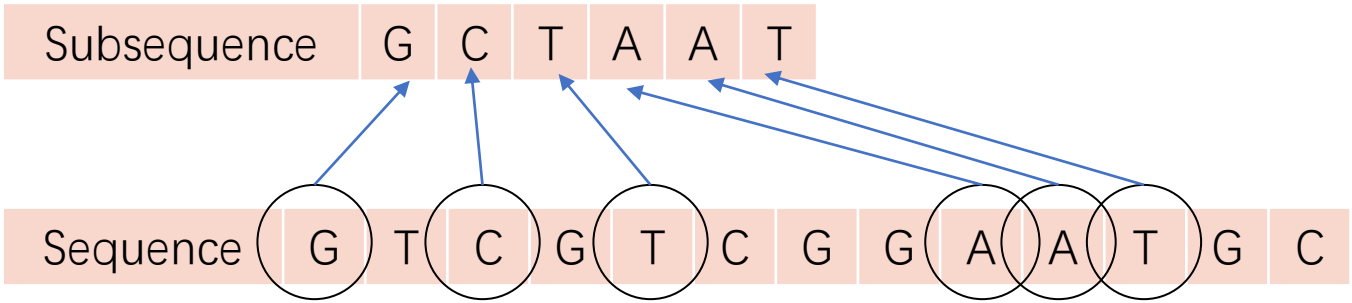
DP recurrence

- **A DP recurrence of the states, with boundary cases**
- **State: represent a unique subproblem**
- **Recurrence (decision): how to compute the current state from other states (subproblems)?**
  - Usually enumerate all possible subproblems, compare them, use the best one
- **Boundary: what are the initial values?**

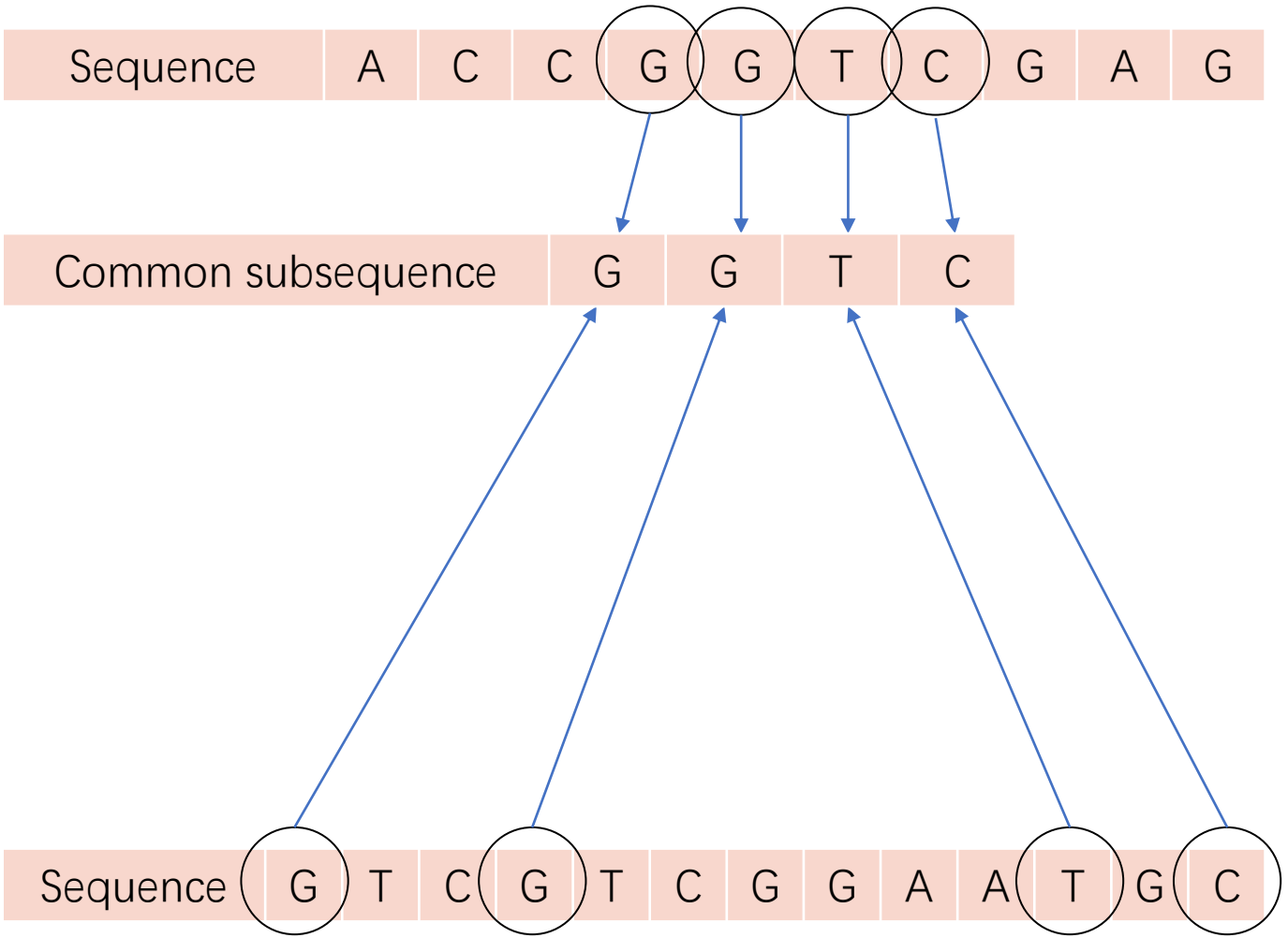
Longest Common Subsequence

Definitions

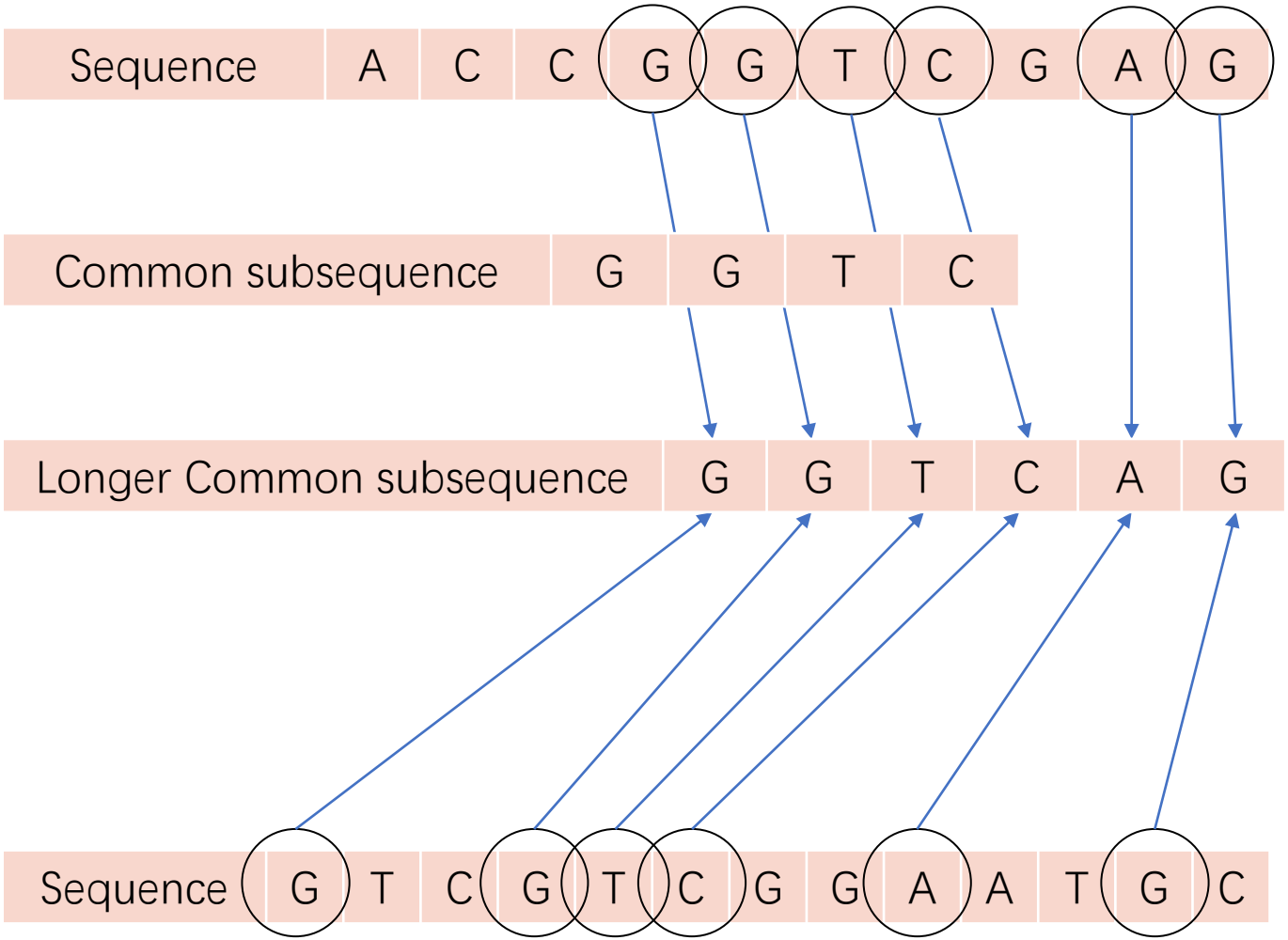




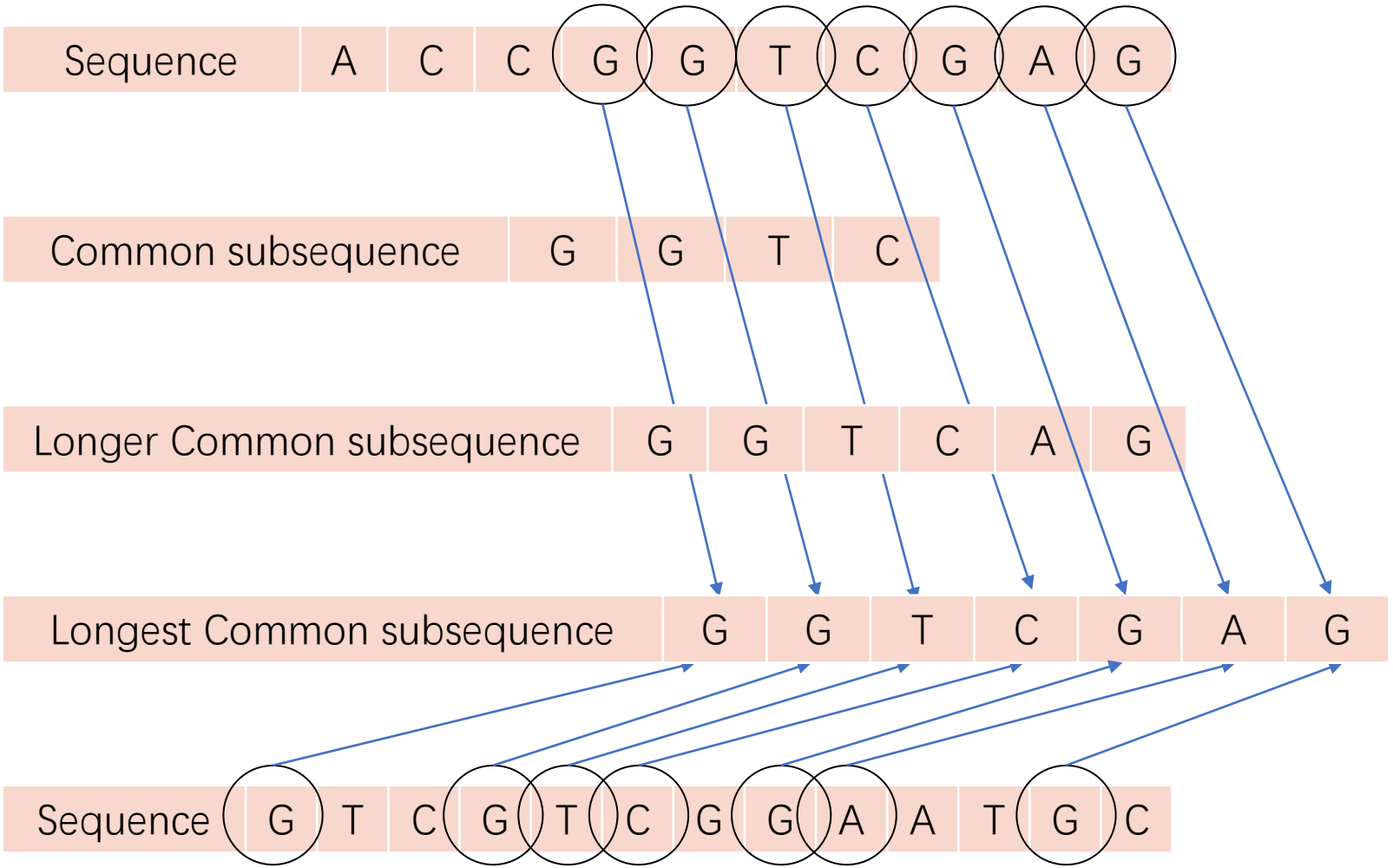
Definitions



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 of X and Y

LCS

- **If we look at $X[1..i]$ and $Y[1..j]$, what is their LCS?**
  - First i characters in X and first j characters in Y
- **Let's compare the last character $X[i]$ and $Y[j]$**
- **Let $s[i,j]$ be the length of LCS of $X[1..i]$ and $Y[1..j]$**
- **What if $X[i] = Y[j]$?**
- **What if $X[i] \neq Y[j]$?**

LCS

- if $X[i] = Y[j] = c$
  - Let's keep the last character c in LCS
  - 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	5	6	7
X =	A	B	C	B	D	A	B
				↑			
Y =	B	D	C	A	B	A	
					↑		

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	4	5	6	7
X =	A	B	C	B	D	A	B
			↑				
Y =	B	D	C	A	B	A	

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 $Y[1..j]$

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

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

Recursive Algorithm

- LCS(i, j)
  - if computed then directly return
  - 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))

Non-Recursive Algorithm

- ans[i][0] = ans[0][j] = 0
- for i = 1 to m do
  - for j = 1 to n do
    - if X[i] == Y[j]
      - ans[i][j] = ans[i-1][j-1] + 1
    - else
      - ans[i][j] = max(ans[i][j-1], ans[i-1][j])

Bottom-up Algorithm

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

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

Bottom-up Algorithm

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

i \ j		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

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

i \ j		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

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

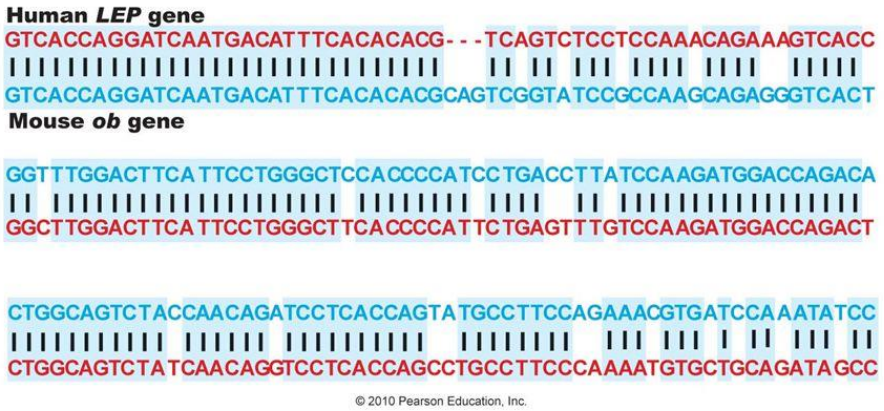
i \ j		B	D	C	A	B	A
	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
A 1	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↑ 1
B 2	0	↖ 1	← 1	← 1	↑ 1	↖ 2	2
C 3	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B 4	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D 5	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A 6	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B 7	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

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
- **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 Relation

- $D_{i,j}$: **The cost of transforming** $X[1..i]$ **to** $Y[1..j]$

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

Summary for Dynamic Programming

Dynamic Programming (DP)

- **Looks hard ☹ it usually takes a few practice 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, …

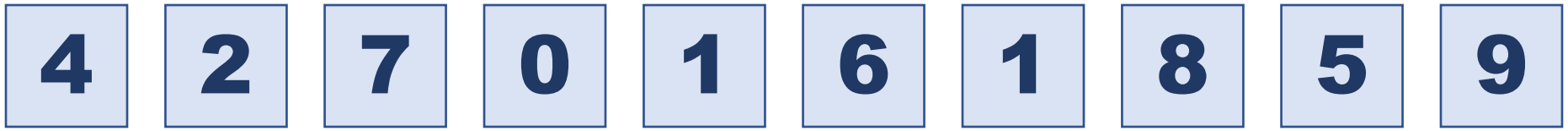
- **And you'll find out they are easy: usually correctness is straightforward**
  - For all states, we compute the solution based on enumerating all possibilities

Dynamic Programming (DP)

- **DP is not an algorithm, but an algorithm design idea (methodology)**
- **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

Longest Increasing Subsequence (LIS) and Other Similar Problems

What is an increasing subsequence?



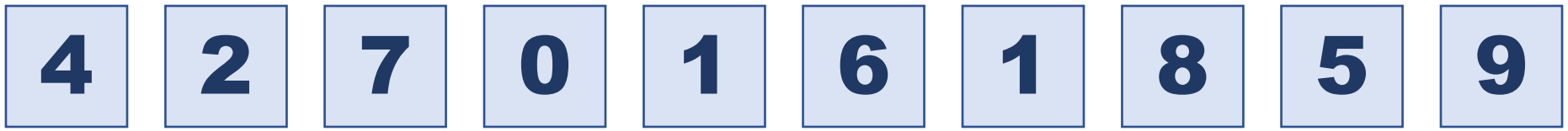
- Increasing subsequence:

- Longest increasing subsequence (LIS):

Why studying LIS?

- **The length of LIS reflect some intrinsic properties of the sequence**
  - Consider it as the “eigenvalue” of a sequence (LIS as the “eigenvector”)
  - Applications in many algorithms and quantum computing

- **Many similar DP algorithms are similar to the DP algorithm for LIS**
  - More examples are given later in this lecture



What are the states for LIS?

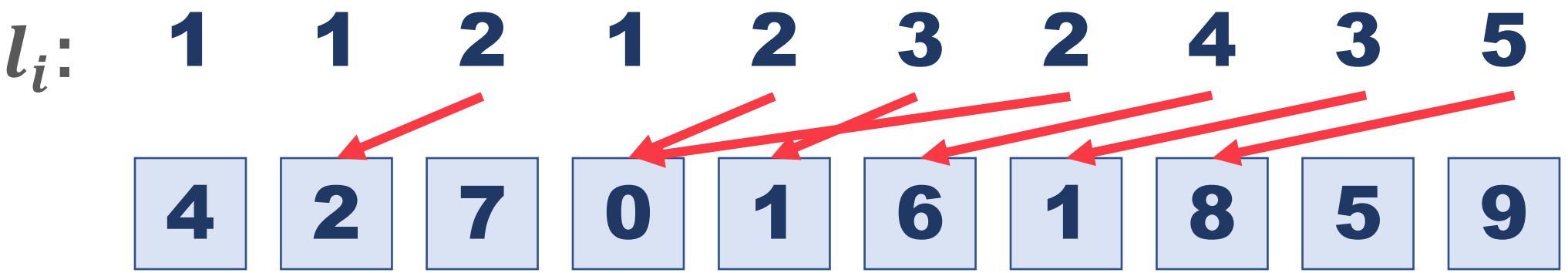
- Let l_i be the LIS for the first i element with i selected (as the last in LIS)
- What is the recurrence of LIS?

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

4	2	7	0	1	6	1	8	5	9

Running the example input

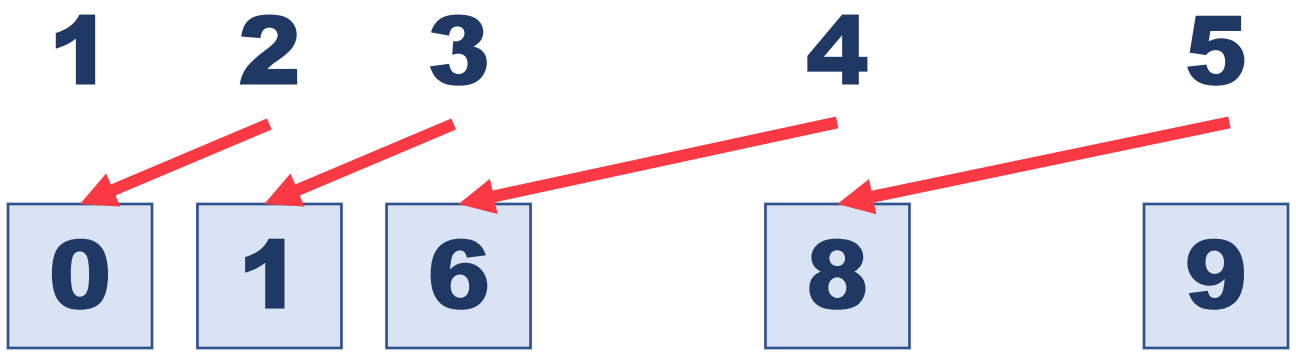
$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$



Running the example input

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

l_i:

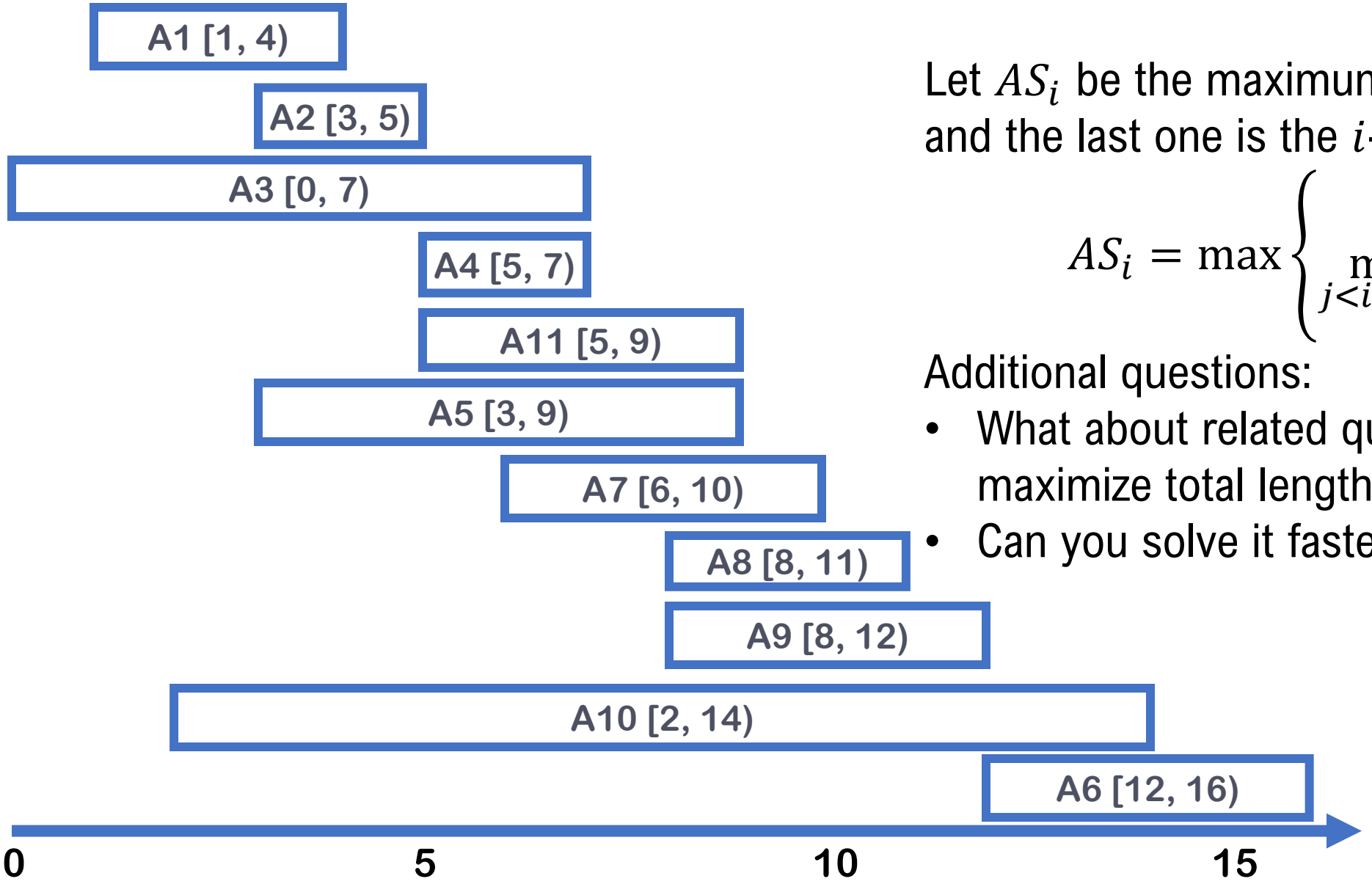


What is the time complexity of LIS?

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

- n element, each takes $O(n)$ time to compute, so $O(n^2)$ cost in total

Revisit: activity selection



Let AS_i be the maximum number of activities and the last one is the i-th activity, then

$$AS_i = \max \begin{cases} 1 \\ \max\limits_{j<i, e_j \le s_i} \{AS_j + 1\} \end{cases}$$

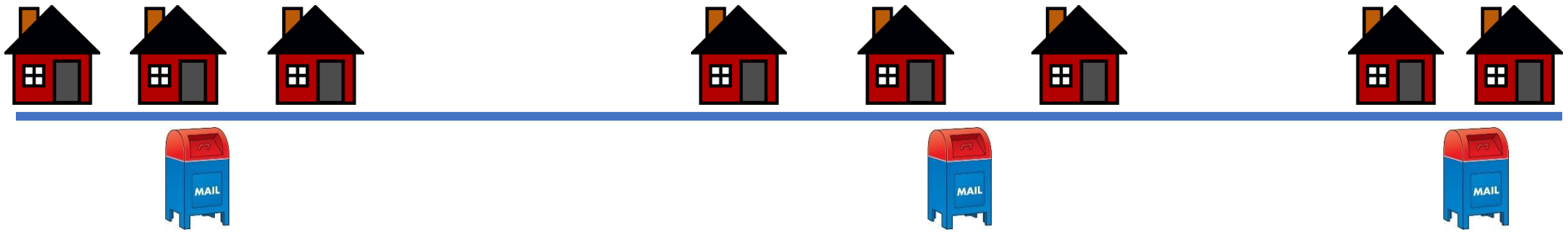
Additional questions:

- What about related questions? Like maximize total length, smallest waiting time
- Can you solve it faster than $O(N^2)$?

Other similar problem: the famous “post-office problem”

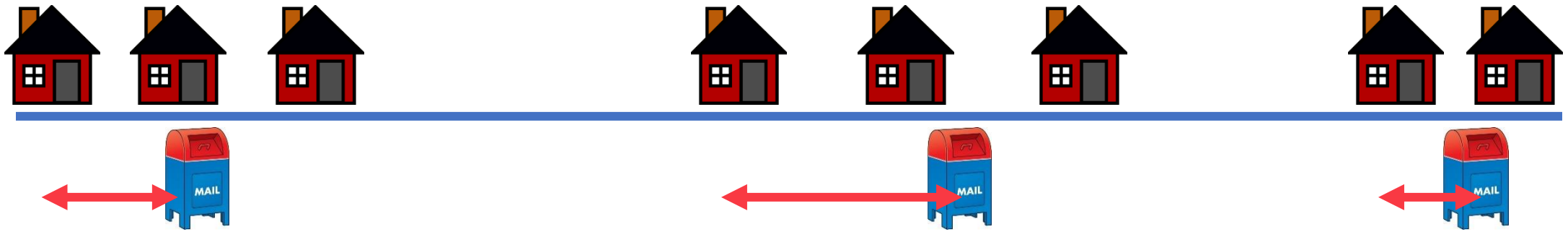
- First proposed by Donald Knuth in vol. 3 of TAOCP (1973)
- Let’s consider the 1d case

- Installing each mailbox has certain cost (installation and maintenance)
- But we also want to minimize the residents’ walking distances



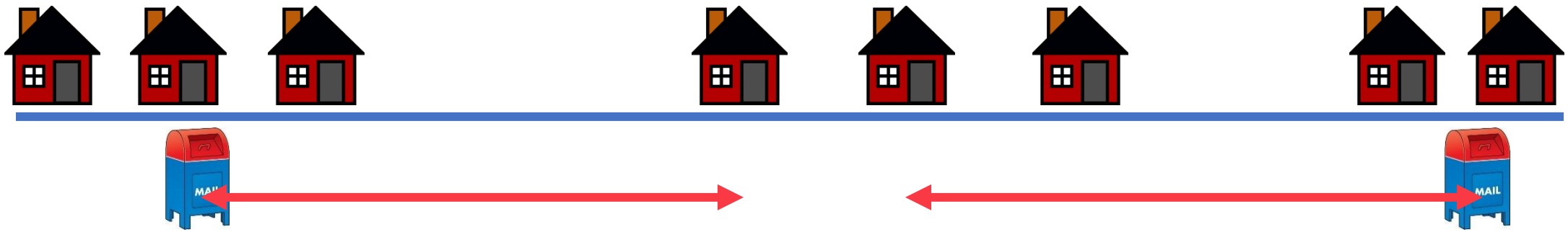
Formalize the problem

- Installing each mailbox has certain cost m
- The residents' unhappiness is the sum of the longest walking distances for each mailbox



Formalize the problem

- Installing each mailbox has certain cost m
- The residents' unhappiness is the sum of the longest walking distances for each mailbox

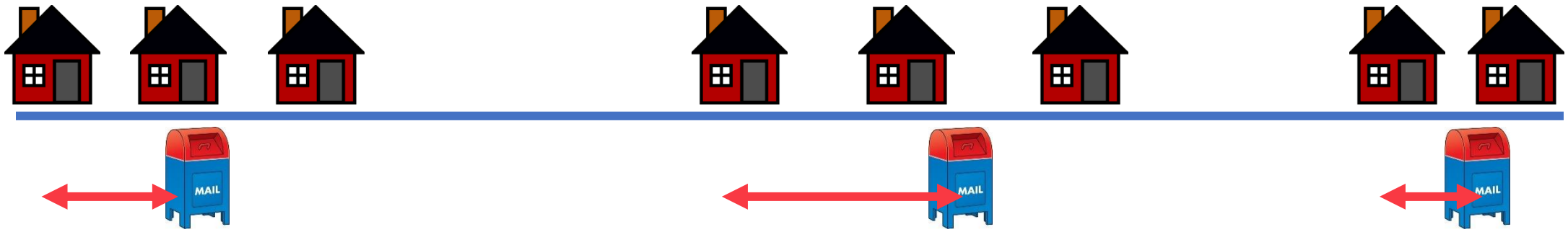


Solving the problem

- Installing each mailbox has certain cost m
- The residents' unhappiness is the sum of the longest walking distances for each mailbox
- Let p_i be the optimal solution of the first i residents:

$$p_i = \min_{j<i} \{p_j + m + (c_i - c_{j+1})/2\}$$

- Boundary: $p_0 = 0$
- Answer: p_n



The line-breaking problem in LaTeX

Randomized Incremental Convex Hull is Highly Parallel

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

ABSTRACT

The randomized incremental convex hull algorithm is one of the most practical and important geometric algorithms in the literature. Due to its simplicity, and the fact that many points or facets can be added independently, it is also widely used in parallel convex hull implementations. However, to date there have been no non-trivial theoretical bounds on the parallelism available in these implementations. In this paper, we provide a strong theoretical analysis showing that the standard incremental algorithm is inherently parallel. In particular, we show that for n points in any constant dimension, the algorithm has $O(\log n)$ dependence depth with high probability. This leads to a simple work-optimal parallel algorithm with polylogarithmic span with high probability.

Our key technical contribution is a new definition and analysis of the configuration dependence graph extending the traditional configuration space, which allows for asynchrony in adding configurations. To capture the "true" dependence between configurations,

the convex hull. A newly-added point either falls into the current convex hull and thus no further action is needed, or it removes existing faces (henceforth facets) that it is *visible* from, while adding new facets. For example, in Figure 1, adding c to the existing hull u-v-w-x-y-z-t would replace edges v-w, w-x, x-y, and y-z with v-c and c-z. Clarkson and Shor, in their seminal work over 30 years ago [28], showed that the incremental convex hull algorithm on n points in d-dimensions has optimal $O(n^{\lfloor d/2 \rfloor} + n \log n)$ expected runtime when points are added in random order. Their results are based on a more general setting, which they also used to solve several other geometry problems, and the work led to over a hundred papers and survey articles on the topic of random incremental algorithms. Their proof has been significantly simplified over the years, and is now described in several textbooks [21, 32, 35, 50]. Analysis techniques, such as backwards analysis [54], were developed in this context and are now studied in many intermediate algorithms courses.

The line-breaking problem in LaTeX

- You have n words in a paragraph with lengths $l_1, \ldots, l_n$
- You want to break them into lines so each line should contain 50 characters
- The penalty for each line is the $|x - 50|$ when x is the number of characters in that line
- You want to find an optimal line-breaking result

The line-breaking problem in LaTeX

- **Let b_i be the optimal penalty for the first i words**

$$b_i = \min_{j<i} \left\{ b_j + \left| i - j - 1 + \sum_{k=j+1}^{i} l_k - 50 \right| \right\}$$

- **Boundary: $b_0 = 0$**
- **Answer: $\min_{i<n} \{b_i + w(i)\}$**
  - $w(i)$ is the penalty for the last line, which is 0 if the last line has no more than 50 letters, or $n - i - 1 + \sum_{k=i+1}^{n} l_k - 50$ otherwise
- **Can add additional penalty to break the words (states changed to letters)**
- **How to implement it in $O(n^2)$ time?**

Can we do better?

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

- n element, each takes $O(n)$ time to compute, so $O(n^2)$ cost in total

Optimize LIS algorithm to O(nlogn)

LIS DP formula

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

- n element, each takes $O(n)$ time to compute, so $O(n^2)$ cost in total

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j\} + 1 \end{cases}$$

- We need to find, for all l_j before l_i, with $a_j < a_i$, which is the largest value

l_i: 1 1 2 1 2 ?

a_i:

4	2	7	0	1	6	1	8	5	9

l_i: 1 1 2 1 2 3 ?

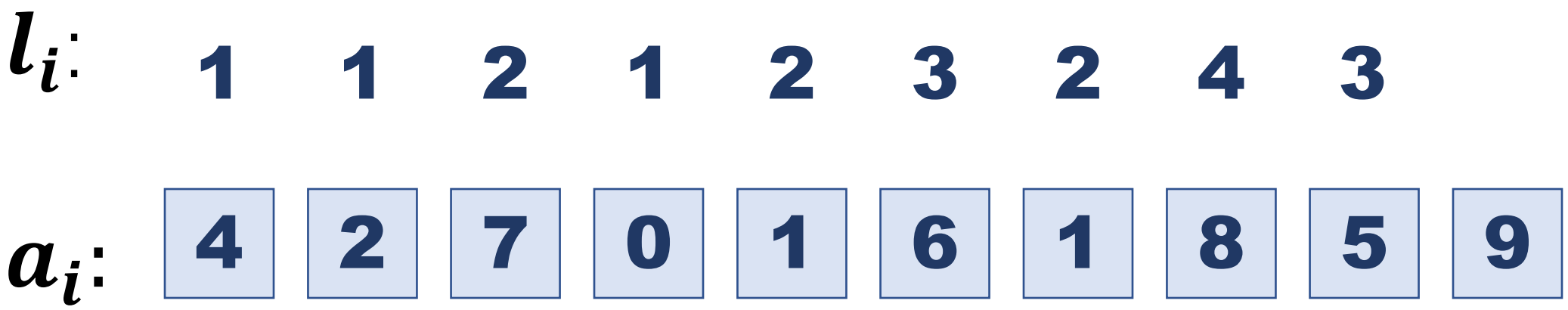
a_i:	4	2	7	0	1	6	1	8	5	9

l_i: 1 1 2 1 2 3 2 ?

a_i: 4 2 7 0 1 6 1 8 5 9

l_i:	1	1	2	1	2	3	2	4	?

a_i:	4	2	7	0	1	6	1	8	5	9


l_i:
1 1 2 1 2 3 2 4 3
a_i:
4 2 7 0 1 6 1 8 5 9

LIS – DP formula

- $l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j\} + 1 \end{cases}$

- **When processing i**
  - For all other elements that have been processed (all $j < i$)
  - We only consider when $a_j < a_i$
  - Find the largest l_j

- **A range-max query?**

LIS + range query

- **Assume we have an ADT D that can deal with range-max query**
  - Store key-value pairs
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key < k, find the maximum value

Key	4	7	8	9	10	12	13	15	16	20	25
value	4	7	9	2	2	11	9	17	10	3	2

range_max(18) = 17

LIS + range query

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j\} + 1 \end{cases}$$

- **When processing i**
  - We need to look at all j before i (processed j) and $a_j < a_i$
  - Find the largest l_j
- **Assume we have an ADT D that can deal with range-max query**
  - Store key-value pairs
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key < k, find the maximum value
  - Key: a_j, value: l_j
- **When processing i**
  - Call range_max(a_i) on D, get v^*, let $l_i = v^* + 1$
  - Insert (a_i, l_i) to D
  - Go to $i + 1$

LIS + range query

- **Assume we have an ADT D that can deal with range-max query**
  - Store key-value pairs
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key<k, find the maximum value
  - Key: a_i, value: l_i
- **We can use an augmented tree to implement D**
- **To compute each value in $l[\cdot]$, we need:**
  - **A range_max query: log n time**
  - **An insert operation: log n time**
  - **Total running time O(n log n)**

Dynamic Programming (DP)

- **Looks hard ☹ it usually takes a few practice 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, a huge class of LIS-style problems

- **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)**
- **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
- **But you need to practice 20-30 problems on finding out the recurrence for seemingly very unrelated problems**