算法设计与分析

Lecture 6: Dynamic Programming

卢杨
厦门大学信息学院计算机科学系
luyang@xmu.edu.cn

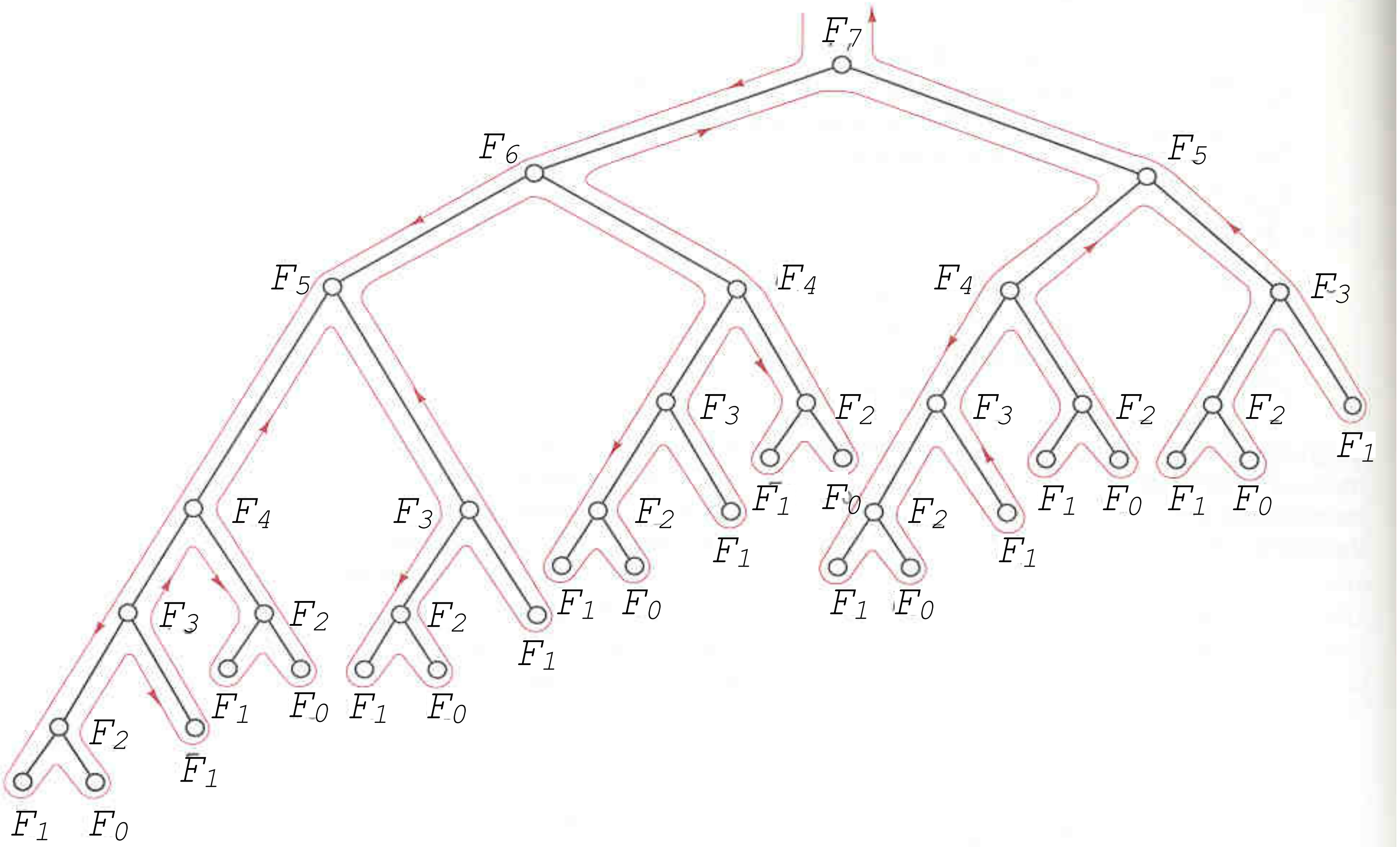
Fibonacci Sequence Revisit

- Fibonacci sequence is defined by

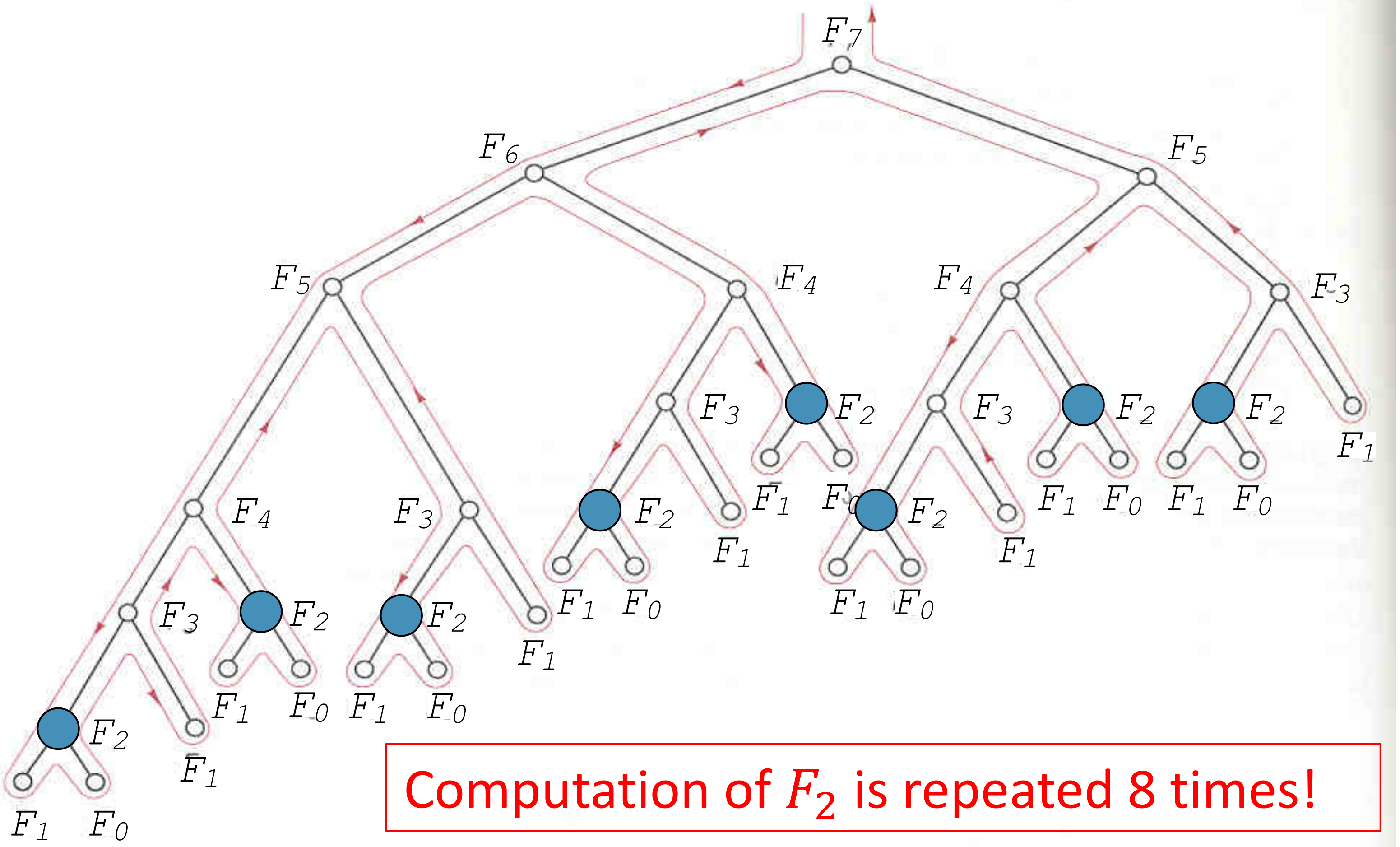
$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} - F_{n-2}, \qquad \text{for } n \geq 2
\end{aligned}
$$

```
Fib1($n$)
if $n \leq 1$ then
    return $n$
else
    return Fib1($n-1$) + Fib1($n-2$)
```

Fibonacci Sequence Revisit

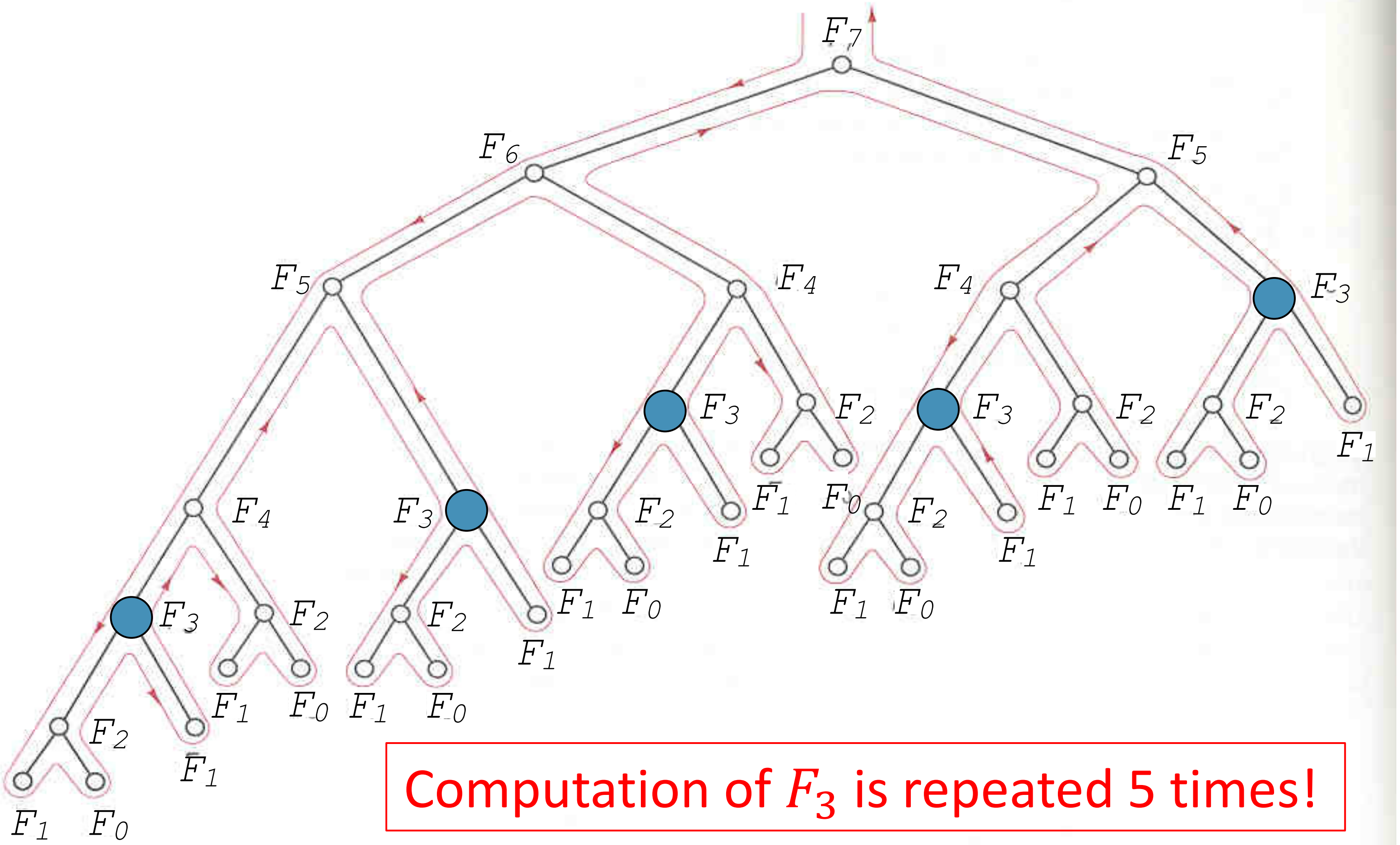


Fibonacci Sequence Revisit



Computation of F_2 is repeated 8 times!

Fibonacci Sequence Revisit



Computation of F_3 is repeated 5 times!

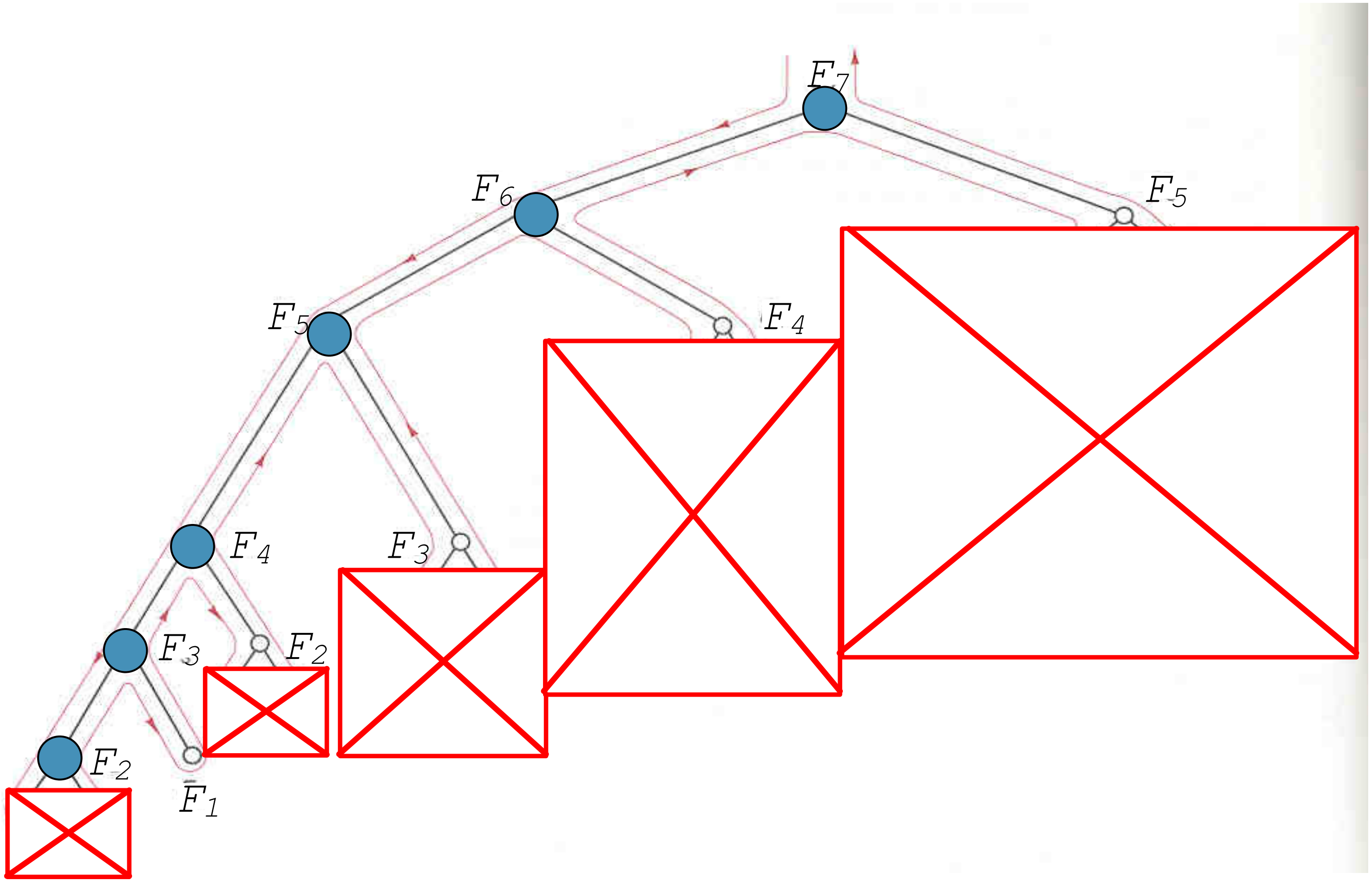
Fibonacci Sequence Revisit

- Idea of improvement: record the values somewhere!
  - Store F_i somewhere after we have computed its value.
  - Afterward, we don't need to re-compute F_i. We can retrieve its value from our memory.

```
Fib2($n$)
**if** $F[n] < 0$ **then**
    $F[n] \leftarrow$ Fib2($n-1$) + Fib2($n-2$)
**return** $F[n]$
```

```
Main()
$F[0] = F[1] \leftarrow 1$
**for** $i \leftarrow 2$ **to** $n$ **do**
    $F[i] = -1$
**print** Fib2($n$)
```

Fibonacci Sequence Revisit



F[0]	1
F[1]	1
F[2]	2
F[3]	3
F[4]	5
F[5]	8
F[6]	13
F[7]	21

Fibonacci Sequence Revisit

- Can we do even better?
  - Although we didn’t waste time on repeated computation, we make a lot of recursive function calls.
  - Must we use recursion?
- Idea to further improve
  - Compute the values in bottom-up fashion.
  - That is, compute F_2 (we already know $F_0 = F_1 = 1$), then F_3, then F_4…

```
Fib3($n$)
$F[0] \leftarrow 1$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to $n$ do
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$
```

This new implementation saves lots of overhead.

Recursion vs. Dynamic Programming

- Recursion:
  - Too slow.
  - Time complexity: $O(2^n)$.

```
Fib1($n$)
if $n \leq 1$ then
        return $n$
else
        return Fib1($n-1$) +
                Fib1($n-2$)
```

- Dynamic Programming:
  - Efficient!
  - Time complexity: $O(n)$.

```
Fib3($n$)
$F[0] \leftarrow 1$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to $n$ do
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$
```

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系
Computer Science Department of Xiamen University

Dynamic Programming

- Write down a formula that relates a solution of a problem instance with those of small instances.
  - E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be stored and retrieved easily in a table (i.e., array).
- Fill the table in some bottom-up manner; start filling the solution of the smallest instance.
  - This ensures that when we solve a particular instance, the solutions of all the smaller instances that it depends are available.

Dynamic Programming

- For historical reasons, we call such methodology Dynamic Programming (动态规划).
- It was developed by American applied mathematician Richard Ernest Bellman in 1950s.
- It is also called Bellman equation in optimization field.

Richard Ernest Bellman
(1920-1984)

Image source: https://en.wikipedia.org/wiki/Richard_E._Bellman

Divide-and-Conquer vs. Dynamic Programming

- Common: Problem is partitioned into one or more subproblem, then the solution of subproblem is combined.
- Divide-and-conquer method
  1. Subproblem is independent.
  2. Subproblem is solved repeatedly.
- Dynamic programming
  1. Subproblem is not independent.
  2. Subproblem is just solved once.
- DP reduces computation by
  1. Solving subproblems in a bottom-up fashion.
  2. Storing solution to a subproblem the first time it is solved.
  3. Looking up the solution when subproblem is met again.

Dynamic Programming

- Key idea:
  - Top-down design: Determine structure of optimal solutions.
  - Bottom-up solve: Avoid repeated computation.
- Dynamic programming is typically applied to optimization problems. It is broken into a sequence of four steps.

  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

ASSEMBLY-LINE SCHEDULING

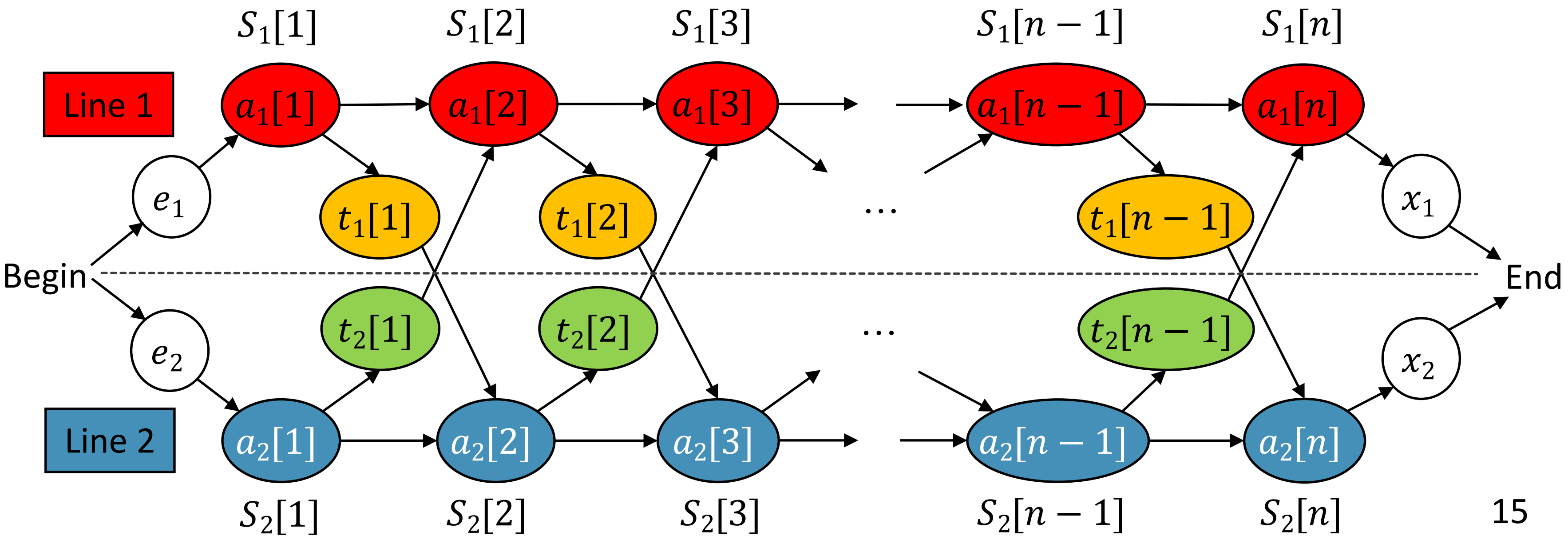
Assembly-Line Scheduling

Assembly-line scheduling (装配线调度) problem:

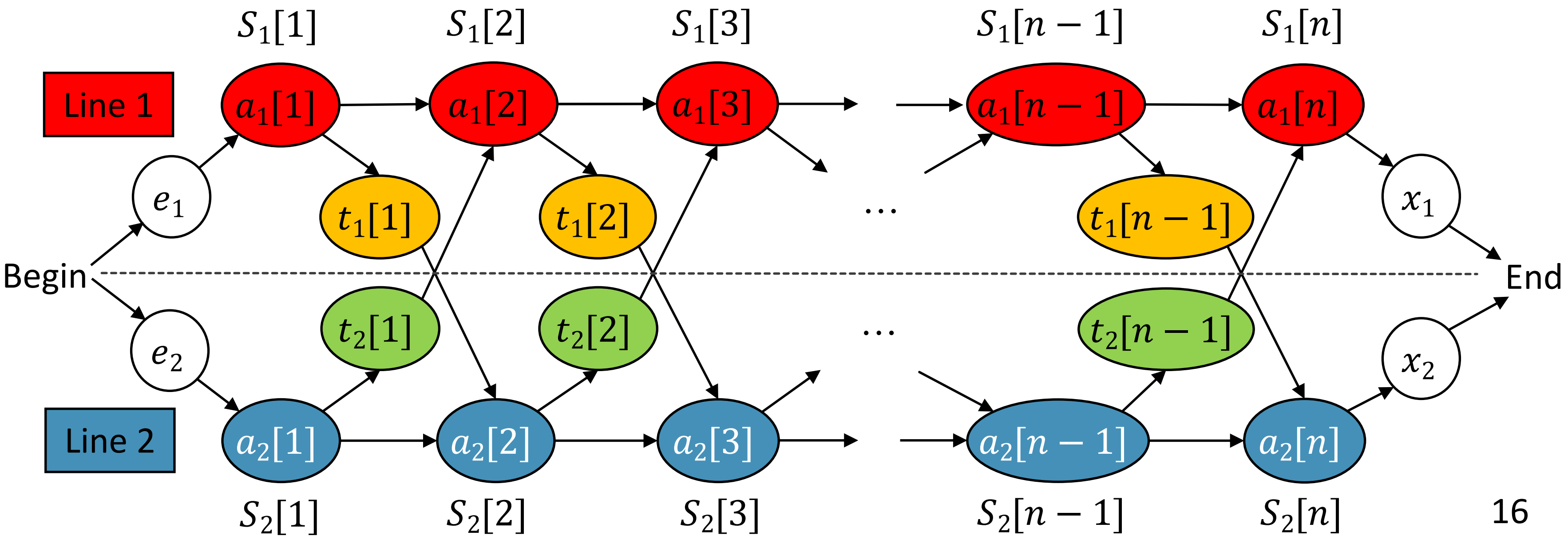
- There are two assembly lines (装配线), each with n stations (装配点).
- An automobile chassis (汽车底盘) enters the factory needs to go though all n stations.

Image source: http://www.auto-assemblyplants.com/sale-11709073-vehicle-assembly-line-automotive-manufacturing-equipment-business-partners.html

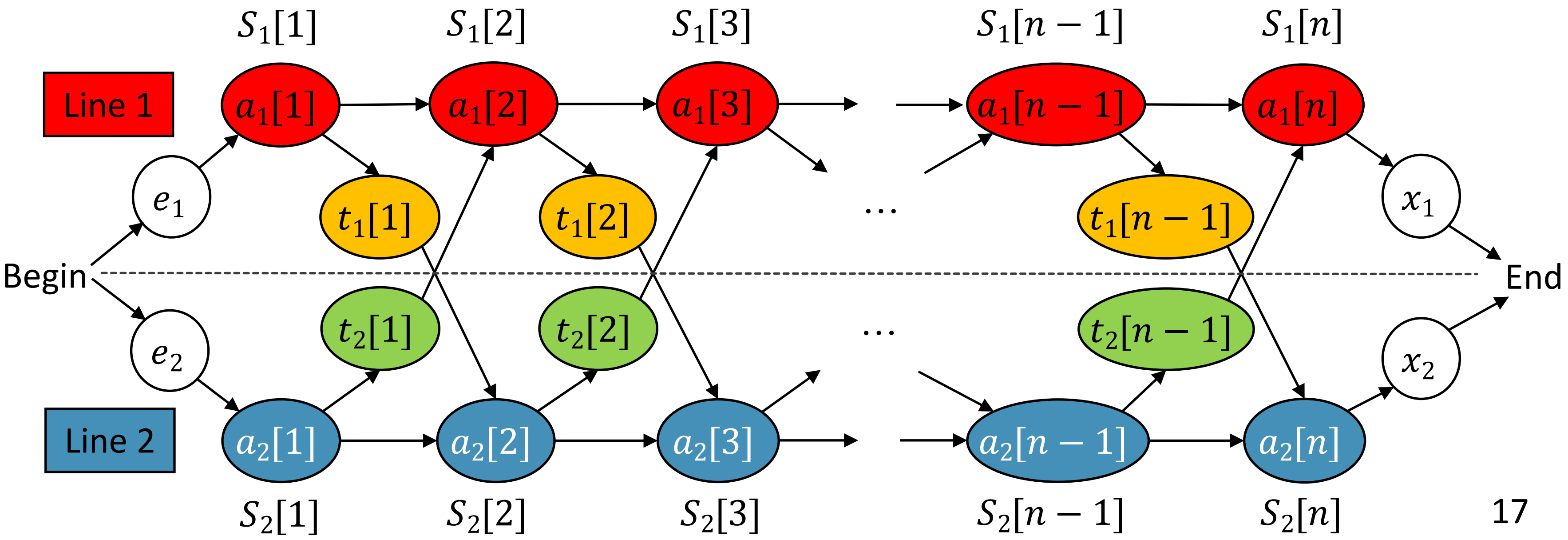
- The jth station on line i ($i = 1$ or 2) is denoted $S_i[j]$ and the assembly time at that station is $a_i[j]$.
- At the beginning, a chassis takes e_i time to enter the factory.
- After going through the jth station on a line, the chassis goes on to the $(j + 1)$st station on either line.



- There is no transfer cost if it stays on the same line, but it takes time $t_i[j]$ to transfer to the other line after station $S_i[j]$.
- After exiting the nth station on a line, it takes x_i time for the completed auto to exit the factory.
- The problem is to determine which stations to choose from line 1 and line 2 in order to minimize the total time.



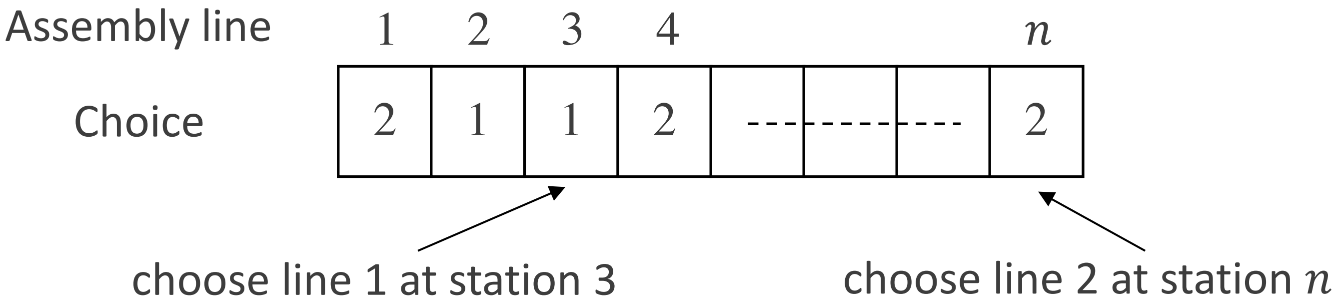
- $S_i[j]$:The jth station on line i ($i = 1$ or 2) .
- $a_i[j]$:The assembly time required at station $S_i[j]$.
- $t_i[j]$: The transfer cost from $S_i[j]$ to another line.
- e_i :Entry time.
- x_i: Exit time.



Brute-Force Solution

Brute-force solution (暴力解法):

- Enumerate all possibilities of selecting stations.
- Compute how long it takes in each case and choose the minimum one.



Assembly line	1	2	3	4				n
Choice	2	1	1	2	---	---	---	2

choose line 1 at station 3

choose line 2 at station n

- How many possible ways? 2^n
- Is it possible for large n? No!

Structure of an Optimal Solution

- Let’s consider the fastest way possible to get from the starting point through station $S_1[j]$.
  - If $j = 1$: determine how long it takes to get through $S_1[1]$.
  - If $j \geq 2$, have two choices of how to get to $S_1[j]$:
    - Through $S_1[j-1]$, then directly to $S_1[j]$.
    - Through $S_2[j-1]$, then transfer over to $S_1[j]$.
- Similar for the case through $S_2[j-1]$.

Structure of an Optimal Solution

- Suppose that the fastest way through station $S_1[j]$ is through station $S_1[j-1]$. The chassis must have taken a fastest way from the starting point through station $S_1[j-1]$.
  - If from $S_1[j-1]$ to $S_1[j]$ is optimal, $S_1[j-1]$ is also optimal.
- Why?
- Prove by contradiction: If there were a faster way to get through station $S_1[j-1]$, we could substitute this faster way to yield a faster way through station $S_1[j]$: a contradiction.
  - If $S_1[j-1]$ is not optimal, from $S_1[j-1]$ to $S_1[j]$ is also not optimal,

Optimal Substructure

- Generalization: an optimal solution to the problem find the fastest way through $S_1[j]$ contains within it an optimal solution to subproblems: find the fastest way through $S_1[j-1]$ or $S_2[j-1]$.
- This is referred to as the optimal substructure property (最优子结构性质).
- Optimal substructure property is one of the hallmarks of the applicability of dynamic programming.
  - We use this property to construct an optimal solution to a problem from optimal solutions to subproblems.

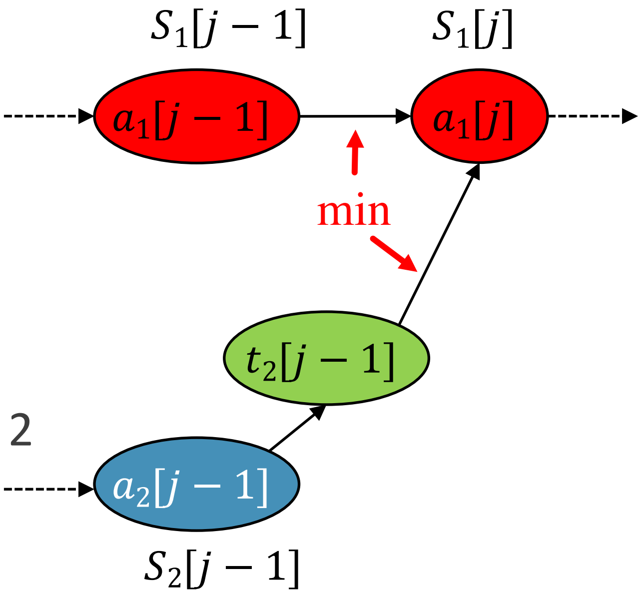
Optimal Substructure

- Denote $f_i[j]$ as the fastest time to get from the starting point through station $S_i[j]$.
- Fastest way through $S_1[j]$ is either:
  - fastest way through $S_1[j-1]$ then directly through $S_1[j]$:

$$f_1[j] = f_1[j - 1] + a_1[j]$$

  - fastest way through $S_2[j-1]$, transfer from line 2 to line 1, then through $S_1[j]$:

$$f_1[j] = f_2[j - 1] + t_2[j-1] + a_1[j]$$



- In summary:

$$f_1[j] = \min(f_1[j - 1] + a_1[j], f_2[j - 1] + t_2[j-1] + a_1[j])$$

Similar for $f_2[j]$

Recursive Equation

- The recursive equation:

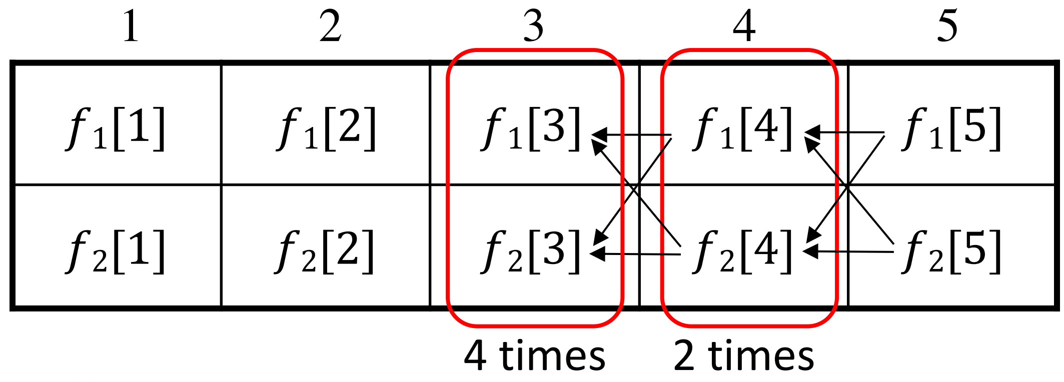
$$f_1[j] = \begin{cases} a_1[1] + e_1 & j = 1 \\ \min(f_1[j-1] + \ a_1[j], f_2[j-1] + t_2[j-1] + a_1[j]) & j > 1 \end{cases},$$

$$f_2[j] = \begin{cases} a_2[1] + e_2 & j = 1 \\ \min(f_2[j-1] + \ a_2[j], f_1[j-1] + t_1[j-1] + a_2[j]) & j > 1 \end{cases}.$$

- The optimal solution when finishing:

$$f^* = \min\{f_1[n] + x_1\,, f_2[n] + x_2\}\,.$$

- How to solve? How about recursion?



	1	2	3	4	5
$n = 5$:	$f_1[1]$	$f_1[2]$	$f_1[3]$	$f_1[4]$	$f_1[5]$
	$f_2[1]$	$f_2[2]$	$f_2[3]$	$f_2[4]$	$f_2[5]$

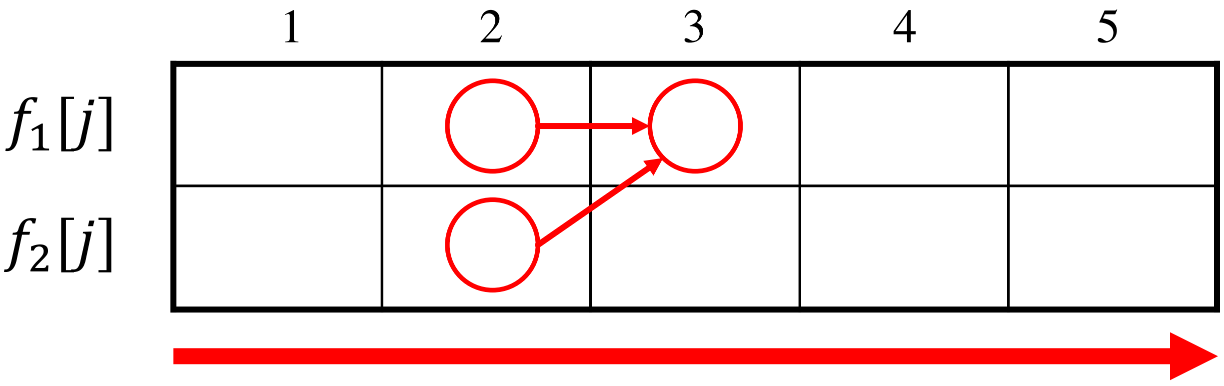
Exponential running time!

Computing the Optimal Solution

- For $j \geq 2$, each value $f_i[j]$ depends only on the values of:

$$f_1[j-1] \text{ and } f_2[j-1].$$

- Compute the values of $f_i[j]$ in increasing order of j.



Calculate column by column in increasing order of j

- Bottom-up approach:
  - First find optimal solutions to subproblems.
  - Find an optimal solution to the problem from the subproblems.

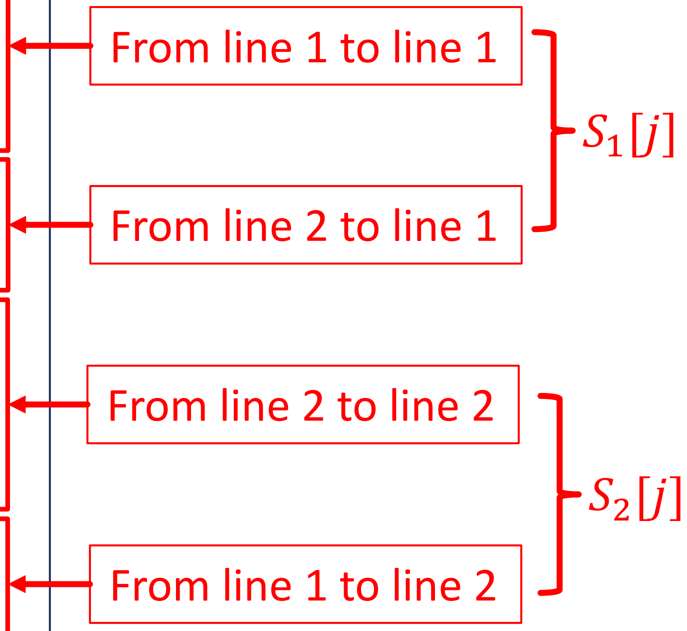
Additional Information

- f^* only records the optimal cost. How can we know the decisions at each step?
- To construct the optimal solution, we need the sequence of what line has been used at each station:
  - $l_i[j]$: the line number (1 or 2) whose previous station ($j-1$) has been used to get in fastest time through $S_i[j]$, $j = 2,3,\dots,n$.
  - l^*: the line whose station n is used to get in the fastest way through the entire factory.

	2	3	4	5
$l_1[j]$				
$l_2[j]$				

Calculate column by column in increasing order of j

```
DPFastestWay($a, t, e, x, n$)
$f_1[1] \leftarrow e_1 + a_1[1]$;  $f_2[1] \leftarrow e_2 + a_2[1]$
for $j \leftarrow 2$ to $n$ do
    if $f_1[j-1] \le f_2[j-1] + t_2[j-1]$ then
        $f_1[j] \leftarrow f_1[j-1] + a_1[j]$
        $l_1[j] \leftarrow 1$
    else $f_1[j] \leftarrow f_2[j-1] + t_2[j-1] + a_1[j]$
        $l_1[j] \leftarrow 2$
    if $f_2[j-1] \le f_1[j-1] + t_1[j-1]$ then
        $f_2[j] \leftarrow f_2[j-1] + a_2[j]$
        $l_2[j] \leftarrow 2$
    else $f_2[j] \leftarrow f_1[j-1] + t_1[j-1] + a_2[j]$
        $l_2[j] \leftarrow 1$
if $f_1[n] + x_1 \le f_2[n] + x_2$ then
    $f^* \leftarrow f_1[n] + x_1$
    $l^* \leftarrow 1$
else $f^* \leftarrow f_2[n] + x_2$
    $l^* \leftarrow 2$
```



From line 1 to line 1

From line 2 to line 1

$\left.\right\} S_1[j]$

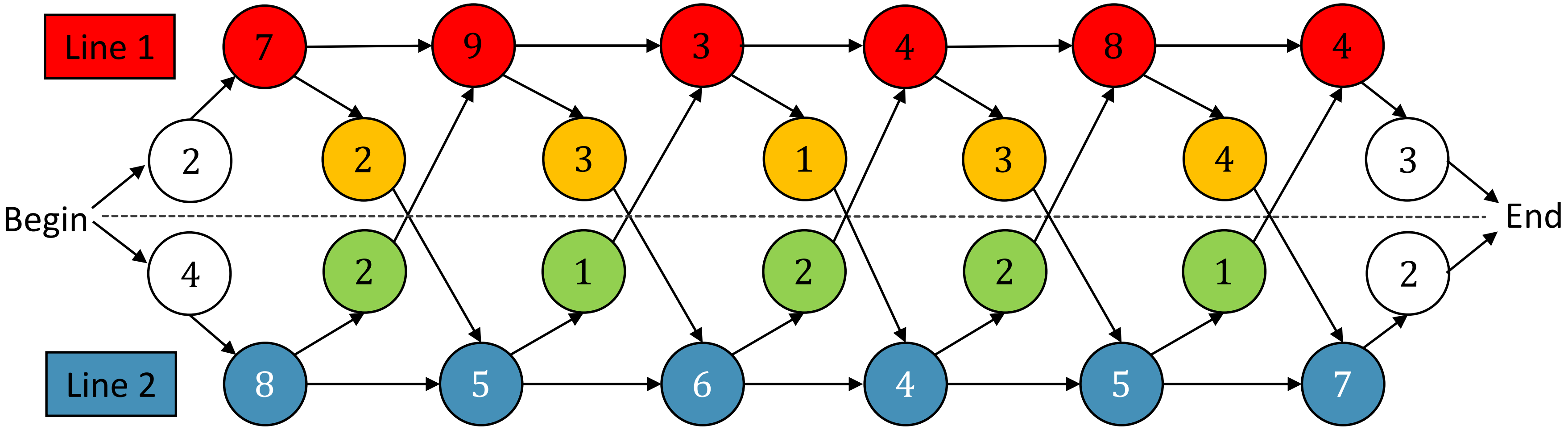
From line 2 to line 2

From line 1 to line 2

$\left.\right\} S_2[j]$

Running time: $\Theta(n)$

Example



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

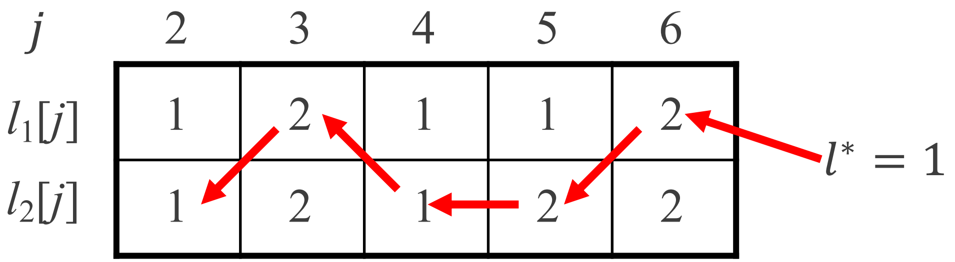
j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

Construct an Optimal Solution

- After calculating, how can we know the optimal solution?

```
PrintStations($l$, $n$)
  $i \leftarrow l^*$
  print "line " $i$ ", station " $n$
  for $j \leftarrow n$ downto 2 do
      $i \leftarrow l_i[j]$
      print "line " $i$ ", station " $j - 1$
```



j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

Output:

line 1, station 6

line 2, station 5

line 2, station 4

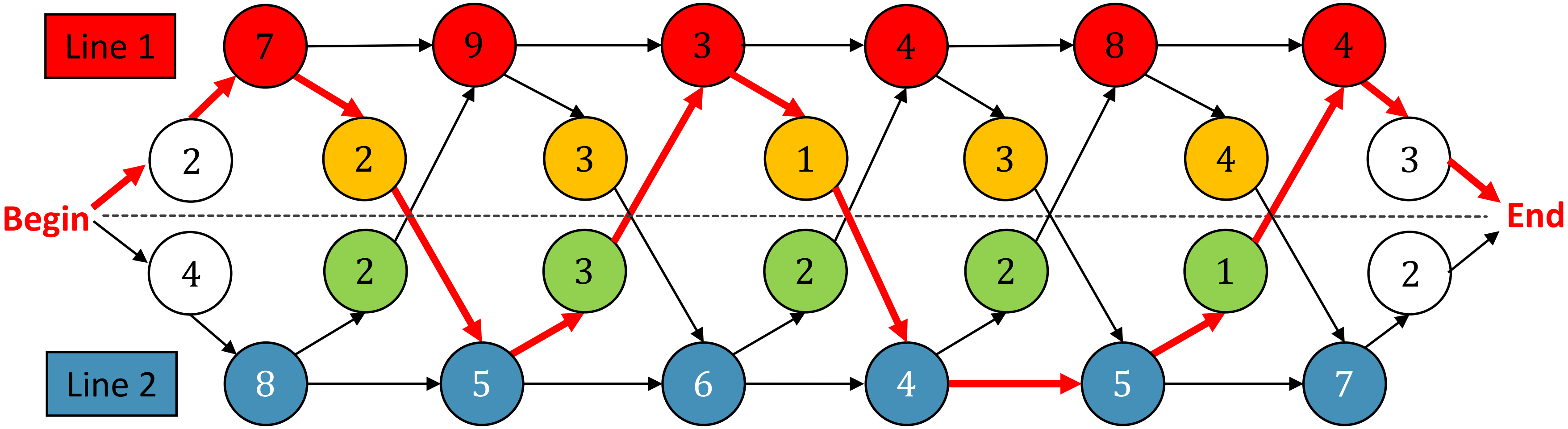
line 1, station 3

line 2, station 2

line 1, station 1

Construct an Optimal Solution

The fastest assembly way: $f^* = 38$



Optimal Solutions

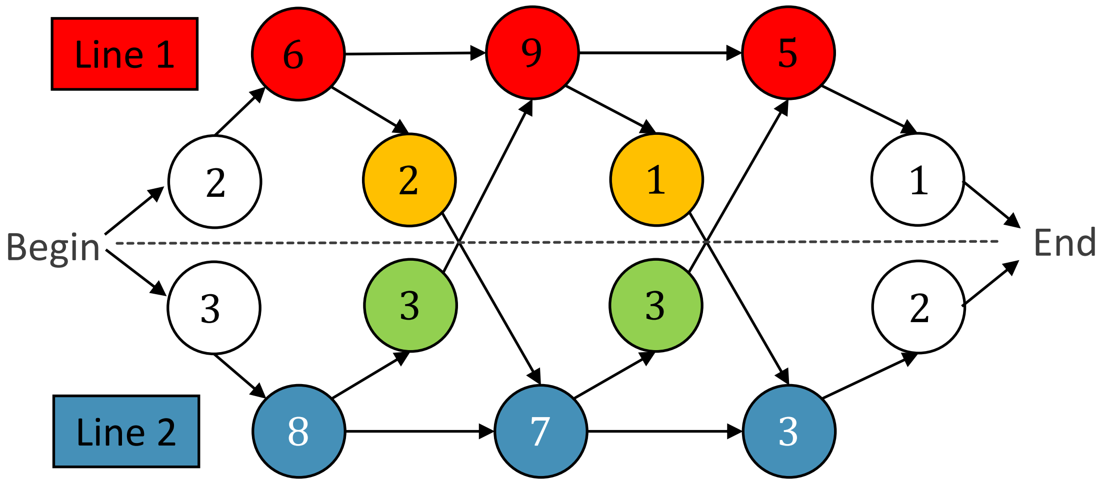
- Dynamic programming not only solves the optimal solution for $n = 6$, but for all n $(n \leq 6)$.
  - Each cell in the table records the optimal solution.
- Every optimal solution is built upon previous optimal solutions. Thus, everything in the table is an optimal solution!

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

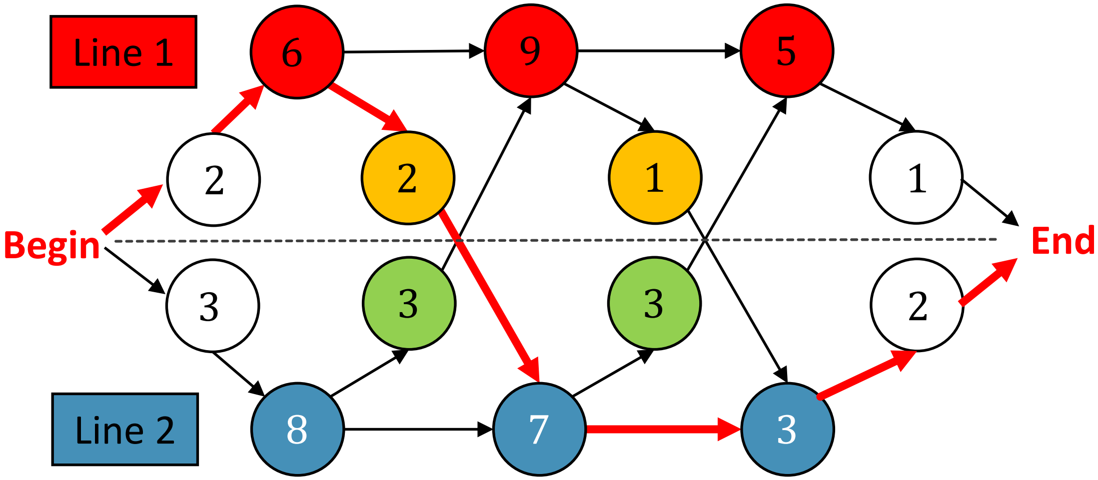
j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

Classroom Exercise

Write the table of $f[j]$ and $l[j]$ for the following problem and determine the optimal solution:



Classroom Exercise



j	1	2	3
$f_1[j]$	8	17	22
$f_2[j]$	11	17	20

$f^* = 22$

j	2	3
$l_1[j]$	1	1
$l_2[j]$	1	2

$l^* = 2$

MATRIX-CHAIN MULTIPLICATION

Matrix-Chain Multiplication

Matrix-chain multiplication (矩阵链乘法) problem:

- Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i = 1,2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

$$\begin{array}{ccccccccc} A_1 & \cdot & A_2 & \cdots & A_i & \cdot & A_{i+1} & \cdots & A_n \\ p_0 \times p_1 & & p_1 \times p_2 & & p_{i-1} \times p_i & & p_i \times p_{i+1} & & p_{n-1} \times p_n \end{array}$$

- Matrix production satisfies the associative law (结合律). Thus, the order of calculation doesn't influence the production result, but influence the efficiency.
- Goal: fully parenthesize (加括号) the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Matrix Multiplication

- To multiply an $n \times m$ matrix with a $p \times q$ matrix using the standard method, it is necessary to do $n \times m \times q$ elementary multiplications.

```
MatrixMultiply($A, B$)
if $m \neq p$ then
    print "Two matrices cannot multiply"
else for $i \leftarrow 1$ to $n$
    for $j \leftarrow 1$ to $q$ do
        $C[i,j] \leftarrow 0$
        for $k \leftarrow 1$ to $m$ do
            $C[i,j] \leftarrow C[i,j] + A[i,k]B[k,j]$
return $C$
```

Running time: $\Theta(nmq)$

Example

- Consider the chained matrix multiplication:

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20\times 2 & & 2\times 30 & & 30\times 12 & & 12\times 8 \end{array}$$

- The total number of elementary multiplications depends on the multiplication order.

$$\begin{aligned}
A\big(B(CD)\big):&\quad 30\times 12\times 8 + 2\times 30\times 8 + 20\times 2\times 8 &&= 3{,}680 \\
(AB)(CD):&\quad 20\times 2\times 30 + 30\times 12\times 8 + 20\times 30\times 8 &&= 8{,}880 \\
A((BC)D):&\quad 2\times 30\times 12 + 2\times 12\times 8 + 20\times 2\times 8 &&= \color{red}{1{,}232} \\
\big((AB)C\big)D:&\quad 20\times 2\times 30 + 20\times 30\times 12 + 20\times 12\times 8 &&= 10{,}320 \\
(A(BC)D):&\quad 2\times 30\times 12 + 20\times 2\times 12 + 20\times 12\times 8 &&= 3{,}120
\end{aligned}$$

Counting the Number of Parenthesizations

- Brute-force solution: Find them all and pick the smallest!
- How many ways can we parenthesize the product of a matrix chain?
  - Denote $P(n)$: The number of alternative parenthesizations of a sequence of n matrices.
  - We can split a sequence of n matrices between the kth and $(k+1)$st matrices for any $k = 1,2,\ldots,n-1$ and then parenthesize the two resulting subsequences independently.

$$(A_1 A_2 \ldots A_k)(A_{k+1} A_{k+2} \ldots A_n)$$

  - Sum over all $k = 1,2,\ldots,n-1$ with recursive calculation.

厦门大学信息学院

SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系

Computer Science Department of Xiamen University

Counting the Number of Parenthesizations

- We obtain the recursive equation:

$$P(n) = \begin{cases} 1 & n = 1 \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & n \geq 2 \end{cases}$$

- The solution to the above recursive equation is $\Omega(2^n)$.

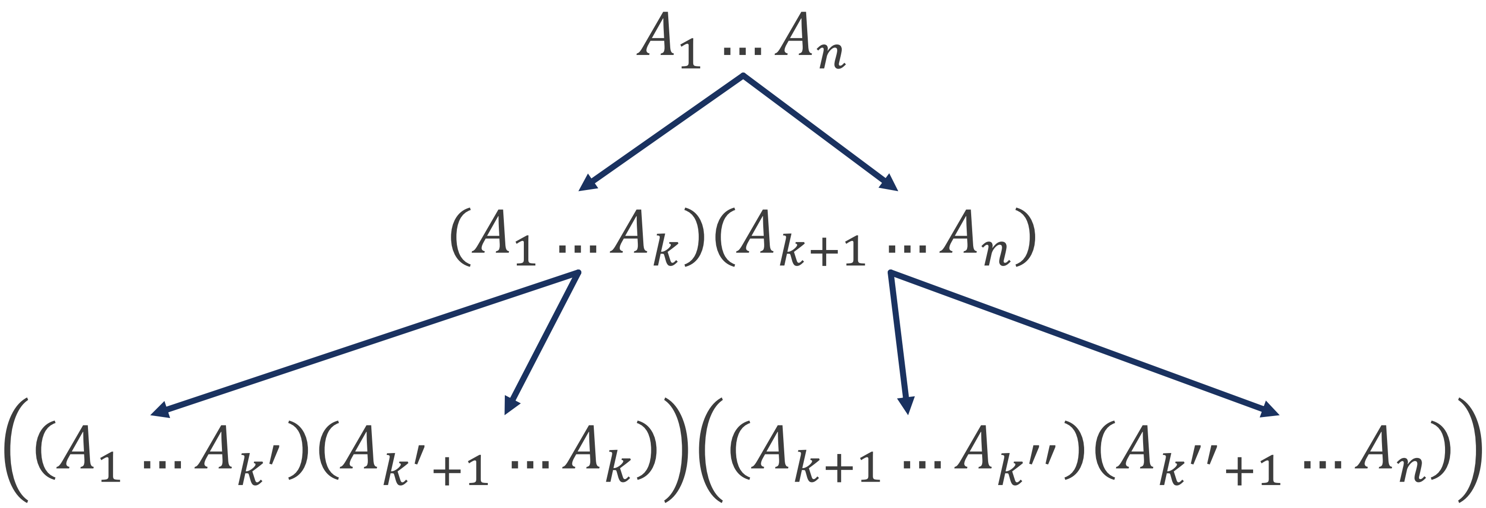
Optimal Substructure

- Assume we have found an optimal solution for parenthesizing $A_1A_2 \dots A_n$.
- The optimal parenthesization must be with some k, for $1 \leq k \leq n-1$:

$$(A_1 \dots A_k)(A_{k+1} \dots A_n)$$

- Then, the parenthesization for $A_1 \dots A_k$ and $A_{k+1} \dots A_n$ must also be optimal.
    - Why? Prove by contradiction again!
- We find the optimal substructure: An optimal solution to an instance of the matrix-chain multiplication is constructed by the optimal solutions to subproblems.

Optimal Substructure



$$A_1 \ldots A_n$$

$$(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$$

$$\big((A_1 \ldots A_{k'})(A_{k'+1} \ldots A_k)\big)\big((A_{k+1} \ldots A_{k''})(A_{k''+1} \ldots A_n)\big)$$

- Therefore, we need to consider the optimal solution for $A_i A_{i+1} \cdots A_j$, for arbitrary $1 \leq i \leq j \leq n$.

Recursive Equation

- We define $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ and let $m[i,j]$=the minimum number of multiplications needed to compute $A_{i\ldots j}$.
- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \leq k < j$:

$$m[i,j] = \ m[i,k] \quad + \quad m[k+1,j] \quad + \quad p_{i-1}p_kp_j$$

Optimal cost to calculate $A_{i\ldots k}$

Optimal cost to calculate $A_{k+1\ldots j}$

Cost to calculate $(A_{i\ldots k})(A_{k+1\ldots j})$

- Now, if we have known all the optimal costs to the small instances, how to construct the optimal cost to the current instance?

Iterate over all k for $i \leq k < j$ and select the minimum one!

Recursive Equation

- There are $j - i$ possible values for k: $k = i, i + 1, \dots, j - 1$.
- Minimizing the cost of parenthesizing $A_{i \dots j}$ becomes:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + p_{i-1}p_k p_j & i < j \end{cases}$$

- The original problem is solved by calculating $m[1,n]$.
- How to calculate?

Recursion does lots of repeated computation, which isn't faster than brute-force approach. We should apply bottom-up approach with dynamic programming!

Filling Table

- To avoid repeated computation, we use a table to store computed values of $m[i,j]$.
- Given the recursive equation

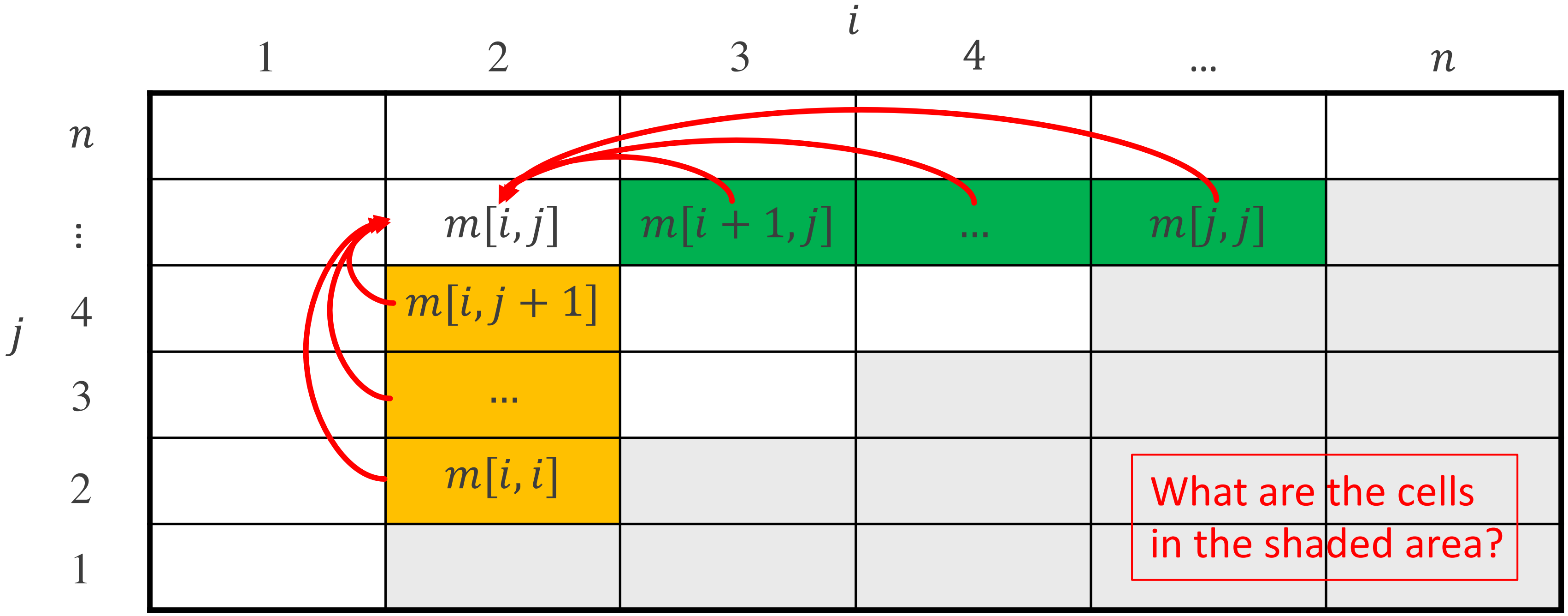
$$m[i,j] = \min_{i \le k < j} m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

what components do we need to calculate $m[i,j]$?

  - Fix i, all $m[i,k]$ for $i \le k < j$.
  - Fix j, all $m[k+1,j]$ for $i \le k < j$.

Filling Table

- Calculating $m[i, j]$ only requires:
  - $m[k+1, j]$ for $i \leq k < j$: The columns behind $m[i, j]$ on the same row.
  - $m[i, k]$ for $i \leq k < j$: The rows below $m[i, j]$ on the same column.



What are the cells in the shaded area?

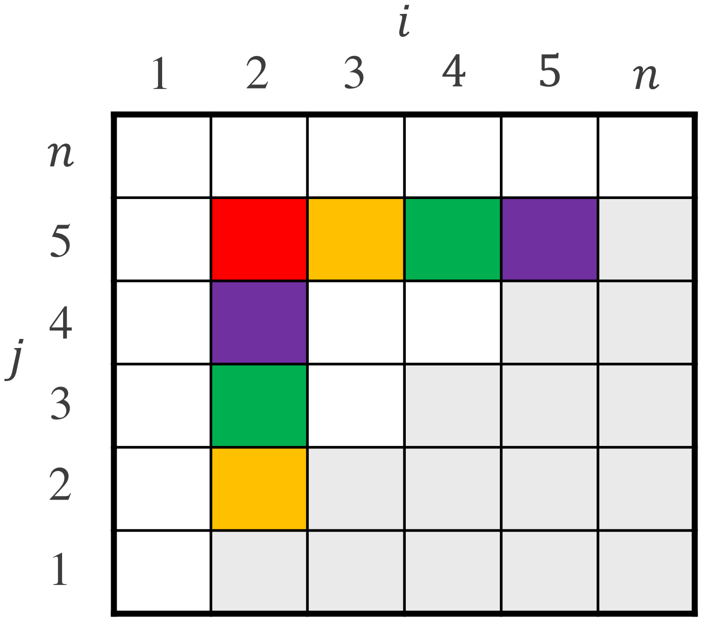
Example

- Given the recursive equation:

$$\min_{i\leq k<j} m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$$

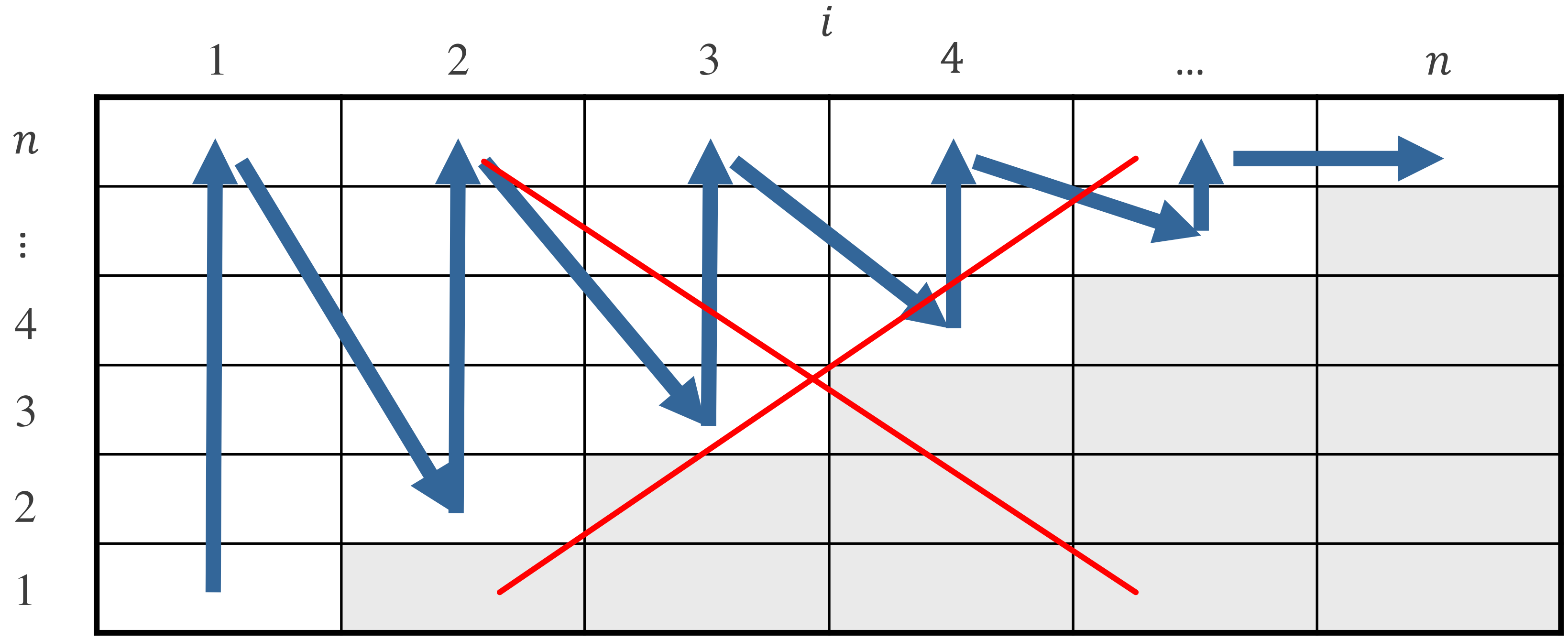
- Calculating $m[2,5]$:

$$\min\begin{cases} m[2,2]+m[3,5]+p_1p_2p_5 & k=2 \\ m[2,3]+m[4,5]+p_1p_3p_5 & k=3 \\ m[2,4]+m[5,5]+p_1p_4p_5 & k=4 \end{cases}$$



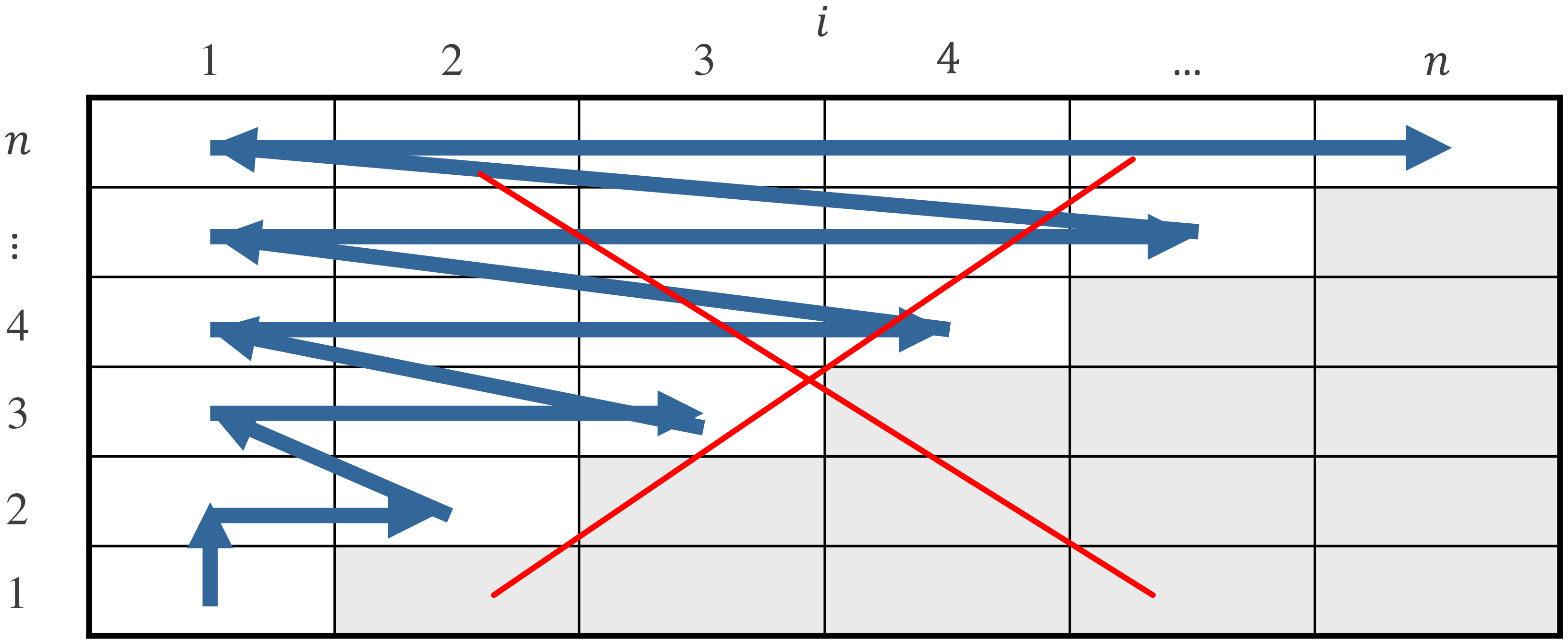
Filling Table

- Can we filling the table in this order?



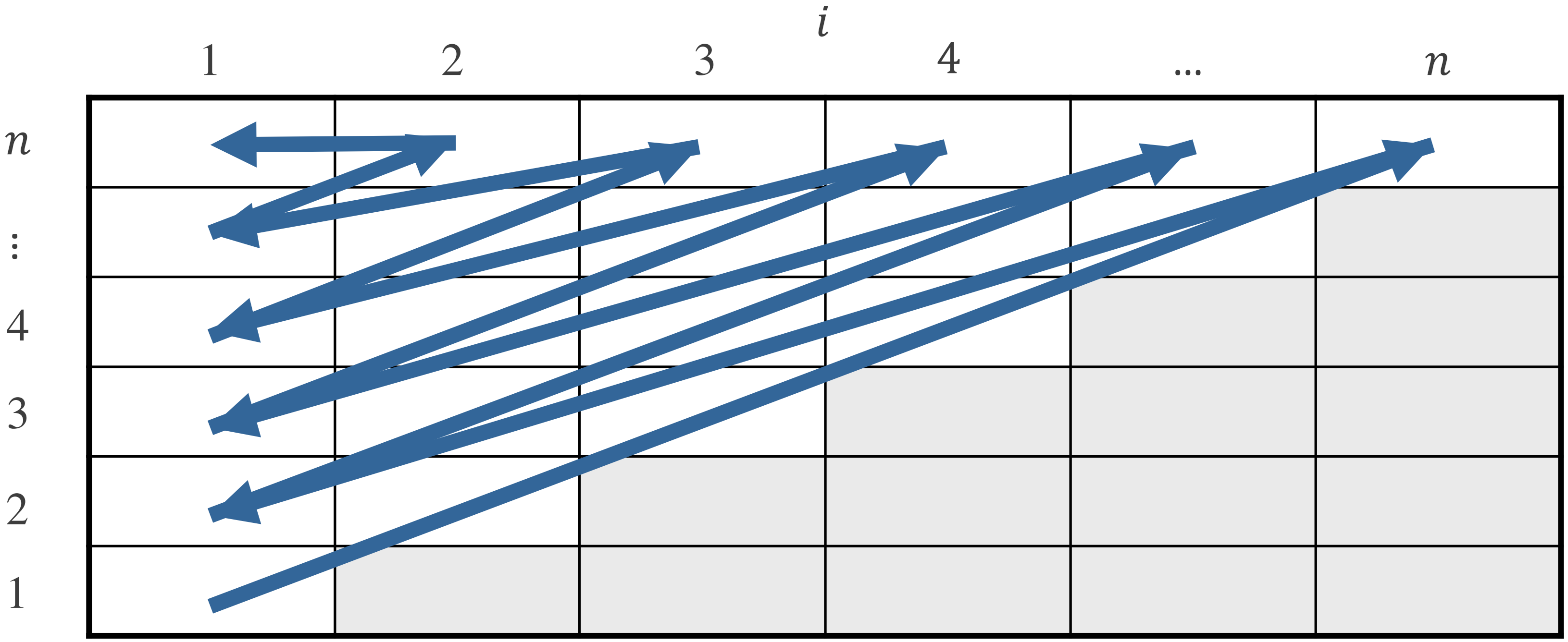
Filling Table

- How about this order?



Filling Table

- Filling the table diagonal by diagonal.



Example

- Given:
  - A_1: 10×100 $(p_0 \times p_1)$
  - A_2: 100×5 $(p_1 \times p_2)$
  - A_3: 5×50 $(p_2 \times p_3)$

	1	2	3
3	7500	25000	0
2	5000	0	
1	0		

- Calculate:
  - $m[i, i] = 0$ for $i = 1{,}2{,}3.$
  - (A_1A_2): $m[1, 2] = m[1{,}1] + m[2{,}2] + p_0p_1p_2 = 0 + 0 + 10 \times 100 \times 5 = 5{,}000.$
  - (A_2A_3): $m[2, 3] = m[2{,}2] + m[3{,}3] + p_1p_2p_3 = 0 + 0 + 100 \times 5 \times 50 = 25{,}000.$
  - $(A_1(A_2A_3))$: $m[1{,}1] + m[2{,}3] + p_0p_1p_3 = 75{,}000.$
  - $((A_1A_2)A_3)$: $m[1{,}2] + m[3{,}3] + p_0p_2p_3 = 7{,}500.$

Reconstructing the Optimal Solution

- Again, we need additional information to maintain the optimal solution to the optimal cost.
- Let $s[i,j]$=a value of k at which we can split the product $A_{i\dots j}$ in order to obtain an optimal parenthesization.

Pseudocode

```
DPMatrixChain($p$)
**for** $i \leftarrow 1$ **to** $n$ **do**
    $m[i,i] \leftarrow 0$
**for** $c \leftarrow 2$ **to** $n$ **do**
    **for** $i \leftarrow 1$ **to** $n - c + 1$ **do**
        $j \leftarrow i + c - 1$
        $m[i,j] \leftarrow \infty$
        **for** $k \leftarrow i$ **to** $j - 1$ **do**
            $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
            **if** $q < m[i,j]$ **then**
                $m[i,j] \leftarrow q;\ s[i,j] \leftarrow k$
**return** $m$ and $s$
```

First diagonal

From 2nd to nth diagonal

Given c, determine the jth column and ith row

Running time: $\Theta(n^3)$

Construct the Optimal Solution

- $s[i,j]$ stores the value of k such that the optimal solution of $A_{i\ldots j}$ splits the product int $A_{i\ldots k}$ and $A_{k+1\ldots j}$.

		i					
		1	2	3	4	5	6
	6	3	3	3	5	5	-
	5	3	3	3	4	-	
	4	3	3	3	-		
j	3	1	2	-			
	2	1	-				
	1	-					

$$A_{1\ldots n} = A_{1\ldots s[1,n]} \cdot A_{s[1,n]+1\ldots n}$$

$$A_{1\ldots 6} = A_{1\ldots 3}\, A_{4\ldots 6}$$

$$A_{1\ldots 3} = A_{1\ldots 1} A_{2\ldots 3}$$

$$A_{4\ldots 6} = A_{4\ldots 5} A_{6\ldots 6}$$

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系
Computer Science Department of Xiamen University

```
PrintParens($s$, $i$, $j$)
if $i = j$ then print "$A_i$"
else print "("
    PrintParens($s$, $i$, $s[i, j]$)
    PrintParens($s$, $s[i, j] + 1$, $j$)
    print ")"
```

		i					
		1	2	3	4	5	6
	6	3	3	3	5	5	-
	5	3	3	3	4	-	
	4	3	3	3	-		
j	3	1	2	-			
	2	1	-				
	1	-					

Function call	Printed value
PrintParens(s, 1, 6)	(
PrintParens(s, 1, 3)	((
PrintParens(s, 1, 1)	$((A_1$
PrintParens(s, 2, 3)	$((A_1($
PrintParens(s, 2, 2)	$((A_1(A_2$
PrintParens(s, 3, 3)	$((A_1(A_2A_3$
PrintParens(s, 4, 6)	$((A_1(A_2A_3))($
PrintParens(s, 4, 5)	$((A_1(A_2A_3))(($
PrintParens(s, 4, 4)	$((A_1(A_2A_3))((A_4$
PrintParens(s, 5, 5)	$((A_1(A_2A_3))((A_4A_5$
PrintParens(s, 6, 6)	$((A_1(A_2A_3))((A_4A_5)A_6$
Finish	$((A_1(A_2A_3))((A_4A_5)A_6))$

Classroom Exercise

- Given the following matrices, fill in the table to get the optimal parenthesization:
  - $A_1: 2\times2$
  - $A_2: 2\times4$
  - $A_3: 4\times2$
  - $A_4: 2\times6$

Classroom Exercise

		i			
		1	2	3	4
	4	48	40	48	0
j	3	24	16	0	
	2	16	0		
	1	0			

$m[i,j]$

		i			
		1	2	3	4
	4	3	3	3	-
j	3	1	2	-	
	2	1	-		
	1	-			

$s[i,j]$

$$((A_1(A_2A_3))A_4)$$

THE LONGEST-COMMON-SUBSEQUENCE PROBLEM

The Longest-Common-Subsequence Problem

- In biological applications, one goal of comparing two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.
- One way to measure the similarity between S_1 and S_2 is by finding a third strand S_3: the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively.

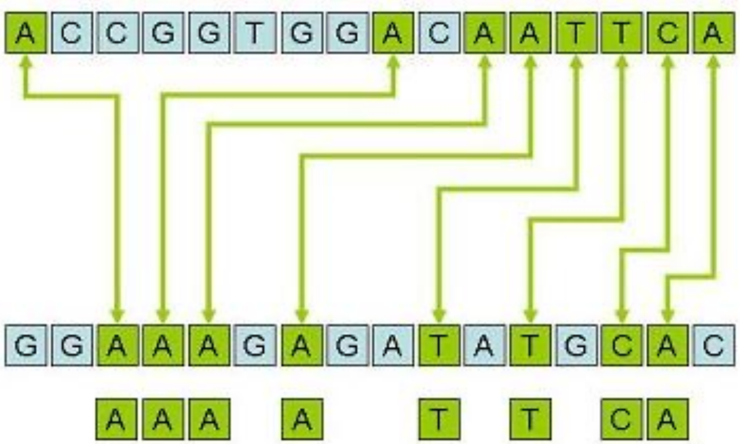
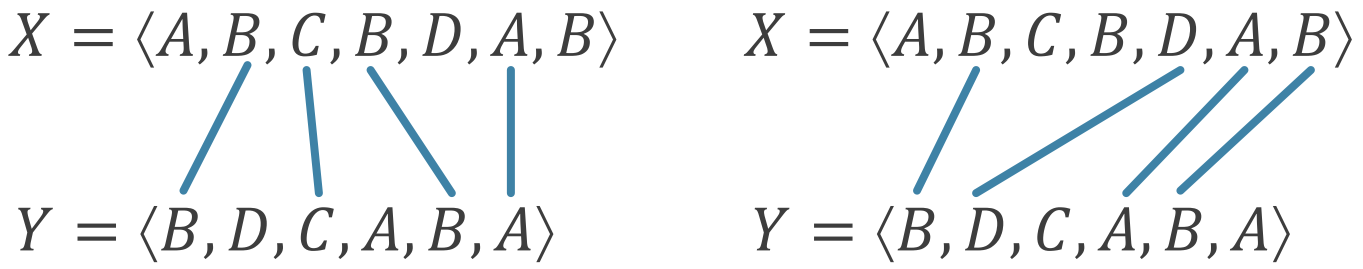


Image source: https://www.8bitavenue.com/dynamic-programming-longest-common-subsequence/

The Longest-Common-Subsequence Problem

- Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a subsequence (子序列) of X if there exists a strictly increasing sequence $i_1, i_2, \dots, i_k$ of indices of X such that for all $j = 1,2, \dots, k$, we have $x_{i_j} = z_j$.
- Given two sequences X and Y, we say that a sequence Z is a common subsequence (公共子序列) of X and Y if Z is a subsequence of both X and Y.
- In the longest-common-subsequence (最长公共子序列) problem, we are given two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y.

Example



$X = \langle A, B, C, B, D, A, B \rangle$ $\quad$ $X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$ $\quad$ $Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y ($length = 4$).
- $\langle B, C, A \rangle$ is a common subsequence, but not the longest.

Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y.
- There are 2^m subsequences of X to check.
- Each subsequence takes $\Theta(n)$ time to check.
  - Scan Y from the first element, and check if it is matched with the subsequence.
- Running time: $\Theta(n2^m)$.

Optimal Substructure

- Given two sequences $X_m = \langle x_1, x_2, \dots, x_m \rangle$ and $Y_n = \langle y_1, y_2, \dots, y_n \rangle$, assume the $Z_k = \langle z_1, z_2, \dots, z_k \rangle$ is a LCS.
- If the last element of X_m and Y_n is same, i.e. $x_m = y_n$, they are also same as the last element of Z_k:

$$z_k = x_m = y_n = \blacksquare$$

$$X_m = \langle X_{m-1}, \blacksquare \rangle, \qquad Y_n = \langle Y_{n-1}, \blacksquare \rangle, \qquad Z_k = \langle Z_{k-1}, \blacksquare \rangle$$

- In this case, Z_{k-1} is also a LCS of X_{m-1} and Y_{n-1}.
  - Prove by contradiction.

Optimal Substructure

- Given two sequences $X_m = \langle x_1, x_2, \dots, x_m \rangle$ and $Y_n = \langle y_1, y_2, \dots, y_n \rangle$, assume the $Z_k = \langle z_1, z_2, \dots, z_k \rangle$ is a LCS.
- If the last element of X_m and Y_n is different, i.e. $x_m \neq y_n$:
  - If $z_k \neq x_m$,

    $$X_m = \langle X_{m-1}, \blacksquare \rangle, \qquad Y_n, \qquad Z_k = \langle Z_{k-1}, \blacklozenge \rangle$$

    no matter $z_k = y_n$ or not, Z_k is a LCS of X_{m-1} and Y_n.
  - If $z_k \neq y_n$,

    $$X_m, \qquad Y_n = \langle Y_{n-1}, \blacksquare \rangle, \qquad Z_k = \langle Z_{k-1}, \blacklozenge \rangle$$

    no matter $z_k = x_m$ or not, Z_k is a LCS of X_m and Y_{n-1}.

Optimal Substructure

- Denote $c[i, j]$ as the length of a LCS of the sequences $X_i = \langle x_1, x_2, \dots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \dots, y_j \rangle$.
- If $x_i = y_j$, the LCS composes of the LCS of X_{i-1} and Y_{j-1} with x_i:

$$c[i, j] = c[i - 1, j - 1] + 1.$$

- If $x_i \neq y_j$, the LCS is either the LCS of X_{i-1} and Y_j, or the LCS of X_i and Y_{j-1}. We choose the longer one:

$$c[i, j] = \max\{c[i - 1, j], c[i, j - 1]\}.$$

Example

- Case 1: $x_i = y_j$

$$X_i = \langle A, B, D, E \rangle, \qquad Y_j = \langle Z, B, E \rangle, \qquad Z_k = \langle B, E \rangle$$

$$X_{i-1} = \langle A, B, D \rangle, \qquad Y_{j-1} = \langle Z, B \rangle, \qquad Z_{k-1} = \langle B \rangle$$

- Case 2: $x_i \neq y_j$

$$X_i = \langle A, B, D, G \rangle, \qquad Y_j = \langle Z, B, D \rangle, \qquad Z_k = \langle B, D \rangle$$

Z_k is either the LCS of X_{i-1} and Y_j ($\langle B, D \rangle$) or the LCS of X_i and Y_{j-1} ($\langle B \rangle$).

Recursive Equation

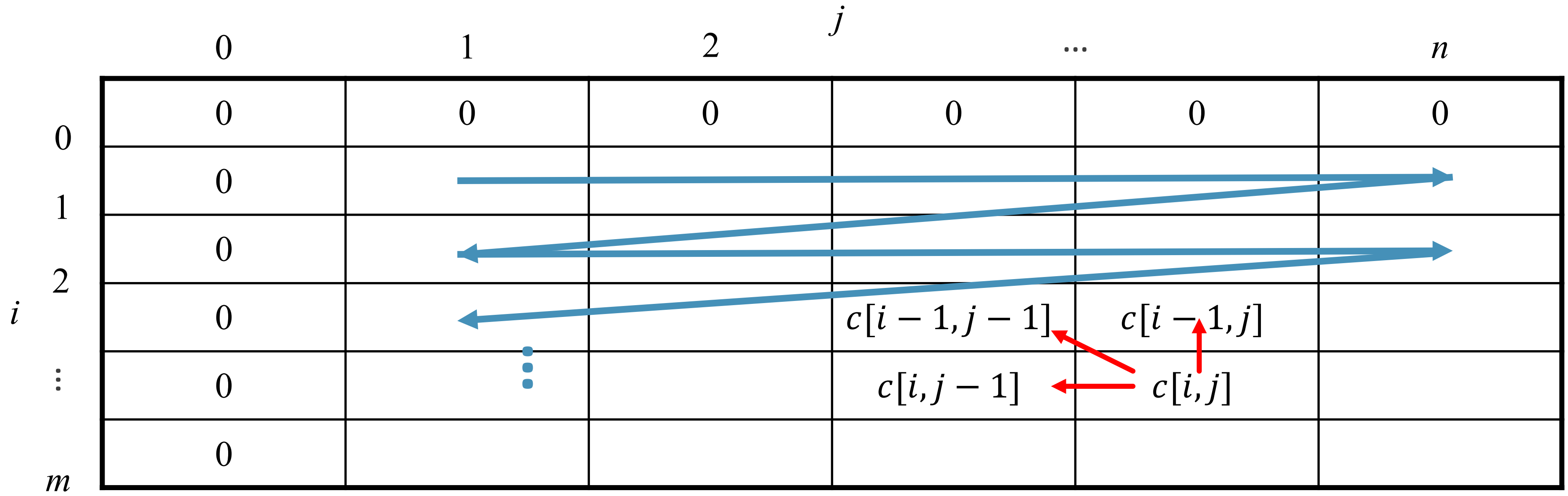
- The recursive equation:

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

  - In the case of $i = 0$ or $j = 0$, the length of LCS is 0 because it is empty.
- Using recursion, is there any repeated computation?

Filling Table

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & i,j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

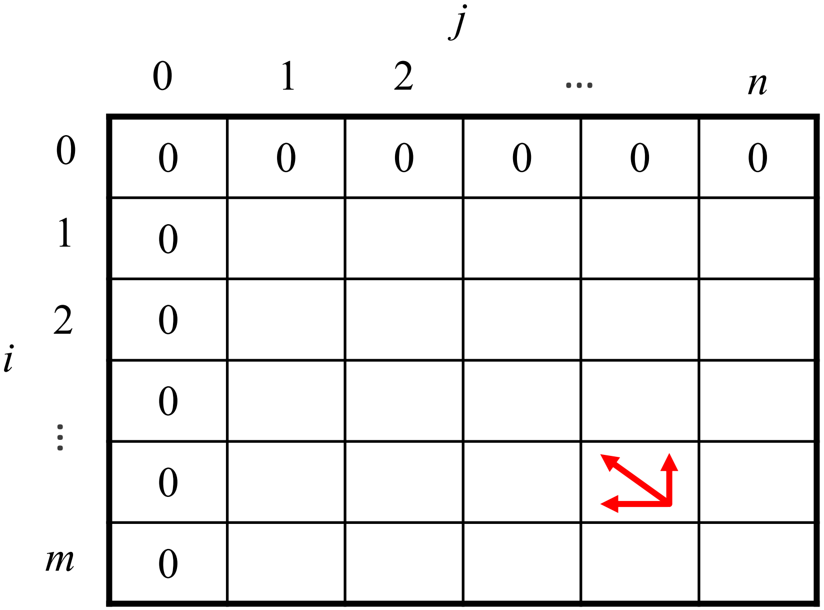


	0	1	2	j	…	n
0	0	0	0	0	0	0
1	0					
2	0					
i	0			$c[i-1,j-1]$	$c[i-1,j]$	
⋮	0	⋮		$c[i,j-1]$	$c[i,j]$	
m	0					

Additional Information

- $c[i,j]$ only records the length of LCS, we need to record additional information for constructing LCS.
- $b[i,j]$ records the choices made to obtain the optimal value.

```
if $x_i = y_j$
    $b[i,j] = $ “ ↖ ”
else if $c[i-1,j] \geq c[i,j-1]$
        $b[i,j] = $ “ ↑ ”
    else
        $b[i,j] = $ “ ← ”
```



	0	1	2		...	n
0	0	0	0	0	0	0
1	0					
2	0					
	0					
⋮	0					
m	0					

Pseudocode

```
LCSLength($X, Y, m, n$)
for $i \leftarrow 1$ to $m$ do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $n$ do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ to $m$ do
    for $j \leftarrow 1$ to $n$ do
        if $x_i = y_j$ then
            $c[i, j] \leftarrow c[i-1, j-1] + 1$
            $b[i, j] \leftarrow$ “ ↖ ”
        else if $c[i-1, j] \geq c[i, j-1]$ then
            $c[i, j] \leftarrow c[i-1, j]$
            $b[i, j] \leftarrow$ “ ↑ ”
        else $c[i, j] \leftarrow c[i, j-1]$
            $b[i, j] \leftarrow$ “ ← ”
return $c$ and $b$
```

Running time:
$\Theta(nm)$

Example

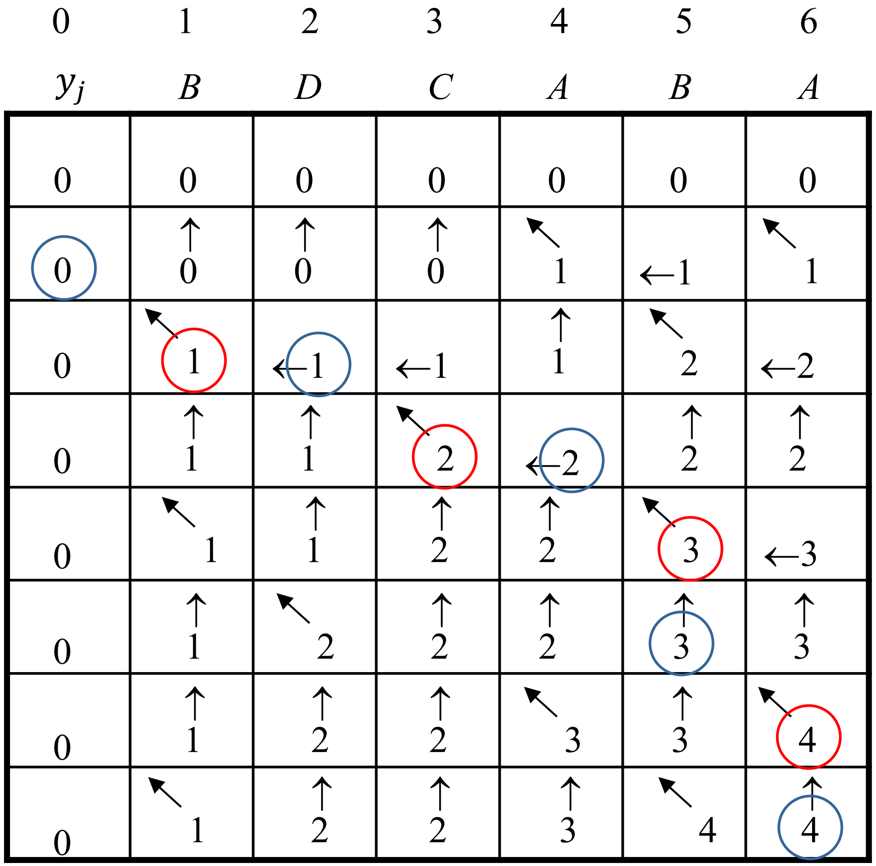
$$X = \langle B, D, C, A, B, A \rangle$$
$$Y = \langle A, B, C, B, D, A \rangle$$

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

Example

- Constructing a LCS: Start at $b[m,n]$ and follow the arrows. When we encounter a "↖" in $b[i,j]$, it means $x_i = y_j$ and it is an element of the LCS.

```
PrintLCS($b, X, i, j$)
if $i = 0$ or $j = 0$ then return 0
if $b[i,j]$ = " ↖ " then
    PrintLCS($b, X, i-1, j-1$)
    print $x_i$
else if $b[i,j]$ = " ↑ " then
    PrintLCS($b, X, i-1, j$)
else PrintLCS($b, X, i, j-1$)
```

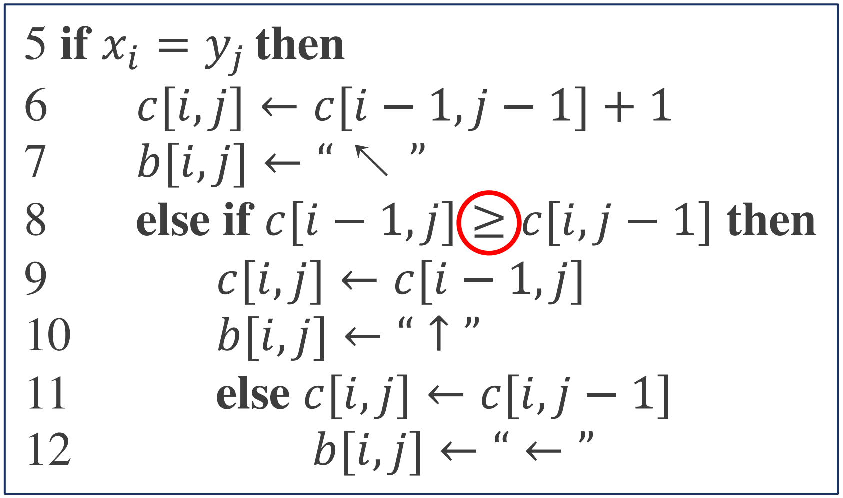


		0	1	2	3	4	5	6
	y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B	0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Output: BCBA

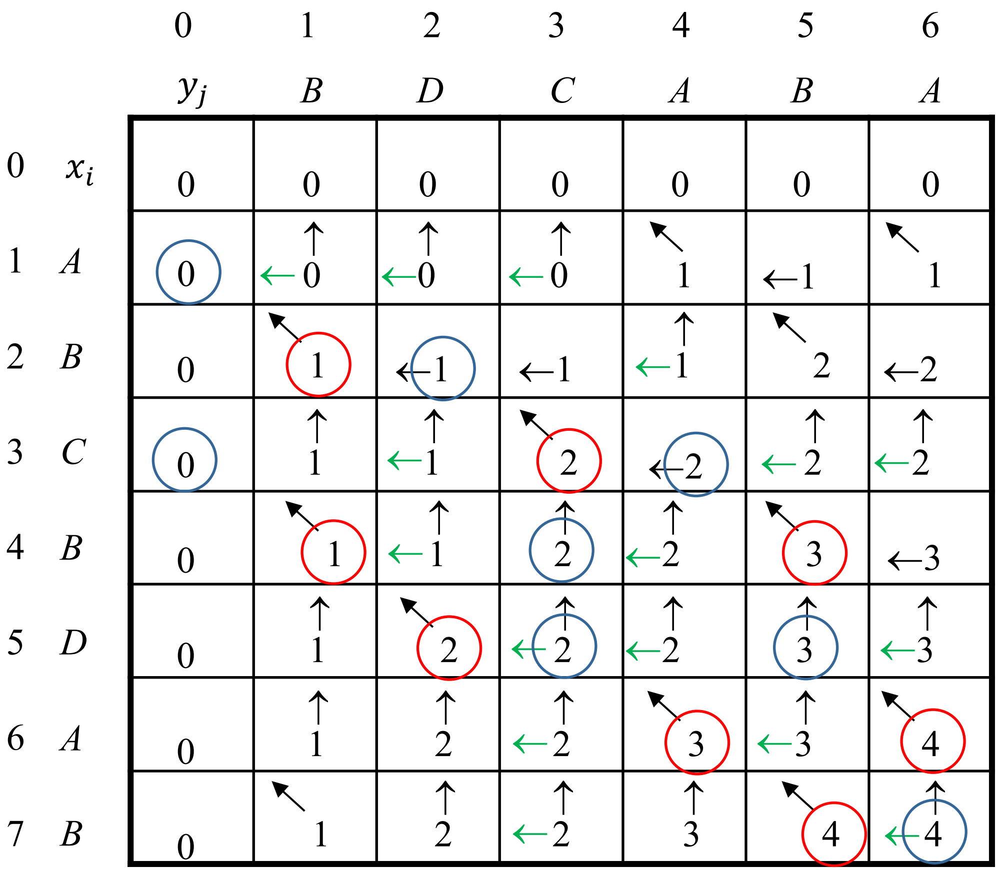
Uniqueness

- This algorithm is deterministic. However, the LCS is not unique. Why?
- We set $b[i,j] \leftarrow$ “ $\uparrow$ ” when $c[i-1,j] = c[i,j-1]$. However, $b[i,j] \leftarrow$ “ $\leftarrow$ ” is also optimal in this case.



```
if $x_i = y_j$ then
    $c[i,j] \leftarrow c[i-1,j-1]+1$
    $b[i,j] \leftarrow$ “ $\nwarrow$ ”
    else if $c[i-1,j] \geq c[i,j-1]$ then
        $c[i,j] \leftarrow c[i-1,j]$
        $b[i,j] \leftarrow$ “ $\uparrow$ ”
        else $c[i,j] \leftarrow c[i,j-1]$
            $b[i,j] \leftarrow$ “ $\leftarrow$ ”
```

Uniqueness



		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑←0	↑←0	↑←0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑←1	↖2	←2
3	C	0	↑1	↑←1	↖2	←2	↑←2	↑←2
4	B	0	↖1	↑←1	↑2	↑←2	↖3	←3
5	D	0	↑1	↖2	↑←2	↑←2	↑3	↑←3
6	A	0	↑1	↑2	↑←2	↖3	↑←3	↖4
7	B	0	↖1	↑2	↑←2	↑3	↖4	↑←4

$BCBA$

$BCAB$

$BDAB$

Space Improvement

- How each entry $c[i, j]$ is computed?
  - It depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$.
- If we only need the length of the LCS, we only need the row being computed and the previous row.
  - We can reduce the asymptotic space requirements by storing only these two rows.

Classroom Exercise

Draw the table of LCS of the following sequences, and find a LCS:

$$X = \langle A, C, D, A \rangle \qquad Y = \langle A, D, C, A \rangle$$

Classroom Exercise

		0	1	2	3	4
		y_j	A	D	C	A
0	x_i	0	0	0	0	0
1	A	0	↖ 1	← 1	← 1	↖ 1
2	C	0	↑ 1	↑ 1	↖ 2	← 2
3	D	0	↑ 1	↖ 2	↑ 2	↑ 2
4	A	0	↖ 1	↑ 2	↑ 2	↖ 3

ACA

0/1 KNAPSACK PROBLEM

0/1 Knapsack Problem

0/1 knapsack (0/1背包) problem:

- There are n items: the ith item is worth v_i dollars and weights w_i kg.
- The capacity of knapsack is W kg.
- Items must be taken entirely or left behind.
- Which items should we take to maximize the total value?

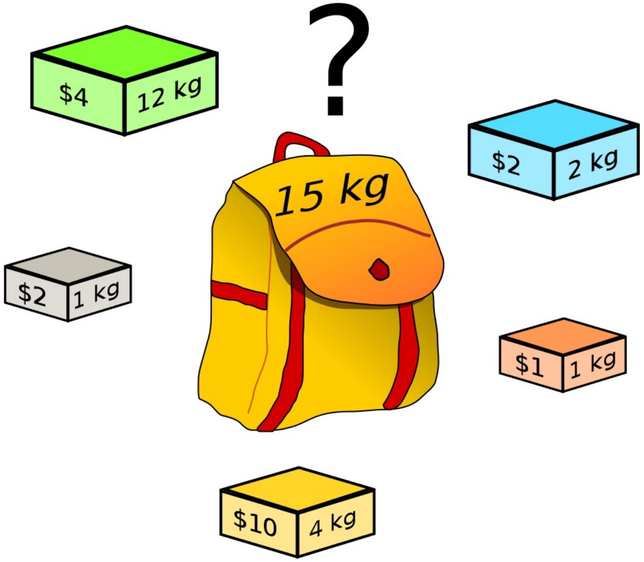


Image source: https://en.wikipedia.org/wiki/Knapsack_problem

0/1 Knapsack Problem

- Mathematical description: Given an item set $s = \langle 1,2,\dots,n \rangle$, and two n-tuples of positive numbers $\langle v_1, v_2, \dots, v_n \rangle$ and $\langle w_1, w_2, \dots, w_n \rangle$, and $W > 0$, we wish to determine the subset $s' \subseteq \{1,2,\dots,n\}$ that

$$\text{maximize} \sum_{i \in s'} v_i$$

$$\text{subject to} \sum_{i \in s'} w_i \leq W$$

Example

- Weight capacity $W = 5\text{kg}$.
- The possible ways to fill the knapsack:
  - {1, 2, 3} has value $37 with weight 4kg.
  - {3, 4} has value $35 with weight 5kg.
  - {1, 2, 4} has value $42 with weight 5kg. (optimal)

i	v_i	w_i
1	$10	1kg
2	$12	1kg
3	$15	2kg
4	$20	3kg

Optimal Substructure

- The variables we should use in recursive equation must be the maximum profit V.
- How a V can be decomposed into the Vs with smaller instance?
  - Item subset?
  - Smaller weight?

Optimal Substructure

- Consider the most valuable load that weights at most w kg.
- If item i is in the load and we remove it, the remaining load must be the most valuable load weighing at most $w - w_i$ that can be taken from the remaining $i - 1$ items.
  - Prove by contradiction: if the remaining load is not the most valuable, there exists a more valuable load and adding item i into it makes the original load not the most valuable.

Optimal Substructure

- $V[i, w]$: the maximum profit that can be obtained from items 1 to i, if the knapsack has size w.
  - Case 1: take item i

$$V[i, w] = V[i-1, w-w_i] + v_i$$

  - Case 2: do not take item i

$$V[i, w] = V[i-1, w]$$

- How to decide whether take item i or not?

Simply compare and select the maximum one.

Recursive Equation

- The recursive equation:

$$V[i,w] = \begin{cases} V[i-1,w] & w_i > w \\ \max\{V[i-1,w], V[i-1,w-w_i]+v_i\} & w_i \leq w \end{cases}$$

  - $w_i > w$: we can’t take w_i more than capacity w.
  - $w_i \leq w$: decide whether to take item i or not.

Additional Information

- $V[i,w]$ only records the optimal value, we need to record additional information to record the items we take.
- $b[i,w]$ records the choices made to obtain the optimal value.
  - Case 1: take item i

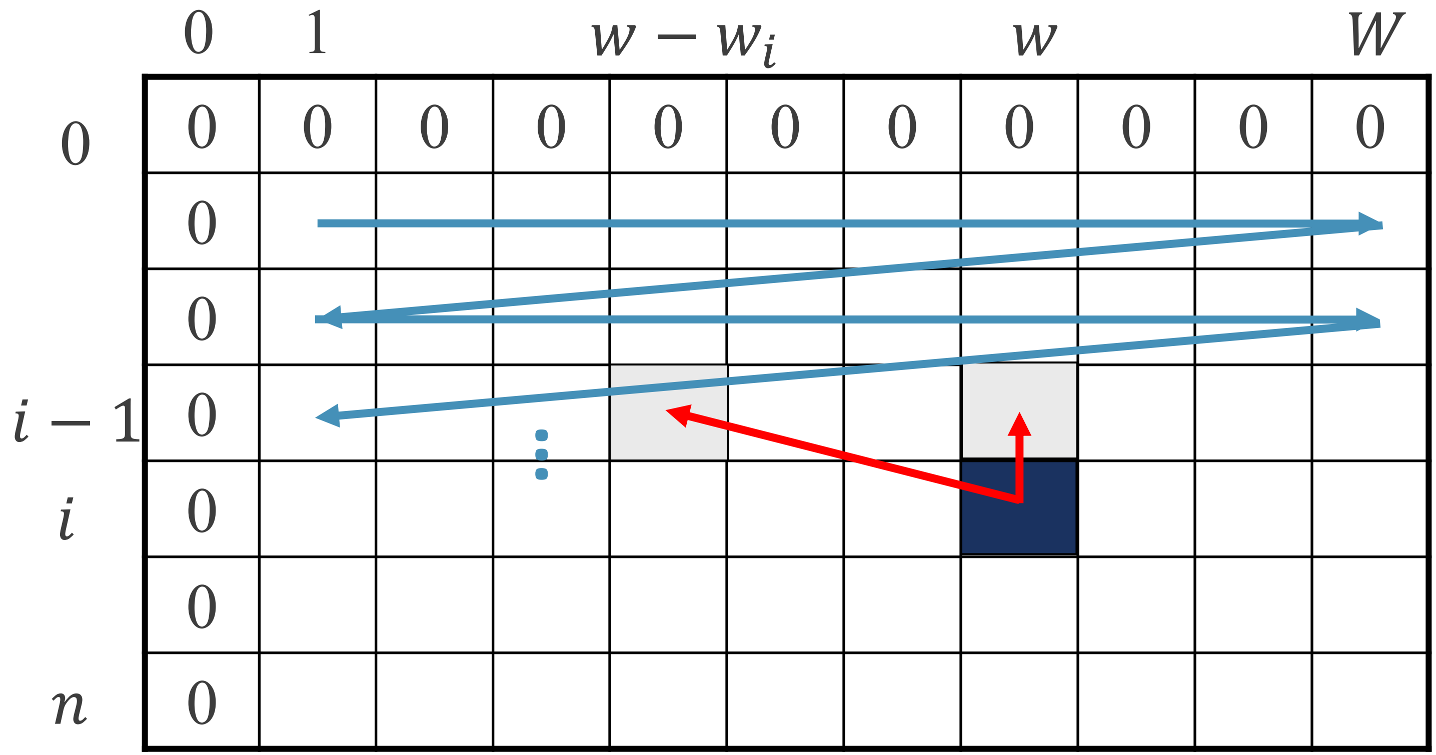
$$b[i,w] = "\nwarrow"$$

  - Case 2: do not take item i

$$b[i,w] = "\uparrow"$$

Filling Table

$$V[i,w] = \max\{V[i-1,w], V[i-1,w-w_i]+v_i\}$$



	0	1			$w-w_i$			w			W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
$i-1$	0										
i	0										
	0										
n	0										

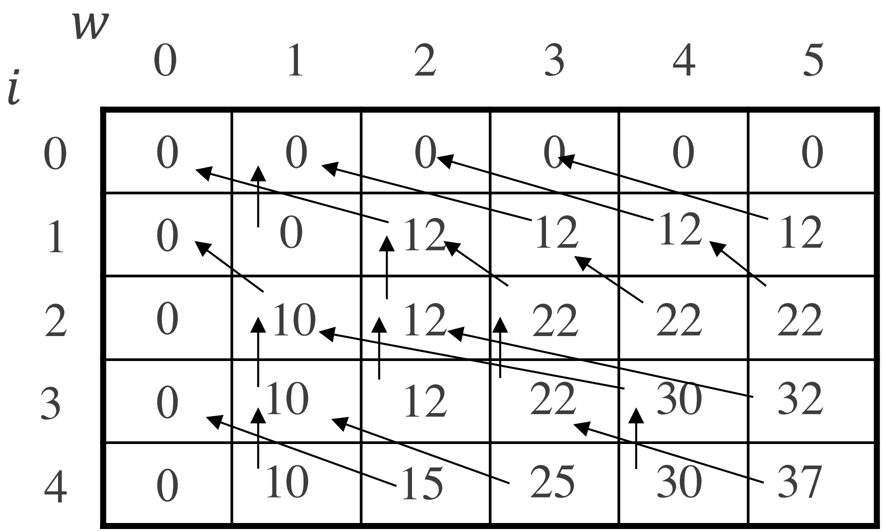
```
DPKnapsack($S, W$)
for $w \leftarrow 0$ to $w_1 - 1$ do $V[1, w] \leftarrow 0$
for $w \leftarrow w_1$ to $W$ do $V[1, w] \leftarrow v_1$
for $i \leftarrow 2$ to $n$ do
    for $w \leftarrow 0$ to $W$ do
        if $w_i > w$ then
            $V[i, w] \leftarrow V[i - 1, w]$
            $b[i, w] \leftarrow " \uparrow "$
        else if $V[i - 1, w] > V[i - 1, w - w_i] + v_i$ then
            $V[i, w] \leftarrow V[i - 1, w]$
            $b[i, w] \leftarrow " \uparrow "$
        else
            $V[i, w] \leftarrow V[i - 1, w - w_i] + v_i$
            $b[i, w] \leftarrow " \nwarrow "$
return $V$ and $b$
```

Running time: $\Theta(nW)$

$W = 5$

Item i	w_i	v_i
1	2	12
2	1	10
3	3	20
4	2	15

$$V[i,w] = \begin{cases} V[i-1,w] & w_i > w \\ \max\{V[i-1,w], V[i-1,w-w_i]+v_i\} & w_i \le w \end{cases}$$



i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

$V[1,1] = V[0,1] = 0$

$V[1,2] = \max\{12+0, 0\} = 12$

$V[1,3] = \max\{12+0, 0\} = 12$

$V[1,4] = \max\{12+0, 0\} = 12$

$V[1,5] = \max\{12+0, 0\} = 12$

$V[2,1] = \max\{10+0, 0\} = 10$

$V[2,2] = \max\{10+0, 12\} = 12$

$V[2,3] = \max\{10+12, 12\} = 22$

$V[2,4] = \max\{10+12, 12\} = 22$

$V[2,5] = \max\{10+12, 12\} = 22$

$V[3,1] = V[2,1] = 10$

$V[3,2] = V[2,2] = 12$

$V[3,3] = \max\{20+0, 22\} = 22$

$V[3,4] = \max\{20+10, 22\} = 30$

$V[4,5] = \max\{20+12, 22\} = 32$

$V[4,1] = V[3,1] = 10$

$V[4,2] = \max\{15+0, 12\} = 15$

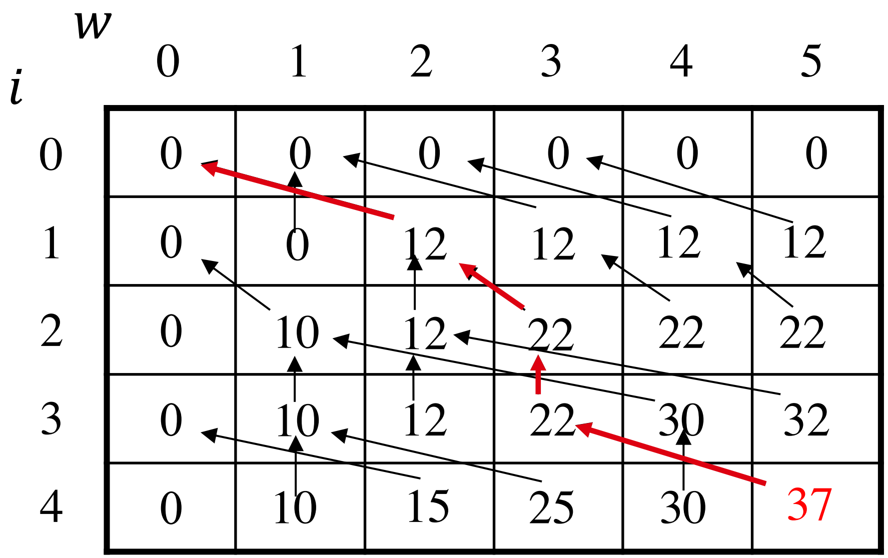
$V[4,3] = \max\{15+10, 22\} = 25$

$V[4,4] = \max\{15+12, 30\} = 30$

$V[4,5] = \max\{15+22, 32\} = 37$

Reconstructing the Optimal Solution

- Start at $V[n,W]$.
- When you go left-up, item i has been taken.
- When you go straight up, item i has not been taken.



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

Classroom Exercise

Fill the table to solve the following 0/1 knapsack problem when $W = 3$.

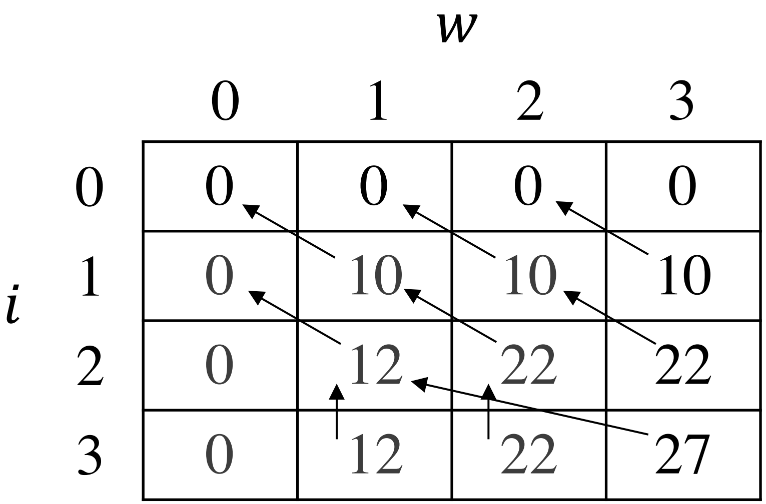
i	v_i	w_i
1	\$10	1kg
2	\$12	1kg
3	\$15	2kg

Classroom Exercise

Solution:

$W = 3$

i	v_i	w_i
1	\$10	1kg
2	\$12	1kg
3	\$15	2kg



w

i	0	1	2	3
0	0	0	0	0
1	0	10	10	10
2	0	12	22	22
3	0	12	22	27

OPTIMAL BINARY SEARCH TREES

Optimal Binary Search Trees

- A binary search tree (BST) (二叉搜索树) is a binary tree of keys that come from an ordered set, such that
  - Each node contains one key.
  - The keys in the left subtree of a given node are less than or equal to the key in that node.
  - The keys in the right subtree of a given node are greater than or equal to the key in that node.

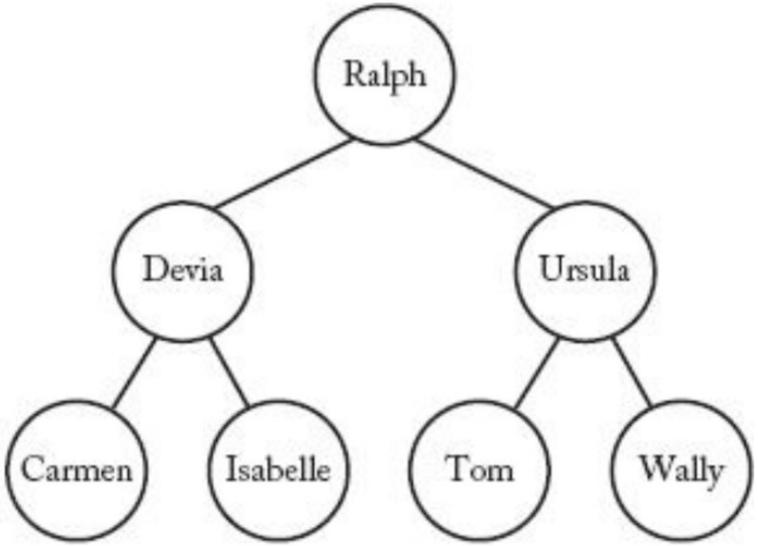


Image source: Figure 3.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees

- The number of comparisons done by search to locate a key is called the search time.
- We want to know the average search time of a BST while the keys do not have the same probability.
  - E.g. Tom is a common name is the United States. It has higher probability to be a search key.
  - Thus, put the node whose key has high probability to lower depth will decrease the average search time.

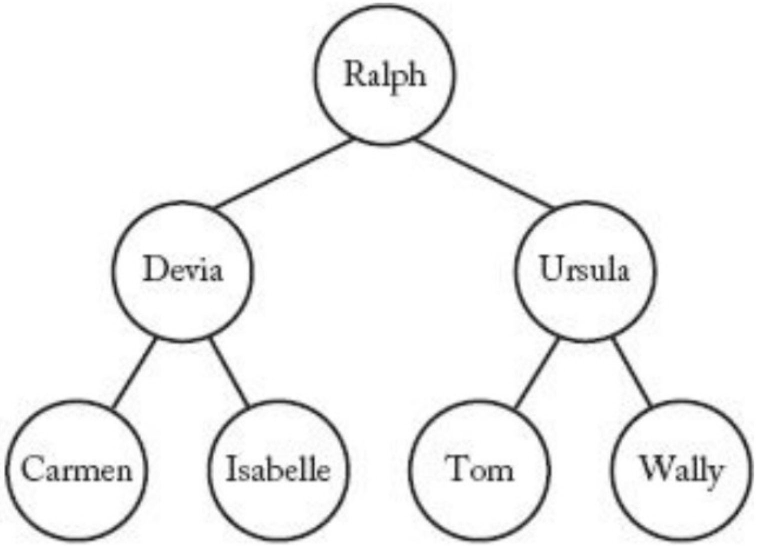
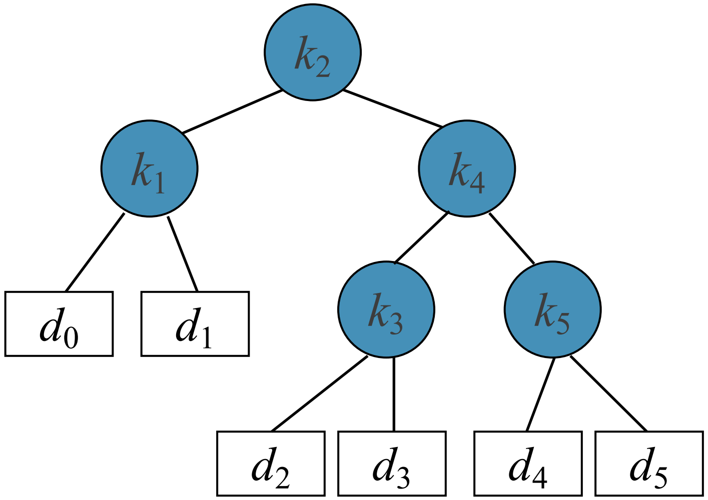


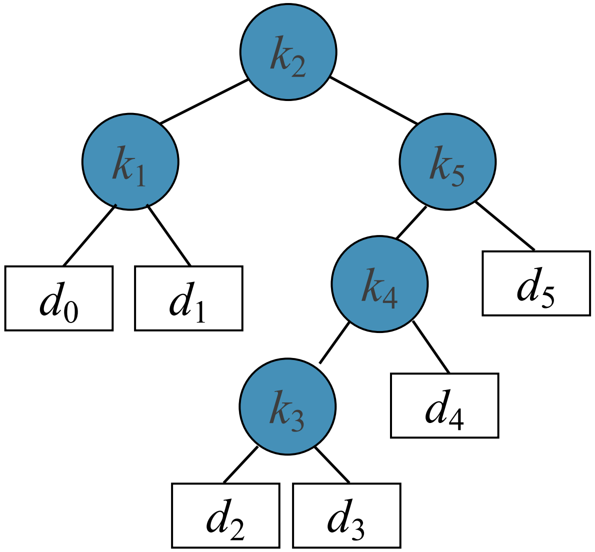
Image source: Figure 3.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees

- Given
  - a sequence $K = \langle k_1, k_2\ \ldots, k_n\rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 \cdots < k_n$)
  - $n + 1$ “dummy keys (虚拟关键字)”$\langle d_0, d_1, d_2, \ldots, d_n\rangle$ when the key is not in K, such that

$$d_0 < k_1 < d_1 < k_2 < \cdots < k_i < d_i < k_{i+1} < \cdots < k_n < d_n$$





Optimal Binary Search Trees

- The search key x is not uniformly distributed. It follows the probability:
  - For each key k_i, we have a probability p_i that a search will be for k_i.
  - For each d_i, we have a probability q_i that a search will correspond to d_i.
- For each search, either some key k_i is found, or some dummy key d_i is found. Therefore, we have:

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Where $\sum_{i=1}^{n} p_i$ is the probability for a successful search and $\sum_{i=1}^{n} q_i$ is the probability for a failed search.

Optimal Binary Search Trees

- Assume that the actual cost of a search is the number of nodes examined, i.e., the depth of the node found by the search in T, plus 1.
- Then the expected cost of a search in T is

$$E(T) = \sum_{i=1}^{n} (d_T(k_i) + 1)\times p_i + \sum_{i=0}^{n} (d_T(d_i) + 1)\times q_i$$

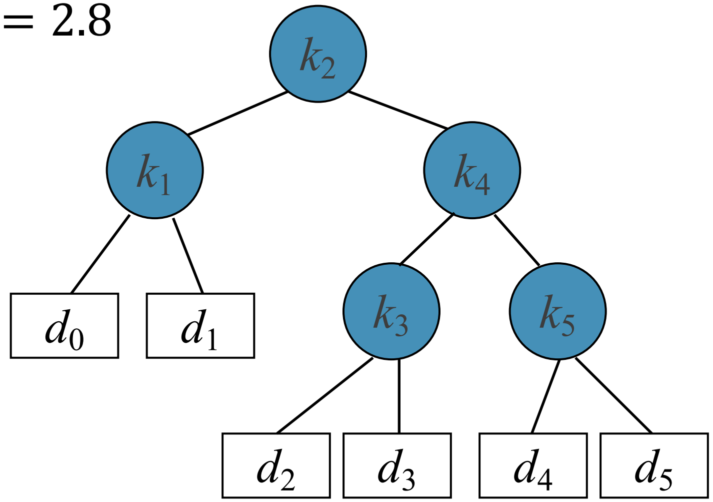
where $d_T(k)$ is the depth of key k in tree T.

- For a given set of probabilities, our goal is to construct a BST whose $E(T)$ is smallest. This tree is called optimal BST (最优二叉搜索树).

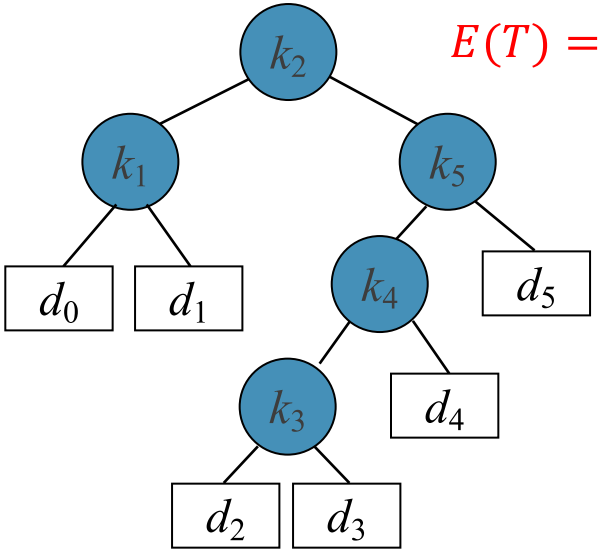
Example

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

$E(T) = 2.8$

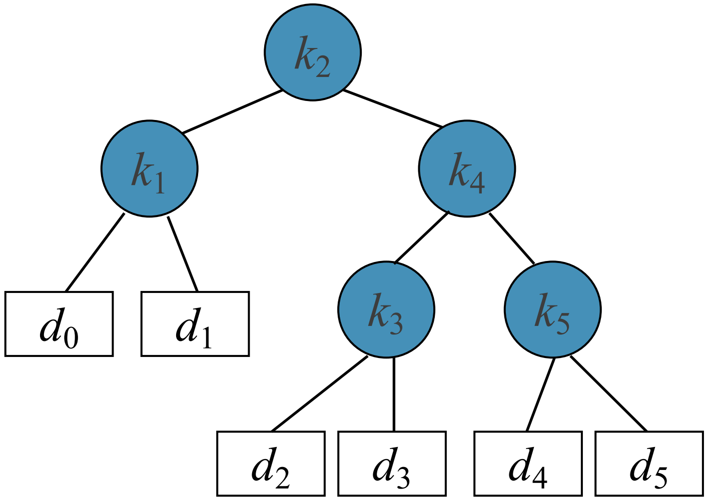


$E(T) = 2.75$



Example

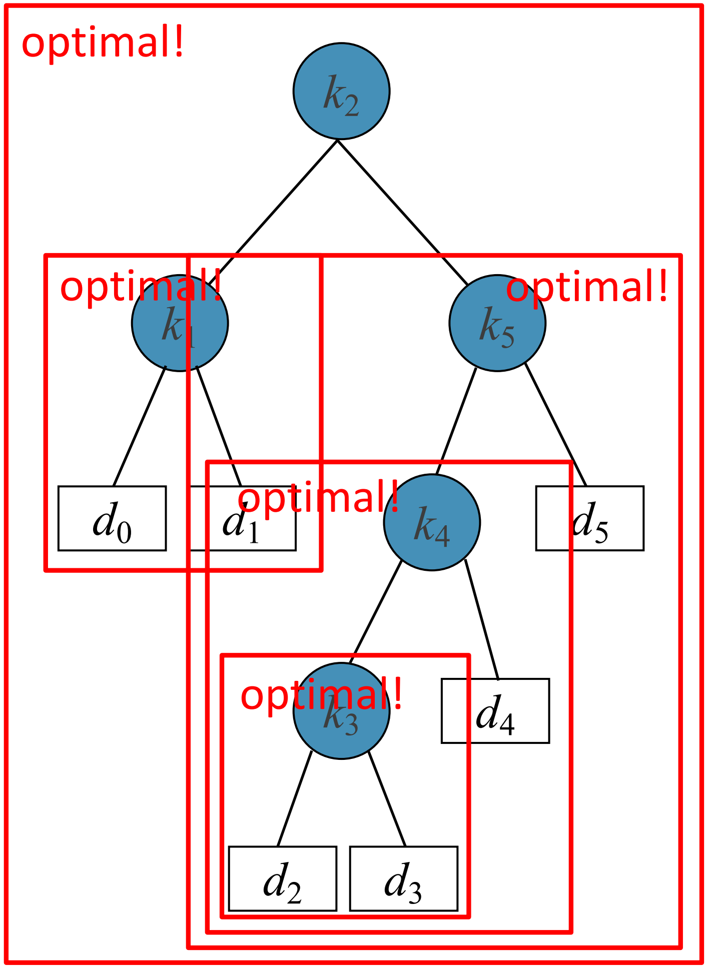
$$E(T) = 2.8$$



node	depth	probability	contribution
k_1	1	0.15	0.3
k_2	0	0.1	0.1
k_3	2	0.05	0.15
k_4	1	0.1	0.2
k_5	2	0.2	0.6
d_0	2	0.05	0.15
d_1	2	0.1	0.3
d_2	3	0.05	0.2
d_3	3	0.05	0.2
d_4	3	0.05	0.2
d_5	3	0.1	0.4

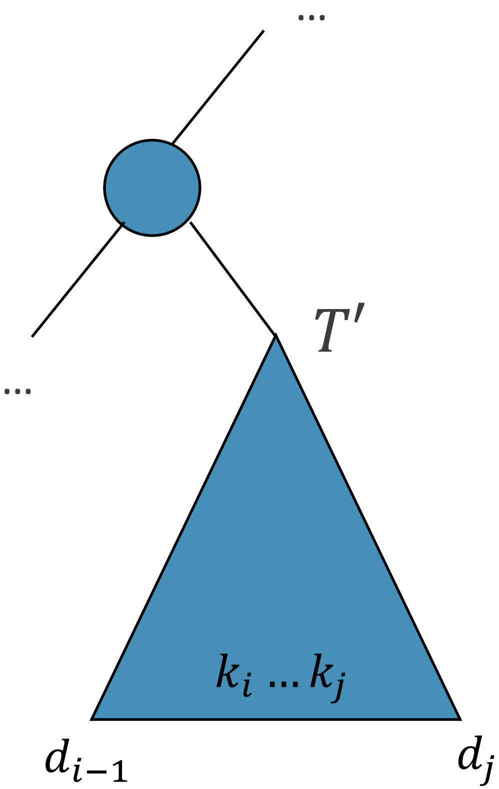
Optimal Substructure

- If a BST is optimal, all its subtrees are also optimal.
- Therefore, we should consider an arbitrary subtree of this problem.



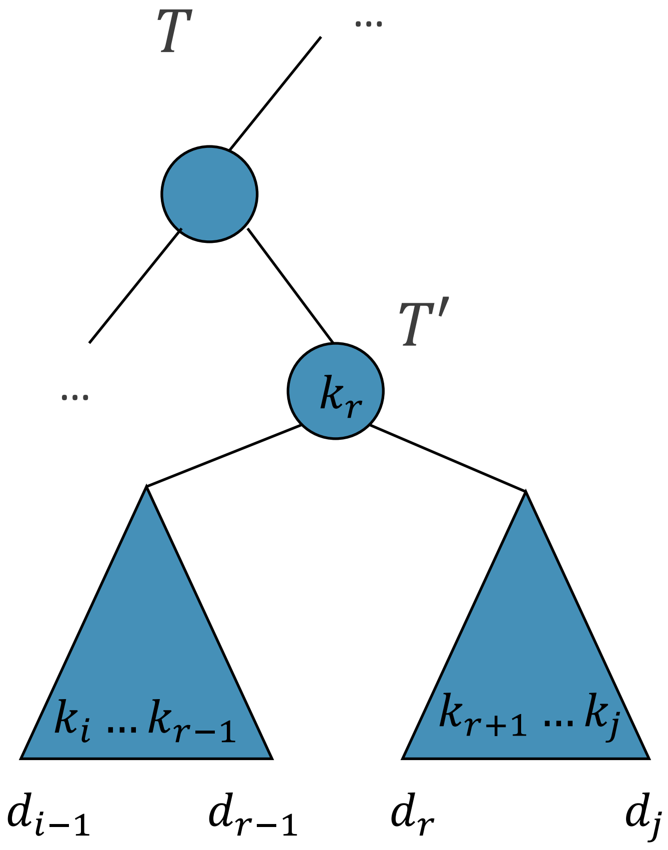
Optimal Substructure

- Consider any subtree T' of a BST. It must contain keys in a contiguous (连续的) range $k_i, \ldots, k_j$, for some $1 \leq i \leq j \leq n$.
- In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.



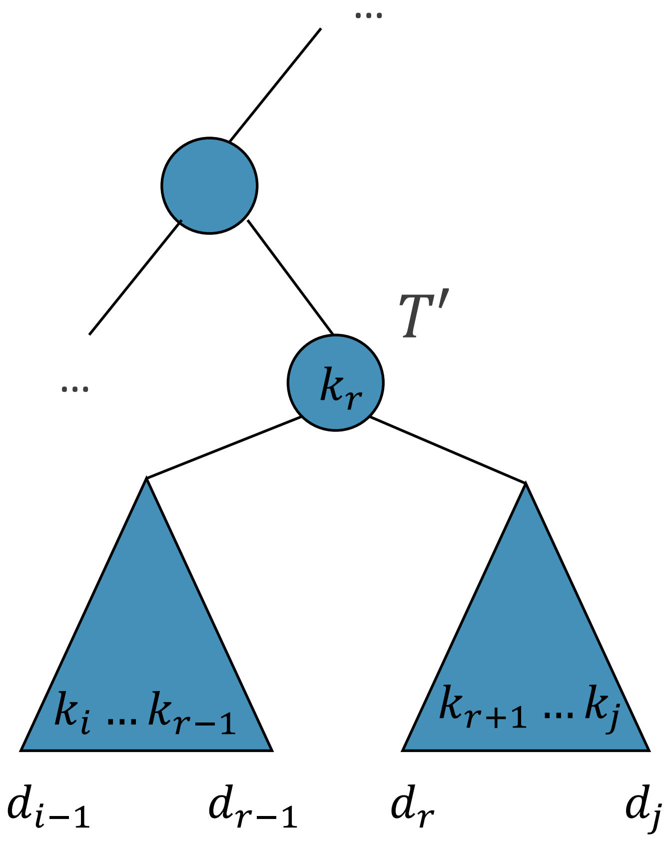
Optimal Substructure

- If an optimal BST T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.
  - Prove in contradiction. If there were a subtree T'' whose expected cost is lower than that of T', then we could cut T' out of T and paste in T'', resulting in a BST of lower expected cost than T, thus contradicting the optimality of T.



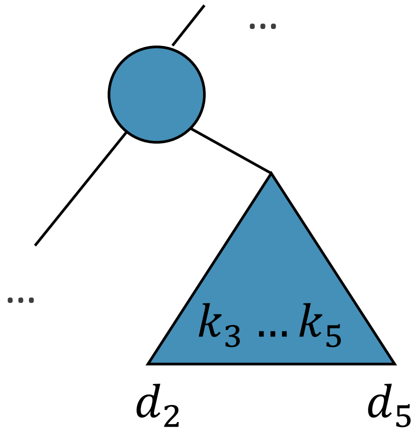
Optimal Substructure

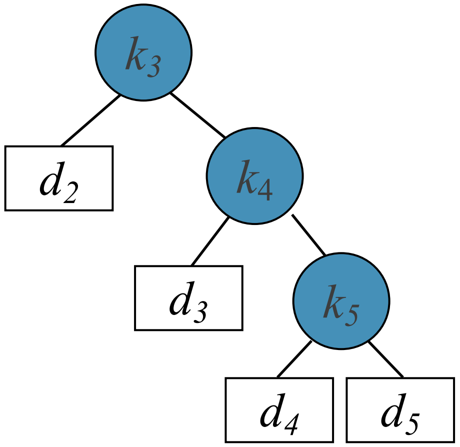
- The subtree T' must have a root k_r for $i \leq r \leq j$.
  - Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
  - Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.
- Given contiguous keys $k_i, \ldots, k_j$, how to recursively find an optimal subtree?
- Examine all candidate roots k_r, for $i \leq r \leq j$, and select the one with minimal cost.



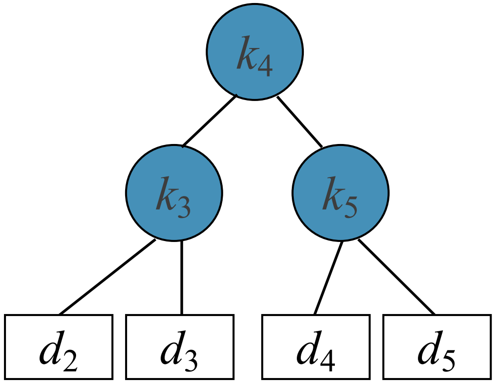
Optimal Substructure

- For example, the subtree T' has contiguous keys $k_3, \dots, k_5$ and dummy keys $d_2, \dots, d_5$.
- We construct all the subtree cases and select the one with minimum expected search time.

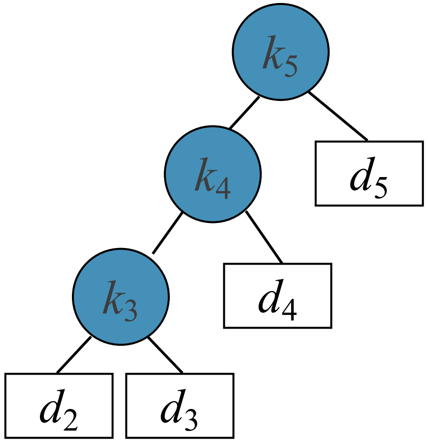




$r = 3$



$r = 4$



$r = 5$

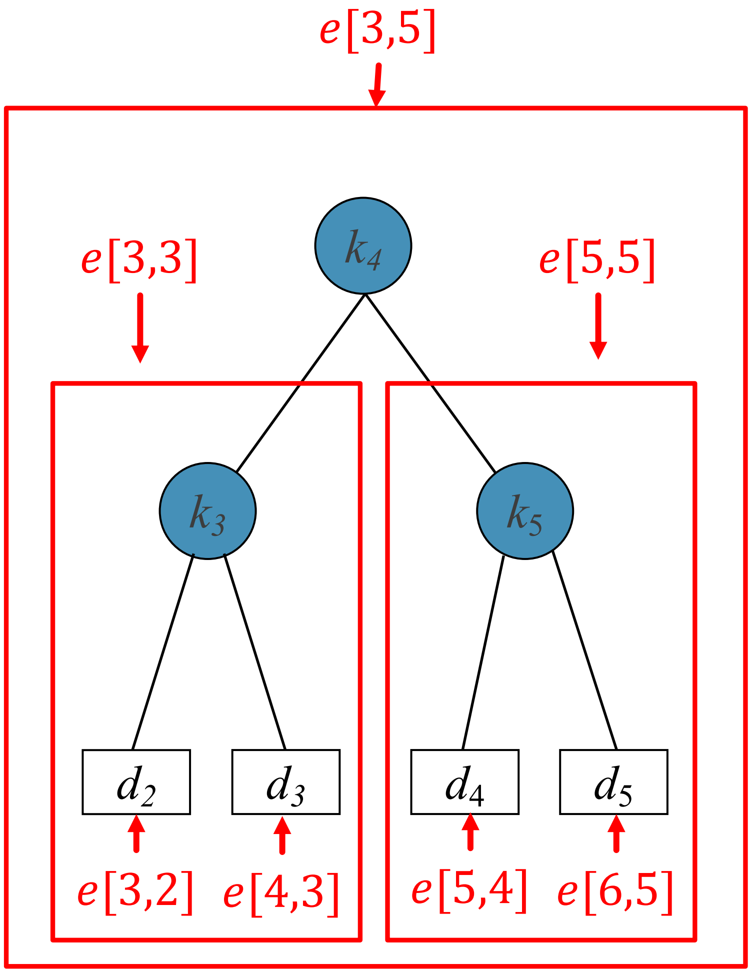
Recursive Equation

- Define $e[i,j]$ as expected search cost of optimal BST for $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_{j.}$
- If $j \geq i$, select a root k_r, for some $i \leq r \leq j$ and recursively make an optimal BST.
  - for $k_i, .., k_{r-1}$ as the left subtree, and
  - for $k_{r+1}, .., k_j$ as the right subtree.

Example

- If k_r is selected as the root of the subtree T', $e[i,j]$ only relates to $e[i,r-1]$ and $e[r+1,j]$.
- If $j=i-1$, the subtree contains no key but a single dummy key, then $e[i,j]=q_{i-1}$.
- Now, does the following equation hold?

$$e[i,j]=e[i,r-1]+e[r+1,j]$$



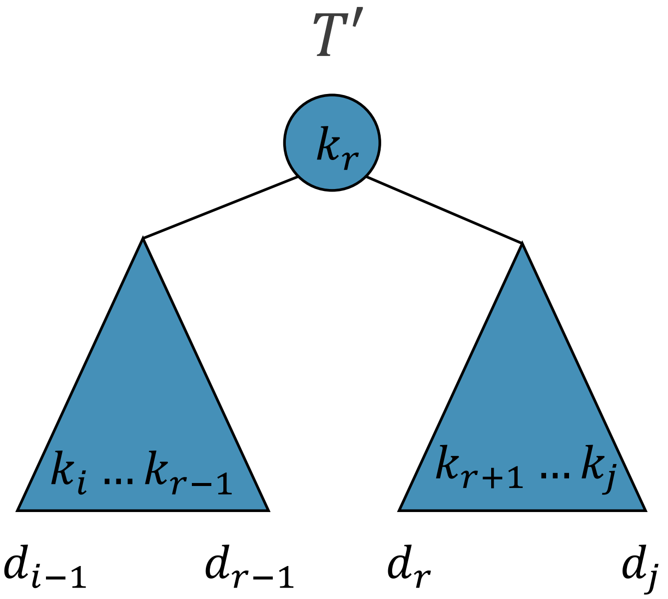
Recursive Equation

- The total search cost is composed by three parts:
  - The search time for the root: p_r.
  - The search time for the left subtree: $e[i, r-1] + \boxed{\sum_{l=i}^{r-1} p_l + \sum_{l=i-1}^{r-1} q_l.}$
  - The search time for the left subtree: $e[r+1, j] + \boxed{\sum_{l=r+1}^{j} p_l + \sum_{l=r}^{j} q_l.}$

Because the subtrees have one more depth, we should add the probabilities of all their keys and dummy keys.

- Let $w[i, j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$, the total is:

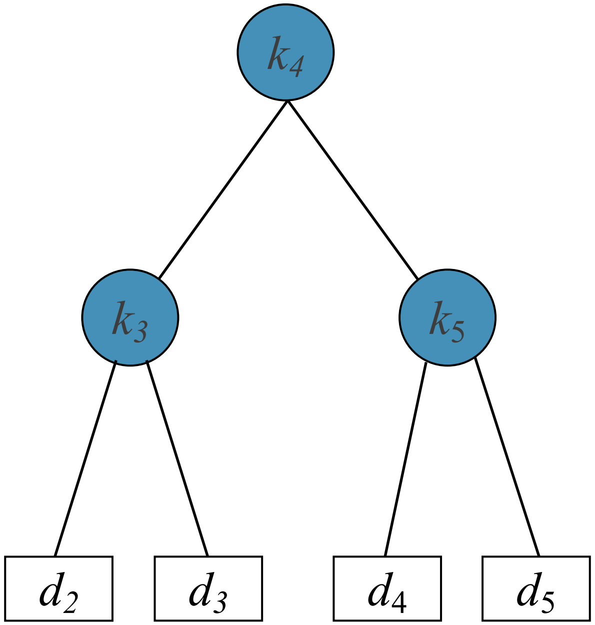
$$e[i, j] = e[i, r-1] + e[r+1, j] + w[i, j].$$



Example

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

$$
\begin{aligned}
e[3,3] &= 1\times p_3 + 2\times(q_2 + q_3) \\
e[5,5] &= 1\times p_5 + 2\times(q_4 + q_5) \\
e[3,5] &= 1\times p_4 \\
&\quad +(1 + 1)\times p_3 + (2 + 1)\times(q_2 + q_3) \\
&\quad +(1 + 1)\times p_5 + (2 + 1)\times(q_4 + q_5) \\
&= e[3,3] + e[5,5] \\
&\quad + \sum_{l=3}^{5} p_l + \sum_{l=2}^{5} q_l \\
&= e[3,3] + e[5,5] + w[3,5]
\end{aligned}
$$



Recursive Equation

- We iterate over all k_r and select the one with minimal cost:

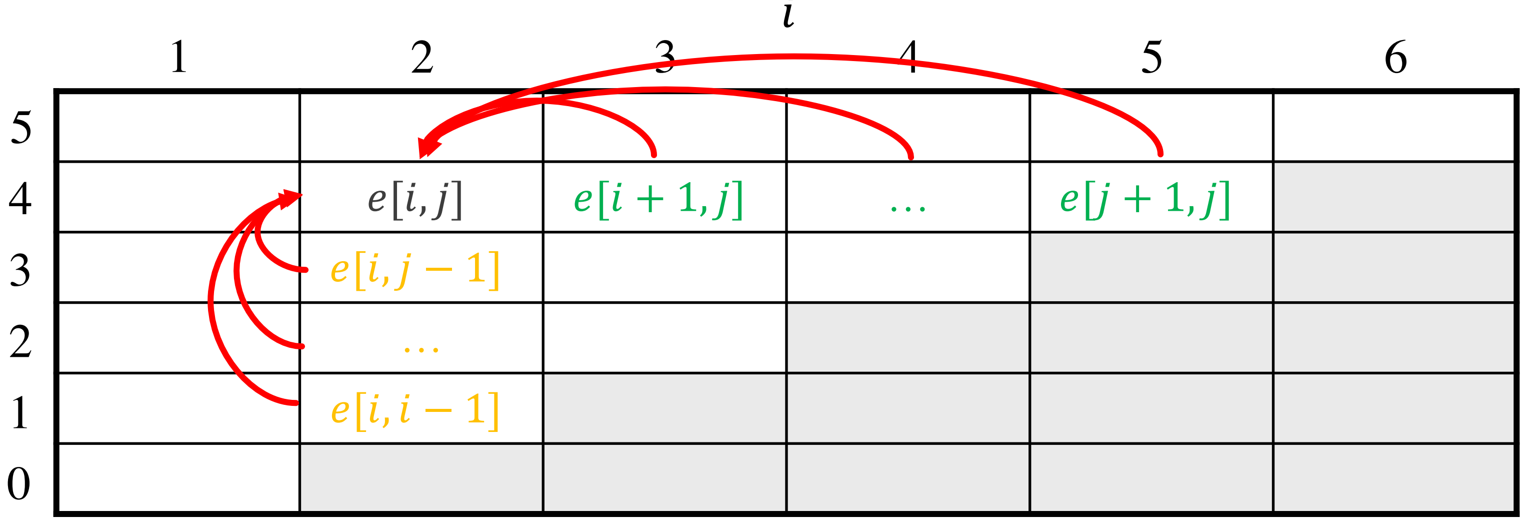
$$e[i,j] = \begin{cases} q_{i-1} & j = i-1 \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & i \le j \end{cases}$$

- To avoid repeated computation, we can also recursively calculate $w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$:

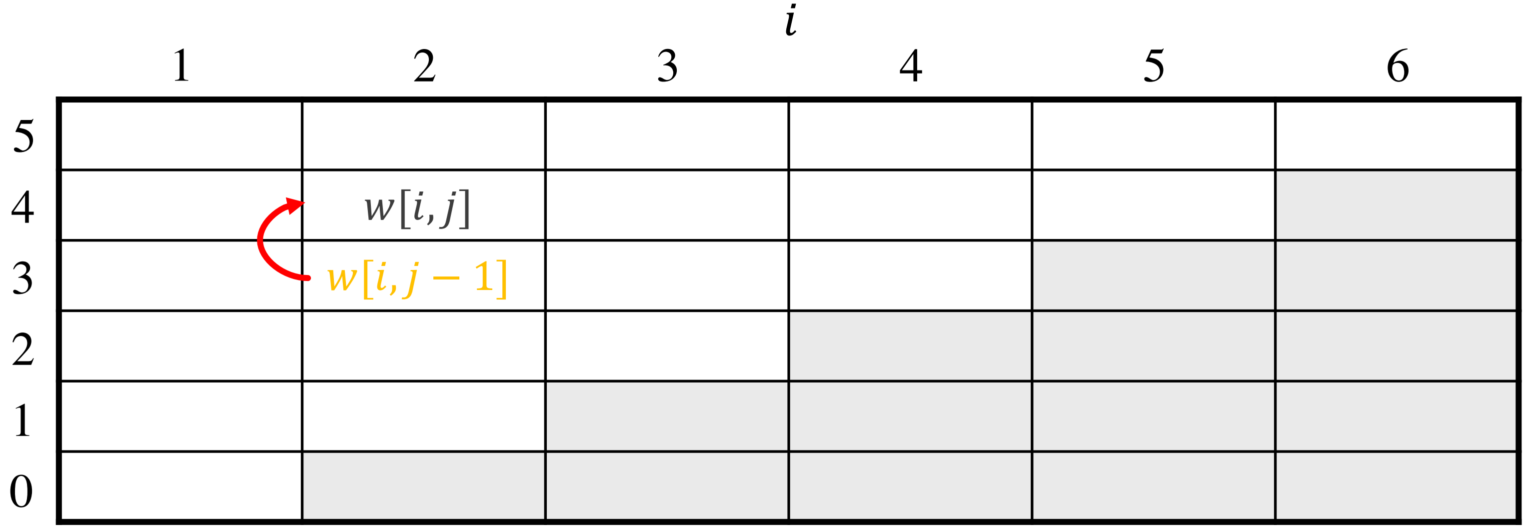
$$w[i,j] = \begin{cases} q_{i-1} & j = i-1 \\ w[i,j-1] + p_j + q_j & 1 \le i \le j \le n \end{cases}$$

- Both $e[i,j]$ and $w[i,j]$ are tables with $i = 1, \ldots, n+1$, $j = 0, \ldots, n$.

$$e[i,j] = \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\}$$



$$w[i,j] = w[i,j-1] + p_j + q_j$$



Pseudocode

```
DPOptimalBST($p, q, n$)
for $i \leftarrow 1$ to $n + 1$ do
    $e[i, i - 1] \leftarrow q_{i-1}$
    $w[i, i - 1] \leftarrow q_{i-1}$
for $c \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n - c + 1$ do
        $j \leftarrow i + c - 1$
        $e[i, j] \leftarrow \infty$
        $w[i, j] \leftarrow w[i, j - 1] + p_j + q_j$
        for $r \leftarrow i$ to $j$ do
            $t \leftarrow e[i, r - 1] + e[r + 1, j] + w[i, j]$
            if $t < e[i, j]$ then
                $e[i, j] \leftarrow t$
                $root[i, j] \leftarrow r$
return $e$ and $root$
```

First diagonal

From 2nd to nth diagonal

Given c, determine the jth column and ith row

Running time: $\Theta(n^3)$

$$e[i,j] = \min_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\}$$

$$w[i,j] = w[i,j-1]+p_j+q_j$$

i

		1	2	3	4	5	6
	5	2.75	2.0	1.3	0.9	0.5	0.1
	4	1.75	1.2	0.6	0.3	0.05	
j	3	1.25	0.7	0.25	0.05		
	2	0.9	0.4	0.05			
	1	0.45	0.1				
	0	0.05					

e

i

		1	2	3	4	5	6
	5	1	0.8	0.6	0.5	0.35	0.1
	4	0.7	0.5	0.3	0.2	0.05	
j	3	0.55	0.35	0.15	0.05		
	2	0.45	0.25	0.05			
	1	0.3	0.1				
	0	0.05					

w

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

i

		1	2	3	4	5
	5	2	4	5	5	5
	4	2	2	4	4	
j	3	2	2	3		
	2	1	2			
	1	1				

$root$

Classroom Exercise

Find the optimal BST with the following key probabilities:

i	0	1	2	3
p_i		0.25	0.3	0.15
q_i	0.05	0.1	0.05	0.1

Classroom Exercise

i

j	1	2	3	4
3	2	1.25	0.45	0.1
2	1.35	0.6	0.05	
1	0.55	0.1		
0	0.05			

e

i

j	1	2	3	4
3	1	0.7	0.3	0.1
2	0.75	0.45	0.05	
1	0.4	0.1		
0	0.05			

w

i

j	1	2	3
3	2	3	3
2	1	2	
1	1		

$root$

Classroom Exercise

游戏绝地求生中有头盔和防弹衣, 分别分为3个等级:

- 头盔: 一级头, 二级头, 三级头.
- 护甲: 一级甲, 二级甲, 三级甲.

在游戏中一开始什么都没有, 但是会在路上随机捡取装备. 假设规定高级装备可以覆盖低级装备, 也可以直接穿上, 但是穿上高级装备后就无法再穿回低级装备. 那么从什么都没有, 到穿上三级头和三级甲, 一共有多少种方式?

例如

- 一级头->二级甲->三级甲->三级头
- 三级头->三级甲
- ...

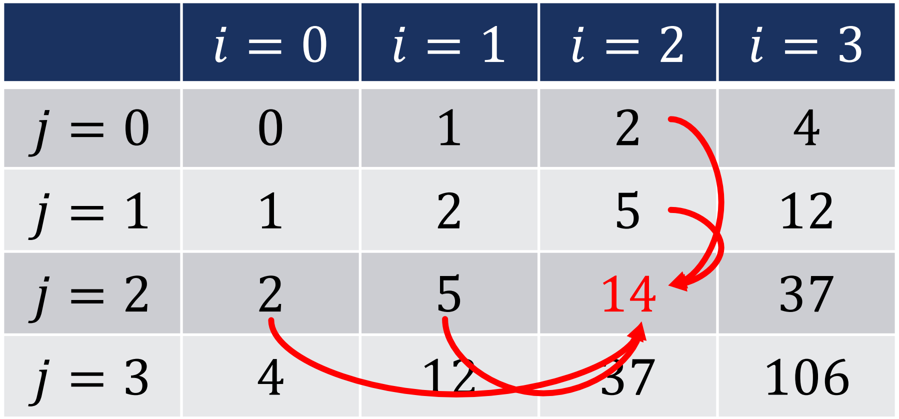
Image source: http://k.sina.com.cn/article_5944198308_1624d44a400100aa8r.html

Classroom Exercise

- 如果只考虑头盔，那么穿上i级头有2^{i-1}种方式. 因为i级头一定要穿上，但是$1, \dots i-1$级头都可以选择穿与不穿.
- 当穿上i级头和j级甲时，可以分解为两种情况:
  - 已经穿上j级甲了，就差i级头了，这时候捡到i级头了.
  - 已经穿上i级头了，就差j级甲了，这时候捡到j级甲了.
- 因此，假设$H(i,j)$为穿上i级头和j级甲的方式数，该问题的递归方程可以写为:

$$H(i,j) = \begin{cases} 2^{i-1} & i = 0 \text{ or } j = 0 \\ \sum\limits_{k<i} H(k,j) + \sum\limits_{k<j} H(i,k) & i > 0, j > 0 \end{cases}$$

Classroom Exercise



	$i = 0$	$i = 1$	$i = 2$	$i = 3$
$j = 0$	0	1	2	4
$j = 1$	1	2	5	12
$j = 2$	2	5	14	37
$j = 3$	4	12	37	106

ELEMENTS OF DYNAMIC PROGRAMMING

Optimal Substructure

Optimal substructure

1. The solution to the problem consists of making a choice. Making this choice leaves one or more subproblems to be solved.
2. For a given problem, you are given the choice that leads to an optimal solution.
3. Given this choice, you determine which subproblems follow and how to best characterize the resulting space of subproblems.
4. Solutions to the subproblems used within the optimal solution to the problem must themselves be optimal by proving by contradiction.

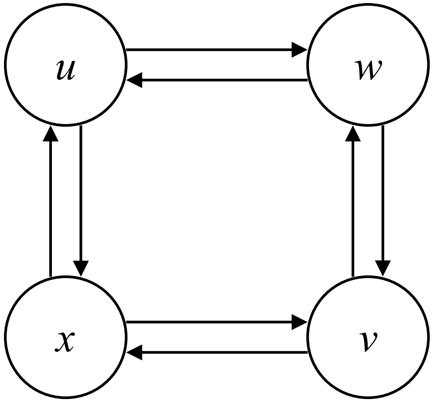
Optimal Substructure

Optimal substructure varies across problem domains in two ways:

1. How many subproblems are used in an optimal solution to the original problem.
   - Assembly line: One subproblem ($f_1[j-1]$ or $f_2[j-1]$)
   - Matrix multiplication: Two subproblems (subproducts $A_{i\ldots k}$ and $A_{k+1\ldots j}$)
2. How many choices we have in determining which subproblem(s) to use in an optimal solution.
   - Assembly line: Two choices (line 1 or line 2)
   - Matrix multiplication: $j-i$ choices for k (splitting the product)

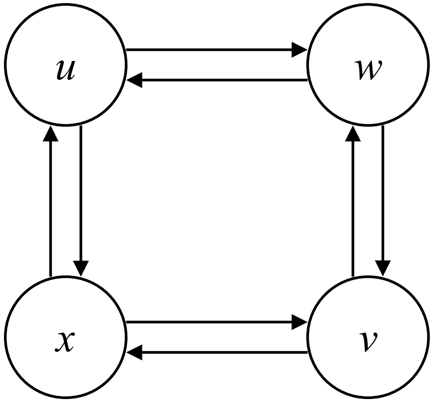
Optimal Substructure

- Dynamic programming uses optimal substructure in a bottom-up fashion.
- One should be careful not to assume that optimal substructure applies when it does not.
- Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.



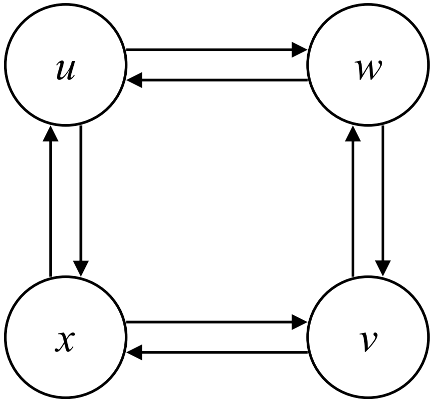
Optimal Substructure

- Unweighted shortest path: Find a path from u to v consisting of the fewest edges.
  - Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.
- This problem has optimal substructure.
  - Assume the shortest path from u to v goes through w. Then $u \rightarrow w$ and $w \rightarrow v$ is also the shortest.



Optimal Substructure

- Unweighted longest simple path: Find a simple path from u to v consisting of the most edges.
  - Simplicity is necessary because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.
- For unweighted shortest path, the problem does not have optimal substructure.
  - Assume the longest path from u to v goes through w. Then $u \rightarrow w$ may not be the longest. $u \rightarrow x \rightarrow v \rightarrow w$ is the longest.



Memorization

- Memorization (备忘录) method still uses recursion but store values in the table after calculation.

```
LookUpChain($p, i, j$)
if $m[i,j] < \infty$ then return $m[i,j]$
if $i = j$ then
    $m[i,j] \leftarrow 0$
else for $k \leftarrow i$ to $j - 1$ do
    $q \leftarrow$LookUpChain($p, i, k$)+
            LookUpChain($p, k+1, j$) + $p_{i+1} p_k p_j$
    if $q < m[i,j]$ then
        $m[i,j] \leftarrow q$
return $m[i,j]$
```

```
MemoizedMatrixChain($p$)
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow i$ to $n$ do
        $m[i,j] \leftarrow \infty$
return LookUpChain($p$, 1, $n$)
```

Easier to implement because no diagonal trick is needed, but more recursive calls are required.

Dynamic Programming vs. Memorization

- Advantages of dynamic programming
  - No overhead (系统开销) for recursion, less overhead for maintaining the table.
  - The regular pattern of table accesses may be used to reduce time or space requirements.
- Advantages of memorization
  - Some subproblems do not need to be solved.

Classroom Exercise

- Given an amount N and unlimited supply of coins with denominations $d_1, d_2, \ldots, d_n$, compute the smallest number of coins needed to get N.

- Example:
  - For $N = 86$ (cents) and $d_1 = 1, d_2 = 2, d_3 = 5, d_4 = 10, d_5 = 25, d_6 = 50, d_7 = 100$.
  - The optimal amount is 4 with changes: one 50, one 25, one 10, and one 1.
- Give the recursive equation and draw the table with the case: $d_1 = 1, d_2 = 4, d_3 = 6$ and $N = 8$.

Classroom Exercise

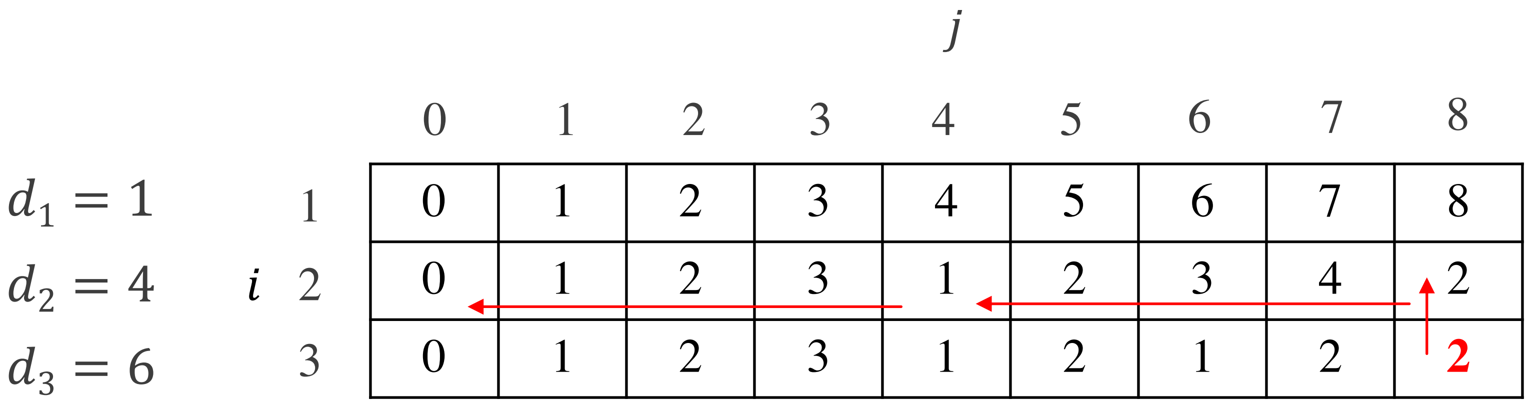
- Assume a set of coins $d_1, d_2, \dots, d_n$ (be sorted) and an amount N.
- Use a table $C[1 \dots n, 0 \dots N]$, where $C[i, j]$ is the smallest number of coins used to pay j cents, using only coins $d_1, \dots, d_i$.
- If $C[i, j]$ is optimal and d_i is used, then $C[i, j - d_i]$ is also optimal.
- $C[i, j]$ is calculated in two ways:
  1. Don’t use coin d_i (even if it’s possible):
  $$C[i, j] = C[i - 1, j]$$
  2. Use coin d_i :
  $$C[i, j] = 1 + C[i, j - d_i]$$

Classroom Exercise

- The recursive equation:

$$C[i, j] = \begin{cases} 0 & j = 0 \\ min(C[i-1, j], 1 + C[i, j - d_i]) & 0 < d_i \leq j \end{cases}$$

- Let $d_1 = 1$, $d_2 = 4$, $d_3 = 6$ and $N = 8$, the dynamic programming table will be:



j

		0	1	2	3	4	5	6	7	8
$d_1 = 1$	1	0	1	2	3	4	5	6	7	8
$d_2 = 4$	i 2	0	1	2	3	1	2	3	4	2
$d_3 = 6$	3	0	1	2	3	1	2	1	2	**2**

Conclusion

After this lecture, you should know:

- The difference between divide-and-conquer and dynamic programming.
- Why is dynamic programming efficient.
- What is optimal substructure.
- The steps of designing a dynamic programming algorithm.

Homework

- Page 92-94

  6.2

  6.3

  6.5

  6.7

  6.9

  6.12

Experiment

- 实现最长公共子序列算法，并应用于文本匹配问题
  - 数据集和要求在spoc上下载.
- 用动态规划算法求解石材切割问题.

谢谢

有问题欢迎随时跟我讨论