算法设计与分析(2024年春季学期)
Dynamic Programming
授课老师: 栾师
南京大学计算机科学与技术系
### Paradigms for Designing Algorithms

#### Greedy algorithm
- Make a greedy choice
- Prove that the greedy choice is safe
- Reduce the problem to a sub-problem and solve it iteratively
- Usually for optimization problems

#### Divide-and-conquer
- Break a problem into many independent sub-problems
- Solve each sub-problem separately
- Combine solutions for sub-problems to form a solution for the original one
- Usually used to design more efficient algorithms
Paradigms for Designing Algorithms

Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse
Recall: Computing the $n$-th Fibonacci Number

- $F_0 = 0$, $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

Fib($n$)

```
1:  F[0] ← 0
2:  F[1] ← 1
3:  for i ← 2 to n do
4:      F[i] ← F[i - 1] + F[i - 2]
5:  return F[n]
```

- Store each $F[i]$ for future use.
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
Recall: Interval Scheduling

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)
- each job has a weight (or value) \( v_i > 0 \)
- \( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** a maximum-size subset of mutually compatible jobs

Optimum value = 220
Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
- Job with the largest $\frac{\text{weight}}{\text{length}}$?

No, when weights are equal, this is the shortest job
- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$
Focus on instance \( \{1, 2, 3, \ldots, i\} \),

- \( \text{opt}[i] \): optimal value for the instance
- Assume we have computed \( \text{opt}[0], \text{opt}[1], \ldots, \text{opt}[i - 1] \)

**Q:** The value of optimal solution that does not contain \( i \)?

**A:** \( \text{opt}[i - 1] \)

**Q:** The value of optimal solution that contains job \( i \)?

**A:** \( v_i + \text{opt}[p_i] \), where \( p_i = \) the largest \( j \) such that \( f_j \leq s_i \)
Designing a Dynamic Programming Algorithm

Q: The value of optimal solution that does not contain \( i \)?

A: \( \text{opt}[i - 1] \)

Q: The value of optimal solution that contains job \( i \)?

A: \( v_i + \text{opt}[p_i] \), \( p_i = \) the largest \( j \) such that \( f_j \leq s_i \)

Recursion for \( \text{opt}[i] \):

\[
\text{opt}[i] = \max \{ \text{opt}[i - 1], v_i + \text{opt}[p_i] \}
\]
Designing a Dynamic Programming Algorithm

Recursion for $\text{opt}[i]$:

$$\text{opt}[i] = \max \{ \text{opt}[i-1], v_i + \text{opt}[p_i] \}$$

- $\text{opt}[0] = 0$
- $\text{opt}[1] = \max \{ \text{opt}[0], 80 + \text{opt}[0] \} = 80$
- $\text{opt}[2] = \max \{ \text{opt}[1], 100 + \text{opt}[0] \} = 100$
- $\text{opt}[3] = \max \{ \text{opt}[2], 90 + \text{opt}[0] \} = 100$
- $\text{opt}[4] = \max \{ \text{opt}[3], 25 + \text{opt}[1] \} = 105$
- $\text{opt}[5] = \max \{ \text{opt}[4], 50 + \text{opt}[3] \} = 150$
Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max \{opt[i - 1], v_i + opt[p_i]\}$$

- $opt[0] = 0$, $opt[1] = 80$, $opt[2] = 100$
- $opt[6] = \max\{opt[5], 70 + opt[3]\} = 170$
- $opt[7] = \max\{opt[6], 80 + opt[4]\} = 185$
- $opt[8] = \max\{opt[7], 50 + opt[6]\} = 220$
- $opt[9] = \max\{opt[8], 30 + opt[7]\} = 220$
Dynamic Programming

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: for $i \leftarrow 1$ to $n$ do
5: \hspace{1em} $opt[i] \leftarrow \max\{opt[i - 1], v_i + opt[p_i]\}$

- Running time sorting: $O(n \lg n)$
- Running time for computing $p$: $O(n \lg n)$ via binary search
- Running time for computing $opt[n]$: $O(n)$
1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: for $i \leftarrow 1$ to $n$ do
5: if $opt[i - 1] \geq v_i + opt[p_i]$ then
6: $opt[i] \leftarrow opt[i - 1]$
7: $b[i] \leftarrow N$
8: else
9: $opt[i] \leftarrow v_i + opt[p_i]$
10: $b[i] \leftarrow Y$

1: $i \leftarrow n, S \leftarrow \emptyset$
2: while $i \neq 0$ do
3: if $b[i] = N$ then
4: $i \leftarrow i - 1$
5: else
6: $S \leftarrow S \cup \{i\}$
7: $i \leftarrow p_i$
8: return $S$
Recovering Optimum Schedule: Example

<table>
<thead>
<tr>
<th>$i$</th>
<th>$opt[i]$</th>
<th>$b[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>105</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>170</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>185</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>220</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>220</td>
<td>N</td>
</tr>
</tbody>
</table>
Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse
Outline

1. Weighted Interval Scheduling

2. Subset Sum Problem
   - Related Problem: Knapsack Problem

3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space

4. Shortest Paths in Directed Acyclic Graphs

5. Matrix Chain Multiplication

6. Optimum Binary Search Tree

7. Summary
Subset Sum Problem

**Input:** an integer bound $W > 0$

a set of $n$ items, each with an integer weight $w_i > 0$

**Output:** a subset $S$ of items that

$$\text{maximizes } \sum_{i \in S} w_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$ 

- Motivation: you have budget $W$, and want to buy a subset of items, so as to spend as much money as possible.

**Example:**

- $W = 35, n = 5, w = (14, 9, 17, 10, 13)$

- Optimum: $S = \{1, 2, 4\}$ and $14 + 9 + 10 = 33$
Candidate Algorithm:
- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below $W$

Q: Does candidate algorithm always produce optimal solutions?

A: No. $W = 100, n = 3, w = (51, 50, 50)$.

Q: What if we change “non-increasing” to “non-decreasing”?

A: No. $W = 100, n = 3, w = (1, 50, 50)$
Design a Dynamic Programming Algorithm

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

Q: The value of the optimum solution that does not contain $i$?

A: $opt[i - 1, W']$

Q: The value of the optimum solution that contains $i$?

A: $opt[i - 1, W' - w_i] + w_i$
Dynamic Programming

- Consider the instance: \( i, W', (w_1, w_2, \cdots, w_i) \);
- \( opt[i, W'] \): the optimum value of the instance

\[
opt[i, W'] = \begin{cases} 
0 & i = 0 \\
 opt[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{ll}
    opt[i - 1, W'] \\
    opt[i - 1, W' - w_i] + w_i 
\end{array} \right\} & i > 0, w_i \leq W'
\end{cases}
\]
Dynamic Programming

1: for $W' \leftarrow 0$ to $W$ do
2: $\text{opt}[0, W'] \leftarrow 0$
3: for $i \leftarrow 1$ to $n$ do
4: for $W' \leftarrow 0$ to $W$ do
5: $\text{opt}[i, W'] \leftarrow \text{opt}[i - 1, W']$
6: if $w_i \leq W'$ and $\text{opt}[i - 1, W' - w_i] + w_i \geq \text{opt}[i, W']$ then
7: $\text{opt}[i, W'] \leftarrow \text{opt}[i - 1, W' - w_i] + w_i$
8: return $\text{opt}[n, W]$
Recover the Optimum Set

1: \textbf{for} \ W' \leftarrow 0 \textbf{ to } W \textbf{ do}
2: \hspace{1em} \text{opt}[0, W'] \leftarrow 0
3: \textbf{for} \ i \leftarrow 1 \textbf{ to } n \textbf{ do}
4: \hspace{1em} \textbf{for} \ W' \leftarrow 0 \textbf{ to } W \textbf{ do}
5: \hspace{2em} \text{opt}[i, W'] \leftarrow \text{opt}[i - 1, W']
6: \hspace{2em} b[i, W'] \leftarrow N
7: \hspace{2em} \textbf{if} \ w_i \leq W' \textbf{ and } \text{opt}[i - 1, W' - w_i] + w_i \geq \text{opt}[i, W']
8: \hspace{3em} \text{then}
9: \hspace{4em} \text{opt}[i, W'] \leftarrow \text{opt}[i - 1, W' - w_i] + w_i
10: \hspace{4em} b[i, W'] \leftarrow Y
11: \textbf{return} \ \text{opt}[n, W]
Recover the Optimum Set

1: \[ i \leftarrow n, W' \leftarrow W, S \leftarrow \emptyset \]
2: \textbf{while} \( i > 0 \) \textbf{do}
3: \hspace{1em} \textbf{if} \( b[i, W'] = Y \) \textbf{then}
4: \hspace{2em} \( W' \leftarrow W' - w_i \)
5: \hspace{2em} \( S \leftarrow S \cup \{i\} \)
6: \hspace{1em} \( i \leftarrow i - 1 \)
7: \textbf{return} \( S \)
Running Time of Algorithm

1. for $W' \leftarrow 0$ to $W$ do
2. \hspace{1em} $opt[0, W'] \leftarrow 0$
3. for $i \leftarrow 1$ to $n$ do
4. \hspace{1em} for $W' \leftarrow 0$ to $W$ do
5. \hspace{2em} $opt[i, W'] \leftarrow opt[i - 1, W']$
6. \hspace{2em} if $w_i \leq W'$ and $opt[i - 1, W' - w_i] + w_i \geq opt[i, W']$ then
7. \hspace{3em} $opt[i, W'] \leftarrow opt[i - 1, W' - w_i] + w_i$
8. return $opt[n, W]$

- Running time is $O(nW)$
- Running time is pseudo-polynomial because it depends on value of the input integers.
Example

- \( n = 4, \ w = (2, 3, 9, 8), \ W = 14 \)

<table>
<thead>
<tr>
<th>( i, W' )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
Avoiding Unnecessary Computation and Memory Using Memoized Algorithm and Hash Map

**compute-opt**\((i, W')\)

1: if \(opt[i, W'] \neq \perp\) then return \(opt[i, W']\)
2: if \(i = 0\) then \(r \leftarrow 0\)
3: else
4: \(r \leftarrow compute-opt(i - 1, W')\)
5: if \(w_i \leq W'\) then
6: \(r' \leftarrow compute-opt(i - 1, W' - w_i) + w_i\)
7: if \(r' > r\) then \(r \leftarrow r'\)
8: \(opt[i, W'] \leftarrow r\)
9: return \(r\)

- Use hash map for \(opt\)
Example Using Memoized Rounding

- $n = 4$, $w = (2, 3, 9, 8)$, $W = 14$

<table>
<thead>
<tr>
<th>$i, W'$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- $i = 0$: No elements.
- $i = 1$: Two elements: $(2, 14)$ and $(3, 14)$.
- $i = 2$: Four elements: $(2, 14)$, $(3, 14)$, $(4, 14)$, $(0, 14)$.
- $i = 3$: Eight elements.
- $i = 4$: Fourteen elements.
Outline

1. Weighted Interval Scheduling

2. Subset Sum Problem
   - Related Problem: Knapsack Problem

3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space

4. Shortest Paths in Directed Acyclic Graphs

5. Matrix Chain Multiplication

6. Optimum Binary Search Tree

7. Summary
Knapsack Problem

**Input:** an integer bound $W > 0$

- a set of $n$ items, each with an integer weight $w_i > 0$
- a value $v_i > 0$ for each item $i$

**Output:** a subset $S$ of items that

$$\text{maximizes } \sum_{i \in S} v_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$ 

- Motivation: you have budget $W$, and want to buy a subset of items of maximum total value
**DP for Knapsack Problem**

- \( opt[i, W'] \): the optimum value when budget is \( W' \) and items are \( \{1, 2, 3, \ldots, i\} \).
- If \( i = 0 \), \( opt[i, W'] = 0 \) for every \( W' = 0, 1, 2, \ldots, W \).

\[
opt[i, W'] = \begin{cases} 
0 & \text{if } i = 0 \\
\text{opt}[i - 1, W'] & \text{if } i > 0, w_i > W' \\
\max \left\{ \text{opt}[i - 1, W'], \text{opt}[i - 1, W' - w_i] + v_i \right\} & \text{if } i > 0, w_i \leq W'
\end{cases}
\]
Exercise: Items with 3 Parameters

**Input:** integer bounds $W > 0, Z > 0$,  
a set of $n$ items, each with an integer weight $w_i > 0$  
a size $z_i > 0$ for each item $i$  
a value $v_i > 0$ for each item $i$

**Output:** a subset $S$ of items that

$$\text{maximizes } \sum_{i \in S} v_i \quad \text{s.t.}$$

$$\sum_{i \in S} w_i \leq W \quad \text{and} \quad \sum_{i \in S} z_i \leq Z$$
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
Subsequence

- $A = bacdca$
- $C = adca$
- $C$ is a subsequence of $A$

**Def.** Given two sequences $A[1..n]$ and $C[1..t]$ of letters, $C$ is called a subsequence of $A$ if there exists integers $1 \leq i_1 < i_2 < i_3 < \ldots < i_t \leq n$ such that $A[i_j] = C[j]$ for every $j = 1, 2, 3, \ldots, t$.

- Exercise: how to check if sequence $C$ is a subsequence of $A$?
Longest Common Subsequence

**Input:** \( A[1 \ldots n] \) and \( B[1 \ldots m] \)

**Output:** the longest common subsequence of \( A \) and \( B \)

Example:
- \( A = 'bacdca' \)
- \( B = 'adbcda' \)
- \( \text{LCS}(A, B) = 'adca' \)

Applications: edit distance (diff), similarity of DNAs
Goal of LCS: find a maximum-size non-crossing matching between letters in $A$ and letters in $B$. 
Reduce to Subproblems

- $A = 'bacdca'$
- $B = 'adbecda'$

either the last letter of $A$ is not matched:
- need to compute LCS('bacd', 'adbecd')

or the last letter of $B$ is not matched:
- need to compute LCS('bacdc', 'adbc')
Dynamic Programming for LCS

- $opt[i, j], 0 \leq i \leq n, 0 \leq j \leq m$: length of longest common sub-sequence of $A[1..i]$ and $B[1..j]$.
- If $i = 0$ or $j = 0$, then $opt[i, j] = 0$.
- If $i > 0, j > 0$, then

$$
opt[i, j] = \begin{cases} 
  opt[i - 1, j - 1] + 1 & \text{if } A[i] = B[j] \\
  \max \left\{ \begin{array}{l} 
  opt[i - 1, j] \\
  opt[i, j - 1]
  \end{array} \right. & \text{if } A[i] \neq B[j]
\end{cases}
$$
Dynamic Programming for LCS

1: \textbf{for} $j \leftarrow 0$ to $m$ \textbf{do}
2: \hspace{1em} $\text{opt}[0, j] \leftarrow 0$
3: \textbf{for} $i \leftarrow 1$ to $n$ \textbf{do}
4: \hspace{1em} $\text{opt}[i, 0] \leftarrow 0$
5: \textbf{for} $j \leftarrow 1$ to $m$ \textbf{do}
6: \hspace{2em} \textbf{if} $A[i] = B[j]$ \textbf{then}
7: \hspace{3em} $\text{opt}[i, j] \leftarrow \text{opt}[i - 1, j - 1] + 1$, $\pi[i, j] \leftarrow \text{“\downarrow\”}$
8: \hspace{2em} \textbf{else if} $\text{opt}[i, j - 1] \geq \text{opt}[i - 1, j]$ \textbf{then}
9: \hspace{3em} $\text{opt}[i, j] \leftarrow \text{opt}[i, j - 1]$, $\pi[i, j] \leftarrow \text{“\leftarrow\”}$
10: \hspace{2em} \textbf{else}
11: \hspace{3em} $\text{opt}[i, j] \leftarrow \text{opt}[i - 1, j]$, $\pi[i, j] \leftarrow \text{“\uparrow\”}$
### Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>d</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>B</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
</tr>
<tr>
<td>1</td>
<td>0⊥</td>
<td>0←</td>
<td>0←</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
<td>1←</td>
</tr>
<tr>
<td>2</td>
<td>0⊥</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
<td>1←</td>
<td>1←</td>
<td>2↖</td>
</tr>
<tr>
<td>3</td>
<td>0⊥</td>
<td>1↑</td>
<td>1←</td>
<td>1←</td>
<td>2↖</td>
<td>2←</td>
<td>2←</td>
</tr>
<tr>
<td>4</td>
<td>0⊥</td>
<td>1↑</td>
<td>2↘</td>
<td>2←</td>
<td>2←</td>
<td>3↖</td>
<td>3←</td>
</tr>
<tr>
<td>5</td>
<td>0⊥</td>
<td>1↑</td>
<td>2↑</td>
<td>2←</td>
<td>3↖</td>
<td>3←</td>
<td>3←</td>
</tr>
<tr>
<td>6</td>
<td>0⊥</td>
<td>1↖</td>
<td>2↑</td>
<td>2←</td>
<td>3↑</td>
<td>3←</td>
<td>4↖</td>
</tr>
</tbody>
</table>
Example: Find Common Subsequence

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>B</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
<td>0⊥</td>
</tr>
<tr>
<td>1</td>
<td>0⊥</td>
<td>0←</td>
<td>0←</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
<td>1←</td>
</tr>
<tr>
<td>2</td>
<td>0⊥</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
<td>1←</td>
<td>1←</td>
<td>2↖</td>
</tr>
<tr>
<td>3</td>
<td>0⊥</td>
<td>1↑</td>
<td>1←</td>
<td>1←</td>
<td>2↖</td>
<td>2←</td>
<td>2←</td>
</tr>
<tr>
<td>4</td>
<td>0⊥</td>
<td>1↑</td>
<td>2↖</td>
<td>2←</td>
<td>2←</td>
<td>3↖</td>
<td>3←</td>
</tr>
<tr>
<td>5</td>
<td>0⊥</td>
<td>1↑</td>
<td>2↑</td>
<td>2←</td>
<td>3↖</td>
<td>3←</td>
<td>3←</td>
</tr>
<tr>
<td>6</td>
<td>0⊥</td>
<td>1↖</td>
<td>2↑</td>
<td>2←</td>
<td>3↑</td>
<td>3←</td>
<td>4↖</td>
</tr>
</tbody>
</table>
Find Common Subsequence

1: $i \leftarrow n, j \leftarrow m, S \leftarrow ()$
2: while $i > 0$ and $j > 0$ do
3: if $\pi[i, j] = \text{“↖”}$ then
4: add $A[i]$ to beginning of $S$, $i \leftarrow i - 1, j \leftarrow j - 1$
5: else if $\pi[i, j] = \text{“↑”}$ then
6: $i \leftarrow i - 1$
7: else
8: $j \leftarrow j - 1$
9: return $S$
# Variants of Problem

## Edit Distance with Insertions and Deletions

**Input:** a string $A$

each time we can delete a letter from $A$ or insert a letter to $A$

**Output:** minimum number of operations (insertions or deletions) we need to change $A$ to $B$?

## Example:

- $A = \text{ocurrance}$, $B = \text{occurrence}$
- 3 operations: insert 'c', remove 'a' and insert 'e'

## Obs.

$\#\text{OPs} = \text{length}(A) + \text{length}(B) - 2 \cdot \text{length}(\text{LCS}(A, B))$
Variants of Problem

**Edit Distance with Insertions, Deletions and Replacing**

**Input:** a string $A$,

each time we can delete a letter from $A$, insert a letter to $A$ or change a letter

**Output:** how many operations do we need to change $A$ to $B$?

**Example:**

- $A = \text{ocurrance}, \ B = \text{occurrence}$.
- 2 operations: insert 'c', change 'a' to 'e'

- Not related to LCS any more
Need to match letters in $A$ and $B$, every letter is matched at most once and there should be no crosses.

However, we can **match two different letters**: Matching a same letter gives score 2, matching two different letters gives score 1.

Need to maximize the score.

DP recursion for the case $i > 0$ and $j > 0$:

$$
op[i, j] = \begin{cases} 
\text{opt}[i - 1, j - 1] + 2 & \text{if } A[i] = B[j] \\
\max \begin{cases} 
\text{opt}[i - 1, j] \\
\text{opt}[i, j - 1] \\
\text{opt}[i - 1, j - 1] + 1 
\end{cases} & \text{if } A[i] \neq B[j]
\end{cases}$$

Relation: $\#\text{OPs} = \text{length}(A) + \text{length}(B) - \text{max\_score}$
Edit Distance (with Replacing): using DP directly

- \( opt[i, j], 0 \leq i \leq n, 0 \leq j \leq m \): edit distance between \( A[1 .. i] \) and \( B[1 .. j] \).
- if \( i = 0 \) then \( opt[i, j] = j \); if \( j = 0 \) then \( opt[i, j] = i \).
- if \( i > 0, j > 0 \), then

\[
opt[i, j] = \begin{cases} 
\begin{align*}
& opt[i - 1, j - 1] & \text{if } A[i] = B[j] \\
& \min \left\{ \begin{align*}
& opt[i - 1, j] + 1 \\
& opt[i, j - 1] + 1 \\
& opt[i - 1, j - 1] + 1
\end{align*} \right. & \text{if } A[i] \neq B[j]
\end{align*}
\end{cases}
\]
Exercise: Longest Palindrome

Def. A palindrome is a string which reads the same backward or forward.

- example: “racecar”, “wasitacaroracatisaw”, ”putitup”

Longest Palindrome Subsequence

Input: a sequence $A$

Output: the longest subsequence $C$ of $A$ that is a palindrome.

Example:

- Input: acbcedeacab
- Output: acedecaa
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
Computing the Length of LCS

1: for $j \leftarrow 0$ to $m$ do
2: \hspace{1em} $opt[0, j] \leftarrow 0$
3: for $i \leftarrow 1$ to $n$ do
4: \hspace{1em} $opt[i, 0] \leftarrow 0$
5: for $j \leftarrow 1$ to $m$ do
6: \hspace{2em} if $A[i] = B[j]$ then
7: \hspace{3em} $opt[i, j] \leftarrow opt[i - 1, j - 1] + 1$
8: \hspace{2em} else if $opt[i, j - 1] \geq opt[i - 1, j]$ then
9: \hspace{3em} $opt[i, j] \leftarrow opt[i, j - 1]$
10: \hspace{2em} else
11: \hspace{3em} $opt[i, j] \leftarrow opt[i - 1, j]$

Obs. The $i$-th row of table only depends on $(i - 1)$-th row.
Reducing Space to $O(n + m)$

**Obs.** The $i$-th row of table only depends on $(i - 1)$-th row.

**Q:** How to use this observation to reduce space?

**A:** We only keep two rows: the $(i - 1)$-th row and the $i$-th row.
Linear Space Algorithm to Compute Length of LCS

1: for $j \leftarrow 0$ to $m$ do
2: \hspace{1em} $opt[0, j] \leftarrow 0$
3: for $i \leftarrow 1$ to $n$ do
4: \hspace{1em} $opt[i \mod 2, 0] \leftarrow 0$
5: for $j \leftarrow 1$ to $m$ do
6: \hspace{2em} if $A[i] = B[j]$ then
7: \hspace{3em} $opt[i \mod 2, j] \leftarrow opt[i - 1 \mod 2, j - 1] + 1$
8: \hspace{2em} else if $opt[i \mod 2, j - 1] \geq opt[i - 1 \mod 2, j]$ then
9: \hspace{3em} $opt[i \mod 2, j] \leftarrow opt[i \mod 2, j - 1]$
10: \hspace{2em} else
11: \hspace{3em} $opt[i \mod 2, j] \leftarrow opt[i - 1 \mod 2, j]$
12: return $opt[n \mod 2, m]$
How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$
- Can recover the LCS using $n$ rounds: time = $O(n^2m)$
- Using **Divide and Conquer** + Dynamic Programming:
  - Space: $O(m + n)$
  - Time: $O(nm)$
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
**Def.** A directed acyclic graph (DAG) is a directed graph without (directed) cycles.

**Lemma** A directed graph is a DAG if and only its vertices can be topologically sorted.
Shortest Paths in DAG

**Input:** directed acyclic graph $G = (V, E)$ and $w : E \rightarrow \mathbb{R}$.
Assume $V = \{1, 2, 3 \cdots , n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

**Output:** the shortest path from 1 to $i$, for every $i \in V$
Shortest Paths in DAG

- \( f[i] \): length of the shortest path from 1 to \( i \)

\[
f[i] = \begin{cases} 
0 & \text{if } i = 1 \\
\min_{j: (j, i) \in E} \{ f(j) + w(j, i) \} & \text{if } i = 2, 3, \cdots, n
\end{cases}
\]
Use an adjacency list for incoming edges of each vertex $i$

**Shortest Paths in DAG**

| 1:  | $f[1] \leftarrow 0$ |
| 2:  | **for** $i \leftarrow 2$ to $n$ **do** |
| 3:  | $f[i] \leftarrow \infty$ |
| 4:  | **for** each incoming edge $(j, i)$ of $i$ **do** |
| 5:  | **if** $f[j] + w(j, i) < f[i]$ **then** |
| 6:  | $f[i] \leftarrow f[j] + w(j, i)$ |
| 7:  | $\pi(i) \leftarrow j$ |

**print-path($t$)**

| 1:  | **if** $t = 1$ **then** |
| 2:  | print(1) |
| 3:  | **return** |
| 4:  | print-path($\pi(t)$) |
| 5:  | print(“,”, $t$) |
Example
Heaviest Path in a Directed Acyclic Graph

**Input:** directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.
Assume $V = \{1, 2, 3, \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

**Output:** the path with the largest weight (the heaviest path) from 1 to $n$.

- $f[i]$: weight of the heaviest path from 1 to $i$

\[
f[i] = \begin{cases} 
0 & i = 1 \\
\max_{j: (j, i) \in E} \{ f(j) + w(j, i) \} & i = 2, 3, \cdots, n 
\end{cases}
\]
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
Matrix Chain Multiplication

**Input:** \( n \) matrices \( A_1, A_2, \ldots, A_n \) of sizes \( r_1 \times c_1, r_2 \times c_2, \ldots, r_n \times c_n \), such that \( c_i = r_{i+1} \) for every \( i = 1, 2, \ldots, n-1 \).

**Output:** the order of computing \( A_1 A_2 \cdots A_n \) with the minimum number of multiplications

**Fact** Multiplying two matrices of size \( r \times k \) and \( k \times c \) takes \( r \times k \times c \) multiplications.
Example:

- $A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$

\[
\begin{align*}
10 \times 100 & \quad 100 \times 5 & \quad 5 \times 50 \\
10 \times 5 & \quad 100 \cdot 5 & \quad 5 \times 50 \\
10 \times 5 & \quad 10 \cdot 5 \cdot 50 & \quad 10 \cdot 100 \cdot 50 \\
\end{align*}
\]

\[
\begin{align*}
10 \times 100 & \quad 100 \times 5 & \quad 5 \times 50 \\
10 \times 5 & \quad 100 \cdot 5 & \quad 5 \times 50 \\
10 \times 5 & \quad 100 \cdot 5 \cdot 50 & \quad 10 \cdot 100 \cdot 50 \\
\end{align*}
\]

\[
\begin{align*}
10 \times 100 & \quad 100 \times 5 & \quad 5 \times 50 \\
10 \times 5 & \quad 100 \cdot 5 & \quad 5 \times 50 \\
10 \times 5 & \quad 10 \cdot 5 \cdot 50 & \quad 10 \cdot 100 \cdot 50 \\
\end{align*}
\]

\[\text{cost} = 5000 + 2500 = 7500\]

\[\text{cost} = 25000 + 50000 = 75000\]

- $(A_1A_2)A_3$: $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$

- $A_1(A_2A_3)$: $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$
Matrix Chain Multiplication: Design DP

- Assume the last step is \((A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)\)
- Cost of last step: \(r_1 \times c_i \times c_n\)
- Optimality for sub-instances: we need to compute \(A_1 A_2 \cdots A_i\) and \(A_{i+1} A_{i+2} \cdots A_n\) optimally
- \(opt[i, j]\) : the minimum cost of computing \(A_i A_{i+1} \cdots A_j\)

\[
opt[i, j] = \begin{cases} 
0 & i = j \\
\min_{k:i \leq k < j} \left( opt[i, k] + opt[k + 1, j] + r_i c_k c_j \right) & i < j
\end{cases}
\]
matrix-chain-multiplication($n, r[1..n], c[1..n]$)

1: let $opt[i, i] \leftarrow 0$ for every $i = 1, 2, \ldots, n$
2: for $\ell \leftarrow 2$ to $n$ do
3: for $i \leftarrow 1$ to $n - \ell + 1$ do
4: $j \leftarrow i + \ell - 1$
5: $opt[i, j] \leftarrow \infty$
6: for $k \leftarrow i$ to $j - 1$ do
7: if $opt[i, k] + opt[k + 1, j] + r_i c_k c_j < opt[i, j]$ then
8: $opt[i, j] \leftarrow opt[i, k] + opt[k + 1, j] + r_i c_k c_j$
9: $\pi[i, j] \leftarrow k$
10: return $opt[1, n]$
Constructing Optimal Solution

Print-Optimal-Order($i, j$)

1: \textbf{if} $i = j$ \textbf{then}
2: \hspace{1em} \text{print(“A”}_i)\newline
3: \textbf{else}
4: \hspace{1em} \text{print(“(”)}\newline
5: \hspace{1.5em} \text{Print-Optimal-Order}(i, \pi[i, j])\newline
6: \hspace{1.5em} \text{Print-Optimal-Order}(\pi[i, j] + 1, j)\newline
7: \hspace{1em} \text{print(“)”)}
<table>
<thead>
<tr>
<th>matrix</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$3 \times 5$</td>
<td>$5 \times 2$</td>
<td>$2 \times 6$</td>
<td>$6 \times 9$</td>
<td>$9 \times 4$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{opt}[1, 2] &= \text{opt}[1, 1] + \text{opt}[2, 2] + 3 \times 5 \times 2 = 30, \\
\pi[1, 2] &= 1 \\
\text{opt}[2, 3] &= \text{opt}[2, 2] + \text{opt}[3, 3] + 5 \times 2 \times 6 = 60, \\
\pi[2, 3] &= 2 \\
\text{opt}[3, 4] &= \text{opt}[3, 3] + \text{opt}[4, 4] + 2 \times 6 \times 9 = 108, \\
\pi[3, 4] &= 3 \\
\text{opt}[4, 5] &= \text{opt}[4, 4] + \text{opt}[5, 5] + 6 \times 9 \times 4 = 216, \\
\pi[4, 5] &= 4 \\
\text{opt}[1, 3] &= \min\{ \text{opt}[1, 1] + \text{opt}[2, 3] + 3 \times 5 \times 6, \\
&\quad \text{opt}[1, 2] + \text{opt}[3, 3] + 3 \times 2 \times 6 \} \\
&= \min\{ 0 + 60 + 90, 30 + 0 + 36 \} = 66, \\
\pi[1, 3] &= 2 \\
\text{opt}[2, 4] &= \min\{ \text{opt}[2, 2] + \text{opt}[3, 4] + 5 \times 2 \times 9, \\
&\quad \text{opt}[2, 3] + \text{opt}[4, 4] + 5 \times 6 \times 9 \} \\
&= \min\{ 0 + 108 + 90, 60 + 0 + 270 \} = 198, \\
\pi[2, 4] &= 2,
\end{align*}
\]
<table>
<thead>
<tr>
<th>matrix size</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 \times 5$</td>
<td>$5 \times 2$</td>
<td>$2 \times 6$</td>
<td>$6 \times 9$</td>
<td>$9 \times 4$</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{opt}[3, 5] &= \min\{ \text{opt}[3, 3] + \text{opt}[4, 5] + 2 \times 6 \times 4, \\
& \quad \text{opt}[3, 4] + \text{opt}[5, 5] + 2 \times 9 \times 4 \} \\
&= \min\{ 0 + 216 + 48, 108 + 0 + 72 \} = 180, \\
\pi[3, 5] &= 4, \\
\text{opt}[1, 4] &= \min\{ \text{opt}[1, 1] + \text{opt}[2, 4] + 3 \times 5 \times 9, \\
& \quad \text{opt}[1, 2] + \text{opt}[3, 4] + 3 \times 2 \times 9, \\
& \quad \text{opt}[1, 3] + \text{opt}[4, 4] + 3 \times 6 \times 9 \} \\
&= \min\{ 0 + 198 + 135, 30 + 108 + 54, 66 + 0 + 162 \} = 192, \\
\pi[1, 4] &= 2,
\end{align*}
\]
<table>
<thead>
<tr>
<th>matrix size</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 × 5</td>
<td>5 × 2</td>
<td>2 × 6</td>
<td>6 × 9</td>
<td>9 × 4</td>
<td></td>
</tr>
</tbody>
</table>

$opt[2, 5] = \min\{opt[2, 2] + opt[3, 5] + 5 \times 2 \times 4,\]

$opt[2, 3] + opt[4, 5] + 5 \times 6 \times 4,$

$opt[2, 4] + opt[5, 5] + 5 \times 9 \times 4}\}

$= \min\{0 + 180 + 40, 60 + 216 + 120, 198 + 0 + 180\} = 220,$

$opt[1, 5] = \min\{opt[1, 1] + opt[2, 5] + 3 \times 5 \times 4,\]

$opt[1, 2] + opt[3, 5] + 3 \times 2 \times 4,$

$opt[1, 3] + opt[4, 5] + 3 \times 6 \times 4,$

$opt[1, 4] + opt[5, 5] + 3 \times 9 \times 4\}

$= \min\{0 + 220 + 60, 30 + 180 + 24,\]

$66 + 216 + 72, 192 + 0 + 108\}$

$= 234,$

$\pi[1, 5] = 2.$
<table>
<thead>
<tr>
<th>matrix</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$3 \times 5$</td>
<td>$5 \times 2$</td>
<td>$2 \times 6$</td>
<td>$6 \times 9$</td>
<td>$9 \times 4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opt, $\pi$</th>
<th>$j = 1$</th>
<th>$j = 2$</th>
<th>$j = 3$</th>
<th>$j = 4$</th>
<th>$j = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>0, /</td>
<td>30, 1</td>
<td>66, 2</td>
<td>192, 2</td>
<td>234, 2</td>
</tr>
<tr>
<td>$i = 2$</td>
<td></td>
<td>0, /</td>
<td>60, 2</td>
<td>198, 2</td>
<td>220, 2</td>
</tr>
<tr>
<td>$i = 3$</td>
<td></td>
<td></td>
<td>0, /</td>
<td>108, 3</td>
<td>180, 4</td>
</tr>
<tr>
<td>$i = 4$</td>
<td></td>
<td></td>
<td></td>
<td>0, /</td>
<td>216, 4</td>
</tr>
<tr>
<td>$i = 5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, /</td>
</tr>
<tr>
<td>( opt, \pi )</td>
<td>( j = 1 )</td>
<td>( j = 2 )</td>
<td>( j = 3 )</td>
<td>( j = 4 )</td>
<td>( j = 5 )</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>( i = 1 )</td>
<td>0, /</td>
<td>30, 1</td>
<td>66, 2</td>
<td>192, 2</td>
<td>234, 2</td>
</tr>
<tr>
<td>( i = 2 )</td>
<td></td>
<td>0, /</td>
<td>60, 2</td>
<td>198, 2</td>
<td>220, 2</td>
</tr>
<tr>
<td>( i = 3 )</td>
<td></td>
<td></td>
<td>0, /</td>
<td>108, 3</td>
<td>180, 4</td>
</tr>
<tr>
<td>( i = 4 )</td>
<td></td>
<td></td>
<td></td>
<td>0, /</td>
<td>216, 4</td>
</tr>
<tr>
<td>( i = 5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, /</td>
</tr>
</tbody>
</table>

Print-Optimal-Order(1,5)

Print-Optimal-Order(1, 2)

Print-Optimal-Order(1, 1)

Print-Optimal-Order(2, 2)

Print-Optimal-Order(3, 5)

Print-Optimal-Order(3, 4)

Print-Optimal-Order(3, 3)

Print-Optimal-Order(4, 4)

Print-Optimal-Order(5, 5)

Optimum way for multiplication: \(((A_1A_2)((A_3A_4)A_5))\)
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
Optimum Binary Search Tree

- $n$ elements $e_1 < e_2 < e_3 < \cdots < e_n$
- $e_i$ has frequency $f_i$
- goal: build a binary search tree for $\{e_1, e_2, \cdots, e_n\}$ with the minimum accessing cost:

$$\sum_{i=1}^{n} f_i \times \text{(depth of } e_i \text{ in the tree)}$$
Optimum Binary Search Tree

Example: $f_1 = 10, f_2 = 5, f_3 = 3$

- $10 \times 1 + 5 \times 2 + 3 \times 3 = 29$
- $10 \times 2 + 5 \times 1 + 3 \times 2 = 31$
- $10 \times 3 + 5 \times 2 + 3 \times 1 = 43$
suppose we decided to let \( e_k \) be the root

\( e_1, e_2, \cdots, e_{k-1} \) are on left sub-tree

\( e_{k+1}, e_{k+2}, \cdots, e_n \) are on right sub-tree

\( d_j \): depth of \( e_j \) in our tree

\( C, C_L, C_R \): cost of tree, left sub-tree and right sub-tree

\( d_1 = 3, d_2 = 2, d_3 = 3, d_4 = 4, d_5 = 1, d_6 = 2, d_7 = 4, d_8 = 3, d_9 = 4, \)

\[ C = 3f_1 + 2f_2 + 3f_3 + 4f_4 + f_5 + 2f_6 + 4f_7 + 3f_8 + 4f_9 \]

\[ C_L = 2f_1 + f_2 + 2f_3 + 3f_4 \]

\[ C_R = f_6 + 3f_7 + 2f_8 + 3f_9 \]

\[ C = C_L + C_R + \sum_{j=1}^{9} f_j \]
\[ C = \sum_{\ell=1}^{n} f_{\ell}d_{\ell} = \sum_{\ell=1}^{n} f_{\ell}(d_{\ell} - 1) + \sum_{\ell=1}^{n} f_{\ell} \]

\[ = \sum_{\ell=1}^{k-1} f_{\ell}(d_{\ell} - 1) + \sum_{\ell=k+1}^{n} f_{\ell}(d_{\ell} - 1) + \sum_{\ell=1}^{n} f_{\ell} \]

\[ = C_{L} + C_{R} + \sum_{\ell=1}^{n} f_{\ell} \]
\[ C = C_L + C_R + \sum_{\ell=1}^{n} f_\ell \]

- In order to minimize \( C \), need to minimize \( C_L \) and \( C_R \) respectively

- \( opt[i, j] \): the optimum cost for the instance \((f_i, f_{i+1}, \cdots, f_j)\)

\[
opt[1, n] = \min_{k:1 \leq k \leq n} \left( opt[1, k - 1] + opt[k + 1, n] \right) + \sum_{\ell=1}^{n} f_\ell
\]

- In general, \( opt[i, j] = \)

\[
\begin{cases} 
0 & \text{if } i = j + 1 \\
\min_{k:i \leq k \leq j} \left( opt[i, k - 1] + opt[k + 1, j] \right) + \sum_{\ell=i}^{j} f_\ell & \text{if } i \leq j
\end{cases}
\]
Optimum Binary Search Tree

1: \(f \text{sum}[0] \leftarrow 0\)

2: \(\text{for } i \leftarrow 1 \text{ to } n \text{ do } f \text{sum}[i] \leftarrow f \text{sum}[i - 1] + f_i\)
   \(\triangleright f \text{sum}[i] = \sum_{j=1}^{i} f_j\)

3: \(\text{for } i \leftarrow 0 \text{ to } n \text{ do } \text{opt}[i + 1, i] \leftarrow 0\)

4: \(\text{for } \ell \leftarrow 1 \text{ to } n \text{ do}\)

5: \(\text{for } i \leftarrow 1 \text{ to } n - \ell + 1 \text{ do}\)

6: \(j \leftarrow i + \ell - 1, \text{opt}[i, j] \leftarrow \infty\)

7: \(\text{for } k \leftarrow i \text{ to } j \text{ do}\)

8: \(\text{if } \text{opt}[i, k - 1] + \text{opt}[k + 1, j] < \text{opt}[i, j] \text{ then}\)

9: \(\text{opt}[i, j] \leftarrow \text{opt}[i, k - 1] + \text{opt}[k + 1, j]\)

10: \(\pi[i, j] \leftarrow k\)

11: \(\text{opt}[i, j] \leftarrow \text{opt}[i, j] + f \text{sum}[j] - f \text{sum}[i - 1]\)
Printing the Tree

**Print-Tree**\((i, j)\)

1. **if** \(i > j\) **then**
2. **return**
3. **else**
4. **print**('(')
5. **Print-Tree**\((i, \pi[i, j] - 1)\)
6. **print**(\(\pi[i, j]\))
7. **Print-Tree**\((\pi[i, j] + 1, j)\)
8. **print**(')')
Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
   - Related Problem: Knapsack Problem
3. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
4. Shortest Paths in Directed Acyclic Graphs
5. Matrix Chain Multiplication
6. Optimum Binary Search Tree
7. Summary
**Dynamic Programming**

- Break up a problem into many *overlapping* sub-problems
- Build solutions for larger and larger sub-problems
- Use a **table** to store solutions for sub-problems for reuse
Comparison with greedy algorithms

- Greedy algorithm: each step is making a small progress towards constructing the solution
- Dynamic programming: the whole solution is constructed in the last step

Comparison with divide and conquer

- Divide and conquer: an instance is broken into many independent sub-instances, which are solved separately.
- Dynamic programming: the sub-instances we constructed are overlapping.
Definition of Cells for Problems We Learnt

- Weighted interval scheduling: \( opt[i] = \) value of instance defined by jobs \( \{1, 2, \cdots, i\} \)
- Subset sum, knapsack: \( opt[i, W'] = \) value of instance with items \( \{1, 2, \cdots, i\} \) and budget \( W' \)
- Longest common subsequence: \( opt[i, j] = \) value of instance defined by \( A[1..i] \) and \( B[1..j] \)
- Shortest paths in DAG: \( f[v] = \) length of shortest path from \( s \) to \( v \)
- Matrix chain multiplication, optimum binary search tree: \( opt[i, j] = \) value of instances defined by matrices \( i \) to \( j \)