CSE 421

Weighted Interval Scheduling, Knapsack, RNA Secondary Structure

Shayan Oveis Gharan
Weighted Interval Scheduling
Interval Scheduling

- Job $j$ starts at $s(j)$ and finishes at $f(j)$ and has weight $w_j$
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

![Interval Scheduling Diagram]
Weighted Job Scheduling by Induction

Suppose 1, ..., n are all jobs. Let us use induction:

IH (strong ind): Suppose we can compute the optimum job scheduling for < n jobs.

IS: Goal: For any n jobs we can compute OPT.

Case 1: Job n is not in OPT.
-- Then, just return OPT of 1, ..., n − 1.

Case 2: Job n is in OPT.
-- Then, delete all jobs not compatible with n and recurse.

Q: Are we done?
A: No, How many subproblems are there? Potentially $2^n$ all possible subsets of jobs.

This idea works for any Optimization problem, e.g., vertex-cover, independent set, etc. For NP-hard problems there is no ordering to reduce # subproblems.

Take best of the two
Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$

IS: For jobs 1,...,n we want to compute OPT

Case 1: Suppose OPT has job n.
  - So, all jobs i that are not compatible with n are not OPT
  - Let $p(n) =$ largest index $i < n$ such that job $i$ is compatible with n.
  - Then, we just need to find OPT of 1, ..., $p(n)$
Sorting to reduce Subproblems

**Sorting Idea:** Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$

IS: For jobs 1,\ldots,n we want to compute OPT

**Case 1:** Suppose OPT has job n.
- So, all jobs $i$ that are not compatible with $n$ are not OPT
- Let $p(n) =$ largest index $i < n$ such that job $i$ is compatible with $n$.
- Then, we just need to find OPT of $1, \ldots, p(n)$

**Case 2:** OPT does not select job n.
- Then, OPT is just the OPT of $1, \ldots, n - 1$

Q: Have we made any progress (still reducing to two subproblems)?
A: Yes! This time every subproblem is of the form $1, \ldots, i$ for some $i$
So, at most $n$ possible subproblems.
Weighted Job Scheduling by Induction

**Sorting Idea:** Label jobs by finishing time \( f(1) \leq \cdots \leq f(n) \)

Def \( \text{OPT}(j) \) denote the OPT solution of 1, ..., \( j \)

To solve \( \text{OPT}(j) \):

**Case 1:** \( \text{OPT}(j) \) has job \( j \).
- So, all jobs \( i \) that are not compatible with \( j \) are not \( \text{OPT}(j) \)
- Let \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
- So \( \text{OPT}(j) = \text{OPT}(p(j)) \cup \{j\} \).

**Case 2:** \( \text{OPT}(j) \) does not select job \( j \).
- Then, \( \text{OPT}(j) = \text{OPT}(j - 1) \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max (w_j + \text{OPT}(p(j)), \text{OPT}(j - 1)) & \text{o. w.}
\end{cases}
\]
Input: $n$, $s(1),...,s(n)$ and $f(1),...,f(n)$ and $w_1,...,w_n$.

Sort jobs by finish times so that $f(1) \leq f(2) \leq \cdots f(n)$.

Compute $p(1), p(2),..., p(n)$

Compute-$Opt(j)$
\[
\begin{align*}
&\text{if } (j = 0) \\
&\quad \text{return } 0 \\
&\text{else} \\
&\quad \text{return } \max(w_j + \text{Compute-$Opt(p(j))$}, \text{Compute-$Opt(j-1)$})
\end{align*}
\]
Recursive Algorithm Fails

Even though we have only $n$ subproblems, we do not store the solution to the subproblems

- So, we may re-solve the same problem many many times.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence

$p(1) = 0$, $p(j) = j - 2$
Algorithm with Memorization

**Memoization.** Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

**Input:** n, s(1), ..., s(n) and f(1), ..., f(n) and w_1, ..., w_n.

Sort jobs by finish times so that f(1) ≤ f(2) ≤ … f(n).

Compute p(1), p(2), ..., p(n)

for j = 1 to n
    M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    return M[j]
}

Memoization
You can also avoid recursion
• recursion may be easier conceptually when you use induction

**Input**: n, s(1),...,s(n) and f(1),...,f(n) and w₁,...,wₙ.

Sort jobs by finish times so that f(1) ≤ f(2) ≤ ⋯ f(n).

Compute p(1),p(2),...,p(n)

Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(wᵢ + M[p(j)], M[j-1])
}

Output M[n]

**Claim**: M[j] is value of OPT(j)

**Timing**: Easy. Main loop is O(n); sorting is O(n log n)
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
\begin{array}{|c|c|c|c|}
\hline
j & w_j & p(j) & OPT(j) \\
\hline
0 & 0 & 0 & 0 \\
1 & 3 & 0 & 0 \\
2 & 4 & 0 & 0 \\
3 & 1 & 0 & 0 \\
4 & 3 & 1 & 1 \\
5 & 4 & 0 & 0 \\
6 & 3 & 2 & 2 \\
7 & 2 & 3 & 3 \\
8 & 4 & 5 & 5 \\
\hline
\end{array}
\]
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

<table>
<thead>
<tr>
<th></th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>( \text{OPT}(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j \).
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).

<table>
<thead>
<tr>
<th>j</th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>OPT($j$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>OPT($j$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$. 

<table>
<thead>
<tr>
<th>j</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Knapsack Problem
Knapsack Problem

Given $n$ objects and a "knapsack."

Item $i$ weighs $w_i > 0$ kilograms and has value $v_i > 0$.

Knapsack has capacity of $W$ kilograms.

**Goal:** fill knapsack so as to maximize total value.

Ex: OPT is $\{3, 4\}$ with value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

$W = 11$

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.

Ex: $\{5, 2, 1\}$ achieves only value $= 35 \Rightarrow$ greedy not optimal.
Dynamic Programming: First Attempt

Let $OPT(i) =$ Max value of subsets of items $1, \ldots, i$ of weight $\leq W$.

**Case 1:** $OPT(i)$ does not select item $i$
- In this case $OPT(i) = OPT(i - 1)$

**Case 2:** $OPT(i)$ selects item $i$
- In this case, item $i$ does not immediately imply we have to reject other items
- The problem does not reduce to $OPT(i - 1)$ because we now want to pack as much value into box of weight $\leq W - w_i$

**Conclusion:** We need more subproblems, we need to strengthen IH.
Stronger DP (Strengthening Hypothesis)

Let $OPT(i, w) = \text{Max value of subsets of items } 1, \ldots, i \text{ of weight } \leq w$

Case 1: $OPT(i, w)$ selects item $i$
• In this case, $OPT(i, w) = v_i + OPT(i - 1, w - w_i)$

Case 2: $OPT(i, w)$ does not select item $i$
• In this case, $OPT(i, w) = OPT(i - 1, w)$.

Therefore,

$$OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i - 1, w) & \text{if } w_i > w \\
\max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)) & \text{o.w.,}
\end{cases}$$
DP for Knapsack

Compute-OPT(i,w)
    if M[i,w] == empty
        if (i==0)
            M[i,w]=0
        else if (w_i > w)
            M[i,w]=Comp-OPT(i-1,w)
        else
            M[i,w]= max {Comp-OPT(i-1,w), v_i + Comp-OPT(i-1,w-w_i)}
    return M[i, w]

for w = 0 to W
    M[0, w] = 0
for i = 1 to n
    for w = 1 to W
        if (w_i > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
    return M[n, W]
DP for Knapsack

\[
\begin{array}{cccccccccccc}
& & & & & & & & & & & \\
& & & & & & & & & & W + 1 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\phi & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\{1\} & 0 & & & & & & & & & & \\
\{1,2\} & 0 & & & & & & & & & & \\
\{1,2,3\} & 0 & & & & & & & & & & \\
\{1,2,3,4\} & 0 & & & & & & & & & & \\
\{1,2,3,4,5\} & 0 & & & & & & & & & & \\
\end{array}
\]

if \( w_i > w \)

\[
M[i, w] = M[i-1, w]
\]
else

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)
DP for Knapsack

\[ W = 11 \]

\[
\begin{array}{c|cccccccccccc}
\text{Item} & \text{Value} & \text{Weight} \\
\hline
1 & 1 & 1 \\
2 & 6 & 2 \\
3 & 18 & 5 \\
4 & 22 & 6 \\
5 & 28 & 7 \\
\end{array}
\]

\[
\text{if } (w_i > w) \\
M[i, w] = M[i-1, w] \\
\text{else} \\
M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\]
DP for Knapsack

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**OPT:** \{4, 3\}
value = 22 + 18 = 40

if \(w_i > w\)
\[M[i, w] = M[i-1, w]\]
else
\[M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}\]
## DP for Knapsack

### Algorithm

```python
if (w_i > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i]}
```

### Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

### Optimal Solution

- **OPT:** \{ 4, 3 \}
- value = 22 + 18 = 40
## DP for Knapsack

### Table

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

### Equation

\[
W = 11
\]

### if-else Case

\[
\text{if } (w_i > w) \\
\quad M[i, w] = M[i-1, w] \\
\text{else} \\
\quad M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\]

### Solution

\[
\text{OPT: } \{4, 3\} \\
\text{value = 22 + 18 = 40}
\]
### DP for Knapsack

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**OPT:** \{4, 3\}

\[
\text{value} = 22 + 18 = 40
\]

If \(w_i > w\)

\[
M[i, w] = M[i-1, w]
\]

Else

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>
Knapsack Problem: Running Time

Running time: $\Theta(n \cdot W)$
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:
There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum in time $\text{Poly}(n, \log W)$. 
DP Ideas so far

• You may have to define an ordering to decrease #subproblems

• You may have to strengthen DP, equivalently the induction, i.e., you have may have to carry more information to find the Optimum.

• This means that sometimes we may have to use two dimensional or three dimensional induction
RNA Secondary Structure
RNA Secondary Structure

RNA: A String $B = b_1b_2...b_n$ over alphabet \{ A, C, G, U \}.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAUGUAACACGUGGCUCACGGCGGAGA

complementary base pairs: A-U, C-G
RNA Secondary Structure (Formal)

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

[Watson-Crick.]

- $S$ is a matching and
- each pair in $S$ is a Watson-Crick pair: A-U, U-A, C-G, or G-C.

[No sharp turns.]: The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

[Non-crossing.]: If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in $S$, then we cannot have $i < k < j < l$.

Free energy: Usual hypothesis is that an RNA molecule will maximize total free energy.

Goal: Given an RNA molecule $B = b_1b_2\ldots b_n$, find a secondary structure $S$ that maximizes the number of base pairs.
Secondary Structure (Examples)

- **Base Pair**: A-U or G-C

- **Sharp Turn**: An angle in the secondary structure that is not allowed.

- **Crossing**: Two base pairs that cross each other, which is also not allowed.

- **Ok**: A structure that is allowed and validated.

Examples:

1. [Diagram 1]
2. [Diagram 2]
3. [Diagram 3]

- Diagram 1 shows a simple base pair between G and C.
- Diagram 2 illustrates a sharp turn that is not allowed.
- Diagram 3 demonstrates a crossing that is not allowed.
First attempt. Let $OPT(n) = \text{maximum number of base pairs in a secondary structure of the substring } b_1b_2...b_n$. 

Suppose $b_n$ is matched with $b_t$ in $OPT(n)$. What IH should we use? 

Diffuculty: This naturally reduces to two subproblems 
- Finding secondary structure in $b_1, \ldots, b_{t-1}$, i.e., $OPT(t-1)$
- Finding secondary structure in $b_{t+1}, \ldots, b_n$
DP: Second Attempt

Definition: \( OPT(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i, b_{i+1}, \ldots, b_j \)

The most important part of a correct DP; It fixes IH

Case 1: If \( j - i \leq 4 \).
- \( OPT(i, j) = 0 \) by no-sharp turns condition.

Case 2: Base \( b_j \) is not involved in a pair.
- \( OPT(i, j) = OPT(i, j-1) \)

Case 3: Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \)
- non-crossing constraint decouples resulting sub-problems
- \( OPT(i, j) = 1 + \max_t \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \} \)