Pole Cutting

Pole-cutting:
- Given is a pole of length $n$
- The pole can be cut into multiple pieces of integral lengths
- A pole of length $i$ is sold for price $p(i)$, for some function $p$

Example:

<table>
<thead>
<tr>
<th>length $i$</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>
</tr>
</thead>
<tbody>
<tr>
<td>price $p(i)$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

![Diagram of pole cutting](image)
Problem: **Pole-Cutting**

1. **Input:** Price table \( p_i \), for every \( i \geq 1 \), length \( n \) of initial pole
2. **Output:** Maximum revenue \( r_n \) obtainable by cutting pole into smaller pieces

How many ways of cutting the pole are there?

![Diagram of different ways to cut the pole]
There are $2^{n-1}$ ways to cut a pole of length $n$.

**Proof.**
There are $n - 1$ positions where the pole can be cut. For each position we either cut or we don’t. This gives $2^{n-1}$ possibilities.

**Problem:**
- Find best out of $2^{n-1}$ possibilities
- We could disregard similar cuts, but we would still have an exponential number of possibilities
- A fast algorithm cannot try out all possibilities
Notation

\[ 7 = 2 + 2 + 3 \]

means we cut a pole of length 7 into pieces of lengths 2, 2 and 3

Optimal Cut

- Suppose the optimal cut uses \( k \) pieces

\[ n = i_1 + i_2 + \cdots + i_k \]

- Optimal revenue \( r_n \):

\[ r_n = p(i_1) + p(i_2) + \cdots + p(i_k) \]
What are the optimal revenues $r_i$?

<table>
<thead>
<tr>
<th>length $i$</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>
</tr>
</thead>
<tbody>
<tr>
<td>price $p(i)$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

$r_1 = 1 = 1$
$r_2 = 5 = 2$
$r_3 = 8 = 3$
$r_4 = 10 = 4 = 2 + 2$
$r_5 = 13 = 5 = 2 + 3$
$r_6 = 17 = 6$
$r_7 = 18 = 7 = 2 + 2 + 3$
$r_8 = 22 = 8 = 2 + 6$
$r_9 = 25 = 9 = 3 + 6$
$r_{10} = 30 = 10 = 10$
Optimal Substructure

Consider an optimal solution to input length $n$

$$n = i_1 + i_2 + \cdots + i_k$$

for some $k$

Then:

$$n - i_1 = i_2 + \cdots + i_k$$

is an optimal solution to the problem of size $n - i_1$

Computing Optimal Revenue $r_n$:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}$$

- $p_n$ corresponds to the situation of no cut at all
- $r_i + r_{n-i}$: initial cut into two pieces of sizes $i$ and $n - i$
Simpler Recursive Formulation: Let \( r_0 = 0 \)

\[
r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) .
\]

Observe: Only one subproblem in this formulation

Example: \( n = 4 \)

\[
r_n = \max\{p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4 + r_0\}
\]
Recall:

\[ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \text{ and } r_0 = 0. \]

Direct Implementation:

```plaintext
Require: Integer n, Array p of length n with prices
if n = 0 then
    return 0
q ← −∞
for i = 1 . . . n do
    q ← max{q, p[i] + CUT-POLE(p, n − i)}
return q
```

Algorithm CUT-POLE(p, n)

How efficient is this algorithm?
Recursion Tree for \textsc{Cut-Pole}

Example: \(n = 5\)

Number Recursive Calls: \(T(n)\)

\[
T(n) = 1 + \sum_{j=0}^{n-1} T(j) \quad \text{and} \quad T(0) = 1
\]
Solving Recurrence

How to Solve this Recurrence?

\[ T(n) = 1 + \sum_{j=0}^{n-1} T(j) \text{ and } T(0) = 1 \]

- Substitution Method: Using guess \( T(n) = O(c^n) \), for some \( c \)
- Trick: compute \( T(n) - T(n - 1) \)

\[
T(n) - T(n - 1) = 1 + \sum_{j=0}^{n-1} T(j) - \left( 1 + \sum_{j=0}^{n-2} T(j) \right)
\]

\[
= T(n - 1) , \text{ hence:}
\]

\[
T(n) = 2T(n - 1).
\]

This implies \( T(i) = 2^i \).
Discussion

Runtime of Cut-Pole

- Recursion tree has $2^n$ nodes
- Each function call takes time $O(n)$ (for-loop)
- Runtime of \texttt{Cut-Pole} is therefore $O(n2^n)$. ($O(2^n)$ can also be argued)

What can we do better?

- Observe: We compute solutions to subproblems many times
- Avoid this by storing solutions to subproblems in a table!
- This is a key feature of dynamic programming
Implementing the Dynamic Programming Approach

**Top-down** with memoization
- When computing $r_i$, store $r_i$ in a table $T$ (of size $n$)
- Before computing $r_i$ again, check in $T$ whether $r_i$ has previously been computed

**Bottom-up**
- Fill table $T$ from smallest to largest index
- No recursive calls are needed for this
Top-down Approach

**Require:** Integer $n$, Array $p$ of length $n$ with prices
Let $r[0 \ldots n]$ be a new array
for $i = 0 \ldots n$ do
  $r[i] \leftarrow -\infty$
return \texttt{Memoized-Cut-Pole-Aux}(p, $n$, $r$)

Algorithm \texttt{Memoized-Cut-Pole}(p, $n$)

- Prepare a table $r$ of size $n$
- Initialize all elements of $r$ with $-\infty$
- Actual work is done in \texttt{Memoized-Cut-Pole-Aux}, table $r$
  is passed on to \texttt{Memoized-Cut-Pole-Aux}
**Top-down Approach (2)**

**Require:** Integer \( n \), array \( p \) of length \( n \) with prices, array \( r \) of revenues

\[
\text{if } r[n] \geq 0 \text{ then} \\
\quad \text{return } r[n] \\
\text{if } n = 0 \text{ then} \\
\quad q \leftarrow 0 \\
\text{else} \\
\quad q \leftarrow -\infty \\
\quad \text{for } i = 1 \ldots n \text{ do} \\
\quad \quad q \leftarrow \max\{q, p[i] + \text{Memoized-Cut-Pole-Aux}(p, n - i, r)\} \\
\quad r[n] \leftarrow q \\
\text{return } q
\]

**Algorithm** \text{Memoized-Cut-Pole-Aux}(p, n, r)

**Observe:** If \( r[n] \geq 0 \) then \( r[n] \) has been computed previously
Require: Integer \( n \), array \( p \) of length \( n \) with prices

Let \( r[0 \ldots n] \) be a new array

\[
r[0] \leftarrow 0
\]

for \( j = 1 \ldots n \) do

\[
q \leftarrow -\infty
\]

for \( i = 1 \ldots j \) do

\[
q \leftarrow \max\{q, p[i] + r[j - i]\}
\]

\[
r[j] \leftarrow q
\]

return \( r[n] \)

Algorithm **BOTTOM-UP-CUT-POLE**(\( p, n \))

Runtime: Two nested for-loops

\[
\sum_{j=1}^{n} \sum_{i=1}^{j} O(1) = O(1) \sum_{j=1}^{n} \sum_{i=1}^{j} 1 = O(1) \sum_{j=1}^{n} j = O(1) \frac{n(n+1)}{2} = O(n^2).
\]
Conclusion

Runtime of Top-down Approach $O(n^2)$

(please think about this!)

Dynamic Programming

- Solves a problem by combining subproblems
- Subproblems are solved at most once, store solutions in table
- If a problem exhibits *optimal substructure* then dynamic programming is often the right choice
- Top-down and bottom-up approaches have the same runtime