**Dynamic Programming**

**Weighted Activity Selection**

Weighted activity selection problem (generalization of CLR 17.1).
- Job requests 1, 2, … , N.
- Job j starts at s_j, 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.

**Activity Selection: Greedy Algorithm**

Recall greedy algorithm works if all weights are 1.

**Greedy Activity Selection Algorithm**

Sort jobs by increasing finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_N \).

\[
S = \emptyset \\
\text{FOR } j = 1 \text{ to } N \\
\quad \text{IF (job } j \text{ compatible with } A) \\
\quad \quad S \leftarrow S \cup \{j\} \\
\text{RETURN } S
\]

**Weighted Activity Selection**

Notation.
- Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_N \).
- Define \( q_j = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j \).
  - \( q_7 = 3, q_2 = 0 \)
Weighted Activity Selection: Structure

Let \( \text{OPT}(j) = \text{value of optimal solution to the problem consisting of job requests } \{1, 2, \ldots, j\} \).

- **Case 1:** \( \text{OPT} \) selects job \( j \).
  - can’t use incompatible jobs \( \{q_j + 1, q_j + 2, \ldots, j-1\} \)
  - must include optimal solution to problem consisting of remaining compatible jobs \( \{1, 2, \ldots, q_j\} \)

- **Case 2:** \( \text{OPT} \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs \( \{1, 2, \ldots, j - 1\} \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max\{w_j + \text{OPT}(q_j), \text{OPT}(j - 1)\} & \text{otherwise}
\end{cases}
\]

Weighted Activity Selection: Brute Force

**Recursive Activity Selection**

**INPUT:** \( N, s_1, \ldots, s_N, f_1, \ldots, f_N, w_1, \ldots, w_N \)

Sort jobs by increasing finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_N \).

Compute \( q_1, q_2, \ldots, q_N \)

\[
r\text{-compute}(j) = \begin{cases} 
\text{RETURN } 0 & \text{if } (j = 0) \\
\max\{w_j + r\text{-compute}(q_j), r\text{-compute}(j-1)\} & \text{otherwise}
\end{cases}
\]

Dynamic Programming Subproblems

Spectacularly redundant subproblems \( \Rightarrow \) exponential algorithms.

```
1, 2, 3, 4, 5, 6, 7, 8
1, 2, 3, 4
1, 2, 3
1

1, 2, 3, 4, 5
1, 2, 3
1

1, 2, 3, 4, 5, 6, 7
1, 2, 3, 4, 5
1

1, 2, 3, 4, 5, 6, 7, 8
1, 2, 3, 4
1

1, 2, 3, 4, 5
1, 2, 3
1

1, 2, 3, 4, 5
1, 2
```

Divide-and-Conquer Subproblems

Independent subproblems \( \Rightarrow \) efficient algorithms.

```
1, 2, 3, 4, 5, 6, 7, 8
1, 2, 3, 4
1

1, 2, 3, 4, 5, 6, 7, 8
5, 6, 7, 8
1

1, 2, 3, 4, 5, 6, 7, 8
5, 6
1

1, 2, 3, 4, 5, 6, 7, 8
7, 8
1
```


Weighted Activity Selection: Memoization

**Memoized Activity Selection**

**INPUT:** \( N, s_1, \ldots, s_N, f_1, \ldots, f_N, w_1, \ldots, w_N \)

Sort jobs by increasing finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_N \).

Compute \( q_1, q_2, \ldots, q_N \)

Global array \( OPT[0..N] \)

FOR \( j = 0 \) to \( N \)

\( OPT[j] = "empty" \)

\( m\text{-compute}(j) \{
\quad \text{IF } (j = 0) \quad OPT[0] = 0
\quad \text{ELSE IF } (OPT[j] = "empty")
\quad \quad \quad \quad OPT[j] = \max(w_j + m\text{-compute}(q_j), m\text{-compute}(j-1))
\quad \quad \quad \quad \text{RETURN } OPT[j]
\quad \}
\)

Weighted Activity Selection: Running Time

Claim: memoized version of algorithm takes \( O(N \log N) \) time.

- Ordering by finish time: \( O(N \log N) \).
- Computing \( q_j \): \( O(N \log N) \) via binary search.
- \( m\text{-compute}(j) \): each invocation takes \( O(1) \) time and either
  - (i) returns an existing value of \( OPT[] \)
  - (ii) fills in one new entry of \( OPT[] \) and makes two recursive calls

Progress measure \( \Phi = \# \text{ nonempty entries of } OPT[] \).
- Initially \( \Phi = 0 \), throughout \( \Phi \leq N \).
- (ii) increases \( \Phi \) by 1 \( \Rightarrow \) at most \( 2N \) recursive calls.

Overall running time of \( m\text{-compute}(N) \) is \( O(N) \).

Weighted Activity Selection: Finding a Solution

\( m\text{-compute}(N) \) determines value of optimal solution.

- Modify to obtain optimal solution itself.

**Finding an Optimal Set of Activities**

**ARRAY:** \( OPT[0..N] \)

Run \( m\text{-compute}(N) \)

\( \text{find-sol}(j) \{ 
\quad \text{IF } (j = 0) \quad \text{output nothing}
\quad \text{ELSE IF } (w_j + OPT[q_j] > OPT[j-1])
\quad \quad \quad \quad \text{print } j
\quad \quad \quad \quad \text{find-sol}(q_j)
\quad \quad \quad \quad \text{ELSE}
\quad \quad \quad \quad \text{find-sol}(j-1)
\quad \}
\)

- \# of recursive calls \( \leq N \Rightarrow O(N) \).

Weighted Activity Selection: Bottom-Up

Unwind recursion in memoized algorithm.

**Bottom-Up Activity Selection**

**INPUT:** \( N, s_1, \ldots, s_N, f_1, \ldots, f_N, w_1, \ldots, w_N \)

Sort jobs by increasing finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_N \).

Compute \( q_1, q_2, \ldots, q_N \)

**ARRAY:** \( OPT[0..N] \)

\( OPT[0] = 0 \)

FOR \( j = 1 \) to \( N \)

\( OPT[j] = \max(w_j + OPT[q_j], OPT[j-1]) \)
Dynamic Programming Overview

Dynamic programming.
  ■ Similar to divide-and-conquer.
  – solves problem by combining solution to sub-problems
  ■ Different from divide-and-conquer.
  – sub-problems are not independent
  – save solutions to repeated sub-problems in table

Recipe.
  ■ Characterize structure of problem.
  – optimal substructure property
  ■ Recursively define value of optimal solution.
  ■ Compute value of optimal solution.
  ■ Construct optimal solution from computed information.

Top-down vs. bottom-up.
  ■ Different people have different intuitions.

Least Squares

Least squares.
  ■ Foundational problem in statistic and numerical analysis.
  ■ Given N points in the plane \( \{ (x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N) \} \),
  find a line \( y = ax + b \) that minimizes the sum of the squared error:

\[
SS = \sum_{i=1}^{N} (y_i - ax_i - b)^2
\]

  ■ Calculus \( \Rightarrow \) min error is achieved when:

\[
a = \frac{N \sum_{i} x_i y_i - \left( \sum_{i} x_i \right) \left( \sum_{i} y_i \right)}{N \sum_{i} x_i^2 - \left( \sum_{i} x_i \right)^2}, \quad b = \frac{\sum_{i} y_i - a \sum_{i} x_i}{N}
\]

Segmented Least Squares

Segmented least squares.
  ■ Points lie roughly on a sequence of 3 lines.
  ■ Given N points in the plane \( p_1, p_2, \ldots, p_N \), find a sequence of lines that minimize:
  – the sum of the sum of the squared errors \( E \) in each segment
  – the number of lines \( L \)
  ■ Tradeoff function: \( e + cL \), for some constant \( c > 0 \).

Optimal solution:
  ■ Last segment uses points \( p_i, p_{i+1}, \ldots, p_j \) for some \( i \).
  ■ Cost = \( e(i, j) + c + OPT(i-1) \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\min_{1 \leq i \leq j} \{ e(i, j) + c + OPT(i-1) \} & \text{otherwise}
\end{cases}
\]

New dynamic programming technique.
  ■ Weighted activity selection: binary choice.
  ■ Segmented least squares: multi-way choice.
Segmented Least Squares: Algorithm

### Bottom-Up Segmented Least Squares

**INPUT:** N, P_1, ..., P_N, c

**ARRAY:** OPT[0..N]

OPT[0] = 0

FOR j = 1 to N

FOR i = 1 to j

compute the least square error e[i, j] for the segment P_i, ..., P_j

OPT[j] = min_1 ≤ i ≤ j (e[i, j] + c + OPT[i-1])

RETURN OPT[N]

**Running time:**
- Bottleneck = computing e(i, n) for O(N^2) pairs, O(N) per pair using previous formula.
- O(N^3) overall.

### Segmented Least Squares: Improved Algorithm

A quadratic algorithm.
- Bottleneck = computing e(i, j).
- O(N^2) preprocessing + O(1) per computation.

**Preprocessing**

\[
a_{ij} = \frac{\sum_{k=1}^{j} x_k y_k}{n} - \left( \frac{\sum_{k=1}^{j} x_k}{n} \right)^2 - \left( \frac{\sum_{k=1}^{j} y_k}{n} \right)^2
\]

\[
b_{ij} = \frac{\sum_{k=1}^{j} y_k - a \sum_{k=1}^{j} x_k}{n}
\]

\[
n_{ij} = j - i + 1
\]

\[
x_s_k = \sum_{k=1}^{i} x_k \quad y_s_k = \sum_{k=1}^{i} y_k
\]

\[
x_{ss} = \sum_{k=1}^{i} x_k^2 \quad y_{ss} = \sum_{k=1}^{i} y_k^2
\]

\[
x y_k = \sum_{k=1}^{i} x_k y_k
\]

\[
e(i, j) = \sum_{k=i}^{j} (y_k - ax_k - b)^2 = (y_{ss} - y_{ss}) + \cdots
\]

Knapsack Problem

**Knapsack problem.**
- Given N objects and a "knapsack."
- Item i weighs w_i > 0 Newtons and has value v_i > 0.
- Knapsack can carry weight up to W Newtons.
- Goal: fill knapsack so as to maximize total value.

<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>

\[
\text{Greedy = 35: \{ 5, 2, 1 \}}
\]

\[
\text{OPT value = 40: \{ 3, 4 \}}
\]

\[
W = 11
\]

### Knapsack Problem: Structure

OPT(n, w) = max profit subset of items {1, ..., n} with weight limit w.
- Case 1: OPT selects item n.
  - new weight limit = w - w_n
  - OPT selects best of {1, 2, ..., n – 1} using this new weight limit
- Case 2: OPT does not select item n.
  - OPT selects best of {1, 2, ..., n – 1} using weight limit w

\[
OPT(n, w) = \begin{cases} 
0 & \text{if } n = 0 \\
OPT(n-1, w) & \text{if } w_n > w \\
\max\{OPT(n-1, w), \ v_n + OPT(n-1, w-w_n)\} & \text{otherwise}
\end{cases}
\]

New dynamic programming technique.
- Weighted activity selection: binary choice.
- Segmented least squares: multi-way choice.
- Knapsack: adding a new variable.
Knapsack Problem: Bottom-Up

Bottom-Up Knapsack

**INPUT**: \( N, W, w_1, \ldots, w_N, v_1, \ldots, v_N \)

**ARRAY**: OPT[0..N, 0..W]

FOR \( w = 0 \) to \( W \)

\[ \text{OPT}[0, w] = 0 \]

FOR \( n = 1 \) to \( N \)

FOR \( w = 1 \) to \( W \)

IF \( (w_n > w) \)

\[ \text{OPT}[n, w] = \text{OPT}[n-1, w] \]

ELSE

\[ \text{OPT}[n, w] = \max \{ \text{OPT}[n-1, w], v_n + \text{OPT}[n-1, w-w_n] \} \]

RETURN \( \text{OPT}[N, W] \)

---

Knapsack Problem: Running Time

Knapsack algorithm runs in time \( O(NW) \).

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is "NP-complete."
- Optimization version is "NP-hard."

Knapsack approximation algorithm.

- There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.
- Stay tuned.

---

Knapsack Algorithm

**Weight Limit**

<table>
<thead>
<tr>
<th>Weight Limit</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>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</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>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>28</td>
<td>29</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>28</td>
<td>29</td>
<td>34</td>
<td>35</td>
<td>40</td>
</tr>
</tbody>
</table>

**Item** | **Value** | **Weight**
---|---|---
1 | 1 | 1
2 | 6 | 2
3 | 8 | 5
4 | 22 | 6
5 | 28 | 7

---

Sequence Alignment

How similar are two strings?

- **ocurrance**
- **occurrence**

5 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Sequence Alignment: Applications

Applications.
- Spell checkers / web dictionaries.
  - occurrence
  - occurrence
- Computational biology.
  - ctgacactct
  - cctgactacat

Edit distance.
- Gap penalty $\delta$.
- Mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

Sequence Alignment: Problem Structure

$$OPT(i, j) = \min \text{ cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j$$
- Case 1: OPT matches $(i, j)$.
  - pay mismatch for $(i, j)$ + min cost of aligning two strings $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
- Case 2a: OPT leaves $m$ unmatched.
  - pay gap for $i$ and min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$
- Case 2b: OPT leaves $n$ unmatched.
  - pay gap for $j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta + \alpha_{x_i, y_j} + OPT(i-1, j-1), & \text{if } i = 0 \\ \min \{ \delta + OPT(i-1, j), \delta + OPT(i, j-1) \}, & \text{otherwise} \\ i\delta + OPT(i, j-1), & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

$O(MN)$ time and space.

Bottom-Up Sequence Alignment

**INPUT**: $M, N, x_1 x_2 \ldots x_M, y_1 y_2 \ldots y_N, \delta, \alpha$

**ARRAY**: $OPT[0..M, 0..N]$

**FOR** $i = 0$ to $M$
  **OPT**[$0$, $i$] = $i\delta$

**FOR** $j = 0$ to $N$
  **OPT**[$j$, $0$] = $j\delta$

**FOR** $i = 1$ to $M$
  **FOR** $j = 1$ to $N$
    **OPT**[$i$, $j$] = $\min(\alpha_{x_i, y_j} + \text{OPT}[i-1, j-1], \delta + \text{OPT}[i-1, j], \delta + \text{OPT}[i, j-1])$

**RETURN** $OPT[M, N]$
Sequence Alignment: Linear Space

Straightforward dynamic programming takes \( \Theta(MN) \) time and space.
- English words or sentences \( \Rightarrow \) may not be a problem.
- Computational biology \( \Rightarrow \) huge problem.
  - \( M = N = 100,000 \)
  - 10 billion ops OK, but 10 gigabyte array?

Optimal value in \( O(M + N) \) space and \( O(MN) \) time.
- Only need to remember \( \text{OPT}(i - 1, \cdot) \) to compute \( \text{OPT}(i, \cdot) \).
- Not clear how to recover optimal alignment itself.

Optimal alignment in \( O(M + N) \) space and \( O(MN) \) time.
- Clever combination of divide-and-conquer and dynamic programming.

Consider following directed graph (conceptually).
- Note: takes \( \Theta(MN) \) space to write down graph.

Let \( f(i, j) \) be shortest path from \((0,0)\) to \((i, j)\). Then, \( f(i, j) = \text{OPT}(i, j) \).

Let \( g(i, j) \) be shortest path from \((0,0)\) to \((i, j)\). Then, \( f(i, j) = \text{OPT}(i, j) \).

\[
\begin{align*}
    f(i, j) &= \min \{ \alpha_{xy} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1) \} \\
    &= \min \{ \alpha_{xy} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1) \} \\
    &= \text{OPT}(i, j)
\end{align*}
\]
Observation 1: the cost of the shortest path that uses \((i, j)\) is \(f(i, j) + g(i, j)\).

Observation 2: let \(q\) be an index that minimizes \(f(q, N/2) + g(q, N/2)\). Then, the shortest path from \((0, 0)\) to \((M, N)\) uses \((q, N/2)\).

**Sequence Alignment: Linear Space**

Divide: find index \(q\) that minimizes \(f(q, N/2) + g(q, N/2)\) using DP.

Conquer: recursively compute optimal alignment in each "half."

\[ T(m, n) = \max \text{ running time of algorithm on strings of length } m \text{ and } n. \]

**Theorem.** \(T(m, n) = O(mn)\).

- \(O(mn)\) work to compute \(f(\cdot, n/2)\) and \(g(\cdot, n/2)\).
- \(O(m + n)\) to find best index \(q\).
- \(T(q, n/2) + T(m - q, n/2)\) work to run recursively.
- Choose constant \(c\) so that:

\[
T(m, 2) \leq cn \\
T(n, 2) \leq cm \\
T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)
\]

- Base cases: \(m = 2\) or \(n = 2\).
- Inductive hypothesis: \(T(m, n) \leq 2cmn\).