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.
Recall greedy algorithm works if all weights are 1.

**Greedy Activity Selection Algorithm**

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

$S = \emptyset$

FOR $j = 1$ to $N$

IF (job $j$ compatible with $A$)

$S \leftarrow S \cup \{j\}$

RETURN $S$
Weighted Activity Selection

Notation.

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

Let \( \text{OPT}(j) = \) 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-compute**($j$) {
    IF ($j = 0$)  
        RETURN 0  
    ELSE  
        return max($w_j + r$-compute($q_j$), $r$-compute($j-1$))  
}
Dynamic Programming Subproblems

Spectacularly redundant subproblems ⇒ exponential algorithms.
Divide-and-Conquer Subproblems

Independent subproblems ⇒ efficient algorithms.

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

1, 2, 3, 4

1, 2

3, 4

3, 4

5, 6

5, 6

7, 8

7, 8

5, 6

5, 6

7, 8

7, 8
**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 \(\text{OPT}[0..N]\)

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

\[
\text{OPT}[j] = "empty"
\]

\(m\)-compute\((j)\) {

IF \((j = 0)\)

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

ELSE IF \((\text{OPT}[j] = "empty")\)

\[
\text{OPT}[j] = \max(w_j + m\text{-compute}(q_j), m\text{-compute}(j-1))
\]

RETURN \(\text{OPT}[j]\)

}
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.
- $m$-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 $\Phi = \#$ 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 $m$-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-compute(N)

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

- \# 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:

$$a = \frac{N \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{N \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{N}$$
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 \(c > 0\).
Segmented Least Squares: Structure

Notation.
- \( \text{OPT}(j) = \) minimum cost for points \( p_1, p_{i+1}, \ldots, p_j \).
- \( e(i, j) = \) minimum sum of squares for points \( p_i, p_{i+1}, \ldots, p_j \).

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

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

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:** \(N, p_1, \ldots, p_N, c\)

**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 \(p_i, \ldots, p_j\)

\(OPT[j] = \min_{1 \leq i \leq j} (e[i, j] + c + OPT[i-1])\)

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.

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

$$b_{ij} = \frac{\sum_{k=i}^{j} y_k - a \sum_{k=i}^{j} x_k}{n}$$

$$n_{ij} = j - i + 1$$

$$\sum_{k=i}^{j} x_k = xs_j - xs_{i-1}$$

$$e(i, j) = \sum_{k=i}^{j} (y_k - ax_k - b)^2$$

$$= (yys_j - yys_{i-1}) + \cdots$$

Preprocessing
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>

\( v_i / w_i \)

\( W = 11 \)

Greedy = 35: \{ 5, 2, 1 \}

OPT value = 40: \{ 3, 4 \}
Knapsack Problem: Structure

\[ OPT(n, w) = \text{max profit subset of items } \{1, \ldots, n\} \text{ with weight limit } w. \]

- Case 1: OPT selects item \( n \).
  - new weight limit = \( w - w_n \)
  - OPT selects best of \( \{1, 2, \ldots, n-1\} \) using this new weight limit

- Case 2: OPT does not select item \( n \).
  - OPT selects best of \( \{1, 2, \ldots, n-1\} \) 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

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

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

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

\[ OPT[0, w] = 0 \]

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

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

IF \( (w_n > w) \)

\[ OPT[n, w] = OPT[n-1, w] \]

ELSE

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

RETURN \( OPT[N, W] \)
## Knapsack Algorithm

<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 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>8</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

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.
Sequence Alignment

How similar are two strings?
- occurrence
- occurrence

5 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Sequence Alignment: Applications

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

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

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

$$2\delta + \alpha_{CA}$$
Problem.
- Input: two strings $X = x_1 x_2 \ldots x_M$ and $Y = y_1 y_2 \ldots y_N$.
- Notation: $\{1, 2, \ldots, M\}$ and $\{1, 2, \ldots, N\}$ denote positions in $X$, $Y$.
- Matching: set of ordered pairs $(i, j)$ such that each item occurs in at most one pair.
- Alignment: matching with no crossing pairs.
  - if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$, then $j < j'$

$$\text{cost}(M) = \sum_{(i, j) \in M} \alpha_{x_i y_j} + \sum_{i : (i, j) \in M} \delta + \sum_{j : (i, j) \in M} \delta$$

- Example: CTACCG vs. TACATG.
  - $M = \{ (2,1) (3,2) (4,3), (5,4), (6,6) \}$

- Goal: find alignment of minimum cost.
Sequence Alignment: Problem Structure

OPT(i, j) = min cost of aligning strings x_1 x_2 \ldots x_i 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}

\[
OPT(i, j) = \begin{cases} 
  j \delta & \text{if } i = 0 \\
  \min \left\{ \alpha_{x_i, y_j} + OPT(i-1, j-1), \right. \\
  \left. \delta + OPT(i-1, j), \delta + OPT(i, j-1) \right\} & \text{otherwise} \\
  i \delta & \text{if } j = 0 
\end{cases}
\]
Sequence Alignment: Algorithm

O(MN) time and space.

<table>
<thead>
<tr>
<th>Bottom-Up Sequence Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT</strong>: M, N, x₁x₂...xₘ, y₁y₂...yₙ, δ, α</td>
</tr>
<tr>
<td><strong>ARRAY</strong>: OPT[0..M, 0..N]</td>
</tr>
</tbody>
</table>

FOR i = 0 to M
    OPT[0, i] = iδ

FOR j = 0 to N
    OPT[j, 0] = jδ

FOR i = 1 to M
    FOR j = 1 to N
        OPT[i, j] = min(α[xᵢ, yⱼ] + OPT[i-1, j-1], δ + OPT[i-1, j], δ + 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 $OPT(i-1, \cdot)$ to compute $OPT(i, \cdot)$.
- 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.
**Sequence Alignment: Linear Space**

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

- **Base case:** $f(0, 0) = \text{OPT}(0, 0) = 0$.
- **Inductive step:** assume $f(i', j') = \text{OPT}(i', j')$ for all $i' + j' < i + j$.
- Last edge on path to (i, j) is either from (i-1, j-1), (i-1, j), or (i, j-1).

$$f(i, j) = \min \{ \alpha_{x_iy_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1) \}$$
$$= \min \{ \alpha_{x_iy_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1) \}$$
$$= \text{OPT}(i, j)$$
Let $g(i, j)$ be shortest path from $(i, j)$ to $(M, N)$.

- Can compute in $O(MN)$ time for all $(i, j)$ by reversing arc orientations and flipping roles of $(0, 0)$ and $(M, N)$.

Sequence Alignment: Linear Space

![Graph diagram showing the alignment process]
Observation 1: the cost of the shortest path that uses \((i, j)\) is \(f(i, j) + g(i, j)\).
Sequence Alignment: Linear Space

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."

\[
\begin{array}{cccc}
\varepsilon & y_1 & y_2 & y_3 \\
\varepsilon & 0-0 & & \\
x_1 & & & \\
x_2 & & & \\
x_3 & & & \\
\end{array}
\]
Sequence Alignment: Linear Space

T(m, n) = max running time of algorithm on strings of length m 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:

\[
egin{align*}
T(m, 2) & \leq cn \\
T(n, 2) & \leq cm \\
T(m, n) & \leq cmn + T(q, n/2) + T(m - q, n/2)
\end{align*}
\]

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

\[
egin{align*}
T(m, n) & \leq T(q, n/2) + T(m - q, n/2) + cmn \\
& \leq 2cq n/2 + 2c(m - q)n/2 + cmn \\
& = cq n + cm n - cq n + cm n \\
& = 2cm n
\end{align*}
\]