Algorithmic Paradigms

**Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
6.1 Weighted Interval Scheduling
Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- 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.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.
**Weighted Interval Scheduling**

**Notation.** Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

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

**Ex:** \( p(8) = 5, p(7) = 3, p(2) = 0. \)
Dynamic Programming: Binary Choice

Notation. \( OPT(j) = \) value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- **Case 1:** \( OPT \) selects job j.
  - collect profit \( v_j \)
  - can't use incompatible jobs \{ p(j) + 1, p(j) + 2, ..., j - 1 \}
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

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

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{ v_j + OPT(p(j)), \ OPT(j-1) \right\} & \text{otherwise}
\end{cases}
\]
Weighted Interval Scheduling: Brute Force

**Brute force algorithm.**

\[
\text{Input: } n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n
\]

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

Compute \( p(1), p(2), \ldots, p(n) \)

\[
\text{Compute-Opt}(j) \{
\text{if } (j = 0) \\
\quad \text{return } 0 \\
\text{else} \\
\quad \text{return } \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))
\}
\]
Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

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

\[ p(1) = 0, \ p(j) = j-2 \]

![Diagram](image-url)
Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

**Input:** $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

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

Compute $p(1), p(2), \ldots, p(n)$

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

M-Compute-Opt($j$) {
  if (M[$j$] is empty)
    M[$j$] = max($v_j + M$-Compute-Opt($p(j)$), M-Compute-Opt($j-1$))
  return M[$j$]
}
Weighted Interval Scheduling: Running Time

**Claim.** Memoized version of algorithm takes $O(n \log n)$ time.
- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- $M$-$\text{Compute-Opt}(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

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

- **Overall running time of** $M$-$\text{Compute-Opt}(n)$ **is** $O(n)$. 

**Remark.** $O(n)$ if jobs are pre-sorted by start and finish times.
Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n)$ after sorting by start time.

- $M\text{-Compute-Opt}(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$,
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \# \text{nonempty entries of } M[]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 ⇒ at most $2n$ recursive calls.

- Overall running time of $M\text{-Compute-Opt}(n)$ is $O(n)$.

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

Input: \(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n\)

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

Compute \(p(1), p(2), \ldots, p(n)\)

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

Output \(M[n]\)

Claim: \(M[j]\) is value of optimal solution for jobs 1..\(j\)
Timing: Easy. Main loop is \(O(n)\); sorting is \(O(n \log n)\)
Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

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

**Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$. 

<table>
<thead>
<tr>
<th>$j$</th>
<th>$v_j$</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>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time

0 1 2 3 4 5 6 7 8 9 10 11
Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing - “traceback”

```plaintext
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

- # of recursive calls ≤ n ⇒ O(n).
Dynamic Programming - iterative approach

Have a collection of subproblems that satisfy a few basic properties:

- Only polynomially many.
- The solution to the original problem can be easily computed from the solutions to the subproblems.
- There is a natural ordering on subproblems from “smallest” to “largest” together with an easy to compute recurrence that allows us to determine the solution to a subproblem from the solution to some number of smaller subproblems.
6.4 Knapsack Problem
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: $\{3, 4\}$ has value 40.

<table>
<thead>
<tr>
<th>#</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: False Start

**Def.** $OPT(i) =$ max profit subset of items $1, \ldots, i$.

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$

- **Case 2:** $OPT$ selects item $i$.
  - accepting item $i$ does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before $i$,
    we don't even know if we have enough room for $i$

**Conclusion.** Need more sub-problems!
Dynamic Programming: Adding a New Variable

**Def.** $OPT(i, w) = \text{max profit subset of items 1, ..., i with weight limit } w.$

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$ using weight limit $w$

- **Case 2:** $OPT$ selects item $i$.
  - New weight limit $= w - w_i$
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$ using this new weight limit

\[
OPT(i, w) = \begin{cases} 
  0 & \text{if } i = 0 \\
  OPT(i - 1, w) & \text{if } w_i > w \\
  \max\{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an n-by-W array.

\( M(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w. \)

**Input:**  \( n, W, w_1, \ldots, w_N, v_1, \ldots, v_N \)

\[
\begin{align*}
\text{for } & \ w = 0 \text{ to } \ W \\
& \ M[0, w] = 0 \\
\text{for } & \ i = 1 \text{ to } \ n \\
& \ \ \text{for } \ w = 1 \text{ to } \ W \\
& \ \ \ \ \text{if } \ (w_i > w) \\
& \ \ \ \ \quad M[i, w] = M[i-1, w] \\
& \ \ \ \ \text{else} \\
& \ \ \ \ \quad M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\} \\
\text{return } & \ M[n, W]
\end{align*}
\]
\( M(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w. \)

**Input:** \( n, W, w_1, \ldots, w_N, v_1, \ldots, v_N \)

\[
\text{for } w = 0 \text{ to } W \\
M[0, w] = 0
\]

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

\text{return } M[n, W]

<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>

\( W = 11 \)
Knapsack Algorithm

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
\phi & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\{1\} & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\{1, 2\} & 0 & 1 & 6 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\
\{1, 2, 3\} & 0 & 1 & 6 & 7 & 7 & 18 & 19 & 24 & 25 & 25 & 25 \\
\{1, 2, 3, 4\} & 0 & 1 & 6 & 7 & 7 & 18 & 22 & 24 & 28 & 29 & 29 \\
\{1, 2, 3, 4, 5\} & 0 & 1 & 6 & 7 & 7 & 18 & 22 & 28 & 29 & 34 & 34 \\
\end{array}
\]

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

W = 11

<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>
\[ M(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w. \]

**Input:** \( n, W, w_1, \ldots, w_N, v_1, \ldots, v_N \)

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 \( M[n, W] \)

How do you find the actual solution once you’re filled out the table?
Dynamic Programming – iterative approach

Have a collection of subproblems that satisfy a few basic properties:

- Only polynomially many (hopefully)

- The solution to the original problem can be easily computed from the solutions to the subproblems.

- There is a natural ordering on subproblems from “smallest” to “largest” together with an easy to compute recurrence that allows us to determine the solution to a subproblem from the solution to some number of smaller subproblems.
Knapsack Problem: Running Time

Running time. $\Theta(nW)$.
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]
6.6 Sequence Alignment
String Similarity

How similar are two strings?

- occurrence
- occurrence

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Edit Distance

Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

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

\[ \alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} \]

\[ 2\delta + \alpha_{CA} \]
Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

Def. An alignment $M$ is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i$-$y_j$ and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$. Don’t allow crossing.

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

Ex: CTACCG vs. TACATG.
Sol: $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$. 
Sequence Alignment: Problem Structure

**Def.** $OPT(i, j) = \min \text{ cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j.$

- **Case 1:** $OPT$ matches $x_i$-$y_j$.
  - pay mismatch for $x_i$-$y_j$ + 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 $x_i$ unmatched.
  - pay gap for $x_i$ 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 $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

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

Sequence-Alignment(m, n, \(x_1 x_2 \ldots x_m\), \(y_1 y_2 \ldots y_n\), \(\delta\), \(\alpha\)) {
    for i = 0 to m
        M[i, 0] = i\delta
    for j = 0 to n
        M[0, j] = j\delta

    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(\(\alpha[x_i, y_j] + M[i-1, j-1]\),
                            \(\delta + M[i-1, j]\),
                            \(\delta + M[i, j-1]\))

    return M[m, n]
}

Analysis. \(\Theta(mn)\) time and space.

English words or sentences: \(m, n \leq 10\).

Computational biology: \(m = n = 100,000\). 10 billions ops OK, but 10GB array?
Dynamic Programming - iterative/bottom-up approach

Have a collection of subproblems that satisfy a few basic properties:

- Only polynomially many.
- The solution to the original problem can be easily computed from the solutions to the subproblems.
- There is a natural ordering on subproblems from “smallest” to “largest” together with an easy to compute recurrence that allows us to determine the solution to a subproblem from the solution to some number of smaller subproblems.