Dynamic Programming

Algorithmic Paradigms

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

**Divide-and-conquer.** Break up a problem into two 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.

Dynamic Programming

**Steps**

- Characterize structure of optimal solution
- Define a recurrence for the value of an optimal solution
- Compute the value of an optimal solution
  - Either bottom-up
  - Top-down with memoization
- Construct the optimal solution from optimal value and intermediate information

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.

Unweighted Interval Scheduling Review

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.

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 $j$.
  - 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.

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

Compute-Opt\( (j) \) {
    if \( j = 0 \)
        return 0
    else
        return max(\( v_j + \text{Compute-Opt}(p(j)) \), \text{Compute-Opt}(j-1))
}

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

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 — global array
    \( M[j] = 0 \)

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

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 \( p() \): \( O(n) \) 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 \( \Rightarrow \) 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.
Automated Memoization

Automated memoization. Many functional programming languages (e.g., Lisp) have library or built-in support for memoization.

```
(def-memo-fun F (n)
  (if (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Java (exponential)

```
static int F(int n) {
    if (n <= 1) return n;
    else return F(n-1) + F(n-2);
}
```

Lisp (efficient)

```
class FibonacciMemo {
    public static int F(int n) {
        if (n <= 1) return n;
        else return memo[n];
    }
}
```

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,...,sn, f1,...,fn, v1,...,vn
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), ..., p(n)
Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(vj + M[p(j)], M[j-1])
}
```

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.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)
Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (vj + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

Knapsack Problem

```
Run M-Compute-Opt(n)
Run Find-Solution(n)
Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (vj + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

- # of recursive calls ≤ n ⇒ O(n).
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: choose subset of items to fill knapsack so as to maximize total value.

**Ex:** \{3, 4\} has 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>
```

**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, ..., i.
- 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.
  - 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

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

**Knapsack Problem: Bottom-Up**

**Input:** $n, w_1, \ldots, w_N, v_1, \ldots, v_N$

```
for w = 0 to W
   M[0, w] = 0
for i = 0 to n
   M[i, 0] = 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]
```
**Knapsack Algorithm**

<table>
<thead>
<tr>
<th></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>$\phi$</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>22</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>34</td>
<td>40</td>
</tr>
</tbody>
</table>

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

value = 22 + 18 = 40

**Knapsack Problem: Running Time**

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

**Finding Optimal Substructure**

A solution to the problem makes a choice
- Leaves one or more subproblems to be solved
- For example, choose whether to use an item

Figure out which subproblems to use if you knew the choice that led to an optimal solution
- Try all possible choices and choose the one that provides the optimal solution

Show that each subproblem in an optimal solution is itself optimal

**Optimal substructure**
- How many subproblems are needed in an optimal solution to the original problem
  - For the examples so far, this has been one
- How many choices must be considered

**RNA Secondary Structure**
RNA Secondary Structure

RNA. String B = b₁b₂…bₙ 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: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA
complementary base pairs: A-U, C-G

RNA Secondary Structure: Examples

Examples.

match bₜ and bₙ

RNA Secondary Structure: Subproblems

First attempt. OPT(j) = maximum number of base pairs in a secondary structure of the substring b₁b₂…bⱼ.

Difficulty. Results in two sub-problems.
• Finding secondary structure in: b₁b₂…bⱼ⁻¹
• Finding secondary structure in: b₁b₂…bₙ⁻¹

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

Goal. Given an RNA molecule B = b₁b₂…bₙ, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure

Secondary structure. A set of pairs S = { (bᵢ, bⱼ) } that satisfy:
• [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: 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 (bᵢ, bⱼ) ∈ S, then i < j - 4.
• [Non-crossing.] If (bᵢ, bⱼ) and (bₖ, bₗ) are two pairs in S, then we cannot have i < k < j < l.

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

Goal. Given an RNA molecule B = b₁b₂…bₙ, find a secondary structure S that maximizes the number of base pairs.
Dynamic Programming Over Intervals

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

- **Case 1.** If $i + j - 4$, $OPT(i, j) = 0$ 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
  - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$
    
    *take max over $t$ such that $i \leq t < j-4$ and $b_t$ and $b_j$ are Watson-Crick complements*

**Remark.** Same core idea in CKY algorithm to parse context-free grammars.

Bottom Up Dynamic Programming Over Intervals

**Q.** What order to solve the sub-problems?
**A.** Do shortest intervals first.

```java
RNA(b_1, ..., b_n) {
    for k = 5, 6, ..., n-1
    for i = 1, 2, ..., n-k
        j = i + k
        Compute M[i, j]
    return M[1, n]  \ using recurrence
}
```

**Running time.** $O(n^3)$.

Dynamic Programming Summary

**Recipe.**
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

**Dynamic programming techniques.**
- Binary choice: weighted interval scheduling.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

**Top-down vs. bottom-up:** different people have different intuitions.

Sequence Alignment
String Similarity

How similar are two strings?
- occurrence
- occurrence

Goal: Given two strings X = x1 x2 . . . xm and Y = y1 y2 . . . yn find alignment of minimum cost.

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

Def. The pair xi- yj and xj- yj’ cross if i < i’, but j > j’.

Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings x1 x2 . . . xi and y1 y2 . . . yj
- Case 1: OPT matches xi- yj:
  - pay mismatch for xi- yj + min cost of aligning two strings
    xi- xj+1 and yj- yj+1
- Case 2a: OPT leaves xi unmatched.
  - pay gap for xi and min cost of aligning xi- xj+1 and yj- yj+1
- Case 2b: OPT leaves yj unmatched.
  - pay gap for yj and min cost of aligning xi- xj+1 and yj- yj+1

Case 1:
OPT(i, j) = \begin{align*}
\min_{i' = i - 1, j' = j} & \alpha_{x_{i'}} + \delta \text{ if } i = 0 \\
\min_{i' = i, j' = j - 1} & \delta + \alpha_{y_{j'}} + \delta \text{ otherwise} \\
\min_{i' = i - 1, j' = j - 1} & \gamma
\end{align*}

Sequence Alignment

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.

Case 2a: OPT leaves x\textsuperscript{i} unmatched.
- pay gap for x\textsuperscript{i} and min cost of aligning x\textsuperscript{i} x\textsuperscript{i+1} and y\textsuperscript{j} y\textsuperscript{j+1}

Case 2b: OPT leaves y\textsuperscript{j} unmatched.
- pay gap for y\textsuperscript{j} and min cost of aligning x\textsuperscript{i} x\textsuperscript{i+1} and y\textsuperscript{j} y\textsuperscript{j+1}

Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings x\textsuperscript{1} x\textsuperscript{2} . . . x\textsuperscript{i} and y\textsuperscript{1} y\textsuperscript{2} . . . y\textsuperscript{j}:
- Case 1: OPT matches x\textsuperscript{i} y\textsuperscript{j}:
  - pay mismatch for x\textsuperscript{i} y\textsuperscript{j} + min cost of aligning two strings
    x\textsuperscript{i} x\textsuperscript{i+1} and y\textsuperscript{j} y\textsuperscript{j+1}
- Case 2a: OPT leaves x\textsuperscript{i} unmatched.
  - pay gap for x\textsuperscript{i} and min cost of aligning x\textsuperscript{i} x\textsuperscript{i+1} and y\textsuperscript{j} y\textsuperscript{j+1}
- Case 2b: OPT leaves y\textsuperscript{j} unmatched.
  - pay gap for y\textsuperscript{j} and min cost of aligning x\textsuperscript{i} x\textsuperscript{i+1} and y\textsuperscript{j} y\textsuperscript{j+1}

Case 1:
OPT(i, j) = \begin{align*}
\min_{i' = i - 1, j' = j} & \alpha_{x_{i'}} + \delta \text{ if } i = 0 \\
\min_{i' = i, j' = j - 1} & \delta + \alpha_{y_{j'}} + \delta \text{ otherwise} \\
\min_{i' = i - 1, j' = j - 1} & \gamma
\end{align*}
Sequence Alignment: Algorithm

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

Sequence Alignment in Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in \( O(m + n) \) space and \( O(mn) \) time.

- Compute \( \text{OPT}(i, \cdot) \) from \( \text{OPT}(i-1, \cdot) \).
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg, 1975] Optimal alignment in \( O(m + n) \) space and \( O(mn) \) time.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Edit distance graph.

- Let \( f(i, j) \) be shortest path from \( (0,0) \) to \( (i, j) \).
- Observation: \( f(i, j) = \text{OPT}(i, j) \).
Sequence Alignment: Linear Space

Edit distance graph.

- Let \( f(i, j) \) be shortest path from \((0,0)\) to \((i,j)\).
- Can compute \( f(\cdot,j) \) for any \( j \) in \( O(mn) \) time and \( O(m + n) \) 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.
- Align \( x_q \) and \( y_{n/2} \).

Conquer: recursively compute optimal alignment in each piece.

Sequence Alignment: Running Time Analysis

Theorem. Let \( T(m, n) = \max \) running time of algorithm on strings of length at most \( m \) and \( n \). \( T(m, n) = \Theta(mn) \).

\[
T(m, n) = T(q, n/2) + T(m-q, n/2) + \Theta(mn)
\]

Note that \((q*n/2) + (m-q)*n/2 = (q+m-q)n/2 = mq/2 \)

Level | Cost
--- | ---
1 | \( cmn \)
2 | \( cmn/2 \)
... | ...
i | \( cmn/2^i \)
... | ...

Total time is \( \Theta(mn) \).

Linear space algorithm due to Hirschberg, 1975