Algorithmic paradigms

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

**Divide-and-conquer.** Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

---

Dynamic programming history

**Bellman.** Pioneered the systematic study of dynamic programming in 1950s.

**Etymology.**

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
Dynamic programming applications

Areas.
- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ....
- ...

Some famous dynamic programming algorithms.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.
- ...

**Example 0:** Fibonacci numbers

\[
F_0 = 1 \\
F_1 = 1 \\
F_2 = 2 \quad (i \geq 2) \\
F_3 = 3
\]

Given \( n \), compute \( F_n \).

**Algol:**

```
Fib(n)
if n = 0 or n = 1
then return 1
else return Fib(n-1) + Fib(n-2)
```

**Example sum:**

\( \text{Fib}(7) \)
Problem: Many Fib(.) values are recomputed! \(\Rightarrow\) slow runtime
Exponential

Solution: Remember & don't recompute! Memoization.

**Algorithm 2:**

Fib(n) make array F[0..n]
F[0] = 0
F[1] = 1
for i = 2 to n
  F[i] = F[i-2] + F[i-1]
end for
return F[n]

Run time: \(O(n)\)

\[
\text{Fib(5)} = 5 \quad F[2] = 2
\]
Example 1: Wall Climbing

Cost of a path = sum of costs of the cells on the path.

Greedy Algorithm?

Dynamic Programming Algorithm

Template:
(1) Describe an array of values (numbers) to compute. Each array entry corresponds to a sub-problem of the original problem.

(2) Give a recurrence to compute the values in the array: a "big" problem can be solved using the solutions to some "small" sub-problems.

(3) Give a program to compute the array values: a "bottom-up" algorithm.

(4) Using the array values, compute an optimal solution to the original problem.

1. \( A(i, j) = \) cost of the cheapest path from bottom row to cell \((i, j)\)

\[ 1 \leq i \leq m, \quad 1 \leq j \leq n \]

Best cost to get to top row:
\[ \min \{ A(m, 1), \ldots, A(m, n) \} \]

2. Recurrence

\( (*) \) \( A(i, j) = c(i, j) + \min \{ A(i-1, j), A(i-1, j+1) \} \)

\( \forall j \)

3. Use \( (*) \) to fill in the array
from bottom row up.

\[ \text{Time: } O(n \cdot m) \]
\[ \text{(const time per array element)} \]

4. Recover an actual cheapest path
(by "tracing" back through our array \( A(i,j) \)).

\[
C(i,j) = \begin{cases} 
C(i, j) & \text{if } i = 1 \\
\min \left( \frac{C(i, j)}{2} + 5, C(i, j) \right) & \text{otherwise}
\end{cases}
\]

\[
A(i, j) = \begin{cases} 
C(i, j) & \text{if } i = 1 \\
C(i, j) + \min \{ A(i-1, j), A(i-1, j+1) \} & \text{otherwise}
\end{cases}
\]

**Alg PrintOpt \((i, j)\)**  
\% print optimal path  
\% ending at cell \((i, j)\)

\[
\text{print } (i, j)
\]
\[
\text{if } i = 1 \text{ then return}
\]
\[
\text{else find } k \in \{j-1, j, j+1\} \text{ such that}
\]
\[
A[i-1, k] = \min \{ A[i-1, j-1], A[i-1, j], A[i-1, j+1] \}
\]
\[
\text{PrintOpt } (i-1, k)
\]
\text{end Alg}
The main call is: \( \text{PrintOpt}(m, j) \)
where \( j \) is such that \( A[m, j] = \min_{1 \leq k \leq n} \{A[m, k]\} \).

The runtime of \( \text{PrintOpt}(m, j) \): \( O(m) \).

The overall time of the DP algo to find a cheapest path on the wall: \( O(m \cdot n) + O(m) \)
\( \leq O(m \cdot n) \)

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.
Earliest-finish-time first algorithm

Earliest finish-time first.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

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

DP Algorithm

1. Array: Define $M[j] =$ the value of an optimal solution for the subset of jobs $1, \ldots, j$

2. Recurrence: Two possibilities: (a) either job $j$ is part of an optimal solution, or (b) job $j$ is not.

Hence, either $M[j] = v[j] + M[p(j)]$, or

So, the recurrence is

$$M[0] = 0$$
$$M[j] = \max\{ v[j] + M[p(j)] , M[j-1] \}$$

(3) **Algorithm to fill in the array:**

```
M[0] = 0
for j = 1 to n
    M[j] = \max\{ v[j] + M[p(j)] , M[j-1] \}
end for
```

(4) **Recover an actual optimal schedule from $M[]$:**

```
Find-Solution(j)
if j = 0
    return \emptyset.
else if (v[j] + M[p(j)] > M[j-1])
    return \{ j \} \cup Find-Solution(p[j]).
else
    return Find-Solution(j-1).
```

**Weighted interval scheduling: finding a solution**

**Q.** DP algorithm computes optimal value. How to find solution itself?

**A.** Make a second pass.

**Analysis.** # of recursive calls $\leq n \Rightarrow O(n)$.

**Runtime:** $O(n \log n)$.

- Preprocessing:
  - Sorting $v(i)$’s: $O(n \log n)$.
  - Computing $p(i)$’s: $O(n \log n)$. 

**Value we want**

```
Knapsack problem

- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \).
- Goal: fill knapsack so as to maximize total value.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w_i )</th>
<th>( v_i )</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>

knapsack instance (weight limit \( W = 11 \))

Ex. \{ 1, 2, 5 \} has value 35.
Ex. \{ 3, 4 \} has value 40.
Ex. \{ 3, 5 \} has value 46 (but exceeds weight limit).

Greedy by value. Repeatedly add item with maximum \( v_i \).
Greedy by weight. Repeatedly add item with minimum \( w_i \).
Greedy by ratio. Repeatedly add item with maximum ratio \( v_i / w_i \).

Observation. None of greedy algorithms is optimal.

Dynamic programming: false start

Def. \( \text{OPT}(i) \) max profit subset of items \( 1, \ldots, i \).

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

Case 2. \( \text{OPT} \) selects item \( i \).
- Selecting 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 subproblems!
Dynamic programming: adding a new variable

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

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

Case 2. $OPT$ selects item $i$.
- New weight limit $= w - w_i$.
- $OPT$ selects best of $\{ 1, 2, \ldots, i-1 \}$ 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

```python
KNAPSACK (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 $w = 0$ TO $W$
        IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.
        ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.
```

$M[1, w] = \begin{cases} 0 & w_i > w \\ v_i & w_i \leq w \end{cases}$
### Knapsack problem: bottom-up demo

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>

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

<table>
<thead>
<tr>
<th>weight limit w</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>subset of items $S \subseteq {1, \ldots, i}$</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></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></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></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></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></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></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>

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

---

**Algorithm**: Print Opt \((i, w)\)

\[
\begin{align*}
\text{% find } S \subseteq \{1, 2, \ldots, i\} \text{ of max value} \\
\text{% st. the weight of } S \leq w \\
\text{% the main call will be: Print Opt (n, W)} \\
\text{if } i = 0 \text{ they return } \emptyset \\
\text{else if } M[i, w] = M[i-1, w] \\
\quad \text{return PrintOpt}(i-1, w) \\
\text{else return } \{i\} \cup \text{PrintOpt}(i-1, w - w_i)
\end{align*}
\]

---

### Knapsack problem: running time

**Theorem.** There exists an algorithm to solve the knapsack problem with \(n\) items and maximum weight \(W\) in \(\Theta(nW)\) time and \(\Theta(nW)\) space.

**Pf.**

- Takes \(\Theta(1)\) time per table entry.
- There are \(\Theta(nW)\) table entries.
- After computing optimal values, can trace back to find solution.
- Takes $O(1)$ time per table entry.
- There are $\Theta(nW)$ table entries.
- After computing optimal values, can trace back to find solution: take item $i$ in $OPT(i, w)$ iff $M[i, w] > M[i-1, w]$.

$$n, w_1, \ldots, w_n, W$$

**Remarks.**
- Not polynomial in input size ("pseudo-polynomial")
- Decision version of knapsack problem is NP-complete. [Chapter 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [Section 11.8]

**Input size:**
$$n (\log \max_v v_i + \log \max_w w_i) + \log W$$

**Actual Runtime:**
$$O(nW)$$

$W \leq n^2 \leq O(n^3)$

$$\# \text{digits of } A = O(\log A)$$

$$+ 1000 = A$$

$$3,045 = B$$

$$9,045 = C$$

Time: $O(\log A)$

$$1111\ldots111$$

1000 times

202K

Time: $O(1)$
"Good" also: poly \((\log A, \log B)\)