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.

---

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD E. BELLMAN

1. Introduction. Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multistage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables; the choice of a decision being identical with the choice of a transformation. The outcomes of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maintaining some function of the parameters describing the initial state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients in a medical clinic, from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories, from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.
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

\[
F_0 = 1 \\
F_1 = 1 \\
F_2 = 2 \\
F_3 = 3 \\
F_i = F_{i-2} + F_{i-1} \quad (i \geq 2)
\]

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

Algorithm 1: \( \text{Fib}(n) \)

\[
\text{if } \quad n = 0 \quad \text{OR} \quad n = 1 \\
\text{then return } \quad 1 \\
\text{else return } \quad \text{Fib}(n-1) + \text{Fib}(n-2)
\]

Example run:

\[
\text{Fib}(7) \\
\quad \text{Fib}(5) \\
\quad \quad \text{Fib}(3) \\
\quad \quad \quad \text{Fib}(1) \quad \text{Fib}(2) \\
\quad \quad \quad \quad \quad \text{Fib}(3) \\
\quad \quad \quad \quad \quad \quad \text{Fib}(3) \quad \text{Fib}(2) \quad \text{Fib}(1)
\]

Problem: Many \( \text{Fib}(\cdot) \) values are recomputed! \( \Rightarrow \) slow runtime exponential
Solution: Remember & don’t recompute! Memoization.

**algo 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: \(O(n)\)

**Example 1:** Wall Climbing

\[ C(i,j) = \text{cost of cell } (i,j) \]
Goal: Get from the first (bottom) row to the last (top) row via a cheapest path.

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.

(2) Give a recurrence to compute the values in the array.

(3) Give a program to compute the array values.

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

1. \( A(i, j) = \text{cost of the cheapest path from bottom row to cell } (i, j) \)
   \[ 1 \leq i \leq n \]
   \[ 1 \leq j \leq m \]
   Best cost to get to top row:
   \[ \min \{ A(n, 1), \ldots, A(n, m) \} \]

2. Recurrence

\[ (\ast) A(i, j) = c(i, j) + \min \{ A(i-1, j), A(i, j-1), A(i-1, j-1) \} \]
\[ A(1, j) = C(1, j), \forall j \]

3. Use (14) to fill in the array from bottom row up.

Time: \( O(n \cdot m) \) (cost time per array element)

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

![Diagram of weighted interval scheduling](image)
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.** $p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

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

![Diagram of weighted interval scheduling](image)

**DP Algorithm**

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

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


   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:

\[
M[0] = 0
\]

\[
\text{for } j=1 \text{ to } n
\]

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

end for

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

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?
A. Make a second pass.

\[
\text{Find-Solution}(j)
\]

\[
\text{if } j = 0
\]

\[
\text{return } \emptyset.
\]

\[
\text{else if } (v[j] + M[p[j]] > M[j-1])
\]

\[
\text{return } \{j\} \cup \text{Find-Solution}(p[j]).
\]

\[
\text{else}
\]

\[
\text{return Find-Solution}(j-1).
\]

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

Runtime: \(O(n \log n)\).
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>( v_i )</th>
<th>( w_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 \( W = 11 \))

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

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

Case 2. \( OPT \) selects item \( i \).
   • 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. $OPT(i, w) = \max$ profit subset of items $1, \ldots, i$ 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

\[
\text{KNAPSACK } (n, \ W, \ w_1, \ \ldots, \ w_n, \ v_1, \ \ldots, \ v_n )
\]

\begin{align*}
\text{FOR } & \ w = 0 \ \text{TO} \ W \\
M[0, \ w] & \leftarrow 0.
\end{align*}

\begin{align*}
\text{FOR } & \ i = 1 \ \text{TO} \ n \\
\text{FOR } & \ w = 0 \ \text{TO} \ W \\
\text{IF } (w_i > w) & \ M[i, \ w] \leftarrow M[i-1, \ w]. \\
\text{ELSE} & \ M[i, \ w] \leftarrow \max \ \{ M[i-1, \ w], \ v_i + M[i-1, \ w-w_i] \}.
\end{align*}

\text{RETURN } M[n, \ W].
Knapsack problem: bottom-up demo

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

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

<table>
<thead>
<tr>
<th>subset of items</th>
<th>weight limit w</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>0  0  0  0  0  0  0  0  0  0  0  0</td>
</tr>
<tr>
<td>{ 1 }</td>
<td>0  1  1  1  1  1  1  1  1  1  1  1</td>
</tr>
<tr>
<td>{ 1, 2 }</td>
<td>0  1  6  7  7  7  7  7  7  7  7  7</td>
</tr>
<tr>
<td>{ 1, 2, 3 }</td>
<td>0  1  6  7  7  18 19 24 25 25 25 25</td>
</tr>
<tr>
<td>{ 1, 2, 3, 4 }</td>
<td>0  1  6  7  7  18 22 24 28 29 29 40</td>
</tr>
<tr>
<td>{ 1, 2, 3, 4, 5}</td>
<td>0  1  6  7  7  18 22 28 29 34 35 40</td>
</tr>
</tbody>
</table>

\(\text{OPT}(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w.\)
Knapsack problem: running time

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

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

**Remarks.**
- Not polynomial in input size! \( \Rightarrow \) "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**]