CSE 421: Intro Algorithms

Dynamic Programming, I
Intro: Fibonacci & Stamps

W. L. Ruzzo
Dynamic Programming

Outline:

- General Principles
- Easy Examples – Fibonacci, Licking Stamps
- Meatier examples
  - Weighted interval scheduling
  - String Alignment
  - RNA Structure prediction
- Maybe others
Some Algorithm Design Techniques, I: Greedy

Greedy algorithms

- Usually builds something a piece at a time
- Repeatedly make the greedy choice - the one that looks the best right away
  
  e.g. closest pair in TSP search

- Usually simple, fast if they work (but often don’t)
Some Algorithm Design Techniques, II: D & C

Divide & Conquer

Reduce problem to one or more sub-problems of the same type, i.e., a recursive solution

Typically, sub-problems are disjoint, and at most a constant fraction of the size of the original

e.g. Mergesort, Quicksort, Binary Search, Karatsuba

Typically, speeds up a polynomial time algorithm
Some Algorithm Design Techniques, III: DP

Dynamic Programming
Reduce problem to one or more sub-problems of the same type, i.e., a recursive solution
Useful when the same sub-problems show up repeatedly in the solution
Sometimes gives exponential speedups
“Dynamic Programming”

Program — A plan or procedure for dealing with some matter

— Webster’s New World Dictionary
Dynamic Programming History

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

Etymology.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - “it’s impossible to use dynamic in a pejorative sense”
  - “something not even a Congressman could object to”

A very simple case: Computing Fibonacci Numbers

Recall $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0$, $F_1 = 1$

Recursive algorithm:

Fibo(n)
    if n=0 then return(0)
    else if n=1 then return(1)
    else return(Fibo(n-1)+Fibo(n-2))
Call tree - start

```
1
0
F (1)  F (0)
1   1
1  0
F (2)  F (1)
F (3)  F (2)
F (4)  F (5)
F (6)
F (4)  F (3)
F (5)
```

Full call tree

many duplicates ⇒ exponential time!
Two Alternative Fixes

Memoization ("Caching")
Compute on demand, but don’t re-compute:
  Save answers from all recursive calls
  Before a call, test whether answer saved

Dynamic Programming (not memoized)
Pre-compute, don’t re-compute:
  Recursion become iteration (top-down → bottom-up)
  Anticipate and pre-compute needed values

DP usually cleaner, faster, simpler data structures
Fibonacci - Memoized Version

initialize: F[i] ← undefined for all i > 1
F[0] ← 0
F[1] ← 1
FiboMemo(n):
   if(F[n] undefined) {
      F[n] ← FiboMemo(n-2)+FiboMemo(n-1)
   }
   return(F[n])
Fibonacci - Dynamic Programming Version

FiboDP(n):

\[
\begin{align*}
F[0] & \leftarrow 0 \\
F[1] & \leftarrow 1 \\
\text{for } i = 2 \text{ to } n \text{ do} & \\
& \quad F[i] \leftarrow F[i-1] + F[i-2] \\
\text{end} & \\
\text{return}(F[n]) & 
\end{align*}
\]

For this problem, keeping only last 2 entries instead of full array suffices, but about the same speed.
Dynamic Programming

Useful when

Same recursive sub-problems occur \textit{repeatedly}
Parameters of these recursive calls anticipated
The solution to whole problem can be solved without knowing the \textit{internal} details of how the sub-problems are solved

“principle of optimality” – more below
Making change

Given:
Large supply of 1¢, 5¢, 10¢, 25¢, 50¢ coins
An amount N

Problem: choose fewest coins totaling N

Cashier’s (greedy) algorithm works:
Give as many as possible of the next biggest denomination
Licking Stamps

Given:
- Large supply of 5¢, 4¢, and 1¢ stamps
- An amount N

Problem: choose fewest stamps totaling N
How to Lick 27¢

<table>
<thead>
<tr>
<th># of 5¢ stamps</th>
<th># of 4¢ stamps</th>
<th># of 1¢ stamps</th>
<th>total number</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Morals: Greed doesn’t pay; success of “cashier’s alg” depends on coin denominations
A Simple Algorithm

At most $N$ stamps needed, etc.

for $a = 0, \ldots, N$
  for $b = 0, \ldots, N$
    for $c = 0, \ldots, N$
      if $(5a+4b+c == N \&\& a+b+c$ is new min)
        {retain $(a,b,c);$}
    output retained triple;

Time: $O(N^3)$
(Not too hard to see some optimizations, but we’re after bigger fish…)}
Better Idea

**Theorem:** If last stamp in an opt sol has value $v$, then previous stamps are *opt sol for* $N-v$.

**Proof:** if not, we could improve the solution for $N$ by using opt for $N-v$.

**Alg:** for $i = 1$ to $n$:

$$M(i) = \min \begin{cases} 
0 & i=0 \\
1 + M(i-5) & i \geq 5 \\
1 + M(i-4) & i \geq 4 \\
1 + M(i-1) & i \geq 1 
\end{cases}$$

where $M(i) = \min$ number of stamps totaling $i \notin$
New Idea: Recursion

\[ M(i) = \min \begin{cases} 
0 & i=0 \\
1 + M(i-5) & i \geq 5 \\
1 + M(i-4) & i \geq 4 \\
1 + M(i-1) & i \geq 1 
\end{cases} \]

Time: > \(3^{N/5}\)
Another New Idea: Avoid Recomputation

Tabulate values of solved subproblems

Top-down: “memoization”

Bottom up (better):

\[
\text{for } i = 0, \ldots, N \text{ do } \quad M[i] = \min \left\{ \begin{array}{ll}
0 & i = 0 \\
1 + M[i-5] & i \geq 5 \\
1 + M[i-4] & i \geq 4 \\
1 + M[i-1] & i \geq 1 \\
\end{array} \right.
\]

Time: $O(N)$
Finding *How Many* Stamps

<table>
<thead>
<tr>
<th>i</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>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(i)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$1 + \text{Min}(3, 1, 3) = 2$
Finding *Which* Stamps: Trace-Back

<table>
<thead>
<tr>
<th>i</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>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(i)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Min}(3, 1, 3) = 2 \]
Trace-Back

Way 1: tabulate all
add data structure storing back-pointers indicating which predecessor gave the min. (more space, maybe less time)

Way 2: re-compute just what’s needed

\[
\text{TraceBack}(i): \\
\text{if } i == 0 \text{ then return; } \\
\text{for } d \text{ in } \{1, 4, 5\} \text{ do } \\
\quad \text{if } M[i] == 1 + M[i - d] \text{ then break; } \\
\text{print } d; \\
\text{TraceBack}(i - d);
\]

\[
M[i] = \min \begin{cases} 
0 & i = 0 \\
1 + M[i - 5] & i \geq 5 \\
1 + M[i - 4] & i \geq 4 \\
1 + M[i - 1] & i \geq 1 
\end{cases}
\]
Complexity Note

$O(N)$ is better than $O(N^3)$ or $O(3^{N/5})$

But still *exponential* in input size ($\log N$ bits)

(E.g., miserable if $N$ is 64 bits — $c \cdot 2^{64}$ steps & $2^{64}$ memory.)

Note: can do in $O(1)$ for fixed denominations, e.g., 5¢, 4¢, and 1¢ (how?) but not in general. See “NP-Completeness” later.
Elements of Dynamic Programming

What feature did we use?
What should we look for to use again?

“Optimal Substructure”
Optimal solution contains optimal subproblems
A non-example: min (number of stamps mod 2)

“Repeated Subproblems”
The same subproblems arise in various ways