CSE 417: Algorithms and Computational Complexity

W. L. Ruzzo

Dynamic Programming, I:
Fibonacci & Stamps
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, least frequent pair in Huffman

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

Often very robust to problem re-definition

Sometimes gives exponential speedups
“Dynamic Programming”

Program – A plan or procedure for dealing with some matter

– Webster’s New World Dictionary

A brief, usually printed, outline of the order to be followed, of the features to be presented, and the persons participating (as in a public performance)

– merriam-webster.com
Dynamic Programming History

Richard 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 $F_0 = 0, F_1 = 1$

$$0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ 34 \ 55 \ 89 \ 144 \ 233 \ ...$$

Recursive algorithm:

FiboR(n)
  if n = 0 then return(0)
  else if n = 1 then return(1)
  else return(FiboR(n-1)+FiboR(n-2))

Note:

Exponential $\uparrow$: $F_n \approx \Phi^n / \sqrt{5}$, $\Phi = (1 + \sqrt{5}) / 2 \approx 1.618...$
Call tree - start

F (6)
  └── F (5)
      ├── F (4)
      │   ├── F (3)
      │   └── F (2)
      │       └── F (1)
      └── F (4)
            └── F (3)
                └── F (2)
                    └── F (1)
                        └── F (0)
Full call tree

many duplicates ⇒ exponential time!

\[ F(n) \approx \Phi^n/\sqrt{5} \]
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 becomes iteration (top-down $\rightarrow$ bottom-up)
  Anticipate and pre-compute needed values

DP usually cleaner, faster, simpler data structs
Fibonacci - Dynamic Programming Version

FiboDP(n):

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

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

Useful when

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

“principle of optimality” – more below, e.g. slide 18
Example: 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
A Few Ways 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.

```plaintext
for a = 0, …, N {
    for b = 0, …, N {
        for c = 0, …, 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 \( N-v \), plus \( v \).

**Alg:** for \( i = 1 \) to \( n \):

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

Claim: \( OPT(i) = \) min number of stamps totaling \( i \notin \)

Pf: induction on \( i \).
New Idea: Recursion

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

function recursive calls

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

Tabulate values of solved subproblems

for \( i = 0, \ldots, N \) do

\[
\text{OPT}(i) = \min \left\{ 0, 1 + \text{OPT}(i-1), 1 + \text{OPT}(i-4), 1 + \text{OPT}(i-5) \right\}
\]

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

![Table and diagram showing the process of finding the minimum number of stamps using dynamic programming.](image)

1 + \(\min(3, 1, 3)\) = 2

Goal
### 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>
</tr>
</thead>
<tbody>
<tr>
<td>OPT[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>
</tr>
</tbody>
</table>

\[ 1 + \text{Min}(3, 1, 3) = 2 \]

\[
OPT(i) = \min \begin{cases} 
0 & i = 0 \\
1 + OPT(i-1) & i \geq 1 \\
1 + OPT(i-4) & i \geq 4 \\
1 + OPT(i-5) & i \geq 5
\end{cases}
\]
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: just re-compute what’s needed

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

\[
OPT(i) = \min \left\{ \begin{array}{ll}
0 & i = 0 \\
1 + OPT(i - 1) & i \geq 1 \\
1 + OPT(i - 4) & i \geq 4 \\
1 + OPT(i - 5) & i \geq 5 \\
\end{array} \right. 
\]
Complexity Note

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

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 (i.e., when denominations and total are both part of the input). 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