Dynamic Programming Intro

Imran Rashid

University of Washington

February 15, 2008

Dynamic Programming

- Outline:
  - General Principles
  - Easy Examples – Fibonacci, Licking Stamps
  - Meatier examples
    - RNA Structure prediction
    - Weighted interval scheduling
    - Maybe others

Some Algorithm Design Techniques, I

- General overall idea
  - Reduce solving a problem to a smaller problem or problems of the same type
- Greedy algorithms
  - Used when one needs to build something a piece at a time
  - Repeatedly make the <u>greedy</u> choice - the one that looks the best right away
    - e.g. closest pair in TSP search
  - Usually fast if they work (but often don't)

Some Algorithm Design Techniques, II

- Divide & Conquer
  - Reduce problem to one or more sub-problems of the same type
  - Typically, each sub-problem is at most a constant fraction of the size of the original problem
    - e.g. Mergesort, Binary Search, Strassen's Algorithm, Quicksort (kind of)

Some Algorithm Design Techniques, III

- Dynamic Programming
  - Give a solution of a problem using smaller sub-problems, e.g. a recursive solution
  - Useful when the same sub-problems show up again and again in the solution

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"

Simple case:Computing Fibonacci Numbers

Recursive Call Tree

Memo-ization (Caching)

- Remember all values from previous recursive calls

Fibonacci - Memoized Version

Fibonacci - Dynamic Programming Version

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”

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¢

A Simple Algorithm

- At most N stamps needed, etc.

---

for $a = 0, \ldots, N$ **do**
    for $b = 0, \ldots, N$ **do**
        for $c = 0, \ldots, N$ **do**
            if $5a + 4b + c == N \&\& a + b + c$ is new min **then**
                retain (a,b,c)
            end if
        end for
    end for
end for

- Time: $O(N^3)$ (Not too hard to see some optimizations, but we're after bigger fish. . .)

The Magic Genie

- a useful way to think about dynamic programming (for me ...)
  - You can ask a magic genie as many thing as you want ...
  - ... but its power falls just short of your question. (Eg., it can onlyfigure how many stamps to use for up to 26¢)

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 $N - v$.

---

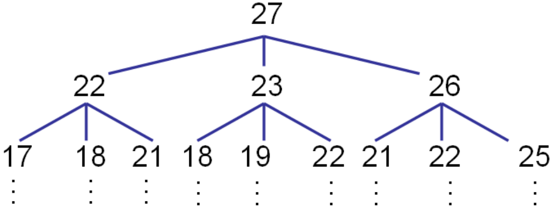
for $i = 1$ to n **do**

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

end for

New Idea: Recursion

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



Time: $O(3^n)$

Another New Idea: Avoid Recomputation

- Tabulate values of solved subproblems
  - Top-down: "memoization"
  - Bottom up:
    for i = 0, . . . , N do ;
- Time: O(N)

Finding How Many Stamps

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M(i)	0	1	2	3	1	1	2	3	2						

Finding Which Stamps: Trace-Back

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M(i)	0	1	2	3	1	1	2	3	2						

Trace-Back

- Way 1: tabulate all
  - add data structure storing back-pointers
- Way 2: re-compute just what's needed

```
procedure TRACEBACK(i)
    if i = 0 then return
    end if
    for d ∈ {1, 4, 5} do
        if M[i] == 1 + M[i − d] then break
        end if
    end for
    print d
    Traceback(i − d)
end procedure
```

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 – $c2^{64}$ steps & 2^{64} memory.)

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