CSE 417: Algorithms and Computational Complexity

4: Dynamic Programming, I
Fibonacci

Winter 2002
Lecture 4
W. L. Ruzzo

A Possible Misunderstanding?

- We have looked at types of complexity analysis:
  - worst-, best-, average-case
- types of function bounds:
  - $O$, $\Omega$, $\Theta$
- These two considerations are independent of each other:
  - one can do any type of function bound with any type of complexity analysis

Another Possible Misunderstanding?

- Insertion sort is not the best sorting algorithm, unless $n$ is < 10 or 20.

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 greedy 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
A 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

Full call tree

Memo-ization (Caching)

- Remember all values from previous recursive calls
- Before recursive call, test to see if value has already been computed
- Dynamic Programming
  - Convert memo-ized algorithm from a recursive one to an iterative one

Fibonacci - Dynamic Programming Version

- `FiboDP(n)`:
  - \( F[0] \leftarrow 0 \)
  - \( F[1] \leftarrow 1 \)
  - for \( i = 2 \) to \( n \) do
    - \( F[i] = F[i-1] + F[i-2] \)
  - endfor
  - return \( (F[n]) \)

Dynamic Programming

- Useful when
  - same recursive sub-problems occur repeatedly
  - Can anticipate the parameters of these recursive calls
  - The solution to whole problem can be figured out with knowing the internal details of how the sub-problems are solved
  - principle of optimality
List partition problem

- **Given**: a sequence of \( n \) positive integers \( s_1, \ldots, s_n \) and a positive integer \( k \)
- **Find**: a partition of the list into up to \( k \) blocks:
  \[ s_1, \ldots, s_i | s_{i+1}, \ldots, s_{i+1} | s_{i+1}, \ldots, s_n \]
  so that the sum of the numbers in the largest block is as small as possible.
  i.e. find spots for up to \( k-1 \) dividers

Greedy approach

- Ideal size would be \( P = \sum_{i=1}^{n} \frac{s_i}{k} \)
- Greedy: walk along until what you have so far adds up to \( P \) then insert a divider
- Problem: it may not exact (or correct)
  - 100 200 400 500 900 700 600 800 600
  - sum is 4800 so size must be at least 1600.
  - Greedy? Best?