Advanced Programming Topics

Algorithm analysis and design - dynamic programming
Lecture 9
James Marshall

In lecture 1 we saw an efficient solution to computing the Fibonacci numbers, and an inefficient solution

But what made the inefficient solution inefficient?

Trace: `f 8 =`

And the efficient solution efficient?

N.B. this change turned an exponential-time algorithm into a polynomial-time one

Dynamic Programming (chapter 15 in CLRS)

Dynamic Programming is a technique for efficient solution of problems having:

1. (*Optimal) substructure —*

   the (optimal) solution to a problem contains the (optimal) solutions to subproblems (‘*optimal*’ refers to the frequent usage of dynamic programming in optimisation problems (next lecture))

2. *Overlapping independent subproblems —*

   subproblems form part of the solution to multiple other problems (*cf.* ‘regular’ divide-and-conquer problems in which each divide step generates new and non-overlapping subproblems)

Subproblem Graphs

Like *recursion trees*, but *subproblem graphs* show the relationships between all the *distinct* subproblems (vertices), with a directed edge from problem *x* to problem *y* showing that a solution to problem *x* requires a solution to problem *y*

Example: Subproblem Graph for Fibonacci Numbers

Identifying and Implementing Dynamic Programming Algorithms in Haskell

To identify that a dynamic programming algorithm may solve a problem, and then implement one:

1. Determine whether the description of the solution to a problem shows that the problem exhibits (optimal) substructure (see above) — *e.g.*

"if we work out that the quickest train route from London to Sheffield includes a stop in Derby, we have also worked out the quickest train route from London to Derby"

2. Determine whether the subproblems can be solved *independently* (does solving one subproblem affect how we solve a different subproblem? Often it doesn't, but sometimes it does…) — *e.g.*

"*working out the optimal route from London to Birmingham doesn't change how we work out the optimal route from London to Nottingham"*

3. Determine whether solutions to subproblems are used in multiple different solutions (if not then simple recursion will suffice) — *e.g.*

"in working out the quickest train route from London to Sheffield we evaluate multiple routes that all require us to know the quickest route from London to Derby"

4. Implement using *top-down recursion with memoization*

    A. start by asking for the answer to the original question (e.g. the 100th Fibonacci number), defined recursively in terms of the answers to smaller questions - hence *top-down*

    B. when a value is computed, it is *memoized* (stored) in an efficient data structure

    C. when a value is required, it is searched for in the structure first; if not found then it is computed recursively

Many languages let you define functions as being automatically memoized. Alternatively, use a data structure to store computed values, *e.g.* Data.Map in Haskell. For simple enough recursions (e.g. the Fibonacci example of lecture 1) it may be possible to keep track of a small number of memoized values at each step in the recursion