Introduction to Algorithms and Data Structures

Lecture 18: Introduction to Dynamic Programming

Mary Cryan

School of Informatics
University of Edinburgh

Divide and Conquer

The Divide and Conquer technique is when we design an algorithm to solve a problem by taking an instance (or input) I (of size n), then

1. Doing some preprocessing with I to construct some number of smaller sub-problems on smaller instances $I_1, \ldots, I_k$;
2. Making k recursive calls to compute the answer for the sub-problems;
3. Take the answers from 2. and do some computation to get the overall answer for the original input I.

Details (of the number of subproblems k, how to combine answers etc) will vary from problem to problem.

In some cases, Divide-and-Conquer can directly give an efficient (polynomial-time) algorithm - for example Mergesort, Quicksort.

Master theorem often features in the analysis.

But the recursive method is not always efficient.

Fibonacci numbers - a toy example

The Fibonacci numbers are defined as

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{(for } n \geq 2\text{)}. \end{aligned}$$

There is an immediate recursive algorithm:

Algorithm Rec-Fib(n)

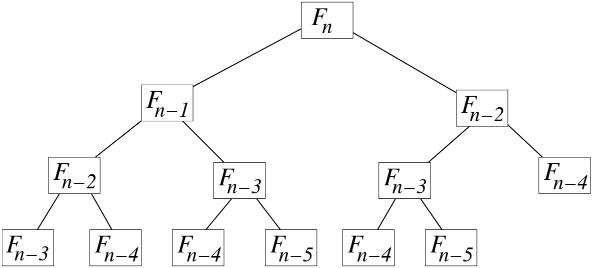
if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
else
    return $\textsc{Rec-Fib}(n-1) + \textsc{Rec-Fib}(n-2)$

Ridiculously slow: **exponentially many** repeated computations of $\textsc{Rec-Fib}(j)$ for small values of j.

Fibonacci numbers (cont'd)

Why is the recursive solution so slow?
Running time $T(n)$ satisfies

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \geq F_n \sim (1.6)^n.$$



(The 1.6 comes from the golden ratio $\frac{1+\sqrt{5}}{2}$. It is a bit easier to prove $F_n \geq \frac{1}{2}(3/2)^n$ for $n \geq 8$.)

Fibonacci numbers (cont'd)

Dynamic Programming Approach

Algorithm Dyn-Fib(n)

```
F[0] = 0
F[1] = 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Build "from the bottom up".
We are "turning recursion upside down".

Running Time is $\Theta(n)$
Very fast in practice - just need an array (of linear size) to store the $\mathrm{F}(i)$ values (in fact don't even need that array ...)

Implementing in 

```
# The plain recursive implementation: SLOW!
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

# Dynamic programming implementation with a 1-dimensional array
def fibDP(n):
    F = [0]*(n+1)
    F[1] = 1
    # The range will be empty if n is 0 or 1
    for i in range(n-1):
        F[i+2] = F[i+1]+F[i]
    return F[n]
```

Test these on the value 44 (say) to see the difference.

Decorators in 

Can get the benefit of the Dynamic Programming via ad-hoc memoization.

```
# The plain recursive implementation:
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

def memoize(f):
    memo = {}
    def check(s):
        if s not in memo:
            memo[s]=f(s)
        return memo[s]
    return check

# now make 'memoize'  a Decorator for 'fib'
fib = memoize(fib)
```

The coin-changing problem (re-visited)

The coin changing problem is the problem, given an input value v ($v \in \mathbb{N}_0$) of calculating a collection of coins (of minimum cardinality) that will sum to v.

This is an *optimization* problem, as we want a solution with as few individual coins as possible (we want to *minimize* the number of coins handed back).

Want to do this for arbitrary systems of coin denominations.

The coin-changing problem

Given: A value $v \in \mathbb{N}_0$, plus a sequence of coin values $c_1, c_2, \ldots, c_k \in \mathbb{N}_0$ (these representing the denominations of the relevant system).

Output: A multiset S of coins whose values sum to v, whose cardinality (size) of S is the minimum possible for v in this coin system.

Return solution as an array S of length k with $S[i]$ being the number of coins of value c_{i+1} for this optimal solution, for each $0 \leq i \leq k-1$.

We saw in lecture 17 that the natural greedy heuristic is not guaranteed to return an optimum set of coins (at least, not for the general case).

We will now develop a recurrence for the (optimal) solution.

We assume some solution definitely exists
(assuming $c_1 = 1$ is enough to ensure this)

The coin-changing problem

Let $C(v)$ denote the number of coins in an optimal solution for value $v \in \mathbb{N}_0$ (with respect to denominations $c_1, c_2, \ldots, c_k \in \mathbb{N}_0$).

Observation:

> *Suppose we have an optimal solution S for our given value v (with respect to our coin values $c_1, \ldots, c_k$), with optimal count $C(v)$.*
>
> *Then there is some initial coin i (maybe, take the lowest one in the solution) which contributes to this solution.*
>
> *Then $C(v) = 1 + C(v - c_i)$ for this coin.*

We will not know which coin c_i is definitely in the optimal solution.

However we can write

$$C(v) = \begin{cases} 1 & v = c_i \text{ for some } 1 \leq i \leq k \\ 1 + \min\{C(v - c_i) : 1 \leq i \leq k, c_i < v\} & \text{otherwise} \end{cases}$$

coin-changing: the algorithm

The recurrence helps me describe the solution (for v) in terms of other values, but that only helps if I already know the value of $C(v - c_i)$ for the various c_i. Would need to have precomputed those.

Solution:

- We will expand our Objective to computing $C(w)$ for every w from 1 to v.
- We will have an array C of length $v + 1$, and $C[w]$ will be computed as the "minimum number of coins to make w" for each w.
- We will compute the solution for small values of w first.
- It will help to have an extra array P to store the "coin used to get the best answer" for each w (to know how to reconstruct).
- At the end we will also use the arrays C and P to build the list of coin values for v (smaller array S of length k).

coin-changing by dynamic programming: example

Consider the case of coin values $1, 5, 7$, and the change-value $v = 18$.

Dynamic programming algorithm

Algorithm Dyn-Coins$(v; c_1, \ldots, c_k)$

initialise array c of length k to hold the c_i values
initialise array S of length k (to 0s)
initialise arrays C, P of length $v+1$ (to ∞)
$C[0] \leftarrow 0,\ C[1] \leftarrow 1,\ P[1] \leftarrow 0$ //Assume $c_1 = c[0] = 1$
for $w \leftarrow 2$ **to** v //We work "bottom-up"
    for $i = 0$ **to** $k-1$ //We try all coin values
        if $(c[i] \leq w)$ **and** $(C[w - c[i]] + 1 < C[w])$
            $C[w] \leftarrow 1 + C[w - c[i]]$
            $P[w] \leftarrow i$
while $v > 0$ //Now we work back to build S
    $i \leftarrow P[v]$
    $S[i] \leftarrow S[i] + 1;\ v \leftarrow v - c[i]$
return $C[v]$ "is the number of coins. The solution is in array S".

Other options?

Recursive implementation:

- A straightforward recursive implementation will show repeated subproblems, as in the case of Fibonacci (though it is less immediate)
- Even with some simple optimizations (like putting an order on considering the "next coin"), still we will get repetitions.
- So there is redundancy in a näive implementation of the recurrence on slide 10.

A "greedy" algorithm:

- The greedy algorithm always chooses the coin of max-value (less than remaining value)
- A very natural heuristic which will work on many coin systems.
- This is *not* guaranteed to be optimal for all systems - try the system with coin values $1, 5, 7$ for the value $v = 18$

Dynamic Programming principles (in general)

4 (related) features we need in order to design an (efficient) Dynamic Programming algorithm:

(dp1) Need is to see that computing the optimum solution for our original instance can be achieved by finding solutions to (smaller) problems of the same type, and combining them.

(Sometimes we will have need to generalise the way we define the problem for this; eg, with coin changing, we think about computing the best solution for every value from 1 to v).

(dp2) Closely related to (dp1), we need the solution to an instance of the problem to be expressible in terms of a recurrence, where the right-hand side contains one or more recursive calls for smaller instances of the same problem.

(for coin-changing, this was the recurrence on slide 11)

Dynamic Programming principles (in general)

(dp3) We need to be able to organise storage for the results for all possible subproblems (identified in dp1/dp2) which will be solved.

It should be possible to store the subproblem results in an table with meaningful indexing - hopefully, 1-dimensional or 2-dimensional. (there are some problems which rely-on 3-dimensional (or greater) tables. That's ok, as long as the space is "polynomially bounded").

(for coin-changing, we use 1-dim arrays of length w and k)

(dp4) We need an algorithm to control the *order* in which subproblems are solved (and results stored in the appropriate cell of the table). This must be done so that all of the subproblems appearing on the right-hand side of the recurrence must be computed and in-the-table *in advance* of computing the left-hand side.

(for coin-changing, all of the sub-problems on the rhs of the recurrence have $w < v$, so the solution is already pre-computed)

Reading and Working

Reading: Neither the [CLRS] nor the [Roughgarden] textbook cover the same Dynamic programming problems as us.

- Section 16.4 of "Algorithms Illuminated" by Tom Roughgarden discusses the Principles of Dynamic programming in a similar way to us.
- Section 6.2 of Kleinberg+Tardos has a similar discussion of general Dynamic Programming principles.

- Dynamic Programming Algorithm for coin-changing is due to J.W. Wright, "The Change Making Problem", *Journal of the ACM*, 1975. `https://dl.acm.org/doi/10.1145/321864.321874`