CIS 575. Introduction to Algorithm Analysis
Material for March 18, 2024

Dynamic Programming: Why and How

The design principle covered in this set of notes, Dynamic Programming, is the topic of *Cormen*'s Chapter 14.

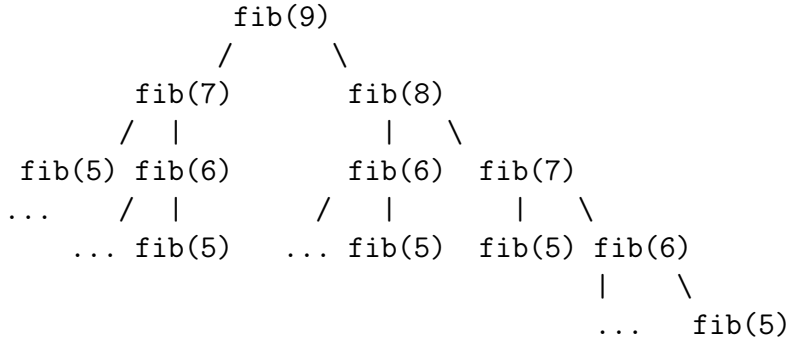
1 Problem: Duplicate Computation

Many problems can be solved by algorithms which, when executed naively, do the same computations over and over and hence may become infeasible as input grows larger.

For example, the *fibonacci* function can be computed by a recursive function `fib` where `fib`(n) is defined as

if $n = 0$ **or** $n = 1$
  return 1
else
  return `fib`$(n-2)$ + `fib`$(n-1)$

Computing say `fib`(9) results in the *call tree* partially depicted below:



and we see that `fib(7)` is called twice; `fib(6)` is called 3 times; `fib(5)` is called 5 times — in fact, it is not hard to see that `fib`(k) is called `fib`$(9-k)$ times! Thus the running time of `fib`(n) is *exponential* in n.

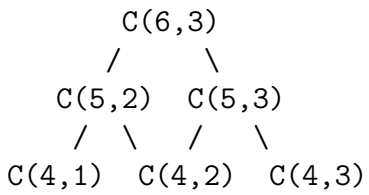
As another example, consider the function $C(n,k)$ that denotes the number of ways we can pick k elements from n elements. (Often one writes $\binom{n}{k}$ rather than $C(n,k)$.) We know that

$$C(n,k) = \frac{n!}{k!\ (n-k)!}$$

but one can also, when $0 < k < n$, write a recursive equation for C:

$$C(n,k) = C(n-1,k-1) + C(n-1,k) \tag{1}$$

which reflects that to pick k elements out of n, one can either pick the nth element together with $k-1$ out of the first $n-1$, or omit the nth element but then pick k out of the first $n-1$. Computing say $C(6,3)$ results in the call tree partially depicted below



where $C(4,2)$ must be computed twice, and it is not hard to see that as we compute $C(n,k)$ for large n and k there will be a lot of entries that are computed over and over.

2 Solution: Dynamic Programming

The examples mentioned above have well-known bottom-up solutions. We have already seen how to compute the fibonacci function by an iterative algorithm:

$i, j \leftarrow 1$
for $k \leftarrow 1$ **to** $n-1$
  $i, j \leftarrow j, i+j$
return j

To compute $C(n,k)$, we can construct *Pascal's triangle* bottom-up. We know that $C(n,0) = C(n,n) = 1$ for all n; by Equation (1), each remaining entry is the sum of the two entries just below. The first entries are

```
1   6  15  20   15  6   1   C(6,0..6)
  1   5  10   10  5   1     C(5,0..5)
    1   4   6   4   1       C(4,0..4)
      1   3   3   1         C(3,0..3)
        1   2   1           C(2,0..2)
          1   1             C(1,0..1)
            1                C(0,0)
```

and we see that $C(6,3) = 20$.
We can generalize the above into a **general recipe**:

- assume our goal is to solve a problem P_0
- assume we have a number of subproblems $P_1 \ldots P_m$
- consider a graph where:
  - the nodes are the problems $P_0 \ldots P_m$;
  - there is an edge from P_i to P_j if the solution to P_j is used directly in the solution of P_i

We will surely expect the graph to be acyclic, and may then solve the problem(s) as follows:

1. first solve those problems from which there are no edges
2. next solve those problems that have edges only to problems solved in step 1
3. next solve those problems that have edges only to problems solved in step 1 or step 2
4. and so on, so on.

Example: if P_0 is to find the value of $C(6,3)$, then the subproblems will be to find the values of $C(n,k)$ when $0 \leq k \leq n \leq 6$, and if $0 < k < n$ there will be an edge from $C(n,k)$ to $C(n-1,k-1)$ and to $C(n-1,k)$.
We have outlined the principle of **Dynamic Programming**. We shall see several instances in the subsequent notes. Observe that

- if many subproblems can be reached from P_0 through many paths then we can expect dynamic programming to be much faster than "top-down" evaluation
- dynamic programming may compute entries not needed, for example it will compute $C(5,1)$ towards computing $C(6,3)$.

Ideally, each subproblem that is needed should be solved once, and each subproblem that is not needed should not be solved at all! This can be accomplished by top-down evaluation with *memoization*, but that involves some overhead so in most situations one will prefer dynamic programming.