CS 341 — Lecture 8

Bartosz Antczak *Instructor: Bin Ma* *October 3, 2017*

8.1 Dynamic Programming

So far we've focused on a few algorithm design techniques: reduction, divide and conquer, and the greedy algorithm. This week we'll focus on another technique: *dynamic programming*.

Example 8.1.1. *Fibonacci numbers*

Let's create an algorithm that returns the nth Fibonacci number. A basic algorithm would look like this:

$F(1) \leftarrow 1, F(2) \leftarrow 1$
if $n \leq 2$ **then**
  return 1
else
  return fibonacci$(n-1)$ + fibonacci$(n-2)$

Note that this algorithm has a run time of $T(n) = T(n-1) + T(n-2) + 1 \in O(2^n)$ — bad! Let's see if we can get more efficient:

$F(1) \leftarrow 1, F(2) \leftarrow 1, F(i) = -1$ for $i > 2$
if $F(n) < 0$ **then**
  $F(n) \leftarrow F(n-1) + F(n-2)$
return $F(n)$

Notice every index is accessed at most once, which means that this algorithm has a run time of $O(n)$.
Now let's try something slightly different. Let's try a bottom-up approach to avoid having to compute $f(n-1)$ and $f(n-2)$:

$F(1) \leftarrow 1, F(2) \leftarrow 1$
for $i = 3$ **to** n **do**
  $F(i) \leftarrow F(i-1) + F(i-2)$
return $f(n)$

This last example took a *dynamic programming* approach. Let's move on to more real examples now.

8.1.1 Weighted Interval Scheduling

Suppose we have times intervals that are requesting access to a meeting room. These intervals have a priority (weight) w_i. Given n intervals, return a set S of disjoint intervals that maximize $\sum_{i \in S} w_i$. We can also assume our intervals (s_i, f_i) are sorted by finish time f_i.
Let $D(k)$ be the weight of the optimal scheduling of the first k intervals. Now, we need to determine the recurrence relation that computes $D(k)$. Observe that $D(k)$ may or may not contain the interval (s_k, f_k) (i.e., the last interval out of our k intervals), so we try both cases:

- **CASE 1:** (s_k, f_k) *is in the optimal solution.*
  In this case, let ℓ_k be the largest integer such that $f_{\ell_k} \leq s_k$ (i.e., the last interval that finishes earlier than s_k). Then the optimal solution consists of the optimal solution of $1, \ldots, \ell_k$ and (s_k, f_k). In other words, $D(k) = D(\ell_k) + w_k$

- **CASE 2:** (s_k, f_k) *is not in the optimal solution.*
  In this case, the optimal solution of the first k intervals is equal to the optimal solution of the first $k-1$ intervals. In other words, $D(k) = D(k-1)$.

The larger of these two cases provides the optimal solution:

$$D(k) = \max\{D(\ell_k) + w_k, D(k-1)\}$$

Now we have a recurrence relation for our algorithm. With it, we can solve our problem:

1. Sort the intervals according to finish time
2. Compute ℓ_k for each k
3. Set $D(0) = 0$

And our algorithm will look like this:

---

for $k = 1$ **to** n **do**
  $D(k) \leftarrow \max\{D(\ell_k) + w_k, D(k-1)\}$
print $D(k)$

Steps 1 and 2 will take $O(n \log n)$ time (for step 2, you perform binary search for every item), and step 4 will take $O(n)$ time, so our total run time is $O(n \log n)$.
Notice that $D(n)$ returns the max weight, but it doesn't output the actual *schedule*! For that, we perform *backtracking*.

Backtracking Algorithm

$k \leftarrow n$
while $k \geq 0$ **do**
  if $D(k) == D(\ell_k) + w_k$ **then**
    print k
    $k \leftarrow \ell_k$
  else
    $k \leftarrow k - 1$

8.1.2 Change Making

In this problem, we have coins with face value $d_1, d_2, \ldots, d_k$. Given a total sum of n, we want to determine the number of coins c_i that we should use: $\sum_{i=1}^{k} c_i \cdot d_i = n$ and we want to minimize $\sum_{i=1}^{k} c_i$.

Let's define $D(i)$ as the minimum number of coins adding up to n. If $n > 0$, then there exists at least one such coin d_i that is used in the optimal solution. We do not know i, so we try all of them:

$$D(n) = 1 + \min_{1 \leq i \leq k} D(n - d_i)$$

The algorithm using dynamic programming would then look like this:

$D(0) \leftarrow 0$
for $j = 1$ **to** n **do**
  $D(n) = 1 + \min_{1 \leq i \leq k} D(n - d_i)$
print $D(n)$
