Notes, 7(a)

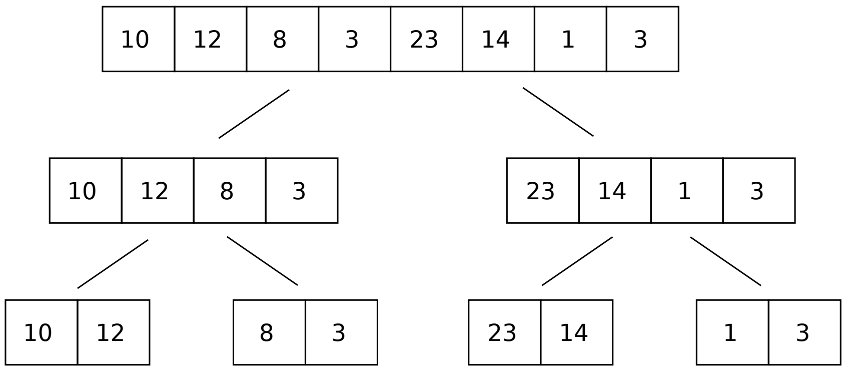
ECE 606

Design Strategy IV: Dynamic Programming

Another strategy that exploits optimal substructure. A bit like divide-n-conquer, but with a crucial difference: overlapping subproblems.

E.g., we can think of MERGE-SORT as realizing a recurrence.

$$\text{Sort}(A[1,\ldots,n]) = \text{Merge}(\text{Sort}(A[1,\ldots,n/2]), \text{Sort}(A[n/2+1,\ldots,n]))$$



Subproblems in MERGE-SORT do not "overlap" — i.e., no subproblem is the same as another.

Now think of change-making problem. Given as input: (i) $a \in \mathbb{Z}^+$, and, (ii) coin values $\langle c_0, \ldots, c_{k-1} \rangle$ where $c_0 = 1$, each $c_i < c_{i+1}$, each $c_i \in \mathbb{Z}^+$, and infinitely many coins of each value, what is the minimum number of coins that makes up a?

This problem demonstrates the following kind of optimal substructure. Suppose the optimal answer, for some input, is m. Think of us handing out the first coin, then the second, ..., and finally the mth coin. I ask: which of the k coin-types is the mth coin?

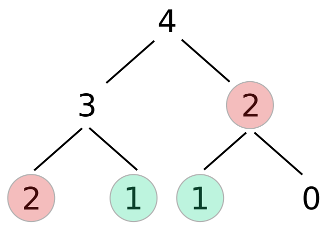
The answer is: the m^{th} coin is c_i if and only if (i) $a - c_i \geq 0$, and, (ii) the minimum number of coins that make up $a - c_i$ is $m - 1$. Thus, we can write the following recurrence for $M[x]$, the minimum number of coins for amount x. Our final solution then is $M[a]$.

$$M[x] = \begin{cases} \infty & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 + \min\limits_{i \in \{0,\ldots,k-1\}} \{M[x - c_i]\} & \text{otherwise} \end{cases}$$

The above recurrence suggests the following natural, recursive algorithm.

```
MinCoins($x, c_0, \ldots, c_{k-1}$)
    if $x < 0$ then return $\infty$
    if $x = 0$ then return $0$
    $min \leftarrow \infty$
    foreach $i$ from $0$ to $k-1$ do
        $tmp \leftarrow$ MinCoins($x - c_i, c_0, \ldots, c_{k-1}$)
        if $tmp < min$ then $min \leftarrow tmp$
    return $1 + min$
```

To demonstrate the phenomenon of overlapping subproblems, consider the simple example $a = 4, k = 2, c_0 = 1, c_1 = 2$. We now draw a few levels of the recursion tree.



The recursion tree shows that to compute $M[4]$, we need to determine each of $M[4 - c_0] = M[3]$ and $M[4 - c_1] = M[2]$ and pick the smaller. To compute $M[3]$, in turn, we compare $M[2]$ and $M[1]$. We represent each subproblem by the amount for which we seek to make change. The overlapping subproblems, i.e., the subproblems which are the same, are shown in circles of the same colour.

Keep in mind that we have one thread of execution only when we run the recursive algorithm. That is, when we invoke MinCoins$(4, \ldots)$, we would first fully explore the subtree rooted at the left child 3 of the root 4 before we explore the subtree rooted at the right child 2 of the root 4. Thus, we will have "forgotten" the intermediate results for MinCoins$(2, \ldots)$ and MinCoins$(1, \ldots)$.

How bad is this issue? Let's continue with this example in which $k = 2, c_0 = 1, c_1 = 2$. Then, our recursion tree for an amount a is a binary tree. The binary tree is full upto depth at least $\lfloor a/2 \rfloor$. Thus, the total number of nodes in the recursion tree up to that depth is: $2^0 + 2^1 + \ldots + 2^{\lfloor a/2 \rfloor} > \left(\sqrt{2}\right)^a$, which is exponential in the input a.

Natural solution this overlapping subproblems issue: remember the intermediate results. There are a of them: $M[0], \ldots, M[a-1]$. Instead of recursing unnecessarily, simply check if we already have an intermediate result and if yes, return it. For example, when we visit the right child 2 of the root 4 above, we will immediately return $M[2]$ from our remembered values, and not recurse any further.

This strategy is called *memoization*.

```
MinCoins-Memoize-EntryPoint($a, c_0, \ldots, c_{k-1}$)
1  $M \leftarrow$ new array $[0, \ldots, a]$
2  foreach $i$ from $0$ to $a$ do  $M[i] \leftarrow -1$
3  MinCoins-Memoize($M, a, c_0, \ldots, c_{k-1}$)
4  return $M[a]$

MinCoins-Memoize($M, x, c_0, \ldots, c_{k-1}$)
5  if $M[x] \geq 0$ then return
6  if $x = 0$ then $M[x] \leftarrow 0$
7  else
8      $M[x] \leftarrow \infty$
9      foreach $i$ from $0$ to $k-1$ do
10         if $x \geq c_i$ then
11             MinCoins-Memoize($M, x - c_i, c_0, \ldots, c_{k-1}$)
12             if $M[x - c_i] + 1 < M[x]$ then $M[x] \leftarrow M[x - c_i] + 1$
```

There is one more improvement we can make, which is not use recursion at all. With that improvement in place, we have exactly what we call *dynamic programming*.

Our final improvement is based on observing that rather than going "top-down," i.e., recursing on $M[a]$, then $M[a - c_i]$, etc., we can go "bottom-up" — compute the $M[\cdot]$ for smaller values first and then the bigger values. This can be done iteratively, and does not need recursion at all.

This, as we say in the previous page, is exactly dynamic programming.

```
MinCoins-DP($a, c_0, \ldots, c_{k-1}$)
1 $M \leftarrow$ new array $[0, \ldots, a]$
2 foreach $i$ from $0$ to $a$ do
3     if $i = 0$ then $M[i] \leftarrow 0$
4     else
5         $M[i] \leftarrow \infty$
6         foreach $j$ from $0$ to $k - 1$ do
7             if $i \geq c_j$ then
8                 if $M[i] > 1 + M[i - c_j]$ then $M[i] \leftarrow 1 + M[i - c_j]$
```

It is easy to characterize the time-efficiency of MinCoins-DP: $O(ak)$. This is the same as for MinCoins-Memoize-EntryPoint.