

Dynamic Programming exercises

Goal: Understand how to reconstruct solution for rod cut, apply techniques for Fibonacci

```
BOTTOM-UP-CUT-ROD( $p, n$ )
  let  $r[0..n]$  be a new array
   $r[0] = 0$ 
  for  $j = 1$  to  $n$ 
     $q = -\infty$ 
    for  $i = 1$  to  $j$ 
       $q = \max(q, p[i] + r[j - i])$ 
     $r[j] = q$ 
  return  $r[n]$ 
```

1) Modify the code above to save the first cut in the optimal solution into a new array $s[0..n]$, and then give a procedure that prints out the sequence of cuts for a rod of size n .

Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

```
EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )
  let  $r[0:n]$  and  $s[1:n]$  be new arrays
   $r[0] = 0$ 
  for  $j = 1$  to  $n$  // for increasing rod length  $j$ 
     $q = -\infty$ 
    for  $i = 1$  to  $j$  //  $i$  is the position of the first cut
      if  $q < p[i] + r[j - i]$ 
         $q = p[i] + r[j - i]$ 
         $s[j] = i$  // best cut location so far for length  $j$ 
     $r[j] = q$  // remember the solution value for length  $j$ 
  return  $r$  and  $s$ 
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION( $p, n$ )
  ( $r, s$ ) = EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )
  while  $n > 0$ 
    print  $s[n]$  // cut location for length  $n$ 
     $n = n - s[n]$  // length of the remainder of the rod
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$			1	2	3	2	2	6	1

A call to PRINT-CUT-ROD-SOLUTION($p, 8$) calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

2) Show that the recurrence for a naive recursive cut rod algorithm, $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$ along with base case $T(0) = 1$, has the solution $T(n) = 2^n$.

Use substitution method, aka proof by induction.

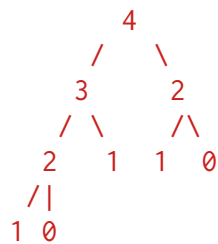
Check base case, then plug in and apply geometric series.

The Fibonacci sequence has the top-down recursive algorithm:

```
Fibonacci(n):  
  if n <= 1  
    return n  
  else  
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

3) Draw a tree that shows the subproblems for $n = 4$. Is this a good candidate for a dynamic programming improvement? (Bonus: count the number of nodes in the tree for arbitrary n and use this to analyze the runtime.)

Did on the board. Yes, repeated subproblems. However, not an optimization problems (not really DP)



4) Write memo-ized top-down and bottom up versions of the Fibonacci program.

```
Bottom-Up(n):  
  if n <= 1  
    return n  
  else  
    init f[0:n]  
    for i = 2:n  
      f[i] = f[i-1] + f[i-2]  
    return f[n]
```

```
Top-Down(n):  
  init f[0:n] = Nil // loop, O(n)  
  f[0] = 0 // base cases in outer  
  f[1] = 1 // can also put into Aux  
  return Aux(n, f)
```

```
Aux(n, f):  
  if f[n] is not Nil  
    return f[n]  
  else  
    f[n] = Aux(n-1, f) + Aux(n-2, f) // f passed in by ref, modified  
    return f[n]
```