Dynamic Programming

Chapter 15

Dynamic Programming

- We can use the divide-and-conquer technique to obtain efficient algorithms
  - But not always
- Divide and Conquer is best used when there are no overlapping subproblems
- Replace a function call with a table lookup!

Fibonacci Numbers

- Fibonacci numbers are defined by the following recurrence:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} \text{ if } n \geq 2 \\ 1 \quad \mathit{if} \quad n = 1 \\ 0 \quad \mathit{if} \quad n = 0 \end{cases}$$

n	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**…**
F_n	1	1	2	3	5	8	13	21	34	55	89	…

A Recursive Algorithm

```
int Fibonacci(int n)
  {
    if ( n <= 1 )
    {
      return 1;
    }
    else
    {
      return Fibonacci(n-1) +
         Fibonacci(n-2);
    }
  }
```

Why is this slow?

Can we do better?

What subproblems are solved twice?

Can we build the recursion tree?

Dynamic Programming

```
int  fibonacci(int n)
{
   int table[          ];




}
```

Dynamic Programming

- Not really dynamic, not really programming
- Name is used for historical reasons
- "Mathematical Programming" - a synonym for optimization.

Space vs Time

Memoization

Dynamic Programming

- Solves each subproblem once and saves the answer in a table

- Used to solve optimization problems
  - Many possible solutions
  - Wish to find a solution with the optimal value

Four Steps for Dynamic Programming

- Characterize
- Recursively
- Compute
- Construct

Rod Cutting

- A company buys long steel rods and cuts them into shorter rods, which it then sells
  - Each cut is free

- What cuts lead to the most money?

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Example

- How many ways can you cut a rod?
- What are the possible ways of cutting a rod of length 4 ($n = 4$)?
- What is the best way?

Initial Optimal Revenues

- Optimal revenues r_i, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Optimal Revenues

- We can determine the optimal revenue r_n by taking the maximum of:
  - p_n: price by not cutting
  - $r_1 + r_{n-1}$: maximum revenue for a rod of length 1 and a rod of length n-1
  - $r_2 + r_{n-2}$: maximum revenue for a rod of length 2 and a rod of length n-2 ….
  - $r_{n-1} + r_1$
  - $\mathbf{r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_{1)}}$

Simplifying

- Every optimal solution has a leftmost cut.
  - produces i and n-i pieces. (i could be zero)
  - Need to divide only the remainder, not the first piece.
  - Leaves only one subproblem to solve, rather than two subproblems.
  - Say that the solution with no cuts has first piece size i = n with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.

$$r_n = \max_{1 \le i \le n} \left(p_i + r_{n-i} \right)$$

Recursive Top-Down Solution

CUT-ROD(p, n)

if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ **to** n
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return q

- Is it correct?
- Is it efficient?

Dynamic-Programming Solution

- "Store, don't recompute"
- Can turn an exponential-time solution to a polynomial-time solution
- Two approaches:
  - Top-down with memoization
  - Bottom up

Top-Down with Memoization

- Solve recursively, but store each result in a table

- Always check the table
  - If there, use it
  - Otherwise, compute it and store in table
  - each subproblem is computed exactly once

Memoized Cut-Rod

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

Bottom-Up

- Sort the subproblems by size and solve the smaller ones first

BOTTOM-UP-CUT-ROD(p, n)

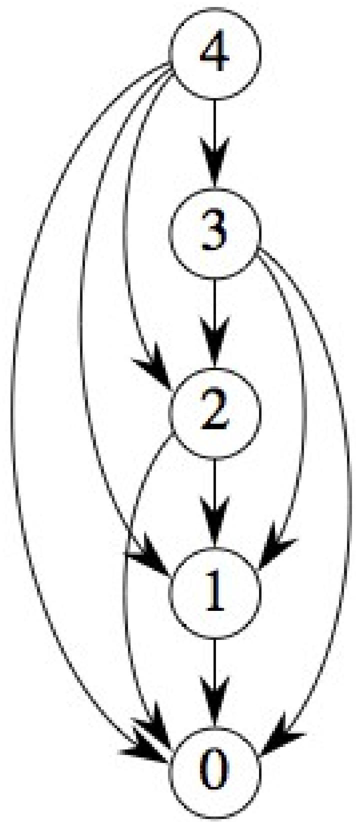
let $r[0\,..\,n]$ be a new array
$r[0] = 0$
for $j = 1$ **to** n
    $q = -\infty$
    for $i = 1$ **to** j
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$

Subproblem graphs

- Directed Graph:
  - One vertex for each distinct subproblem
  - Has a directed edge (x, y) if computing an optimal solution to subproblem x directly requires knowing an optimal solution to subproblem y

Subproblem Graph for Rod-Cutting

- When n = 4:



Reconstructing a Solution

- We have only computed the value of an optimal solution
  - i.e. When $n = 4$, $r_n = 10$
- We still don't know how to cut up the rod!

Rod-Cutting

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
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]
    n = n − s[n]
```

Example

- PRINT-CUT-ROD-SOLUTION(p, 369)

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