Dynamic Programming

Chapter 15

Dynamic Programming

- We know that we can use the divide-and-conquer technique to obtain efficient algorithms
- Sometimes, the direct use of divide-and-conquer produces really bad and inefficient algorithms

Fibonacci Numbers

- Fibonacci numbers are defined by the following recurrence:

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

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

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

- What is the running time?

Finonacci

- Why is it so slow?

- Can we do better?
- Recursion is not always best!

Dynamic Programming

- Not really dynamic
- Not really programming
- Name is used for historical reasons
- It comes from the term “mathematical programming”, which is a synonym for optimization.

Dynamic Programming

- Dynamic programming improves inefficient recursive algorithms
- How?
  - Solves each subsubproblem 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 the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution, typically in a bottom-up fashion
- Construct an optimal solution from computed information

Rod Cutting

- A company buys long steel rods and cuts them into shorter rods, which it then sells
- Each cut is free
- The management wants to know the best way to cut up the rods to make 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

- Can cut up a rod in 2^{n-1} different ways
  - You can choose to cut or not cut after the first n-1 inches
- 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
  - …
  - $r_{n-1} + r_1$
- $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$

Optimal Substructure

- To solve a problem of size n, solve problem of smaller sizes. After making a cut, we have two subproblems. The optimal solution to the original problem incorporates optimal solutions to the subproblems.
- Example

Simplifying

- Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length n - i on the right
  - 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 $r_0 = 0$.

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

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

- Don’t solve same subproblems repeatedly
- “Store, don’t recompute”
  - time-memory trade-off
- 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
- To find the solution to a subproblem, first look in the table.
  - If there, use it
  - Otherwise, compute it and store in table

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 $r[n]$

Running Time

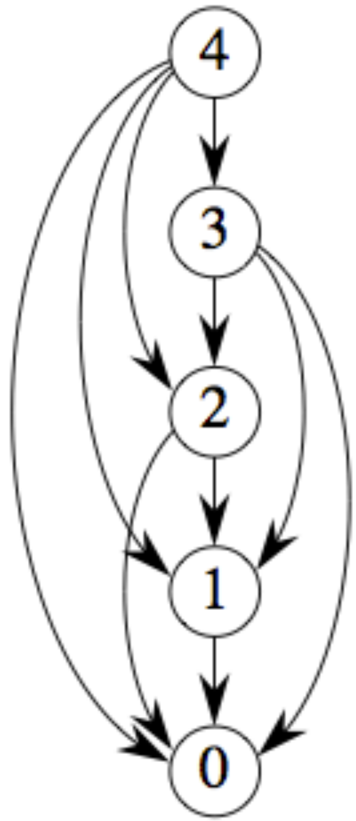
- What is the running time of the previous two algorithms?

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, 8)

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

Problem

- Do exercise 15.1-5 on page 370

Summary

- Divide and Conquer is best used when there are no overlapping subproblems
- Otherwise, use dynamic programming!