Dynamic Programming

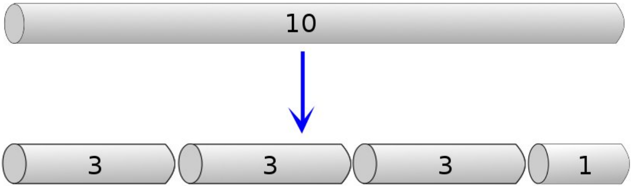
Dr Timothy Kimber

February 2018

Back To Solving Problems

The Rod Cutting Problem

- A business buys steel rods in a variety of lengths
- They will cut the rods into smaller pieces to sell on
- Each rod size has a different market value
- What is the maximum revenue $R(N)$ for a rod of length N?

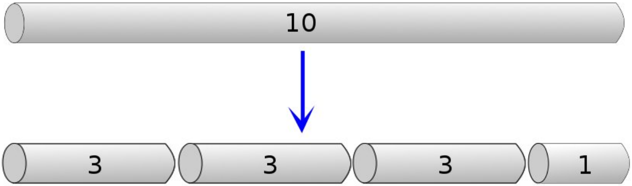


Is $p(3) + p(3) + p(3) + p(1) > p(4) + p(4) + p(2)$?

Instance of The Problem

If the selling prices for each size of rod up to 10 are

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30



Then the answer for $N = 10$ is 32 ($1 \times 6 + 4 \times 1$, or 2×5)

Rod Cutting

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30

Question

Given an array of prices $P = [P_1, \ldots, P_k]$ and an integer N between 1 and k, how can $R(N)$ be computed?

Design

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30

Possible outline:

- Choose some sizes $s = \langle s_1, \ldots, s_j \rangle$ that sum to N
- (Values can repeat in s)
- Compute $R_s = P[s_1] + \cdots + P[s_j]$
- For all possible s
- Update current best $R(N)$ as you go

Design

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30

Question

How do you generate (only) sequences s that sum to N?

At this point it will be useful to think about reducing the problem to solving one or more smaller subproblems.

Design

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30

Choosing sizes:

- Pick an s_1
- Then s is s_1 followed by $\langle s_2, \ldots \rangle$ that sum to $N - s_1$

Can now see the structure of the problem:

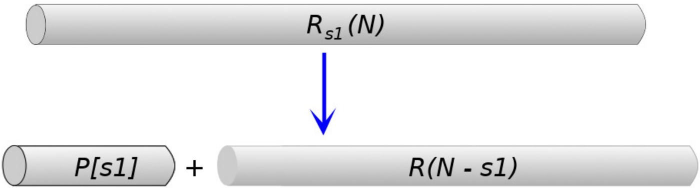
- For each possible s_1
- Find all solutions for $N - s_1$, and combine with s_1
- Base case: only sequence that sums to 0 is $\langle \rangle$

Design

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30

Re-evaluate overall design:

- Pick an s_1
- Max revenue using s_1 is $P[s_1] + R(N - s_1)$
- $R(N - s_1)$ is overall solution for rod length $(N - s_1)$
- One option per value for s_1



A Simple Recursive Solution

SimpleRodCut(Input: N, $P = [P_1, \ldots, P_k]$)

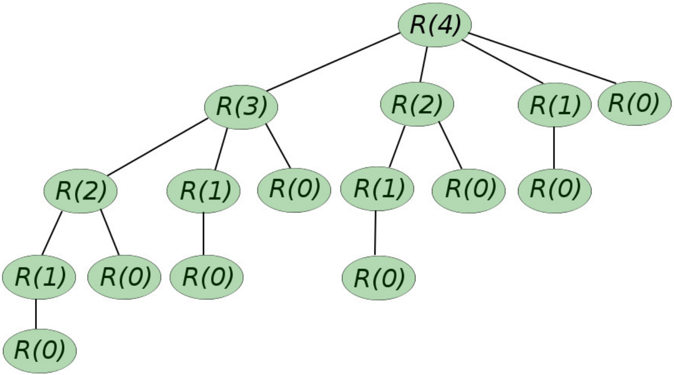
```
if N == 0
  return 0
else
  for i = 1 to N
    choices[i] = P[i] + SimpleRodCut(N-i, P)
    return max(choices)
```

- *choices* collects total for each s_1
- *max* finds the maximum of the choices

How does this run?

Simple Rod Cut — Reflection

WOW that was sloooooowww.



Question

Solving $R(0)$ takes $\Theta(1)$ time. What about $R(N)$?

Time for Simple Solution

The time taken by SimpleRodCut is

$$T(0) = \Theta(1)$$
$$T(N) = 2T(N-1) + \Theta(1), \text{ for } N > 0$$

or

$$T(N) = 2^{N-1}T(0) + \Theta(1)$$

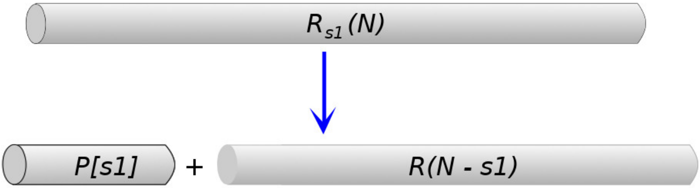
so $T(N) = \Theta(2^N)$.

- The running time grows exponentially.
- This is not a practical solution.

Divide & Conquer?

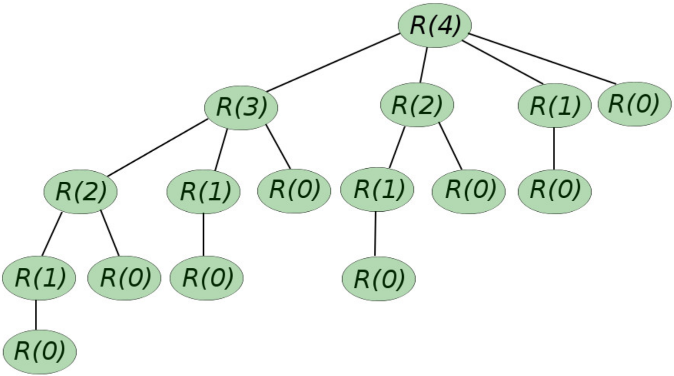
- Can we divide the problem? (and conquer?)

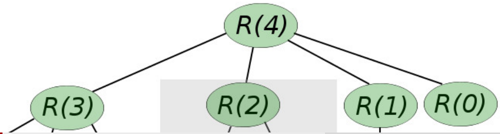
size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30



New Strategy

What is there that we can take advantage of?





Dynamic Programming

Dynamic Programming makes a space–time tradeoff

- Do not want to recompute the answer to $R(i)$ every time
- Compute it once and save the answer in a table
- Check the table before computing each subproblem

This is called memoisation (we are making a note for later)

MemoisedRodCut(Input: N, $P = [P_1, \ldots, P_k]$)

```
for i = 0 to N
  R[i] = 0
return MemoiseAux(N, P, R)
```

- R is the table to be filled in

Memoisation

MemoiseAux(Input: N, $P = [P_1, \ldots, P_k]$, $R = [R_0, \ldots, R_{N'}]$)

```
if N == 0
  return 0
if R[N] > 0
  return R[N]
for i = 1 to N
  choices[i] = P[i] + MemoiseAux(N-i, P, R)
R[N] = max(choices)
return R[N]
```

- If $R[N]$ was already computed ($R[N] > 0$) it is returned immediately
- Otherwise we compute it, save it, and then return it
- Also called Top Down (set out to solve the biggest problem)

The 'Bottom Up' Method

We know which problems depend on which others

- so we can just complete the table in order
- this will be more efficient than recursion

BottomUpRodCut(Input: N, $P = [P_1, \ldots, P_k]$)

```
R[0] = 0
for i = 1 to N
  choices = [0,...,0]
  for j = 1 to i
    choices[j] = P[j] + R[i-j]
  R[i] = max(choices)
return R[N]
```

- What is the running time?

Dynamic Programming

Dynamic programming can be applied to a problem if

- The problem has optimal substructure
- The problem has overlapping subproblems

A problem has optimal substructure if

- the problem can be decomposed into subproblems
- an optimal solution uses optimal solutions to the subproblems

In rod cutting the optimal solution for N was one of

- $P[i] + R[N-i]$, where $1 \leq i < N$

and each $R[N-i]$ was an optimal solution for $N-i$.

Optimal Substructure

Problems may appear to have optimal substructure when they do not

Problem *(Unweighted Shortest Path)*

Input: graph $G = (V, E)$.

Input: vertices $u, v \in V$.

Output: the simple path from u to v containing the fewest edges

Problem *(Unweighted Longest Path)*

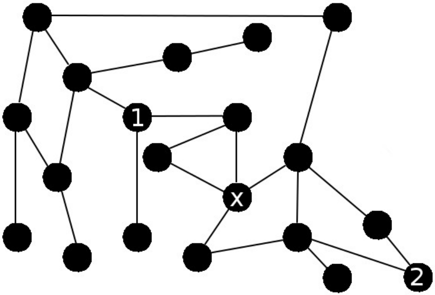
Input: graph $G = (V, E)$.

Input: vertices $u, v \in V$.

Output: the simple path from u to v containing the most edges

Optimal Substructure

A shortest path is composed of optimal solutions to subproblems

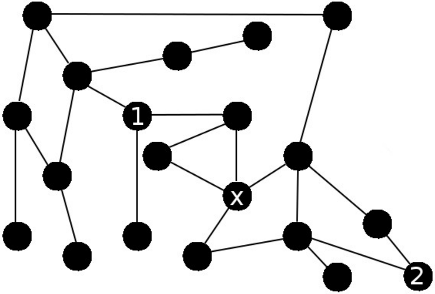


The shortest path from 1 to 2 (via x) is

- shortest path from 1 to x
- plus the shortest path from x to 2

Optimal Substructure

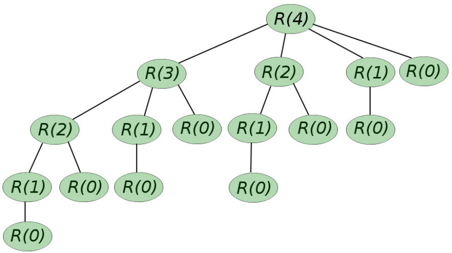
How about a longest path?



- Independent subproblem solutions do not make an optimal solution
- In an optimal solution the subproblems will interfere

Overlapping Subproblems

The second property we need when applying dynamic programming is overlapping subproblems



- The same problems are generated over and over
- The subproblems must still be independent
- The set of all subproblems is the subproblem space
- The smaller the subproblem space the quicker the (dynamic) algorithm