Dynamic Programming I - Rod Cutting

1 Rod cutting problem

You work for a company that buys steel rods and cuts them into shorter rods, which it then sells.

You are asked to determine the best way to cut the rods to maximise profit.

You know, for $i = 1, ..., n$, the price p_i in euros that the company charges for a rod of length i centimetres (Rod lengths are always an integer number of centimetres)

E.g.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

1.1 More formally

Definition: Rod cutting problem

Given a rod of length n and a table of prices p_i for $i = 1, ..., n$, determine the maximum revenue r_n obtained by cutting up the rod and selling the pieces

E.g. with $n = 4$ and $p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9$

By brute force: list all possible ways of cutting up $n = 4$

Partition	4	1 + 3	2 + 2	1 + 1 + 2	1 + 1 + 1 + 1
Revenue	9	9	10	7	4

Maximum revenue if we cut two rods of length 2, and we have $r_4 = 10$

1.2 A note on integer partitions

Definition: Integer partition

An integer partition of a positive integer n is a list of positive integers $\langle a_1, ..., a_k \rangle$ such that $a_1 \leqslant a_2 \leqslant ... \leqslant a_k$ and

$$\sum_{i=1}^{k} a_i = n$$

Let $p(n)$ denote the number of integer partitions of n, then

$$p(1) = 2, p(2) = 2, p(3) = 3, p(4) = 5...$$

Hardy and Ramanujan proved that

$$p(n) \sim \frac{1}{4n\sqrt{3}} \exp(\pi \sqrt{\frac{2n}{3}})$$

Therefore, brute force has an exponential running time

2 Dynamic Programming

2.1 Optimal substructure

More generally, we have for all $n \geqslant 1$

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$$

Idea: p_n corresponds to no cuts and $p_i + r_{n-i}$ to cutting into a rod of length i and another rod of length $n - i$

To solve the original problem of size n, we solve independent smaller problems of the same type, but of smaller sizes

Definition: Optimal substructure

Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently

2.2 Recursive top-down implementation

Input: an array $p[1..n]$ of prices and an integer n

Output: The maximum revenue possible for a rod of length n

Listing 1: ROD(p,n)

```
1 if n==0 then
2     return 0
3 q ← -1
4 for i ← 1 to n do
5     q=max{q,p[i]+ROD(p,n-i)}
6 return q
```

The running time for this is exponential again (there are 2^n calls of the form ROD(p,k) for $n \geqslant 1$)

2.3 Using dynamic programming for optimal rod cutting

The naive recursion solution is slow because it repeatedly solves the same subproblems

We arrange the subproblems in order to solve them only one, saving its solution. If we need that solution, we just look it up: we needn't recompute it

Dynamic programming thus uses additional memory to save computation: it is an example of time-memory trade-off

A dynamic programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial and we can solve each subproblem in polynomial time

2.4 Overlapping subproblems

In the naive top-down method, we keep on solving the same subproblems: we say that these subproblems overlap. Memoization prevents that. Careful

- Two subproblems are independent if they do not share resources
- Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems

3 Top-down vs Bottom-up

First approach: top-down with memoization

We write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array)

The procedure checks if it has previously solved this subproblem; if not, the procedure computes the value in the usual manner

We say the recursive procedure has been memoized: it remembers what results it has computed previously.

3.1 Rod cutting: Top-down with memoization

Listing 2: MEMOIZED-ROD(p,n)

```
1  Let r[0..n] be a new array
2  for i ← 0 to n do
3      r[i] ← -1
4  return MEMOIZED-ROD-AUX(p,n,r)
```

Listing 3: MEMOIZED-ROD-AUX(p,n,r)

```
1  if r[n]⩾ 0 then
2      return r[n]
3  if n==0 then
4      q ← 0
5  else
6      q ← -1
7      for i ← 1 to n do
8          q ← max{q,p[i]+MEMOIZED-ROD-AUX(p,n-i,r)}
9  r[n] ← q
10 return q
```

3.2 Second approach

Second approach to implement dynamic programming: Bottom-up method

It depends on the idea of the ”size” of a subproblem, such that solving any particular subproblem relies on solving smaller subproblems

We sort the subproblems by size and solve them in size order, smallest first. We save the solutions.

We solve each subproblem only once, and when we first see it, we have solved all its prerequisite subproblems.

Listing 4: BOTTOM-UP-ROD(p,n)

```
1  Let r[0..n] be a new array
2  r[0] ← 0
3  for j ← 1 to n do
4      q ← -1
5      for i ← 1 to j do
6          q ← max{q,p[i]+r[j-i]}
7      r[j]⩽ q
8  return r[n]
```

3.3 Running time

For rod cutting, both approaches have a worst case running time of $\Theta(n^2)$

Easy to see for bottom-up

The memoized approach solves each subproblem for sizes 0,1,...,n once. To solve a problem of size n, the for loop iterates n times. The total number of iterations is an arithmetic series.

Usually, both approaches have the same complexity, but the bottom-up approach has lower constants, due to less overhead for procedure calls