Introduction to Dynamic Programming

Optimization Problem: Rod Cutting

Claim: Not all optimization problems have optimal greedy solutions.

Rod Cutting Problem . Consider a rod of length n units, made out of relatively valuable metal. The rod needs to be cut into a number of pieces to be sold separately. Selling a piece of rod of length i units earns $P[i]$ dollars. The Rod Cutting problem is formulated as follows: given n and the $P[i]$ table for $i = 1 \ldots n$, find the set of cuts of the rod, i.e., the set/sequence of lengths $L = (l_1, \ldots, l_k)$, such that:

$$\sum_{j=1}^{k} l_j = n;$$

$$Cost(L) = \sum_{j=1}^{k} P[l_j] = \max_{\{l'_1+\ldots+l'_s=n\}} \left(\sum_{j=1}^{s} P[l'_j] \right),$$

i.e., find the set of rod cuts that lead to the largest generated amount of money.

Example. Consider an instance of the Rod Cutting problem depicted on Figure 1. We have a rod of size n units, and the prices of rods of lengths 1 — 9 are listed in the array $P[i]$. We can see right away that not all ways to cut the rods lead to the same amount of money brought in. For example, $L = \{9\}$ has the total cost $Cost(L) = P[9] = 13$, whereas $L' = \{1, 1, 1, 1, 1, 1, 1, 1, 1\}$ has the total cost $Cost(L') = 9 \cdot P[1] = 9 < 13 = Cost(L)$. From this we may observe that it is more profitable to sell the rod as a whole than to split it into nine rods of size 1 unit each.

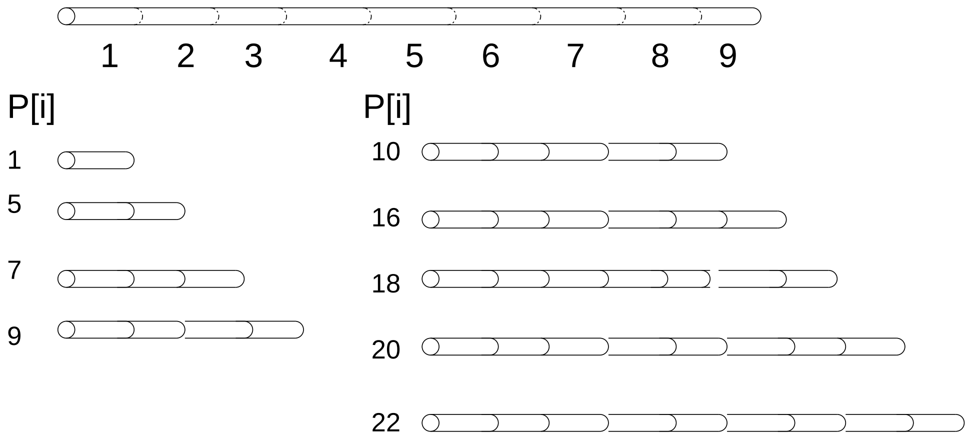

Figure 1: An instance of the Rod Cutting problem for $n = 9$.

i	1	2	3	4	5	6	7	8	9
$P[i]$	1	5	7	9	10	16	18	20	22
$\frac{P[i]}{i}$	1.0	2.5	2.333	2.25	2.0	2.666	2.57	2.5	2.55

Table 1: Price per unit of rod length.

Solving Rod Cutting

Steps:

- Try greedy techniques. Convince yourselves they do not quite work.
- Solve "small" versions of the problem.
- Can solving a smaller problem help you solve a more complex problem?

Example. Consider again the instance of the Rod Cutting problem from Figure 1. We note that the standard greedy algorithm (select the most expensive by unit of length rod length) does not always work. For example, we could have the following input:

$n = 5$

i	$P[i]$	$\frac{P[i]}{i}$
1	1	1
2	5	2.5
3	8	$2\frac{2}{3}$
4	10	2.5

The greedy algorithm picks the solution $\{3, 1\}$ (as $P[i]/i$ is maximized by $i = 3$). However $Cost(\{3, 1\}) = 8 + 1 = 9 < 10 = Cost(\{2, 2\})$.

Top-Down Solution

Observe the following:

- For input n, there are $\frac{n}{2}$ possibilities to *make the first cut*:

  $\{n\}$ (no cut)
  $\{n-1, 1\}$
  $\{n-2, 2\}$
  ...
  $\{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil\}$

- Once the first cut $\{x, y\}$ $(x + y = n)$ is made, the problem of finding the best cut for the rod of length n is reduced to the problem of finding the best solutions for rods of lengths x and y.
- Thus, Rod Cutting appears to have the optimal substructure property.
- But NOT, as we noted earlier, the greedy choice property.
- We cannot simply pick one possible split and use it. (What if some other split is better?).
- **But we can find the best cost for each split and pick the one with the best cost.**

This gives rise to the following *top-down* algorithm for finding the best cost (w/o finding the actual set of cuts matching it):

```
ALGORITHM Cut(P[1..n],n)
Cost[0] ← P[n];
for i = 1 to n/2 do
   Cost[i] ← Cut(P[], i) + Cut(P[], n-i);
endfor
BestCost ← FindMax(Cost[]);
return BestCost;
```

Bottom-Up Solution

There is one **major** problem with the top-down solution:

> Each recursive call to Cut(P[], i) will be repeated **multiple times**.
>
> In fact, Cut(P[],i) will be repeated once for each call of Cut(P[], j) where $j > i$.

For example, for $n = 5$, the numbers of repetitions will be:

Call	Number of repetitions
Cut(P[],5)	1
Cut(P[],4)	1
Cut(P[],3)	2
Cut(P[],2)	4
Cut(P[],1)	8

However, one call is sufficient to determine the optimal cut of a given size.

This can be avoided by using a *bottom-up* iterative approach. All we need to do is *remember* Cost[j] and use it to construct the costs of larger cuts. To do this, we need to start with the simplest cuts and compute Cost[0] before computing Cost[1] before computing Cost[2] etc. The iterative algorithm for computing the costs this way is shown below.

```
ALGORITHM CutIterative(P[1..n],n)
Cost[1] ← P[1];
for j = 2 to n do
   Cost[j] ← P[j];
   for i = 1 to n/2 do
     if Cost[j] < Cost[i] + Cost[j-i] then
      Cost[j] ← Cost[i] + Cost[j-i];
    endif
  endfor
endfor
return Cost[n];
```