28. Dynamic Programming

Idea: Like divide & conquer:
↳ Solve problem by combining solutions to subproblems.

Assume subproblems overlap
↳ Do not use recursion and repeatedly solve the same subproblem.
↳ Solve subproblems once, store results in a table.

Application: Often optimization problems
↳ May have many solutions, each having a value.
↳ Find solution with optimal value, an optimal solution

Approach: Four steps

1. Characterize structure of an optimal solution (should involve optimal solutions to subproblems, more on this later).
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution (two strategies, see below).
4. Construct optimal solution from computed information.

Steps 1–3 in every dynamic programming (if we only need the value of an optimal solution).

Step 4, if also the solution itself is required.
In this case, one usually maintains additional information in Step 3.

28.1 Rod Cutting

ROD:
Given: Rod of length n, table of prices p_i for rods of length i, $1 \le i \le n$.
Problem: Determine the maximal revenue
(obtainable by cutting the rod and selling the pieces).

Example:

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

$4 \rightsquigarrow 9$ $\quad 2+2 \rightsquigarrow 5+5=10$ $\quad 1+3 \rightsquigarrow 1+8=9$ (3+1)

$1+1+2 \rightsquigarrow 1+1+5=7$ $\quad 1+1+1+1 \rightsquigarrow 4$.
(1+2+1, 2+1+1)

- Note that a rod of length n can be cut in 2^{n-1} ways.
  We can cut/not cut at every i inches from the left.
- Now, if an optimal decomposition is
  $n = i_1 + \dots + i_k$ // with cuts in additive notation
  then the revenue is
  $r_n = p_{i_1} + \dots + p_{i_k}$.

Insight:
We can obtain the optimal revenues r_n
in terms of the optimal revenues of shorter rods:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1\}$$

Either · make no cut or
· cut the given piece into two
and obtain optimal revenues for those pieces.

⇒ The problem has optimal substructure
↳ Optimal solutions incorporate optimal solutions to subproblems.
↳ These subproblems may be solved independently.
↳ Both aspects are discussed below.

The recursion can even be simplified.

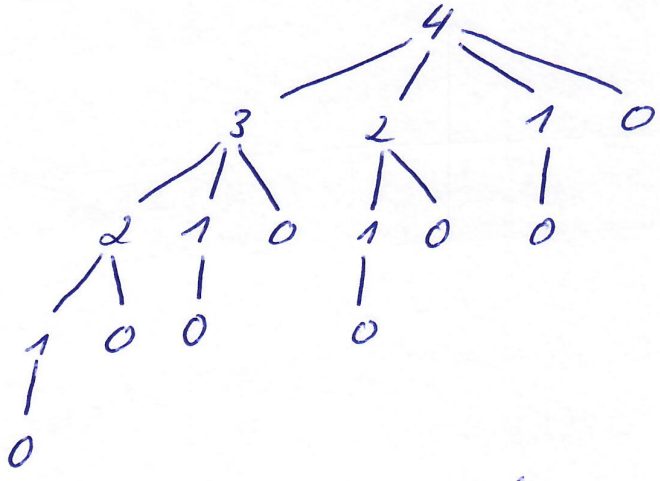
Make the first cut i inches from the left and optimize for the remaining length:

$$r_n = \max\{p_i + r_{n-i} \mid 1 \le i \le n\}$$

The recursive top-down implementation is immediate:

```
CUT(p, n){   // prices given as an array p
  if n==0 then return 0;
  q = -∞;
  for i=1 to n do
    q = max(q, p[i] + CUT(p, n-i));
  od
  return q;
}
```

The call tree of this implementation with n as the parameter:



- A path from root to leaf corresponds to one of the 2^{n-1} ways of cutting.
- The tree has 2^n nodes and 2^{n-1} leaves.

Indeed, let $T(n)$ denote the total number of calls (made to CUT) when called with initial parameter n.

We have

$$T(0) = 1 \quad \text{and} \quad T(n) = 1 + \sum_{j=0}^{n-1} T(j).$$

One can show by induction that $T(n) = 2^n$.

With dynamic programming

↳ Arrange subproblems so that they are solved only once.
↳ Save the solution rather than recomputing it.

Step back:
We use additional memory to save computing time.
→ time-memory trade-off

Dynamic programming may transform
an exponential-time solution into a polynomial-time solution.
More precisely:

> Dynamic programming runs in polynomial time
> when the number of distinct subproblems is polynomial in the input
> and each subproblem can be solved in polynomial time.

Two ways of implementing dynamic programming

Top-down implementation with memoization (from memo):

- Write the procedure recursively
- Save results to subproblems in an array or hash table.
- If a subproblem occurs, first check the table
  - ↳ If the subproblem is found, return the saved value.
  - ↳ If the subproblem is not found,
    compute the value as usual and save the result.

On the example

```
MEMOCUT(p,n){
    let r[0..n] be a new array;
    for i=0 to n do r[i]=-∞ od
    return MEMOCUTAUX(p,n,r);
}
MEMOCUTAUX(p,n,r){
    if r[n]≥0 then return r[n]; // Look-up.
    if n==0 then
        q=0;
    else
        q=-∞
        for i=1 to n do
            q=max(q, p[i]+MEMOCUTAUX(p,n-i,r)); // Compute as usual
        od
    fi;
    r[n]=q;     // Save computed value.
    return q;
}
```

Bottom-up implementation:

- Need a notion of size for subproblems
- Solving a subproblem should only depend on solving smaller subproblems.
- Then sort the problems according to their size and solve them in increasing order.

(when we arrive at a problem, we have already solved all smaller subproblems it depends on (and saved the solutions/values))

On the example

```
BOTTOM-UP-CUT(p, n) {
    let r[0..n] be a new array
    r[0] = 0;
    for j = 1 to n do
        q = -∞;
        for i = 1 to j do
            q = max{q, p[i] + r[j-i]};
        od
        r[j] = q;
    od
    return r[n];
}
```

Analysis:

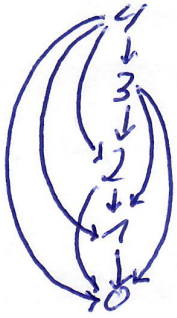
1) Typically, both implementations have the same asymptotic complexity.
   (sometimes, top-down can avoid to recurse into all subproblems).
2) Bottom-up typically has better constant factors
   as it avoids overhead for procedure calls.

In the example: $\Theta(n^2)$.

Subproblem graphs

Goal: Understand the set of subproblems involved in a dynamic programming problem.
and how the subproblems depend on one another.

In the example:



Note that this is a collapsed version of the above tree.

Definition

The subproblem graph (V, E) is defined by

V = sizes of subproblems

$E \ni (x, y)$, if we need an optimal solution value to the subproblems of size y when computing an optimal solution to the subproblem of size x.

Coming back to the implementation:

↳ Bottom-up dynamic programming considers the vertices of the subproblem graph in reverse topological order.

↳ Top-down dynamic programming is depth-first search of the subproblem graph.

The size of the subproblem graph can help us determine the running time of dynamic programming algorithms.

↳ Since we solve each subproblem once,

$$\text{running time} = \sum_{\text{subproblems } p} \text{running time}(p).$$

↳ Number of subproblems = number of vertices.

↳ Running time to solve a subproblem is proportional to the degree of the vertex in the subproblem graph.

In this setting, running time $\approx$ #vertices $\cdot \max\{\deg(v) \mid v \in V\}$.

Linear in the number of edges.

Reconstructing a solution

Goal: Not only compute the revenue,
but also the list of pieces.

Idea: • Store the choice that led to an optimal value.
• Technically, introduce an auxiliary array $s[0..n]$,
where $s[j]$ is the optimal size of the first piece
to cut-off from a rod of length j.

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

```
let r[0..n] and s[0..n] be arrays;
r[0] = 0 = s[0];
for j = 1 to n do
    q = -∞;
    for i = 1 to j do
        if q < p[i] + r[j-i] then
            q = p[i] + r[j-i];
            s[j] = i;
        fi
    od
    r[j] = q;
od
return r and s;
}
```

In the example:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Question: How to determine the space requirement of an algorithm
from the structure of the subproblem graph?

28.2 Dynamic Programming Theory

Question: What are the criteria an optimization problem should satisfy for dynamic programming to apply.

Answer: Optimal substructure and overlapping subproblems.

28.2.1 Optimal Substructure

Recall: In Step 1, we characterize the structure of an optimal solution.

Definition:
A problem exhibits optimal substructure, if an optimal solution contains within it optimal solutions to subproblems.

Hence, when constructing a solution we must make sure that the range of subproblems includes those used in an optimal solution.

Example:
Optimally cutting a rod involves optimally cutting two pieces resulting from the first cut.

How to discover optimal substructure?

1. A solution to the problem involves making a choice (choosing an initial cut).
2. Given a choice, determine what subproblems arise and how to characterize the space of subproblems.
3. Show that subproblems used in an optimal solution need to have themselves optimal solutions.

Characterizing the space of subproblems:

- Keep the characterization as right as possible, expand it when necessary.
- In rod cutting: Optimally cutting length i, for each i. Sometimes two end-points of the input vary.

Parameters of optimal substructure:

1. How many subproblems does an optimal solution to the original problem use.
2. How many choices do we have to determine which subproblems to use in an optimal solution.

In the example, the recurrence that we implemented used one subproblem but had n choices.

Both factors determine the running time as follows:

#subproblems overall × #choices.

In the example: $O(n) \times n = O(n^2)$.

The running time can also be derived from the subproblem graph.

Note:

- Dynamic programming has similarities with greedy algorithms. In particular, problems to which greedy algorithms apply have optimal substructure.
- The difference is that instead of first finding optimal solutions to subproblems and then returning an informed choice, greedy algorithms make a choice and then solve the subproblem, without bothering with alternative subproblems. In some cases, this works!

28.2.2 Independence

Consider the following problems:

USSP:

Given: Graph (V, E), $u, v \in V$.

Problem: Find an unweighted shortest (simple) path from u to v.

ULSP:

Given: Graph (V, E), $u, v \in V$.

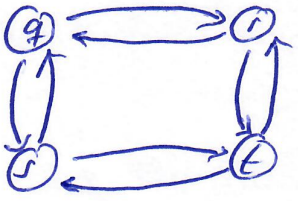
Problem: Find an unweighted longest simple path from u to v.

USSP exhibits optimal substructure:

- Any path from u to $v \neq u$ contains an intermediary vertex w (may be u or v).
- Then a shortest path $u \overset{p}{\rightsquigarrow} v$ can be decomposed into $u \overset{p_1}{\rightsquigarrow} w$ and $w \overset{p_2}{\rightsquigarrow} v$ such that both are USSP solutions.

ULSP does not exhibit optimal substructure:

Example:



A longest path from q to t is $q \to r \to t$.

Is $q \to r$ a longest simple path from q to r?

No, by $q \to s \to t \to r$.

Similarly, $r \to t$ is not optimal by $r \to q \to s \to t$.

Not only are the components of an optimal solution themselves not optimal, and so the problem lacks optimal substructure

optimal solutions to the subproblems cannot be composed at all to form a solution to the original problem.

Why? Because $q \to s \to t \to r \to q \to s \to t$ is not simple.

USLP is NP-complete.

Definition:
Two subproblems invoked in a recursion are independent, if the solution to one subproblem does not affect the solution to the other.

What does induce dependence? Resources!

- Resources used to solve one subproblem are not available for the other.
- USSP does not share resources, USLP shares vertices as resources.

28.2.3 Overlapping subproblems

Definition:
An optimization problem has overlapping subproblems if a recursive algorithm revisits the same subproblem repeatedly.

- For dynamic programming to be efficient, the space of subproblems has to be small (polynomial in input size, will not hold in the FPT-world).
- When we encounter brand-new subproblems at each recursion, use divide & conquer (or something else).