Dynamic Programming

Dynamic programming is a method developed by Bellman from 1950-57 for creating algorithms for **combinatorial optimization problems**. It's a specific form of the **divide-and-conquer** approach, using recursion to solve larger problems by breaking them down into smaller ones.

Combinatorial Optimization Problems

These involve finding the best structure out of many possibilities, where the structure is made of a finite number of parts.

Examples include:

- Finding the quickest route from point A to point B.
- Maximizing profit when packing a truck or ship.
- Creating a school schedule with minimal after-hours teaching.
- Developing a production plan that minimizes late deliveries.

Core Idea of Dynamic Programming

The main idea behind dynamic programming is to:

> Create a table of solutions to subproblems, solving each only once. This typically reduces runtime from exponential to polynomial.

More generally, dynamic programming involves:

> Developing recursive solutions for optimization problems where subproblems are only reduced by $O(1)$ in size.

The challenging part is finding the recursive solution; implementation is usually similar across problems.

Finding the Recursive Solution

A good approach involves:

1. Describing the problem size using integer indices, creating a table with corresponding dimensions.
2. Analyzing how an optimal solution for a given problem size can be divided into:
    - A "last part."
    - "The rest," which should be an optimal solution for a smaller, similar problem.

This last property is called "optimal subproblems."

Example: Malte's Problem

Imagine you have a gold chain with n links that you can divide into smaller lengths to sell. A goldsmith buys chains of different lengths at different prices.

How do you divide your long chain to maximize your selling price?

There are 2^{n-1} different ways to divide the chain, so trying every option is not efficient.

Optimal Subproblems in Malte's Problem

Any division of a chain of length n consists of:

- A last piece of length $k \leq n$.
- A division of the remaining chain of length $n - k$.

Observation (Optimal Subproblems):

> An optimal division of a chain of length n requires the division of the remaining part to be optimal for a chain of length $n - k$. If a better division of the remainder existed, it could replace the current one, improving the overall solution for the chain of length n.

Let $r(n)$ be the value of an optimal division of a chain of length n. Clearly, $r(0) = 0$. The goal is to find $r(n)$ for $n > 0$.

Recursive Formula for r(n)

An optimal division T for length n includes:

- A last piece of length $k \leq n$.
- An optimal division of the remaining chain of length $n - k$.

The value $r(n)$ of T equals $p_k + r(n - k)$, suggesting recursion. However, the value of k is unknown.

Solution:

Let T_i (for $i = 1...n$) be a division with a last piece of length i and an optimal division of the rest. The value of T_i is $p_i + r(n - i)$.

- At least one of $T_1, T_2, T_3, \ldots, T_n$ is optimal for length n.
- No T_i can have a value better than the optimal value.

Therefore:

$r(n) = max(p_i + r(n - i))$, with $r(0) = 0$ where $1 \leq i \leq n$

Calculating Optimal Values

$r(n) = max(p_i + r(n - i))$, with $r(0) = 0$ where $1 \leq i \leq n$

This means $r(n)$ is recursively defined based on smaller instances.

However, using recursion directly isn't efficient. The recursion tree has $1 + (1 + 2 + 4 + \cdots + 2^{n-1}) = 2^n$ nodes, leading to a runtime of $\Theta(2^n)$ due to repeated subproblems.

Using a Table

Focus on a table of optimal solution values.

Start with $r(0) = 0$:

n	0	1	2	3	4	5	6	7	8	9	10
r(n)	0										

Each field can be filled using $r(n) = max(p_i + r(n - i))$ if the preceding fields are filled.

This dependency allows calculating $r(n)$ bottom-up, i.e., for increasing values of n.

Runtime

$r(n) = max(p_i + r(n-i))$, with $r(0) = 0$ where $1 \leq i \leq n$

This computation can be done with two simple for-loops:

```
r[0] = 0
for k = 1 to n:
  max = -∞
  for i = 1 to k:
    x = p[i] + r[k - i]
    if x > max:
      max = x
  r[k] = max
```

The time complexity is $O(1 + 2 + 3 + 4 + \cdots + n) = \Theta(n^2)$.

Example Calculation

Using $r(n) = max(p_i + r(n-i))$, with $r(0) = 0$ and the prices p_i:

length i	1	2	3	4	5	6	7	8	9	10
price pi	1	5	8	9	10	17	17	20	24	26

Fill the table for $r(n)$ from left to right:

n	0	1	2	3	4	5	6	7	8	9	10
r(n)	0	1	5	8	10	13	17	18	22	25	27

Finding the Solution

$r(n)$ only provides the value of the optimal solution. To find the actual solution (the lengths of the chain pieces):

> Store the length $s(n)$ of the last piece for the optimal solution of length n.
> Save the value i that maximizes the recursive equation.

$r(n) = max(p_i + r(n-i))$, with $r(0) = 0$ where $1 \leq i \leq n$

length i	1	2	3	4	5	6	7	8	9	10
price pi	1	5	8	9	10	17	17	20	24	26
length n	0	1	2	3	4	5	6	7	8	9
optimal value r(n)	0	1	5	8	10	13	17	18	22	25
last length s(n)	0	1	2	3	2	2	6	1	2	3

```
while n > 0:
  print s[n]
  n = n - s[n]
```

Memoization

Memoization combines recursion with table storage.

- Recursion: $\Theta(2^n)$
- Structured table filling: $\Theta(n^2)$

Combine them:

```
GuldKæde(n)
  if n == 0:
    return 0
  else if r[n] is already filled in table:
    return r[n]
  else:
    x = max (p[i] + GuldKæde(n − i)) where 1 ≤ i ≤ n
    r[n] = x
    return x
```

An arrow showing a subproblem's dependency on others becomes an edge in the recursion tree exactly once (the first time the subproblem is reached). This results in the same $\Theta(n^2)$ runtime and $\Theta(n)$ space complexity as bottom-up table filling, though with a slightly worse constant factor in practice.