Dynamic programming

Combinatorial optimization problems

Combinatorial structure: A structure composed of a finite number of individual parts. Examples:

- Route from A to B.
- Loading of a truck or container ship.
- Teaching schedule.
- Production plan for y orders and x production machines.

Combinatorial optimization problem: one wants to find the best combinatorial structure among many possible ones. Examples:

- Fastest route from A to B.
- Most profitable loading of trucks or container ship.
- Teaching schedule with the least amount of teaching after 4 PM.
- Production plan for y orders and x production machines with the fewest delivery deadline overruns.

Dynamic programming

Dynamic programming [Bellman, 1950-57]: a method for developing algorithms for combinatorial optimization problems.

Dynamic programming is a special case of the Divide-and-Conquer method i.e, it is a recursive method that builds solutions to larger problems based on solutions to smaller problems.

Observation:

- Normally in recursive methods: subproblems are typically half the size, and there are no repetitions of subproblems in different parts of the recursion tree.

- In some recursive methods: subproblems are called where the size is only reduced by one. This often leads to repetitions of subproblems in different parts of the recursion tree, which often causes the runtime to become exponential.

Dynamic programming

The core idea of dynamic programming is the following:

- Create a table of solutions to subproblems so that each is only solved once. This usually changes the runtime from exponential to polynomial.

2^n n^2

More generally, the term dynamic programming is used for:

- Developing recursive solutions for optimization problems, where some subproblems in the recursion are only reduced by O(1) in size. The idea above is then used to implement the recursive solution efficiently.

The creative part is finding the recursive formulation of the solution. Applying the idea above afterward is quite similar from one problem to another.

Dynamic programming

The creative part is finding the recursive description of the solution.

The following is often a good approach:

1. What would be a good way to describe the size of a problem, expressed using one, two, or possibly more integer indices? This results in a table with one, two, or more dimensions.

2. Analyze how an optimal solution for a given problem size must consist of a “final part” and “the rest,” where you can argue that “the rest” must be an optimal solution for a smaller problem of the same type. In this way, a recursive description of the solutions can be formed.

This latter property is known as having **“optimal subproblems.”**

The principle is best understood through examples.

Example: Malte's problem

A lot of gold chains in surplus:

Photo:©Kaspar Wenstrup

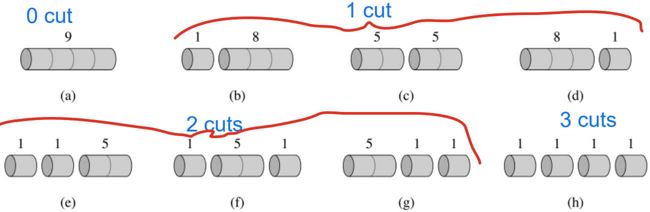
Example: Malte's problem

You have a gold chain with *n* links. It can be divided into smaller lengths (with *n* links in total, i.e., no links are lost). The goldsmith buys gold chains of different lengths at different prices:

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

example of Pk

How should you divide your long gold chain to maximize your selling price?



There are 2^{n-1} different ways to divide it, so simply trying all of them is not an efficient algorithm. meaning expontial runtime

here n is the number of elements in the chain, so 2^3 =8 options

Optimal subproblems

Any division of a chain of length n must consist of:

- A final piece of length $k \leq n$.
- A division of the remainder, i.e., a division of a chain of length $n - k$.

Observation (optimal subproblems):

For an optimal division of the chain of length *n*, the division of the remainder must itself be optimal for a chain of length $n - k$.
For if there existed a truly better division of the remainder, one could use that instead, thereby improving the "optimal" division of the chain of length *n*.

Let the value of an optimal division of a chain of length *n* be denoted $r(n)$

Clearly, r(0) = 0. We now want to find r(n) for n > 0.

base case

Recursive formula for $r(n)$

partisoning something of length n in a optimalway to maximaze the value of r(n)

Summary: An optimal partition T for length n consists of:

- A final piece of length $k \leq n$.
- An optimal partition of the remainder, i.e., an optimal partition of a chain of length $n - k$.

Pk is a tabel given

k = size of the final piece

The value $r(n)$ of T is therefore equal to $p_k + r(n - k)$, so it has a hint of recursion.

optimal value = value of a single piece + the optimal value for solving the remaining subproblem

But: unfortunately, we don't know k!

Recursive formula for $r(n)$

The value $r(n)$ of T is therefore equal to $p_k + r(n - k)$, so it has a hint of recursion, but unfortunately, we do not know k.

Therefore, we proceed as follows:

we look at all leghts i from 1 to n

Let T_i (for i = 1, ..., n) be a partition consisting of a final piece of length i and an optimal partition of the remainder.

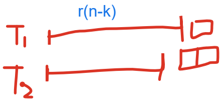
The value of T_i is $p_k + r(n - i)$

T_k has the value $p_k + r(n - k)$, just like T, and is therefore optimal for length n.

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

Skriv tekst her

$$r(n) = \max_{1 \le i \le n} (p_i + r(n - i)),\ r(0) = 0$$



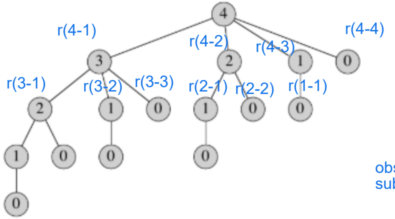
Calculate the optimal values

$$r(n) = \max_{1 \leq i \leq n} (p_i + r(n - i)), \quad r(0) = 0$$

That is, r(n) is (mathematically speaking) recursively defined based on smaller instances.
But is recursion also a good solution from an algorithmic point of view?

node = subproblem

branch = recursive calls



n = 4
when the node is 4 it must have 4 resurive calls to solve all subproblems

observe that we need to solve the same sub problem multiple times: r(2), r(1)

One can show by induction that there are $1 + (1 + 2 + 4 + \cdots + 2^{n-1}) = 2^n$

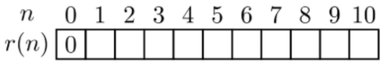
nodes in the recursion tree. So the runtime will be $\Theta(2^n)$. for n = 4, 2^4

The problem is repetitions among the subproblems of subproblems.

Use a table

Instead, focus on a table of the values of the optimal solutions.

Start: r(0) = 0

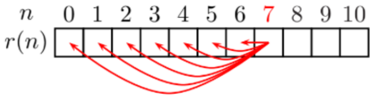


n	0	1	2	3	4	5	6	7	8	9	10
$r(n)$	0										

A cell can be filled in

$$r(n) = \max\,(p_i + r(n - i))$$

if the preceding cells are filled in. We can illustrate this dependency:

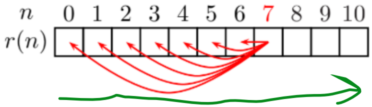


It follows from this that we can compute r(n) bottom-up, i.e., for increasing values of n.

Running time

$$r(n) = \max_{1 \le i \le n} (p_i + r(n-i)),\quad r(0) = 0$$

"bottom up method"



for r(7) we need to go thorugh all the numbers before

work from left to right

This computation can be done with two simple for-loops (the outer loop goes through the table cell by cell, the inner loop finds the max for each cell):

```
       r[0] = 0
O(n)   for $k = 1$ to $n$:          green arrow
           $max = -\infty$
O(n)       for $i = 1$ to k:        red arrows
               $x = p[i] + r[k-i]$:
               if $x > max$:
                   $max = x$
           $r[k] = max$
```

Time: $O(1 + 2 + 3 + 4 + \cdots + n) = \Theta(n^2)$

Example

Use

$$r(n) = \max_{1 \le i \le n} (p_i + r(n - i)),\ r(0) = 0$$

and the prices p_i :

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

To fill in the table of r(n) from right to left:

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

fx for n = 8: take max{1 + r(8-1) = 1 + 18, 5+r(8-2) = 5+17, 8+r(8-3)=8+13 .... }

Find the solution itself

The number r(n) is only the value of the optimal solution. But what if we want the actual solution (the individual lengths the gold chain should be broken into)?

Store the length s(n) of the last piece for an optimal solution of length n. That is, store the i that gives the maximum in the recursive equation.

$$r(n) = \max_{1 \le i \le n} (p_i + r(n - i)),\ r(0) = 0$$

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

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

s(n) what ever was lest of n, for the maximum solution? fx for lenght n = 8, r(n) =22 was derived from n = 6, r(n) = 17. so s(6)=6. s(8) = 8-s(6)

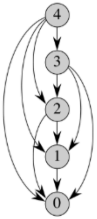
```
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

Memoization

Recursion: Θ(2^n). Structured table filling: Θ(n^2)

Can the two be combined? Yes.

```
GuldKæde($n$)
    if $n = 0$
        return 0
    else if $r[n]$ allerede udfyldt i tabel
        return $r[n]$
    else
        $x = \max_{1 \le i \le n}(p_i + \text{GuldKæde}(n - i))$
        $r[n] = x$
        return $x$
```



An arrow in the figure, showing a subproblem's dependency on others, will become an edge in the recursion tree exactly once (the first time the subproblem is reached).

So the same runtime Θ(n^2) and space usage Θ(n) as in bottom-up table filling. But likely with a slightly worse constant in practice.