Scuola universitaria professionale della Svizzera italiana

SUPSI

Università della Svizzera italiana

Istituto Dalle Molle di studi sull'intelligenza artificiale

Algorithms and Data Structures

Dynamic programming

Matteo Salani
matteo.salani@idsia.ch

Motivation

Similarly to divide and conquer algorithms, dynamic programming algorithms solve a problem by iteratively solving subproblems.

- Dynamic programming algos are best suited when problems share subproblems and each solution can be computed just once and then stored for further use.
- Dynamic programming applies to **optimization/combinatorial problems**

Common steps

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

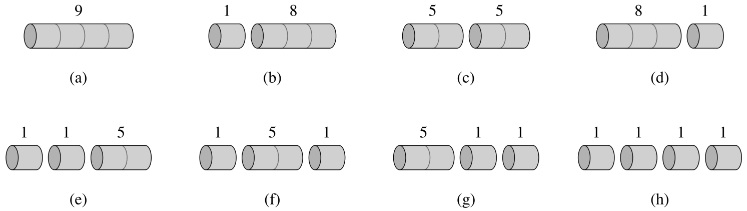
Example: rod cutting

Rod cutting problem

- Given a rod of length n and sell prices p_i
- Let p_i the sell price of a rod of length i
- Determine the maximum revenue r_n

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

Possible cuts of a rod of length $n = 4$



Example: rod cutting

We can cut a rod of length n in 2^{n-1} ways. As at each i we can decide to cut or not (*actually, if we impose the cut length in non-decreasing sizes, the number of solutions equals* $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$*, according to the partition function*)

We call a solution a **decomposition** with additive notation for some $1 \leq k \leq n$

$$n = i_1 + i_2 + \cdots + i_k$$

with revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

Example: rod cutting

We define the value of an optimal solution recursively:

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

Observe we solve smaller problems of the same type.

Theorem:

optimal-substructure The rod-cutting problem has the optimal-substructure, that is the optimal solution incorporates optimal solutions of subproblems

If we simplify the problem saying it is decomposed in a uncut left part and further divided right part we can write

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Recursive implementation

```c
int cutrod(int p[], int n){
    if (n == 0)
        return 0;

    int q = INT_MIN;
    for (int i=0; i<n; i++)
        q = max(q, p[i] + cutrod(p, n-i-1));
    return q;
}
```

Listing 1: cut

Why is that inefficient? Show recursion tree on the blackboard.

Efficient implementation

Dynamic programming makes the computation efficient by memorizing the values of already computed solutions (trade-off time-memory).
A dynamic-programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.
There are actually two techniques:

- Top-down memoization (it is not misspelled), natural recursive way but with saved values
- Bottom-up method sort problems by size and solve smaller first, then bigger problems are solved composing solutions of smaller problems.

Efficient implementation

```
int mem_cutrod(int p[], int n) {
    int r[n];
    for (int i = 0; i < n; i++)
        r[i] = -1;
    return mem_cutrod_aux(p, n, r);
}
int mem_cutrod_aux(int p[], int n, int r[]) {
    if (n == 0) return 0;
    if (r[n-1] >= 0) return r[n-1];

    int q = INT_MIN;
    for (int i = 0; i < n; i++)
        q = max(q, p[i] + mem_cutrod_aux(p, n-i-1, r));
    r[n - 1] = q;
    return q;
}
```

Listing 2: mem cut

We make use of an auxiliary array *r* storing the results of previous revenue computations.

The recursive call is developed to the leaves only once

Efficient implementation - bottom up

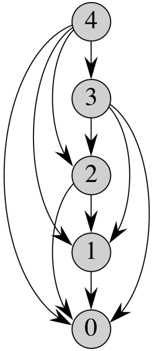
```c
int bottom_up_cutrod(int p[], int n) {
    int r[n];
    for (int j = 0; j < n; j++)
        r[j] = 0;

    r[0] = p[0];
    for (int j = 1; j < n; j++) {
        int q = INT_MIN;
        for (int i = 0; i < j; i++) {
            q = max(q, p[i] + r[j - i - 1]);
        }
        r[j] = q;
    }
    return r[n-1];
}
```

Listing 3: cut

Optimal solution is constructed incrementally.

Cut rod subproblem graph example

To design a dynamic-programming algorithm, we should understand the set of subproblems involved and how subproblems depend on one another. The **subproblem graph** embodies this information. It represents the relative needs of a subproblem size to other subproblem sizes to be solved.



- The subproblem graph is a "reduced" version of the recursion tree.
- The size of the graph helps in determining the running time as every problem node is solved just once.

Cut rod - building a solution

Discuss with the class.

Generalization

When the method applies?

- **Optimal substructure**: an optimal solution to a problem contains the optimal solution of subproblems.
- **Overlapping subproblems property**: the space of subproblems is small, that is a recursive algorithm would solve the same subproblem many times.

Optimal substructure

Examples and contradiction of common substructure:

1. Unweighted shortest simple path from s to t in a graph.
2. Unweighted longest simple path from s to t in a graph.

Proof on the blackboard.

Optimal substructure

The subproblems in finding the longest simple path are not independent, whereas for shortest paths they are.
The solution to one subproblem does not affect the solution to another subproblem of the same problem.
Why does that hold for shortest path? I.e. why subproblems are independent for the shortest path?
Because a node cannot appear twice in any optimal path.
If w is on a shortest path p from u to v we can join *any* shortest path from u to w with *any* shortest path from w to v.
Prove it: suppose a node x appears in both subproblems...

Overlapping subproblems

Typically the total number of distinct subproblems is polynomial in the input size.

- When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.
- In contrast, a problem for which a divide-and conquer approach is suitable usually generates **brand-new** problems at each step of the recursion.

A relevant application in Biology

Comparing DNA sequences of organisms is a relevant problem in biology. A strand of DNA consists in a sequence of bases {denine, guanine, cytosine, and thymine}.
For example:

- $S_1 =$ DACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- $S_2 =$ DGTCGTTCGGAATGCCGTTGCTCTGTAAA

A subsequence of a given sequence is the given sequence with zero or more elements left out.
Formally, given a sequence $X = \{x_1, x_2, \ldots, x_m\}$ another sequence $Z = \{z_1, z_2, \ldots, z_k\}$ with $z \leq m$ is a **subsequence** of X if there exist a strictly increasing sequence of indexes $I = \{i_1, i_2, \ldots, i_k\}$ such that for all $j \in 1, \ldots, k$ $x_{i_j} = z_j$.

A relevant application in Biology

Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

In the **longest common subsequence problem** we want to find a common subsequence of maximal length.

Brute force and prefixes

Enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y

A subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ therefore X has 2^m this approach requires exponential time.

Given a sequence $X = \{x_1, x_2, \ldots, x_m\}$ we define the $i - th$ **prefix** of X as $X_i = \{x_1, x_2, \ldots, x_i\}$.

X_0 is the empty sequence.

Optimal substructure of LCS

Theorem: Optimal substructure of LCS

Let $X = \{x_1, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ be sequences and let $Z = \{z_1, z_2, \ldots, z_k\}$ be any LCS of X and Y

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is LCS of X and Y_{n-1}

If $z_k \neq x_m$... prove (1), (2) and (3) by contradiction...

Therefore, an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an **optimal-substructure property**!

A recursive solution

We examine *one* or *two* subproblems for finding LCS of $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$.

If $x_m = y_n$, then find LCS of X_{m-1} and Y_{n-1} and append $x_m = y_n$

If $x_m \neq y_n$, solve two subproblems: find LCS of X_{m-1} and Y, find LCS of X and Y_{n-1} and keep the maximum.

This recursion has the overlapping property!

Let $c[i,j]$ be the length of an LCS of sequences X_i, Y_j
We obtain the recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if i=0 or j=0,} \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j, \\ max(c[i-1,j], c[i,j-1]) & \text{if } i,j>0 \text{ and } x_i \neq y_j, \end{cases}$$

Bottom up dynamic programming for LCS

```
LCS-LENGTH($X, Y, m, n$)
  let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
  for $i = 1$ to $m$
      $c[i, 0] = 0$
  for $j = 0$ to $n$
      $c[0, j] = 0$
  for $i = 1$ to $m$
      for $j = 1$ to $n$
          if $x_i == y_j$
              $c[i, j] = c[i-1, j-1] + 1$
              $b[i, j] = $ "$\nwarrow$"
          else if $c[i-1, j] \geq c[i, j-1]$
                  $c[i, j] = c[i-1, j]$
                  $b[i, j] = $ "$\uparrow$"
              else $c[i, j] = c[i, j-1]$
                  $b[i, j] = $ "$\leftarrow$"
  return $c$ and $b$
```

Complexity $\Theta(mn)$.

Example

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Bottom up dynamic programming for LCS

```
PRINT-LCS($b, X, i, j$)
  if $i$ == 0 or $j = 0$
      return
  if $b[i, j]$ == "$\nwarrow$"
      PRINT-LCS($b, X, i-1, j-1$)
      print $x_i$
  elseif $b[i, j]$ == "$\uparrow$"
      PRINT-LCS($b, X, i-1, j$)
  else PRINT-LCS($b, X, i, j-1$)
```

Complexity $\Theta(m+n)$.