Dynamic Programming

ACM Seminar in Algorithmics

Marko Genyk-Berezovskyj
berezovs@fel.cvut.cz

Tomáš Tunys
tunystom@fel.cvut.cz

CVUT FEL, K13133

February 27, 2013

Problem: **Longest Increasing Subsequence**

Input: A list of numbers, $a_1, a_2, \ldots, a_n$.

Output: Find the largest k for which there are indices $i_1, i_2, \ldots, i_k$ with $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$.

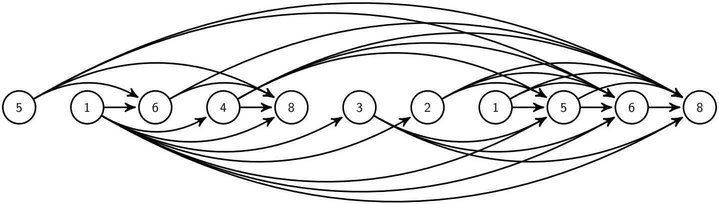
Exercise: What is the longest increasing subsequence in the following list of numbers?

5, 1, 6, 4, 8, 3, 2, 1, 5, 6, 8

Problem: **Longest Increasing Subsequence**

On the first sight it may seem a bit difficult to come up with a solution but as we will see there is *an underlying structure* in the problem that allows us to solve it "fast".

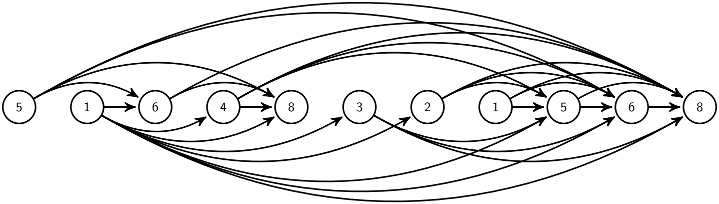
If you are faced with the following



can you tell what the underlying structure is and what its properties are?

Problem: **Longest Increasing Subsequence**

Let $G = (V, E)$ is a directed graph with vertices $v_1, v_2, \ldots, v_n$ which are labelled with the numbers from the list $a_1, a_2, \ldots, a_n$. There is an edge $(v_i, v_j) \in E$ *iff* the corresponding numbers satisfy $a_i < a_j$.



Two important things to notice:

- The graph G is a directed acyclic graph, so-called DAG.
- There is one-to-one correspondence between paths in the graph G and the increasing subsequences in the list $a_1, a_2, \ldots, a_n$.

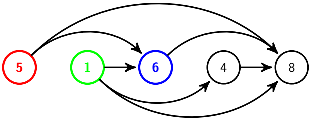
Problem: **Longest Increasing Subsequence**

Solution: Let us denote the length of a longest path ending in the vertex v_i as $L(i)$. Then the following algorithm computes $L(i)$ for every $i \in \{1, \ldots, n\}$:

```
for $j \leftarrow 1$ to $n$ do
    $L(j) \leftarrow 1 + \max\{L(i) \mid (i, j) \in E(G)\}$
end
return $\max_{i=1}^{n} L(i)$
```

Example:

$L(3) = 1 + max\{L(1), L(2)\}$



Dynamic Programming "Definition"

Dynamic programming is a problem solving technique based upon the following two principles:

1. Identification of subproblems.
2. Using the answer to "smaller" subproblems to solve the "bigger" ones.

In the case of the *Longest Increasing Subsequence* we have according to these principles:

1. $L(i)$ - the length of the longest increasing subsequence ending with a_i.
2. $L(i) \leftarrow 1 + \max_{j<i}\{L(j) \mid a_j < a_i\}$.

When Does Dynamic Programming Work?

A problem is solvable with dynamic programming if it has:

1. Optimal substructure. *An optimal solution of the whole problem can be build out of optimal solutions to its subroblems.*
2. Overlapping subproblems.

Remember, when you are faced with a DP problem the DAG is *implicit* - you are the one to define the vertices (subproblems) and the edges (relations) between them.

How do we find out a given problem is solvable using DP?

When Does Dynamic Programming Work?

A problem is solvable with dynamic programming if it has:

1. Optimal substructure. *An optimal solution of the whole problem can be build out of optimal solutions to its subroblems.*
2. Overlapping subproblems.

Remember, when you are faced with a DP problem the DAG is *implicit* - you are the one to define the vertices (subproblems) and the edges (relations) between them.

How do we find out a given problem is solvable using DP?

Through Experience!

Problem: **Maximum Subarray Problem**

Input: An array of numbers, $a_1, a_2, \ldots, a_n$.

Output: Find a continuous subarray within the given array which has the largest sum.

Solution:

Problem: **Maximum Subarray Problem**

Input: An array of numbers, $a_1, a_2, \ldots, a_n$.

Output: Find a continuous subarray within the given array which has the largest sum.

Solution:

1. Subproblems: $S[i]$ - the largest sum of a subarray ending at i-th element (inclusive).
2. Induction: $S[i] \leftarrow a_i$ **if** $S[i-1] <= 0$ **else** $a_i + S[i-1]$.

Problem: **Longest Common Subsequence**

Input: Two strings $a_1 a_2 \cdots a_n$ and $b_1 b_2 \cdots b_m$.

Output: The largest k for which there are indices

$$1 \leq i_1 < i_2 < \cdots < i_k \leq n$$

and

$$1 \leq j_1 < j_2 < \cdots < j_k \leq m$$

with

$$a_{i_1} a_{i_2} \cdots a_{i_k} = b_{j_1} b_{j_2} \cdots b_{j_k}.$$

Solution:

Problem: **Longest Common Subsequence**

Input: Two strings $a_1 a_2 \cdots a_n$ and $b_1 b_2 \cdots b_m$.

Output: The largest k for which there are indices

$$1 \le i_1 < i_2 < \cdots < i_k \le n$$

and

$$1 \le j_1 < j_2 < \cdots < j_k \le m$$

with

$$a_{i_1} a_{i_2} \cdots a_{i_k} = b_{j_1} b_{j_2} \cdots b_{j_k}.$$

Solution:

1. Subproblems: $S[i][j]$ - the longest common subsequence of the prefixes of the two strings $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$.
2. Induction: ($\mathit{diff}(a, b)$ is 1 **if** $a = b$ **else** 0)
   $S[i][j] \leftarrow \max\{S[i-1][j], S[i][j-1], S[i-1][j-1] + \mathit{diff}(a_i, b_j)\}$.

Problem: **Knapsack Problem**

Input: A list of items with their weights $w_1, w_2, \ldots, w_n$ and costs $c_1, c_2, \ldots, c_n$, a total weight W the knapsack can hold (capacity).

Output: A subset of items for which the sum of their costs is maximum and the knapsack can hold them, i.e. $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} c_i$ is maximum and $\sum_{i \in I} w_i <= W$.

Solution:

Problem: **Knapsack Problem**

Input: A list of items with their weights $w_1, w_2, \ldots, w_n$ and costs $c_1, c_2, \ldots, c_n$, a total weight W the knapsack can hold (capacity).

Output: A subset of items for which the sum of their costs is maximum and the knapsack can hold them, i.e. $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} c_i$ is maximum and $\sum_{i \in I} w_i <= W$.

Solution:

1. Subproblems: $C[\widehat{w}][i]$ - the maximum achievable value of the items from a set $\{1, 2, \ldots, i\}$ with the knapsack capacity $\widehat{w}$.
2. Induction: $C[\widehat{w}][i] = \max\{C[\widehat{w}][i-1], C[\widehat{w} - w_i][i-1] + c_i\}$ **if** $\widehat{w} \geq w_i$ **else** $C[\widehat{w}][i-1]$.

Top-Down vs. Bottom-up Approach

Consider the knapsack problem again. The following solution of it is referred to as *the bottom-up approach*:

```
for $i \leftarrow 0$ to $n$ do
    $C[0][i] \leftarrow 0$
end

for $\widehat{w} \leftarrow 0$ to $W$ do
    $C[\widehat{w}][0] \leftarrow 0$
end

for $i \leftarrow 1$ to $n$ do
    for $\widehat{w} \leftarrow 1$ to $W$ do
        if $\widehat{w} < w_i$ then
            $C[\widehat{w}][i] \leftarrow C[\widehat{w}][i-1]$
        else
            $C[\widehat{w}][i] \leftarrow C[\widehat{w} - w_i][i-1] + c_i$
        end
    end
end

return $C[W][n]$
```

Top-Down vs. Bottom-up Approach

Now consider the following which is referred to as *the top-down approach*:

```
function solve($\widehat{w}$,$i$)
    if $C[\widehat{w}][i] = -1$ then
        if $\widehat{w} \geq w_i$ then
            $C[\widehat{w}][i] \leftarrow$solve($\widehat{w} - w_i$,$i - 1$)$+c_i$
        end
        $C[\widehat{w}][i] \leftarrow \max\{C[\widehat{w}][i]$, solve($\widehat{w}$,$i - 1$)$\}$
    end
    return $C[\widehat{w}][i]$
end

for $\widehat{w} \leftarrow 0$ to $W$ do
    for $i \leftarrow 0$ to $n$ do
        $C[\widehat{w}][i] \leftarrow -1$
    end
end

return solve($W$,$n$)
```

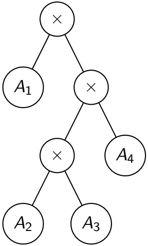
Problem: **Chain Matrix Multiplication**

Input: An expression $A_1 \times A_2 \times \cdots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$.

Output: A parenthesization of the expression such that the number of multiplications needed to be done in order to evaluate it is minimum.

Hint:

$$A_1 \times ((A_2 \times A_3) \times A_4)$$



Problem: **Chain Matrix Multiplication**

Solution:

1. Subproblems: $C[i][j]$ - the minimum number of multiplications to evaluate $A_i \times A_{i+1} \times \cdots \times A_j$.
2. Induction: $C[i][j] \leftarrow \min_{i \leq k < j}\{C[i][k]+C[k][j]+m_{i-1} \cdot m_k \cdot m_j\}$.

DP Problem Complexities

The problems that have been presented are typical representants of the following complexity classes:

$O(n)$	*Maximum Subarray Sum*
$O(mn)$	*Longest Common Subsequence*
$O(n^3)$	*Chain Matrix Multiplication*