Efficient Algorithms and Problem Complexity
– Dynamic Programming –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. What is Dynamic Programming?

2. Example: The Longest Common Subsequence

3. Example: Floyd’s Algorithm
Dynamic Programming

- Divide and Conquer solves a problem instance by dividing it into smaller instances and solving them.
- Conceptually, this is a top-down approach.
- Dynamic Programming (DP) is similar, but bottom-up (conceptually).
- Solutions to small sub-instances are computed and then used to solve larger sub-instances.
- Major strength: avoid to compute things repeatedly.

...but the name is badly chosen...
The Dynamic Programming Pattern

Suppose the problem instances are from a set $\mathcal{I}$. Thus, with input $I \in \mathcal{I}$, a program $P$ is supposed to produce a solution $P(I)$.

The DP idea, if $|I| = n$:

- Define auxiliary parameters $i_1, \ldots, i_k$ to “cut out” smaller sub-instances $I_{i_1 \ldots i_k}$ from $I$.
- The smallest sub-instances $I_{i_1 \ldots i_k}$ should be easy to solve directly.
- Larger $I_{i_1 \ldots i_k}$ should be solvable by making use of the solutions of the smaller sub-instances.
- Strategy:
  - Solve the smallest sub-instances $I_{i_1 \ldots i_k}$ and save the results.
  - Using the saved results, solve larger and larger sub-instances until $I$ itself is solved.
Iterative Dynamic Programming

Filling a table “from one corner”
(dimensions = number of auxiliary parameters)
Dynamic Programming by Memoization

Filling a table recursively “by need”.
Memoization of results of earlier calls avoids exponential blow-up.
Parallelization Opportunities

Usually, DP solutions can nicely be parallelized!

The red squares can be computed in parallel!
What is Dynamic Programming?

Straightforward Recursion vs. Dynamic Programming

Typical situation in which DP is worth considering:

- There is a recursive solution that breaks down instances into smaller ones (and combines their solutions to bigger ones).
- The recursion creates a combinatorial explosion of calls.
  Example:
  \[
  f(a_1 \cdots a_n) \Rightarrow f(a_1 \cdots a_{n-1}) \oplus f(a_2 \cdots a_n)
  \]
yields \(O(2^n)\) calls.
- Many recursive calls are redundant (as they are identical).
  Example: Above, there are only \(O(n^2)\) different calls.
- In such situations, simple recursion is a huge waste of time.
  DP computes each sub-solution only once.

Question: Why is divide and conquer not a huge waste of time, then?
What is Dynamic Programming?

Dynamic Programming and Optimization

The Optimal Substructure Property (Johnsonbaugh, Schaefer)

If $S$ is an optimal solution to the problem, then the components of $S$ are optimal solutions to subproblems.

In order for a DP algorithm to solve an optimization problem correctly, this property must hold.

Let’s discuss what this means...
The Longest Common Subsequence

Terminology: A subsequence of a string $a_1 \cdots a_m$ is a string $a_{i_1} \cdots a_{i_l}$ such that $1 \leq i_1 < \cdots < i_l \leq m$ (not necessarily a substring).

Problem: Longest Common Subsequence

Input: Strings $u = a_1 \cdots a_m$ and $v = b_1 \cdots b_n$.
Output: A subsequence of both $u$ and $v$ that is maximal in length.

Recursive solution:

$$lcs(u, v) = \begin{cases} 
0 & \text{if } |u| = 0 \text{ or } |v| = 0 \\
\max\{lcs(u', v), lcs(u, v')\} & \text{if } u = u'a, v = v'b, a \neq b \\
lcs(u', v') + 1 & \text{if } u = u'a, v = v'a.
\end{cases}$$

⇒ Exponentially many recursive calls. 😞
The Longest Common Subsequence by DP, Step 1

Computing the length of \( lcs(u', v') \) for all prefixes \( u', v' \) of \( u, v \):

\[
\text{lcs\_len}(u, v) \quad \text{where} \quad (m, n) = (|u|, |v|)
\]

create new integer array \( len[0 \cdots m, 0 \cdots n] \) initialized to 0

for \( i = 1 \) to \( m \) do

for \( j = 1 \) to \( n \) do

if \( u[i] = v[j] \) then

\[
len[i, j] \leftarrow len[i - 1, j - 1] + 1
\]

else

\[
len[i, j] \leftarrow \max(len[i - 1, j], len[i, j - 1])
\]

\( \Rightarrow \) this computes \( lcs\_len(u, v) = len(m, n) \) in \( O(mn) \) steps.
The Longest Common Subsequence by DP, Step 2

Constructing $\text{lcs}(u, v) = \text{extract\_lcs}(m, n)$ from $\text{len}[0 \cdots m, 0 \cdots n]$:

\[
\text{extract\_lcs}(i, j) \text{ where } u, v \text{ are global variables}
\]

\[
\begin{align*}
\text{if } i = 0 \text{ OR } j = 0 \text{ then} & \quad \text{return empty string} \\
\text{if } u[i] = v[j] \text{ then} & \quad \text{return } \text{extract\_lcs}(i-1, j-1) \cdot u[i] \quad \text{// dot denotes concatenation} \\
\text{else if } \text{len}[i, j-1] = \text{len}[i, j] \text{ then} & \quad \text{return } \text{extract\_lcs}(i, j-1) \\
\text{else} & \quad \text{return } \text{extract\_lcs}(i-1, j)
\end{align*}
\]

Question: Would it be better to avoid the detour via $\text{lcs\_len}$?
All-Pairs Shortest Paths

Problem: All-Pairs Shortest Paths

Input: The adjacency matrix of a weighted graph \( G = (V, E, w) \).

Output: A matrix \( dist \) such that \( dist[u, v] \) is the length of the shortest path from \( u \) to \( v \), for all \( u, v \in V \).

DP idea (where \( u, v \in V = \{1, \ldots, n\} \) and \( k \in \{0, \ldots, n\} \)): Let

\[
Floyd(u, v, k) = \text{length of the shortest path from } u \text{ to } v \text{ that does not pass any of } k+1, \ldots, n
\]

Observation: For \( k > 0 \),

\[
Floyd(u, v, k) = \min\{Floyd(u, v, k - 1),
Floyd(u, k, k - 1) + Floyd(k, v, k - 1)\}
\]
Floyd’s Algorithm

**Example: Floyd’s Algorithm**

\[ \text{Floyd}_0(G) \text{ where } G \text{ is given as a matrix, } G[u, v] = w(u, v) \in \mathbb{N}^\infty \]

\[
\begin{align*}
\text{for } k = 1 \text{ to } n \text{ do} \\
\quad &\text{for } u = 1 \text{ to } n \text{ do} \\
\quad &\quad \text{for } v = 1 \text{ to } n \text{ do} \\
\quad &\quad \quad G[u, v] \leftarrow \min(G[u, v], G[u, k] + G[k, v])
\end{align*}
\]

\[ \text{Floyd}_1(G) \text{ where } \ldots \text{ and } \text{high}[u, v] \text{ is initialized to } 0 \]

\[
\begin{align*}
\text{for } k = 1 \text{ to } n \text{ do} \\
\quad &\text{for } u = 1 \text{ to } n \text{ do} \\
\quad &\quad \text{for } v = 1 \text{ to } n \text{ do} \\
\quad &\quad \quad \text{if } G[u, v] > G[u, k] + G[k, v] \text{ then} \\
\quad &\quad \quad \quad G[u, v] \leftarrow G[u, k] + G[k, v] \\
\quad &\quad \quad \quad \text{high}[u, v] \leftarrow k
\end{align*}
\]
Floyd’s Algorithm, Extracting the \((u, v)\)-Path

\[
\text{getPath}(u, v) \text{ where array } high \text{ has been filled using } Floyd_1
\]

\[
k = high[u, v]
\]

\[
\text{if } k = 0 \text{ then}
\]

\[
\text{return empty list}
\]

\[
\text{else}
\]

\[
\text{return getPath}(u, k) \cdot [k] \cdot getPath(k, v) \quad \text{// list concatenation}
\]

Notes & questions

- The version in the textbook is slightly different.
- Why not store paths instead of \(high[u, v]\) right away?
Warshall’s Algorithm

Computing the transitive closure of a graph (or Boolean matrix):

\[
\text{Warshall}(G) \text{ where } G \text{ is given as a matrix, } G[u, v] \in \{T, F\}
\]

\[
\text{for } k = 1 \text{ to } n \text{ do}
\]
\[
\quad \text{for } u = 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \text{for } v = 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \quad G[u, v] \leftarrow G[u, v] \lor (G[u, k] \land G[k, v])
\]

Note what is important in both algorithms:

- Initially, each \(G[u, v]\) contains the value for \(k = 0\).
- \(G[u, v] = G[u, v] \oplus (G[u, k] \odot G[k, v])\) uses \(\odot\) to combine \(G[u, k]\) and \(G[k, v]\), and \(\oplus\) to combine \(G[u, k] \odot G[k, v]\) with the old \(G[u, v]\).
- Floyd uses edge weights and the operations \(\text{min}\) and \(+\).
- Warshall uses \(T, F\) and the operations \(\lor\) and \(\land\).
Converting FA to REGEXP (for those interested)

Have a look at (or recall) the algorithm that converts a finite automaton into an equivalent regular expression. It uses once more the same algorithm, but for a completely different problem (mainly by choosing other initial values and operations $\oplus$ and $\odot$).
Please read the chapters about dynamic programming in the textbooks.