Lecture 9: Dynamic programming

In **dynamic programming**, a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to smaller problems to solve larger ones, until all subproblems are solved.

We introduce this powerful design principle through several example problems:

- Shortest paths in DAGs
- Longest increasing subsequences
- Edit distance
- Knapsack
- Chain matrix multiplication
- Shortest paths
- Travelling salesman problem
- Independent sets in trees
Example 1: Shortest paths in DAGs (1/4)

- Shortest paths are particularly easy in DAGs (see Lecture 6 / Chapter 4 in the course book for general shortest paths algorithms).
- This is because vertices in a DAG can be linearised: there is an ordering of the vertices such that all arcs (directed edges) go from left to right.
Example 1: Shortest paths in DAGs (2/4)

- So how do we determine the distance from a source vertex $s$ to a given vertex $v$ in a digraph (specifically a DAG)?
- If we know the shortest distances from $s$ to all the immediate predecessors $u_1, \ldots, u_t$ of $v$, this is easy:
- The minimum distance is obtained through an immediate predecessor $u_i$ for which the sum (the distance from $s$ to $u_i$) + (the distance from $u_i$ to $v$) is minimum, that is,

$$\text{dist}(v) = \min \{ \text{dist}(u_1) + \ell(u_1, v), \ldots, \text{dist}(u_t) + \ell(u_t, v) \}$$
Example 1: Shortest paths in DAGs (3/4)

- If the vertices are linearised, and we compute the dist values in the given linear order, we are guaranteed to have all the required information for a vertex $v$ when we get there.

- Now we can compute all distances in a single pass:

**Algorithm 1: Algorithm for shortest paths in DAGs**

1. initialize all $\text{dist}(\cdot)$ values to $\infty$
2. $\text{dist}(s) = 0$
3. for each $v \in V \setminus \{s\}$ in linearised order do
   4. $\text{dist}(v) = \min\{\text{dist}(u) + \ell(u, v) : (u, v) \in E\}$
4. end

- Note that (a properly implemented version of) the algorithm runs in linear time in the size of the input graph.
In this very simple example of dynamic programming, the subproblems are to determine the values \( \{ \text{dist}(v) : v \in V \} \), and a subproblem \( \text{dist}(u) \) is “smaller than” subproblem \( \text{dist}(v) \) if vertex \( u \) is a predecessor of \( v \) in the linearised order.
Example 2: Longest increasing subsequence (1/4)

➢ In this problem, the input is a sequence of numbers $a_1, a_2, \ldots, a_n$.

➢ A subsequence is any subset of these numbers taken in order $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots i_k \leq n$.

➢ An increasing subsequence is one in which the numbers get strictly larger.

➢ The task is to find an increasing subsequence of greatest length.

➢ For example, the longest increasing subsequence of

$$5, 2, 8, 6, 3, 6, 9, 7$$

is

$$2, 3, 6, 9$$
Example 2: Longest increasing subsequence (2/4)

- The solution space can be analysed using a digraph $G = (V, E)$ where for each element $a_i$ in the sequence there is a vertex $i \in V$, and there is an edge $(i, j) \in E$ whenever $i < j$ and $a_i < a_j$.

- Now there is a one-to-one correspondence between increasing subsequences and paths in this DAG.
- So the task is to find the longest path in the DAG.
Example 2: Longest increasing subsequence (3/4)

Algorithm 2: Algorithm for longest increasing subsequence

1. for $j = 1, 2, \ldots, n$ do
2.     $L(j) = 1 + \max\{L(i) : i < j \land a_i < a_j\}$
3. end
4. return $\max_j L(j)$

where $\max\{} = 0$

Basic idea:

- Let $L(j)$ be the length of the longest path ending at $j$ (+1).
- Then $L(j)$ is the length of the longest path to any one of $j$’s predecessors + 1.
Example 2: Longest increasing subsequence (4/4)

- This is again dynamic programming.
- This time our collection of subproblems are the $L(j)$, $1 \leq j \leq n$, and these satisfy:
  - there is an ordering on the subproblems and a relation that shows how to solve a subproblem given the answers to subproblems that appear earlier in the ordering.
- The running time is $O(n^2)$:
  - Computing $L(j)$ takes time proportional to the indegree of $j$, the sum of these is linear in $|E|$ and at most $O(n^2)$ (given that predecessors of a vertex $j$ are known).
- The $L$ values give only the length of an optimal subsequence. To recover the subsequence itself some further bookkeeping about the longest paths is needed (see Lecture 6 / Chapter 4 in the course book).

1 Also a slightly more complicated $O(n \log n)$ algorithm exists.
Dynamic programming and subproblem DAGs

- Dynamic programming algorithms can in general be viewed as operating on an (abstract, implicit) subproblem DAG.

- The vertices of this DAG are the subproblems that we define, and the edges indicate the dependences between subproblems: if to solve subproblem B, we need the answer to subproblem A, then there is a (conceptual) edge from A to B.

- In this case A is considered a “smaller” subproblem than B.
Example 3: Edit distance (1/7)

- In this problem the task is to determine how close two given strings, i.e. sequences of symbols are.
- A natural measure of distance is the extent to which the strings can be aligned.
- For example, two possible alignments of SNOWY and SUNNY:

  S - N O W Y    - S N O W - Y
  S U N N - Y    S U N - - N Y

  Cost: 3        Cost: 5

- The symbol “-” indicates a gap; any number of these can be placed in either string.
- The cost of an alignment is the number of columns in which the letters differ, or there is a gap in one string and a symbol in the other.
Example 3: Edit distance (2/7)

- The **edit distance** between two strings is the cost of their best possible alignment.
- The edit distance can also be seen as the minimum number of edits (insertions, deletions, and substitutions of characters) needed to transform the first string to the second.
Example 3: Edit distance (3/7)

- What are the subproblems?
- The input is two strings $x[1 \cdots m]$ and $y[1 \cdots n]$.
- We shall consider the edit distances between each prefix $x[1 \cdots i]$ of $x$ and each prefix $y[1 \cdots j]$ of $y$.
- Call this subproblem $E(i, j)$, corresponding to the associated optimal cost (edit distance) between the two prefixes.
- The goal is to solve subproblem $E(m, n)$.

For instance, the subproblem $E(7, 5)$:

```
EXPONENTIAL
POLYNOMIAL
```
Example 3: Edit distance (4/7)

- How to express $E(i, j)$ in terms of “smaller” subproblems?
- Consider the alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$ at its rightmost column; there are three possible cases:

$$
\begin{array}{c|c|c}
  x[i] & - & y[j] \\
  \hline
  - & y[j] & \end{array}
$$

- In the first case, the cost is $1 +$ that of the remaining alignment $x[1 \cdots i - 1]$ and $y[1 \cdots j]$, i.e. $1 + E(i - 1, j)$.
- In the second case, the cost is $1 +$ that of the remaining alignment $x[1 \cdots i]$ and $y[1 \cdots j - 1]$, i.e. $1 + E(i, j - 1)$.
- In the third case, the cost is that of the remaining alignment $x[1 \cdots i - 1]$ and $y[1 \cdots j - 1]$ plus the matching cost of $x[i]$ and $y[j]$, viz. $E(i - 1, j - 1) + 1$ if $x[i] \neq y[j]$ and $+ 0$ if $x[i] = y[j]$. 

Example 3: Edit distance (5/7)

► Hence, we have expressed the subproblem $E(i, j)$ in terms of three smaller subproblems $E(i - 1, j), E(i, j - 1), E(i - 1, j - 1)$.

► We do not know which of these leads to the best solution, but we can try them all and pick the best:

$$E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), \text{diff}(i, j) + E(i - 1, j - 1)\}$$

where $\text{diff}(i, j) = 0$ if $x[i] = y[j]$ and otherwise 1.
Example 3: Edit distance (6/7)

- The subproblems $E(i, j)$ can be represented as a two-dimensional table which should be filled in an order where $E(i - 1, j)$, $E(i, j - 1)$, $E(i - 1, j - 1)$ are filled in before $E(i, j)$.

- But what are the base cases $E(i, 0)$ and $E(0, j)$?

- $E(i, 0)$ is the edit distance between the 0-length prefix of $y$ (the empty string) and the first $i$ letters of $x$; this distance is clearly $i$.

- Similarly, $E(0, j) = j$. 
Example 3: Edit distance (7/7)

Algorithm 3: Algorithm for edit distance

1. for $i = 0, 1, 2, \ldots, m$ do
2. \hspace{1em} $E(i, 0) = i$
3. end
4. for $j = 0, 1, 2, \ldots, n$ do
5. \hspace{1em} $E(0, j) = j$
6. end
7. for $i = 1, 2, \ldots, m$ do
8. \hspace{1em} for $j = 1, 2, \ldots, n$ do
9. \hspace{2em} $E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), \text{diff}(i, j) + E(i - 1, j - 1)\}$
10. \hspace{1em} end
11. end
12. return $E(m, n)$
In the edit distance problem, the subproblems are of the form \((i, j)\) and there are edges from \((i - 1, j), (i, j - 1), (i - 1, j - 1)\) to \((i, j)\).

In this graph edge lengths can be set so that edit distance corresponds to shortest path!

Set lengths to 1 for all edges except for those edges from \((i - 1, j - 1)\) to \((i, j)\) with \(x[i] = y[j]\) for which the length is set to 0 (dotted lines in the figure).

Now edit distance is the length of the shortest path from \((0, 0)\) to \((m, n)\).
Example 4: Knapsack (1/2)

- In this problem the input is a set of $n$ items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, together with a weight limit $W$.
- The task is to select a set of items of at most total weight $W$ that has the most value.
- We consider two versions of the problem:
  - In knapsack with repetition there are unlimited quantities of each item available.
  - In knapsack without repetition only one copy of each item is available.
Example 4: Knapsack (2/2)

- For example, take $W = 10$ and

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

- For knapsack with repetition the optimal solution is: . . . ?
pick one copy of item 1 and two of item 4 (total value 48).

- For knapsack w/out repetition the optimal solution is: . . . ?
pick items 1 and 3 (total value 46).
Example 4.1: Knapsack with repetition (1/2)

- What are the subproblems?
- We can shrink the original problem in two ways:
  - look at fewer items (1, 2, ..., j for some $j \leq n$) or
  - consider a smaller weight limit $w \leq W$
- Consider the latter approach first.
- Let $K(w) =$ maximum value achievable with a knapsack of capacity $w$.
- If the optimal solution to $K(w)$ includes a copy of item $i$, then removing this copy leaves an optimal solution to $K(w - w_i)$.
- This means that $K(w) = K(w - w_i) + v_i$ for some $i$
- ... but we do not know which one so we try them all:

$$K(w) = \max_{i: w_i \leq w} \{ K(w - w_i) + v_i \}$$
Example 4.1: Knapsack with repetition (2/2)

**Algorithm 4**: Algorithm for knapsack with repetition

1. \( K(0) = 0; \)
2. for \( w = 1 \) to \( W \) do
   3. \( K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\} \)
3. end
4. return \( K(W) \)

The algorithm fills in a one-dimensional table of length \( W + 1 \) in left-to-right order. Each entry can take up \( O(n) \) time to compute. Thus, the overall running time is \( O(nW) \).
Example 4.2: Knapsack w/out repetition (1/2)

- For the case without repetition, a better way of forming subproblems is to limit the items (one of each only allowed).
- Let $K(w, j) =$ maximum value achievable using a knapsack of capacity $w$ and items $1, \ldots, j$.
- Now for each item $j$, either $j$ is needed or not needed to achieve the optimal value:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$
Example 4.2: Knapsack w/out repetition (2/2)

Algorithm 5: Algorithm for knapsack without repetition

1. Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$
2. for $j = 1$ to $n$ do
   3. for $w = 1$ to $W$ do
      4. if $w_j > w$ then
         5. $K(w, j) = K(w, j - 1)$
      6. else
         7. $K(w, j) = \max\{K(w, j - 1), K(w - w_j, j - 1) + v_j\}$
   8. end
3. end
4. return $K(W, n)$

The algorithm fills in a two-dimensional table with $W + 1$ rows and $n + 1$ columns. Each entry can be computed in constant time. Thus, the overall running time is $O(nW)$. 
Example 5: Chain matrix multiplication (1/4)

- Suppose we want to multiply a number of matrices with different dimensions.
- Because matrix multiplication is associative, i.e.
  \[ A \times (B \times C) = (A \times B) \times C \]
  different orders are possible.
- Multiplying an \( m \times n \) matrix with an \( n \times p \) matrix gives an \( m \times p \) matrix and takes roughly \( mnp \) multiplications.
- Different evaluation orders (parenthesizations) of chained multiplications lead to different costs.
Example 5: Chain matrix multiplication (2/4)

Consider matrices:
\[ A(50 \times 20), \ B(20 \times 1) \ C(1 \times 10), \ D(10 \times 100) \]

Now evaluation \( A \times B \times C \times D \) has different costs in different evaluation orders:

<table>
<thead>
<tr>
<th>Parenthesization</th>
<th>Cost computation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \times ((B \times C) \times D) )</td>
<td>[ 20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 ]</td>
<td>120.200</td>
</tr>
<tr>
<td>((A \times (B \times C)) \times D )</td>
<td>[ 20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100 ]</td>
<td>60.200</td>
</tr>
<tr>
<td>((A \times B) \times (C \times D) )</td>
<td>[ 50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100 ]</td>
<td>7.000</td>
</tr>
</tbody>
</table>

Observe that the natural greedy approach to always perform the cheapest matrix multiplication available (second option) does not lead to an optimal solution.
Example 5: Chain matrix multiplication (3/4)

- So the input is a sequence of matrices $A_1, \ldots, A_n$ with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$ and the task is to give an ordering (parenthesization) so that the cost of the matrix multiplication $A_1 \times \cdots \times A_n$ is the lowest.
- What are the subproblems?
- For $1 \leq i \leq j \leq n$ define:

$$C(i, j) = \text{minimum cost of multiplying } A_i \times \cdots \times A_j$$

- Now for any $k$ ($i \leq k < j$), this problem can be divided in two subproblems: multiplying $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$. So the cost $C(i, j)$ is the cost of the two subproblems plus the cost of combining the results.
- We do not know which is the best way of splitting, but we can try them all:

$$C(i, j) = \min_{1 \leq k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}$$
Example 5: Chain matrix multiplication (4/4)

Algorithm 6: Algorithm for chain matrix multiplication

1 for $i = 1$ to $n$ do
2 \hspace{1em} $C(i, i) = 0$
3 end
4 for $s = 1$ to $n - 1$ do
5 \hspace{1em} for $i = 1$ to $n - s$ do
6 \hspace{2em} $j = i + s$
7 \hspace{2em} $C(i, j) = \min\{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j : i \leq k < j\}$
8 \hspace{1em} end
9 end
10 return $C(1, n)$

The algorithm fills in a two-dimensional table whose entries take $\mathcal{O}(n)$ time to compute. Thus, the overall running time is $\mathcal{O}(n^3)$. 
Example 6: All-pairs shortest paths (1/3)

- If we want to find the shortest path between all pairs of vertices, the straightforward approach of running the general shortest-path algorithm $|V|$ times, once for each starting vertex, is not the most economical.

- A better alternative is the $O(|V|^3)$ Floyd-Warshall algorithm.

- The idea: subproblems are obtained by restricting the permissible intermediate vertices on the path.

- Number the vertices in $V$ as $\{1, 2, \ldots, n\}$ and let $\text{dist}(i, j, k)$ denote the length of the shortest path from $i$ to $j$ in which only vertices $\{1, 2, \ldots, k\}$ can be used as intermediates.

- Initially, $\text{dist}(i, j, 0) = \ell(i, j)$ if $(i, j) \in E$ and otherwise $\text{dist}(i, j, 0) = \infty$. 
Example 6: All-pairs shortest paths (2/3)

If we now consider a new intermediate vertex $k$, then the shortest path from $i$ to $j$ uses it once or not at all:

$$\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1), \text{dist}(i, j, k-1)\}$$
Example 6: All-pairs shortest paths (3/3)

Algorithm 7: Algorithm for all-pairs shortest paths

1 for $i = 1$ to $n$ do
2     for $j = 1$ to $n$ do
3         dist($i, j, 0$) = $\infty$
4     end
5 end
6 for all $(i, j) \in E$ do
7     dist($i, j, 0$) = $\ell(i, j)$
8 end
9 for $k = 1$ to $n$ do
10     for $i = 1$ to $n$ do
11         for $j = 1$ to $n$ do
12             dist($i, j, k$) =
13                 \[ \min\{ \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1) \} \]
14         end
15     end
16 end
Example 7: The travelling salesman problem (1/3)

- In this problem the input is a set of \( n \) cities and a matrix \( D = (d_{ij}) \) of intercity distances.
- The task is to find a tour that:
  1. starts and ends at city 1,
  2. visits all other cities exactly once, and
  3. has the minimum total length.
- This is a very difficult problem with no guaranteed polynomial time algorithm known.
- The brute force technique would examine all possible tours but would have \( \mathcal{O}(n!) \) time complexity.
- Using dynamic programming we can do a bit better (but not polynomial time).
Example 7: The travelling salesman problem (2/3)

- A suitable subproblem:
  for a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1 and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each vertex in $S$ exactly once, starting at 1 and ending at $j$.

- If $|S| \geq 2$, we can set $C(S, 1) = \infty$.

- Now $C(S, j)$ can be determined from subproblems by considering the second-to-last city, which has to be some $i \in S$, with the overall path length equal to distance from 1 to $i$ plus the distance from $i$ to $j$.

- We again try all such $i$ and choose the best:

  $$C(S, j) = \min_{i \in S: i \neq j} \{C(S - \{j\}, i) + d_{ij}\}$$

- The subproblems can be ordered by $|S|$.
Example 7: The travelling salesman problem (3/3)

Algorithm 8: Algorithm for the travelling salesman problem

1. \( C(\{1\}, 1) = 0 \)
2. \( \text{for } s = 2 \text{ to } n \text{ do} \)
3. \( \quad \text{for all subsets } S \subseteq \{1, 2, \ldots, n\} \text{ of size } s \text{ and containing } 1 \)
4. \( \quad \quad C(S, 1) = \infty; \)
5. \( \quad \quad \text{for all } j \in S, j \neq 1 \text{ do} \)
6. \( \quad \quad \quad C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\} \)
7. \( \quad \quad \end \)
8. \( \end \)
9. \( \text{return } \min\{C(\{1, 2, \ldots, n\}, j) + d_{j1} : 2 \leq j \leq n\} \)

There are at most \(2^n \cdot n\) subproblems and each takes linear time to compute. Hence, the total run time is \(O(n^2 2^n)\).
Example 8: Independent sets in trees (1/3)

- A subset of vertices $S \subseteq V$ is an **independent set** of graph $G = (V, E)$ if there are no edges between the vertices in $S$.

- Finding a largest independent set in a graph is a very difficult (“NP-complete”) problem with no guaranteed polynomial time algorithm known.

For this graph one largest independent set is $\{2, 3, 6\}$:
Example 8: Independent sets in trees (2/3)

- However if the graph is a tree, a linear time dynamic programming algorithm is available:
- Select one of the vertices in the tree as the root \( r \).
- In a rooted tree, each vertex \( u \) defines a subtree, that is, the subtree induced by the vertices \( v \) for which the path from \( v \) to \( r \) contains \( u \). (Or, the subtree “hanging from” \( u \), when we draw the tree so that the root \( r \) is the topmost vertex.)
- Hence, a suitable subproblem:
  \( I(u) = \) size of the largest independent set in the subtree hanging from \( u \).
- Now dynamic programming can proceed bottom-up in the rooted tree and the final goal is \( I(r) \) where \( r \) is the root of the entire tree.
Example 8: Independent sets in trees (3/3)

- Suppose we know the largest independent sets of all subtrees below a certain vertex $u$.
- Now computing $l(u)$ has two cases: any independent set either includes $u$ or it does not. Hence,

$$l(u) = \max\{1 + \sum_{\text{grandchildren } w \text{ of } u} l(w), \sum_{\text{children } w \text{ of } u} l(w)\}$$
Summary

- Dynamic programming is a general algorithmic technique that is applicable to a wide range of problems.
- In dynamic programming, a problem is divided into a set of subproblems such that there is
  1. an ordering on the subproblems, and
  2. a rule that shows how to solve a subproblem given the answers to subproblems that appear earlier in the ordering.
- Every dynamic programming algorithm has an underlying DAG structure where the vertices are subproblems and edges capture the precedence constraints; that is, a subproblem can only be solved once the answers to its predecessors in the DAG are known.