1 Examples

1.1 Longest increasing subsequence

Problem. The input is a sequence of numbers $a_1, \ldots, a_n$. A subsequence is any subset of these numbers taken in order. An increasing subsequence is one in which the numbers are getting strictly larger. The task is to find the length of the longest increasing subsequence.

(1) **Subproblems definition.** The subproblems that we consider are the suffixes of the given input.

\[
\begin{array}{c}
 a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\
\end{array}
\]

We call $\text{OPT}(i)$ the length of the longest increasing subsequence starting at $a_i$.

Then we will return the biggest $\text{OPT}(i)$.

(2) **Recursive formulation.**

**Question.** Starting at $a_i$, what can be the next item?

**Options.** It can be any element

- following $a_i$,
- and greater than $a_i$.

**Go through the options.** Let’s consider one option: let $a_j$ be the next item in the longest increasing subsequence starting at $a_i$. Then the length of this sequence $a_i \rightarrow a_j \rightarrow \ldots$ will be:

\[
\begin{array}{c}
1 + \text{OPT}(j) \\
\end{array}
\]
for the step $a_i \rightarrow a_j$ length of the longest increasing subsequence starting at $a_j$

**The optimal option.** We want the option that gives the longest length: the maximum of all the lengths given by these options.

\[
\text{OPT}(i) = 1 + \max_{j > i \text{ if } a_j > a_i} (\text{OPT}(j))
\]

length of the longest increasing subsequence starting at $a_i$ going through each following item $a_j$ greater than $a_i$ length of the longest increasing subsequence starting at $a_j$

(3) **Pseudocode.**

for $i$ from 1 to $n$:

\[
\text{OPT}[i] = 1 + \max(\{\text{OPT}[j] \text{ for } j > i \text{ if } a[i] < a[j]\})
\]

return $\max(\{\text{OPT}[i] \text{ for } i \text{ from } 1 \text{ to } n\})$
(4) **Time complexity.**

- We have $O(n)$ subproblems (main loop in the algorithm),

- and for each subproblem, we go through all the following items to get the maximum, which is $O(n)$.

So the total complexity of the main loop is $O(n^2)$. Then getting the maximum of all $OPT$s takes $O(n)$ operations, that can be ignored.

Therefore the total complexity is $O(n^2)$.

### 1.2 Edit distance

(1) **Problem.** The cost of an alignment of 2 strings is the number of columns in which the letters differ. And the edit distance between two strings is the cost of their best possible alignment. The task is to find the edit distance between 2 given strings $x[1..m]$ and $y[1..n]$.

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>O</th>
<th>W</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>S U N N - Y</td>
<td>S U N - N Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cost 3</td>
<td>cost 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2) **Subproblems definition.** The subproblems that we consider are the edit distance $OPT(i, j)$ between some prefix of the first string, $x[1...i]$, and some prefix of the second, $y[1...j]$.

(3) **Recursive formulation.**

**Question.** What options do we have for the rightmost column?

**Options.** We have 3 possibilities:

- **First option**: $(x[i], -)$. The cost of this alignment would be:

  \[
  1 + \text{OPT}(i - 1, j)
  \]

  for the alignment of the rightmost column:

  \[x[1...i - 1] \text{ and } y[1...j]\]

- **Second option**: $(-, y[j])$. The cost of this alignment would be:

  \[
  1 + \text{OPT}(i, j - 1)
  \]

  for the alignment of the rightmost column:

  \[x[1...i] \text{ and } y[1...j - 1]\]

- **Third option**: $(x[i], y[j])$. The cost of this alignment would be:

  \[
  \text{diff}(x[i], y[j]) + \text{OPT}(i - 1, j - 1)
  \]

  for the alignment of the rightmost column:

  \[x[1...i - 1] \text{ and } y[1...j - 1]\]
The optimal option. We want the option that gives the smallest cost: it will be the minimum between all the costs given by these 3 options.

\[
\text{OPT}(i, j) = \min \left( \begin{array}{c}
1 + \text{OPT}(i-1, j), \\
1 + \text{OPT}(i, j-1), \\
\text{diff}(x[i], y[j]) + \text{OPT}(i-1, j-1)
\end{array} \right)
\]

\text{edit distance between } x[1...i] \text{ and } y[1...j]

We can now fill the matrix of values for \text{OPT}(i, j) and return the last value \text{OPT}(m, n).

(4) Pseudocode.

```
for i from 0 to m:
    OPT[i, 0] = i
for j from 0 to n:
    OPT[j, 0] = j

for i from 1 to m:
    for j from 1 to n:
        E[i, j] = \min(1 + E[i-1, j], 1 + E[i, j-1], \text{diff}(i, j) + E[i-1, j-1])

return OPT[m, n]
```

(5) Complexity.

- There are \(mn\) cells in the matrix,
- and for each cell, you are making 3 operations.

So the overall complexity will be \(O(mn)\).