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 the given input.

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad \boxed{a_5 \quad a_6 \quad a_7 \quad a_8 \quad a_9 \quad a_{10}}$$

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

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

Method

We are going to find the solutions to **all** the subproblems. Then we will return the last one.
We assume we know already the solutions of some subproblems ($\mathbf{OPT}(j)$, in blue) to find the solution to the next subproblem ($\mathbf{OPT}(i)$, in red).

(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 $a_i \to a_j \to \ldots$ will be:

$$\boxed{1} + \boxed{\mathbf{OPT}(j)}$$

for the step $a_i \to a_j$

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 theses options.

$$\boxed{\mathbf{OPT}(i)} = 1 + \max_{\boxed{\substack{j>i \\ \text{if } a_j > a_i}}} \left(\boxed{\mathbf{OPT}(j)} \right)$$

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:
  OPT[i] = 1 + max([OPT[j] for j > i if a[i] < a[j]])
return max([OPT[i] for i from 1 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 $y[1..n]$.

```
S - N O W Y      - S N O W - Y
S U N N - Y      S U N - - N Y
  cost 3             cost 5
```

(2) Subproblems definition. The subproblems that we consider are the edit distance $\mathbf{OPT}(i,j)$ 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:

```
x[i]   or    -     or    x[i]
 -          y[j]         y[j]
```

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

$$\underbrace{\boxed{1}}_{\substack{\text{for the alignment of}\\ \text{the rightmost column}}} + \underbrace{\boxed{\mathbf{OPT}(i-1,j)}}_{\substack{\text{edit distance of}\\ x[1...i-1] \text{ and } y[1...j]}}$$

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

$$\underbrace{\boxed{1}}_{\substack{\text{for the alignment of}\\ \text{the rightmost column}}} + \underbrace{\boxed{\mathbf{OPT}(i,j-1)}}_{\substack{\text{edit distance of}\\ x[1...i] \text{ and } y[1...j-1]}}$$

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

$$\underbrace{\boxed{\mathrm{diff}(x[i],y[j])}}_{\substack{\text{for the alignment of}\\ \text{the rightmost column:}\\ 1 \text{ if } x[i] \neq y[j]\\ 0 \text{ if } x[i] = y[j]}} + \underbrace{\boxed{\mathbf{OPT}(i-1,j-1)}}_{\substack{\text{edit distance of}\\ x[1...i-1] \text{ and } y[1...j-1]}}$$

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.

$$\underbrace{\boxed{\color{red}{\mathbf{OPT}(i,j)}}}_{\color{red}{\text{edit distance between } x[1...i] \text{ and } y[1...j]}} = \min\Big(\underbrace{\boxed{1+\mathbf{OPT}(i-1,j)}}_{\text{First option}}, \underbrace{\boxed{1+\mathbf{OPT}(i,j-1)}}_{\text{Second option}}, \underbrace{\boxed{\text{diff}(x[i],y[j])+\mathbf{OPT}(i-1,j-1)}}_{\text{Third option}} \Big)$$

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

(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], 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 $O(mn)$.