# Week 6, Lecture 1

**Algorithm Analysis and Design**

**Pratyay Suvarnapathaki, 2020111016**

## The Edit Distance Problem

The edit distance between two strings can be alternatively termed as the *cost of alignment*.
For instance, the words `SUNNY` and `SNOWY` can be aligned in the following way to have the minimum edit distance.

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

Here, the edit distance is 3 (as the number of differing columns is 3).
Hence, given two strings $x[1 \cdots m]$ and $y[1 \cdots n]$, the minimum edit distance between them is to be found, as efficiently a possible.

---

## The Dynamic Programming Solution

Given the above two strings, a natural subproblem to approach would be one, where the **edit distance of some prefix** of the two strings is to be found.
Let us term the edit distance between $x[1 \cdots i]$ and $y[1 \cdots j]$, where $i \leq m$ and $j \leq n$ as $E(i, j)$.
The final goal is finding $E(m, n)$.

### Subprobleming the subproblem

Now, in order to express $E(i, j)$ into further smaller subproblems, we look at the rightmost columns of the two substrings. The possibilities are:
`x[i]` and `-` => Cost = 1 => $E(i, j) = 1 + E(i-1, j)$
`-` and `y[j]` => Cost = 1 => $E(i, j) = 1 + E(i, j-1)$
`x[i]` and `y[j]` => Cost = 1 if `x[i]!=y[j]`, else 0 => $E(i, j) = diff(x[i], y[j]) + E(i-1, j-1)$

## Ordering

In accordance with the principle of dynamic programming, the smaller subproblem should be solved before the larger one. i.e. $E(i-1,j), E(i,j-1), E(i-1,j-1)$ should all be solved before solving $E(i,j)\backslash$

## Algorithm

Since we have termed our subproblems with respect to the **rightmost columns** of the substrings, the base cases we need to separately take care of are $E(i,0)$ and $E(0,j)$.
But these cases are trivial, as the edit distance between a nonzero length string and an ampty string, is the nonzero length itself.
Hence, we have $E(i,0) = i$ and $E(0,j) = j$
In accordance with the notation $E(i,j)$, we implement this solution using a memoization table, which we shall conveniently call `E`.

Algorithm:

```
# E[m+1][n+1]
# diff is a function that returns 0 if the characters at the string indices
corresponding to the two arguments are equal, 1 if they are not

def EditDistance():
for i in range(m+1):
E[i][0]=i
for j in range(n+1):
E[0][j]=j
for i in range(m+1):
for j in range(n+1):
E[i][j]=min(1+E[i-1][j],1+E[i][j-1],diff(i,j)+E[i-1][j-1])
return E[m][n]
```

The complexity of the above algorithm is naturally $O(mn)$
The underlying DAG for this problem is such that:

- Its nodes correspond to the table values of $E$.
- The edges (dependencies) between nodes reflect the 'ordering' constraints above, i.e. $E(i-1,j), E(i,j-1), E(i-1,j-1)$ should all be solved before solving $E(i,j)$

---