Ahora veremos más ejemplos de programaciones dinámicas ahora enfocadas en Strings

Longest common subsequence.

Definimos las subsecuencia más larga en común entre dos strings y y x como la subsecuencia más larga que aparece tanto en y como en x. Ejemplos:

$TOUR$ y $OPERA$ es OR con longitud 2.

$ACAATCC$ y $AGCATGC$ es $ACATC$ con longitud 5.

$apple$ y $people$ es $pple$ con longitud 4.

Ya que tenemos una idea de lo que trata el problema podemos tratar de abordarlo.

El que estemos en la parte de programación dinámican nos es un gran spoiler pues sabemos que la solución incluyen programación dinámica, pero ahora veamos como. La característica de la programación dinámica es poder resolver un problema desde subproblemas menores,

Definimos $LCS(i,j)$ como la longest common subsequence con prefijos $x[0...i]$ y $y[0...j]$ entonces en cada momento si la los string en la posicion coinciden entonces tenemos una posición i,e $i = j$ entonces obtenemos un elemento más en nuestra subsecuencia, y nos reduciría el tamaño en cada string por uno, si no coinciden buscamos que opción nos da más rendimiento si eliminar un elemento de la del primer string o del segundo string.

$$LCS(i,j) = \begin{cases} LCS(i,j) = LCS(i-1,j-1)+1 & si \quad x[i] = y[i] \\ max(LCS(i,j-1), LCS(i-1,j) & si \quad x[i] \neq y[i] \\ 0 & si \quad i = 0 \vee j = 0 \end{cases}$$

La implementación de la función queda de la siguiente forma implementada con memorización.

En Top Down.

```
int lcs(int i, int j){
        if( i == 0 || j == 0) return 0;
        if(memo[i][j] != -1) return memo[i][j];
        if(u[i - 1] == v[j - 1]) return memo[i][j] = lcs(i - 1, j - 1) + 1;
        return memo[i][j] = max(lcs(i, j - 1), lcs(i - 1, j));
}
```

En bottom Up

tomamos $dpstring[i][j] = lcs(i,j)$

```
for(int i = 0 ; i<= u.length(); i++){
    for(int j = 0; j<= v.length(); j++){
        if( i== 0  || j == 0){
            dpstring[i][j] = 0;
            continue;
        }else{
            if(u[i - 1] == v[j - 1]){
                dpstring[i][j] = dpstring[i - 1][j - 1] + 1;
            }else{
                dpstring[i][j] = max(dpstring[i][j - 1],dpstring[i - 1]
[j]);
            }
        }
    }
}
```

Complejidad $\mathbb{O}(|x||y|)$

EDIT DISTANCES(Levenshtein distance) / STRING ALIGMENT

Los problemas son prácticamente idénticos por lo que los abordaremos únicamente el de edit distance y el problema de string aligment deberá ser inmediato.

Podemos definir la edit distance entre dos string x y y como el mínimo número de operaciones con las cuales se puede transformar el primer string en el segundo string.

Para que lo anterior sea concreto necesitamos definir las operaciones permitidas.

- Insertar un caracter (Ejemplo HOLA -> HOLAS)
- Remover un caracter (EJemplo HOLA -> HLA)
- Modificar un caracter (Ejemplo HOLA -> HULA)

Cada una de las operaciones tienen un costo de una operación debemos minimizar el costo de trasnformar el primer string en el segundo string.

EJEMPLO

`LOVE -> MOVIE` Lo podemos hacer a través de `LOVE -> MOVE -> MOVIE` cambiar caracter y agregar caracter. Por lo que tiene un costo de 2 unidades.

Definamos $edit(i,j)$ como la mínima cantidad de unidades necesarias para transformas $x[0..i]$ y $y[0...j]$

Notemos que las 3 operaciones anteriores se pueden representar de la forma en que:

- Insertar caracter como `x[i]` toma el valor de `y[j]`
- Insertar `y[j]` en `x`
- Borrar $x[i]$

para cada una las recuerrencias queda como

- Reemplazar el subproblema es `edit(i - 1,j - 1) + cost(i,j)` donde `cost(i,j)` es igual a cero si son iguales y uno si no.
- Si ingresamos `y[j]` en x como agragamos x sigue igual por lo que tenemos `edit(i, j - 1) + 1`
- Si eliminamos `x[i]` se quita un elemento de `x` y `y` queda igual lo cual queda `edit(i - 1, j) + 1)`.

Por lo cual queda de la forma:

$$edit(i,j) = \begin{cases} \min(edit(i-1,j-1)+1, edit(i-1,j)+1, edit(i,j-1)+1) & si \quad x[i] \neq y[j] \\ edit(i-1,j-1) & si \quad x[i] = y[i] \\ j & si \quad i = 0 \\ i & si \quad j = 0 \end{cases}$$

IMPLEMENTACIÓN TOP DOWN

```
int edit(int i, int j){
        if(memo2[i][j] != -1) return memo2[i][j];
        if( i == 0) return memo2[i][j] = j;
        if( j == 0) return memo2[i][j] = i;
        if(a[i-1] == b[j-1]) return memo2[i][j] = edit(i - 1, j - 1);
        int opcion1 = edit(i, j - 1) + 1;
        int opcion2 = edit(i - 1, j) + 1;
        int opcion3 = edit(i - 1, j - 1) + 1;
        return memo2[i][j] = min(min(opcion1, opcion2),opcion3);
}
```

IMPLEMENTACIÓN BOTTOM UP

```
for(int j=0 ; j <= b.length(); j++) dpedit[0][j] = j;
for(int i=0 ; i <= a.length(); i++) dpedit[i][0] = i;

for(int i=1 ; i<= a.length(); i++){
    for(int j=1; j<= b.length(); j++){
        if(a[i-1] == b[j-1]){
            dpedit[i][j] = dpedit[i - 1][j - 1];
        else{
            int opcion1 = dpedit[i - 1][j] + 1;
            int opcion2 = dpedit[i][j - 1] + 1;
            int opcion3 = dpedit[i - 1][j - 1] + 1;
            dpedit[i][j] = min(opcion1,min(opcion2,opcion3));
        }
    }
}
```

COMPLEJIDAD $\mathbb{O}(|x||y|)$