Algorithm 1: Space_Efficient_Dynamic_Programming_Edit_Distance

Input: X , Y

```
/* Parameter :                                                                    */
/*    x,y :  strings.                                                             */
/* Return :                                                                       */
/*    l[1,:]  :  the last line of the array created by the dynamic programming approach.  */
```

1 **begin**

```
    /* Blank letter added to X and Y corresponds to the uppest row and leftest column of the
       array ℓ.                                                                   */
```

2 X ←concat(∅,X);

3 Y ←concat(∅,X);

4 n ←len(X);

5 m ←len(Y);

6 Create Array ℓ of size [0..1 , 0..n];

7 Initialize ℓ :

```
    /* Initializing ℓ with uppest row equal to 0,1,...,n-1 and leftest column equal to
       0,1,...,m-1.  As ℓ is composed of two rows (for space efficiency), the edge case
       corresponding to the leftest column must be handelded in the main loop.    */
```

8 ℓ[0 , i] = i for i∈[0..n];

9 ℓ[1 , 0] = 1;

```
    /* Algorithm's main loop.                                                     */
```

10 **for** *i ←1 to m* **do**

11 **for** *j ←1 to n* **do**

```
        /* Edge case :  the i-nth leftest element must be equal to i              */
```

12 **if** *X[j] == Y[i]* **then**

13 ℓ[1 , j] ←ℓ[0 , j-1];

14 **else**

15 $\ell[1,j] \leftarrow 1 + \min \begin{vmatrix} \ell[0,j] \\ \ell[0,j-1] \\ \ell[1,j-1] \end{vmatrix}$

```
    /* Make uppest row equal to the lowest before starting again.                 */
```

16 ℓ[0 , :] ←ℓ[1 , :]

17 **return** ℓ[1 , :]

Algorithm 2: Backward_Space_Efficient_Dynamic_Programming_Edit_Distance

Input: X , Y

```
/* Parameter :                                                              */
/*    x,y :  strings.                                                       */
/* Return :                                                                 */
/*    r[0,:]  :  the first line of the array created by the backward dynamic programming
   approach.                                                                */
```

1 **begin**

```
   /* Blank letter added to X and Y corresponds to the lowest row and rightest column of the
      array r.                                                              */
```

2 X ←concat(∅,X);

3 Y ←concat(∅,X);

4 n ←len(X);

5 m ←len(Y);

6 Create Array r of size [0..1 , 0..n];

7 Initialize r :

```
   /* Initializing r with lowest row equal to 0,1,...,n-1 and rightest column equal to
      0,1,...,m-1.  As r is composed of two rows (for space efficiency), the edge case
      corresponding to the rightest column must be handelded in the main loop.  */
```

8 r[1 , i] = n-i for i∈[1..n+1];

9 r[0 , n-1] = 1;

```
   /* Algorithm's main loop.                                                */
```

10 **for** *i ←m-2 to -1 step -1* **do**

11 **for** *j ←n-1 to -1 step -1* **do**

```
      /* Edge case :  the (m-i)-nth rightest element must be equal to (m-i-1)  */
```

12 **if** *X[j] == Y[i]* **then**

13 r[0 , j] ←r[1 , j+1];

14 **else**

15 $$r[1,j] \leftarrow 1 + \min \begin{vmatrix} r[1,j] \\ r[1,j+1] \\ r[0,j+1] \end{vmatrix}$$

```
   /* Make lowest row equal to the uppest before starting again.            */
```

16 r[1 , :] ←r[0 , :]

17 **return** r[0 , :]

Algorithm 3: Divide_and_Conquier_Edit_Distance

Input: X , Y

```
/* Parameter :                                                                    */
/*    x,y :  strings, the two words we work with.                                 */
/* Return :                                                                       */
/*    {'p' :  p, 'ed' :  ed} dictionnary :                                        */
/*    • p :  the collection of points of interest to compute the alignement between x and y.  */
/*    • ed :  the edit distance between x and y.                                  */
1  begin
2      n ←len(X);
3      m ←len(Y);
4      if n < 1 or m < 2 then
           /* Edge cases, if (len(X) == 0,1) or (len(Y) == 0,1)                   */
5          if m == 1 and n > 1 then
6              l ←Space_Efficient_Dynamic_Programming_Edit_Distance(X,Y);
7              s ←l + [n,n-1,...,0];
8              mini ←leftest index minimizing s;
9              p = [(mini , m)];
10         else if n == 1 and m > 0 then
               /* if X is in Y then there are m-1 operations to do to go from Y to X, else there are
                  m changes to do.                                                 */
11             ed = m-1 if x in y else m;
12             p = [(X , Y[0])];
13         else
14             Handle case X empty or Y empty;
15         return { 'p' : p , 'ed' : ed };
16     else
17         ℓ ←Space_Efficient_Dynamic_Programming_Edit_Distance(X,Y[:⌊m/2⌋]);
18         r ←Backward_Space_Efficient_Dynamic_Programming_Edit_Distance(X,Y[⌊m/2⌋:]);
19         s ←ℓ+ r;
20         mini ←leftest index minimizing s;
21         ed = s[mini];
22         p = [(mini , ⌊m/2⌋)];
23
24           Concatenate(p , Divide_and_Conquier_Edit_Distance(X[:mini] , Y[:⌊m/2⌋])['p']) ;
           // top-left
25           Concatenate(p , Divide_and_Conquier_Edit_Distance(X[mini:] , Y[⌊m/2⌋:])['p']) ;
           // bottom-right
           /* concatenate :  w.r.t.  original X and Y coordinates and such that the final p is still
              sorted*.                                                             */
26         return { 'p' : p , 'ed' : ed };
           /* Post-treat p to get alignement from the points of interest.         */
           /* * :  here                                                           */
```