**X[1…k]** & **Y[1.. n]** are 2 input strings where **k ≤ n**.

// **LSS** returns thelength of the longest subsequence of **Y** that is not a super-sequence of **X**.

1. def **LSS** ( **X[1...k]** , **Y[1…n]** ):
2. if( |Y| = = 0 ) // when Y exhausts
3. return 0;
4. if ( X[1] == Y[1] )
5. {
6. if (|X| == 1) //only 1 element remaining in **X**
7. return **LSS**( **X[1...1]** , **Y[2...n]** ) //*do not consider the last element of* ***X***

*for the subsequence in* ***Y****, so that it doesn't become a supersequence of* ***X****.*

1. else
2. return **max** **{** 1 + **LSS** ( **X[2...k]** , **Y[2...n] ),**  **LSS** ( **X[1...k]** , **Y[2...n] )** **}**
3. }
4. else
5. return { 1 + **LSS** ( **X[2...k]** , **Y[2...n] ) }** // if the 1st elements of **X** & **Y** are different, then definitely consider the element in **Y** for the subsequence, as this is a maximization problem

The **LSS** function takes the arrays **X** & **Y** as input & computes the length of the longest subsequence in **Y** that is not a supersequence of **X.** At each recursive call, the first element of the 2 arrays are compared, then the function makes a decision based on the fact of their equality. Accordingly, the 2 arrays are pruned for the next call.

Since the problem is a maximization one, the method aims to add every element of **Y** to the subsequence, provided it does not violate the constraint defined above. It continues adding the elements until **Y** exhausts. The array **X** can be pruned to the extent of containing a single element only. *The algorithm is designed in such a way that if the array* ***X*** *has been reduced to a single element, it means that all its previous elements have been already included in the pursuit of finding the longest subsequence in* ***Y.***  Hence, the last element of **X** will never be included in the final subsequence of **Y.**

**Rough justification of proof:**

*Line* ***1:*** The **LSS** is called as **LSS** ( **X[1...k]** , **Y[1…n]** ).

*Lines* ***2-3:*** The above method tries to add every element of **Y** to the subsequence, provided it does not violate the constraint. So when the array **Y** exhausts, then we have already figured out the length of the longest subsequence, hence return 0.[**Base case**]

*Line* ***4*** checks if the first element of the 2 arrays are equal or not. If they are not equal, then the flow goes to line **12,** where the first element of **Y** is certainly considered for the longest subsequence. That element (i.e. character) may/may not be present later in **X,** but considering it for the longest subsequence, *at this stage,* will not violate the given constraint.

Hence, the algorithm prunes both the arrays off their first elements & calls the recursion with the reduced arrays.

*Lines* ***6-7:*** If the array **X** reduces to a single element only, then we need to be cautious about whether including it or not. As explained earlier, at this stage all the previous elements of **X** have already been included, hence we will never include that element in the final subsequence of **Y,** otherwise, it will be a supersequence of **X.** Hence, we prune only **Y** & call the recursion. Since **X** has only one element now, it is passed as **X[1…1]**.

*Lines* ***8-9:*** When the first element of the 2 arrays are equal & the array **X** has more than one elements, the algorithm need to consider 2 cases:

* Case 1: The element in **X** is considered in the pursuit of creating the longest subsequence in Y. In this case, the algorithm prunes both the arrays off their first elements & calls the recursion with the reduced arrays.
* Case 2: The element in **X** is *not* considered in the pursuit of creating the longest subsequence in **Y**. In this case, the algorithm prunes the array **Y** only & calls the recursion. The array **X** is passed unchanged.

Since, the problem is a maximization one, we choose the **maximum** of the above 2 cases for computing the length of the longest subsequence.

**Proof of Complexity:**

Let the time complexity of the **LSS** algorithm be **T(p,q),** where **p** & **q** denotes the length of the arrays **X** & **Y** respectively**.**

Lines **2-6** either takes constant time operation or is a no-op, depending on the conditionals.

Hence it takes **O(1)** time. **…(A)**

**Base Case:** When the array **Y** exhausts, then we have already figured out the length of the longest subsequence, hence return 0. The length of **X** does not matter here. Hence **T(p,0)=1.**

Line **7** makes a recursive call with the size of **Y** reduced by 1. **X** is already reduced to a single element set. Hence it takes **T(1,q-1)** time.  **…(B)**

Line **9** makes 2 recursive calls:

* One with the size of both **X** & **Y** reduced by 1. Hence it takes **T(p-1,q-1)** time.
* Another, with the size of **Y** only reduced by 1. **X** remains unchanged. Hence it takes **T(p,q-1)** time.

So the overall time taken will be **T(p-1,q-1) + T(p,q-1). …(C)**

Line **12** makes a recursive call with the size of both **X** & **Y** reduced by 1. Hence it takes **T(p-1,q-1)** time.  **…(D)**

Although the above conditional cases **(B , C & D)** are mutually exclusive, part **(A)** is common to all cases.

Hence the time complexity will be :

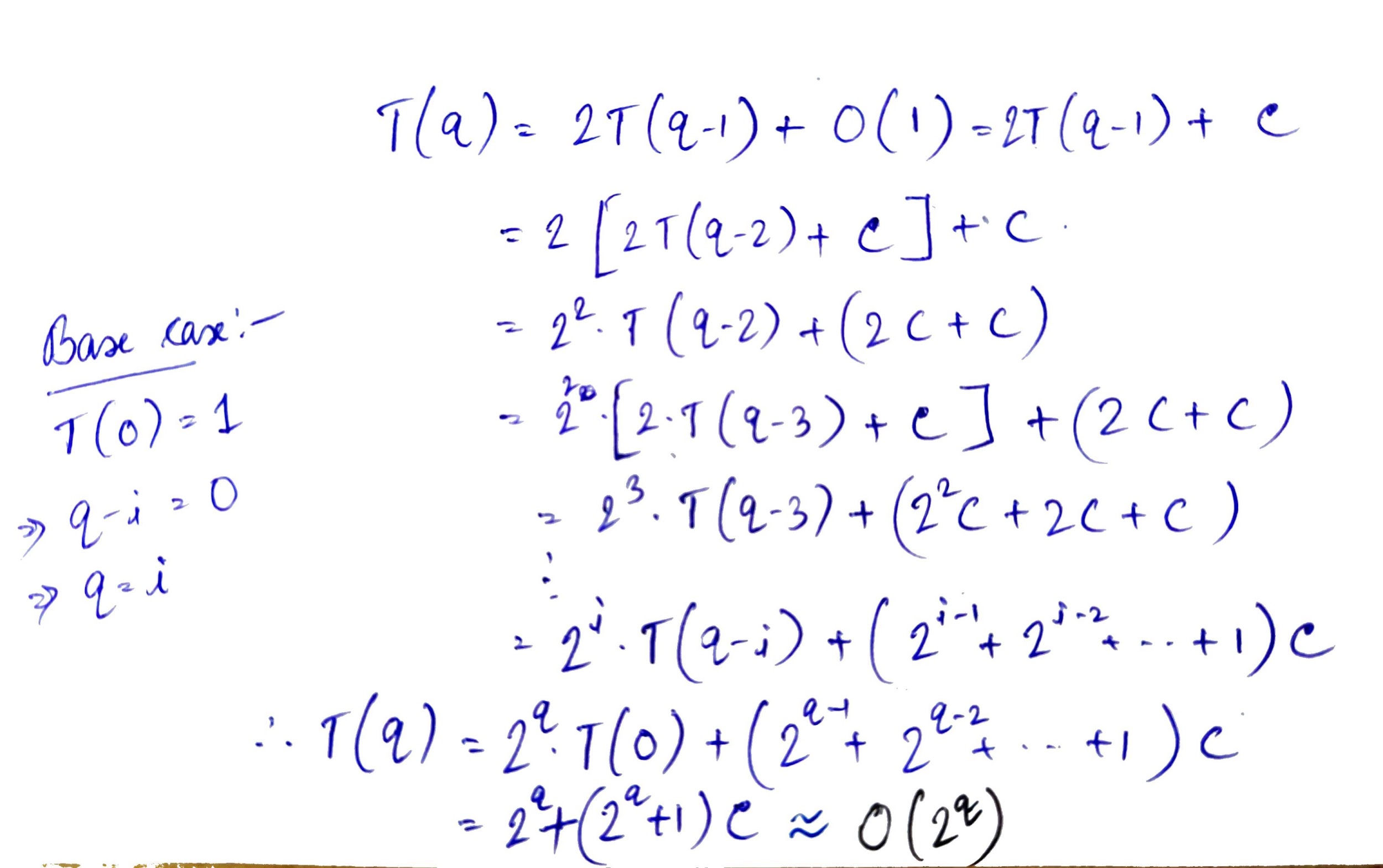
**T(p,q) = maximum { (T(1,q-1)** + **O(1)) , T(p-1,q-1) + T(p,q-1) + O(1) ) , (T(p-1,q-1)** + **O(1) ) }**

**T(p,q) = T(p-1,q-1) + T(p,q-1) + O(1),** as it is the most costliest operation in the worst case.

Since, the recursion tree for this method runs only on the size of **Y** (i.e. only on **q**)**,** and not at all on the size of **X** (i.e. not on **p**)**,** hence we can drop the variable **p** for further calculations.

Therefore, **T(q) = T(q-1) + T(q-1) + O(1) =>**

Now the base case becomes: **T(0) = 1**

****

The solution of the above recurrence is **O( 2q ).**