# Common Child

A string is said to be a child of a another string if it can be formed by deleting 0 or more characters from the other string. Given two strings of equal length, what's the longest string that can be constructed such that it is a child of both?

For example, `ABCD` and `ABDC` have two children with maximum length 3, `ABC` and `ABD`. They can be formed by eliminating either the `D` or `C` from both strings. Note that we will not consider `ABCD` as a common child because we can't rearrange characters and `ABCD` $\neq$ `ABDC`.

**Function Description**

Complete the *commonChild* function in the editor below. It should return the longest string which is a common child of the input strings.

commonChild has the following parameter(s):

- *s1, s2*: two equal length strings

**Input Format**

There is one line with two space-separated strings, $s1$ and $s2$.

**Constraints**

- $1 \leq |s1|,\ |s2| \leq 5000$
- All characters are upper case in the range ascii[A-Z].

**Output Format**

Print the length of the longest string $s$, such that $s$ is a child of both $s1$ and $s2$.

**Sample Input**

```
HARRY
SALLY
```

**Sample Output**

```
2
```

**Explanation**

The longest string that can be formed by deleting zero or more characters from $HARRY$ and $SALLY$ is $AY$, whose length is 2.

**Sample Input 1**

```
AA
BB
```

**Sample Output 1**

```
0
```

**Explanation 1**

$AA$ and $BB$ have no characters in common and hence the output is 0.

**Sample Input 2**

```
SHINCHAN
NOHARAAA
```

**Sample Output 2**

```
3
```

**Explanation 2**

The longest string that can be formed between $SHINCHAN$ and $NOHARAAA$ while maintaining the order is $NHA$.

**Sample Input 3**

```
ABCDEF
FBDAMN
```

**Sample Output 3**

```
2
```

**Explanation 3**

$BD$ is the longest child of the given strings.

Editorial by fionalmatters

Let's discuss the example of two anagrams

**Example Case**

```
TERRACED
CRATERED
```

This makes it a little more compact. What we are basically trying to do here is to find the length of the longest common subsequence of these two strings while maintaining order (which is why the anagram case isn't just trivial.)

I find the easiest way to visualise this is to form a grid with the first string along the top and with the other string along the side.

**Grid Form**

```
  T E R R A C E D
C - - - - - - - -
R - - - - - - - -
A - - - - - - - -
T - - - - - - - -
E - - - - - - - -
R - - - - - - - -
E - - - - - - - -
D - - - - - - - -
```

You then simply sweep along from top left to bottom right searching for matches and recording any that you do find

**Row 1**

```
  T E R R A C E D
C - - - - - C C C
R - - - - - - - -
A - - - - - - - -
T - - - - - - - -
E - - - - - - - -
R - - - - - - - -
E - - - - - - - -
D - - - - - - - -
```

Once you find a match you continue that to the end so that it can be used later

**Row 2**

```
  T E R R A C E D
C - - - - - C C C
R - - R R R R R R
A - - - - - - - -
T - - - - - - - -
E - - - - - - - -
R - - - - - - - -
E - - - - - - - -
D - - - - - - - -
```

Here we can only use one `R` as we only have one available in `CR` (although there are two by this point in `TERR`). To account for this I either use the information to the left or the information from above plus my current character. This prevents clashes from the right. For the last 3 letters in row two we could have used either `C` or `R`.

This method effectively stops us from trying to use `CR` as the `C` is not available to the top or left at the point which we match the `R`

**Row 3**

```
  T E R R  A  C  E  D
C - - - -  -  C  C  C
R - - R R  R  R  R  R
A - - R R  RA RA RA RA
T - - - -  -  -  -  -
E - - - -  -  -  -  -
R - - - -  -  -  -  -
E - - - -  -  -  -  -
D - - - -  -  -  -  -
```

We now have a string of length 2 and could no longer have included the `C` as it comes after both `R` and `A` in `TERRACED`

**Row 4 & 5**

```
  T E  R  R  A  C  E  D
C - -  -  -  -  C  C  C
R - -  R  R  R  R  R  R
A - -  R  R  RA RA RA RA
T T T  R  R  RA RA RA RA
E T TE TE TE RA RA RAE RAE
R - - - -  -  -  -  -
E - - - -  -  -  -  -
D - - - -  -  -  -  -
```

We are now up to a string of length 3 - note that we could not have `TEE` as that would use the `E` twice - looking to the left

**Row 6**

```
  T E  R   R   A   C  E   D
C - -  -   -   -   C  C   C
R - -  R   R   R   R  R   R
A - -  R   R   RA  RA RA  RA
T T T  R   R   RA  RA RA  RA
E T TE TE  TE  RA  RA RAE RAE
R T TE TER TER TER TER TER TER
E - -  -   -   -   -  -   -
D - -  -   -   -   -  -   -
```

We now have a different string of length 3. We will be extending it shortly. It doesn't actually matter as the question only asks for the length of a string but it is generally better to favour strings from the left over those from above as it gives you more options as you go down.

**Row 7**

```
  T E  R   R   A   C   E    D
C - -  -   -   -   C   C    C
R - -  R   R   R   R   R    R
A - -  R   R   RA  RA  RA   RA
T T T  R   R   RA  RA  RA   RA
E T TE TE  TE  RA  RA  RAE  RAE
R T TE TER TER TER TER TER  TER
E T TE TER TER TER TER TERE TERE
D - -  -   -   -   -   -    -
```

This is an interesting line as it shows how to cope with clashes from above. In this case when we hit the first `TE` from above we can not add our `E` to it as it is already being used for the first `E`. If we have a match we actually have to add to the cell above and to the left (unless we are at the first column, it which case we can only have a match one letter long). Otherwise we will use more letters than we have. By the time we come to the second `E` we are fine to add it however and now have a string of length 4.

**Final Row**

```
  T E  R   R   A   C   E    D
C - -  -   -   -   C   C    C
R - -  R   R   R   R   R    R
A - -  R   R   RA  RA  RA   RA
T T T  R   R   RA  RA  RA   RA
E T TE TE  TE  RA  RA  RAE  RAE
R T TE TER TER TER TER TER  TER
E T TE TER TER TER TER TERE TERE
D T TE TER TER TER TER TERE TERED
```

On processing our final row we have the result. TERED - 5 letter common string. The answer appears in the bottom right cell. While this does work it would not be efficient for large strings. It is however easy to convert to a length count.

**Converting to Length Counts**

In this case we fill the array with zeros and on a match we take the maximum of either the value in the cell to the left or the cell above left plus one (for our match). If there is no match we simply take the maximum of the cell to the left and the cell above.

Our example therefore becomes

**Number Version**

```
  T E R R A C E D
C 0 0 0 0 0 1 1 1
R 0 0 1 1 1 1 1 1
A 0 0 1 1 2 2 2 2
T 1 1 1 1 2 2 2 2
E 1 2 2 2 2 2 3 3
R 1 2 3 3 3 3 3 3
E 1 2 3 3 3 3 4 4
D 1 2 3 3 3 3 4 5
```

A little harder to follow but much more concise.

My code for this is in Python - note that it actually times out in the python interpreter which really doesn't like for loops. However it runs in time with the PyPy interpreter. So I've felt justified to add it here.

I also don't usually dump code in one big file (well not huge) - but it did cut the time down significantly to not have the dp as a class variable.

In order to save space I have simply used two rows for the matrix. This is fine as we only ever look up one row anyway. This is often true in dynamic programming and can save a significant amount of space. (Say in a case where the string lengths are 5000)