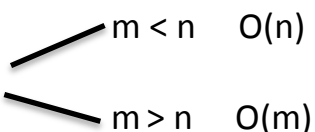


Task3:

First I decided to create an hashtable to store the dictionary, because search and insert time complexity is $O(1)$. Storing chars from the dictionary into the hashtable would cost $O(1*n) = O(n)$ (n is number of chars in dictionary), then went to through each char in the document which is $O(m)$ (m is the number of elements in document), and for each char we search in hashtable to find it. Since in each list in hashtable we can have couple of char stored and the average cost of a lookup depends only on the average number of char per list search can take up to $O(m)$. $O(m)$ is the worst case when all entries are stored in one list. But since we've chosen the hashtable to be as large as the dictionary, numbers of chars that in average are gonna be in a linked list is quiet small in comparison to other variables hence we can consider it $O(1)$. So search in dictionary for each document is $O(m)$.

Total time : $O(n) + O(m)$ 

Task4:

(to avoid confusion length of words in average is mentioned as "t")

Because the length of a string is probably quite small in comparison to other available variables so we ignore string lengths. Therefore hash operations will be $O(1)$.

I inserted the dictionary in hashtable as explained above it took $O(n)$. also created the hashtable for edits to store all the edits made through the process which took $O(n)$. Then went through document and first searched the hashtable of dictionary chars and again as explained above took $O(1*m)$.

If word is not in dictionary, first make all edits by performing one deletion, one substitution and one insertion function. In the function made for this operations we look up each made edit in dictionary and then add the edit that appears first to our hashtable for edits. Based on the explanation above about length of strings the time complexity for each of these function would be $O(1)$ and for m chars its $O(m)$

Then to optimize my code we check the hashtable for edits if for a query there isn't any one-edit away edits we calculate the 2 edit distance away edits. This would reduce the number of edits made by eliminating the ones that have already found their corrected versions. This is much faster compare to method of creating 2 edit distance away edits for all queries.

Then to even optimize the code more I calculated the number of 2 edit distance away edits based on the length of the string, if number of edits is less than the size of the dictionary we create them otherwise its faster to look through the dictionary. This makes huge difference cause for long words the number of edits made are much bigger than the dictionary.

The functions to create 2 edit distance away edits cost $O(t^2)$ each cause string length matters and wont be assumed to be $O(1)$. So if char's 2 edit distance away edits in total are less than size of dictionary its time complexity would be $O(t^2)$ for each query .

But if the number of edits are bigger than dictionary we need to look through dictionary and it'd be $O(n)$.

We repeat the same thing for 3 edit distance. We check the hashtable for edits, if for a query there isn't any one-edit away, or 2 edit away edits we calculate the 3 edit distance edits.

Based on length of strings we calculate the total number of edits. If its less than size of dictionary like previous section it'll be $O(t^2)$ if its more it'll be $O(n)$.

Diagram illustrating the breakdown of time complexity for edit distance calculations:

- Insert dic in hashtable** points to $O(n)$
- one edit-dis edits** points to $O(m * 1)$
- 2-edit distance** points to $O(t^2 * m)$ or $O(n * m)$
- 3-edit distance** points to $O(t^2 * m)$ or $O(n * m)$
- Checking hashtable to find the char** points to $O(m)$
- Creating edits** points to $O(t^2 * m)$ or $O(n * m)$
- checking dic** points to $O(n * m)$

$$O(n) + [O(m * 1) + O(m) + O(t^2 * m) \text{ or } O(n * m) + O(t^2 * m) \text{ or } O(n * m)]$$

$$= O(n) + [O(m) + O(t^2 * m) \text{ or } O(n * m) + O(t^2 * m) \text{ or } O(n * m)]$$