Task 5 Designs of Algorithms

D = dictionary size, **Q** = document size, **P** = ratio of duplicated rate in document, **N** = average length of strings in both lists, **R_x** = ratio of fail to find matched strings for distance x in task 4 hash table;

Generally, there are three ways to complete task 3 & 4: using linear searching to get Levenshtein distance between words in dictionary and documents one by one, using hash table to process if words or its distanced words are in the dictionary, OR combining two methods according to the size of dictionary D and size of the hash table (for task 4).

Assume for every stage to find matched string in the distance x. There is ($1-R_x$) percentage of words successfully found its matched strings in distance x and print the result, and R_x percentage found no matched string at these distance and gone further. Assume P percentage of string in documents has tested before and in the history. Since average length of strings is N. Assume length of string is O(N), so string compare is O(N). Therefore **Time complexity for hash table is:**

Hash table has: O(N) Hash table put: O(N) Hash table get: O(N)

Task 3:

(1) Linear searching:

1. Loop every string in the document **loop Q times**
2. Loop every string in the dictionary **loop D times**
3. Compare every string in the document and find if the string occurs in the dictionary. **O(1)**

Total: O(Q*D*N).

(2) Hash table:

1. Put all strings in dictionary into a hash table **D*O(N)**
2. Loop every string in document **loop Q times**
3. Use hash table has to find if there is a same string in hash table **O(N)**

Total: O(D*N+Q*N) = O(N*(D+Q))

Conclusion: Obviously **O(Q*D*N) > O(N*(D+Q))** so hash table runs much quicker than linear searching and is a better method.

Task 4:

(1) Linear searching:

1. Loop every string in document **loop Q times**
2. Build history hash table to store tested strings. **O(1)**
3. Loop every string in dictionary **loop D times**
4. If the string is in history hash table. Directly print answer. **P*O(N)**
5. Else: **(1-P)**
6. Calculate distance between strings in dictionary and in document if its difference is in 0 to 3, store it. Then from distance 0 to 3 see if they have matched strings and print it out and record it into history hash table **$O(N^2)+O(N)$**

   Total: $O(Q*D*(P*N+(1-P)(N^2+N))$
   $= O(Q*D*N^2)$

(2) Hash table

1. Put all strings in dictionary into a dictionary hash table **D*O(N)**
2. Build history hash table to store tested strings. **O(1)**
3. Loop every string in the document **Loop Q times**
4. If the string is in history hash table. Directly print answer. **P*O(N)**
5. Else: **(1-P)**
6. Use hash table has to find if there are same strings in the dictionary hash table **O(N)**
7. If there are not, build hash table to store all distance-1 strings for failed strings in document. **$R_0*N*O(N)$**
8. Use hash table has to find if there are matched distance-1 strings in dictionary hash table, print it and record it in hash table history. **$R_0*N^2*O(N)$**
9. If there are not, build hash table to store all distance-2 strings for failed strings in document. **$R_1*N^2*O(N)$**
10. Use hash table has to find if there are matched distance-2 strings in dictionary hash table, print it and record it in hash table history. **$R_1*N^3*O(N)$**
11. If there are not, build hash table to store all distance-3 strings for failed strings in document

R_2*N^3*O(N)

12. Use hash table has to find if there are matched distance-3 strings in dictionary hash table, print it and record it in hash table history. **R_2*N^4*O(N)**
13. If there are not, print string with "?" and record in hash table history. **R_3*O(1)**

Total: $O(D*N+Q*(P*N+(1-P)*(N+R_0*N^2+2R_1*N^3+2R_2*N^4+ R_2*N^5+R_3)))$
$= O(D*N+Q*N+Q*R_0*N^2+Q*R_0*N^3+Q*R_1*N^4+Q*R_2*N^5+R_3)$
$= O(D*N+Q*N^5)$

(3) Combination of (1) & (2):

1. Put all strings in dictionary into a dictionary hash table **D*O(N)**
2. Build history hash table to store tested strings. **O(1)**
3. Loop every string in the document **Loop Q times**
4. If the string is in history hash table. Directly print answer. **P*O(N)**
5. Else: **(1-P)**
6. Use hash table has to find if there are same strings in the dictionary hash table **O(N)**
7. If there are not, build hash table to store all distance-1 strings for failed strings in document. **R_0*N*O(N)**
8. Use hash table has to find if there are matched distance-1 strings in dictionary hash table, print it and record it in hash table history. **2R_0*N*O(N)**
9. If there is not, compare dictionary size with size of distance-2 hash table. **O(1)**
10. If hash table is larger, directly use linear searching to store first distance 2 and 3 strings, if distance 2 string is recorded, print and store in history, else if distance 3 string is recorded, print and store in history, else print original words with "?" and recorded in history hash table. **R_1*O(D*N^3)**
11. If dictionary size is larger, use hash table has to find if there are matched distance-2 strings in dictionary hash table, print it and record it in hash table history. **R_1*N^2*O(N)**
12. If there is not, use linear searching to store first 3 strings, if distance 3 string is recorded, print and record in history, else print original words with "?" and record in history hash table. **R_2*O(D*N^3)**

Total: $O(D*N+Q*(P*N+(1-P)*(N+2*R_0*N^2+R_1*D*N^3+R_1*N^3+R_2*Q*D*N^3)))$
$= O(D*N + Q*N + Q*R_0*N^2+Q^2*R_1*D*N^3+Q*R_1*N^3+Q^2*R_2*D*N^3)$
$= O(D*N + Q*D*N^3)$

Conclusion: according to the calculated time complexity, if we assume length is O(1) and all action under hash table and string are O(1), linear searching is O(Q*D), hash table is O(Q+D), and combination is O(D+Q*D). It looks like combination has highest time complexity since it add a lot of improving functions. However, under different situations, combination will have different performance. When size of distance-2 hash table is larger than size of dictionary, $N^2 > D$, linear search will be faster than hash table since $O(D*N+Q*N^5) > O(D*N+Q*D*N^3)$, than combination will be quicker than hash table. This is especially useful when dictionary is small such as 100K and 250K. When D is much larger than other list and length, D >> L. N, $O(Q*D*N^2) > O(D*N+Q*D*N^3)$ since larger number is better to be added than multiplied. As conclusion, combination will perform as hash table when D is very large and as linear search as D is small, always choose a better way to get lower time complexity. Therefore I choose the combination method.

Space manipulating: for both tasks linear searching only compares on two lists without using other space, but hash table will also need space to save buckets for hash functions. However, both spaces used by lists and buckets are directly related to document and dictionary size O(Q) and O(D), so they have the same increasing rate and the difference does not increase as size increases. In a word, they have the same big O for space but hash tables always need more space than linear searching.

Reality Situation: To test the validity of algorithm of task 4. I compile three methods independently and run their time for document jabberwocky.txt under all dictionaries. The average consuming time is:

	100K	250K	1M	5M
(1) Linear searching	0.2sec	1.4sec	25.3sec	80.1sec
(2) Hash table	4.8sec	2.7sec	8.9sec	22.8sec
(3) Combination	0.2sec	0.3sec	2.4sec	5.4sec

Which best illustrates my induction.