Design and Analysis of Algorithm for finding meaningful words in a Dictionary

Nairit Banerjee
IIT2016505

Kamal Nayan Chaubey
IIT2016510

Ashutosh Vishwakarma
IIT2016017

Mandeep Chakma
IIT2016012

February 14, 2018

Abstract

This document is a final report of our study on finding meaningful words in a Dictionary from a given set 7 of randomly generated characters. We are given a set of seven ranomly generated characters and we have to find how many meaningful words can be formed from them of length 2 to 7.

Keywords: *Next permutation,Rand function, Time Complexity, Space Complexity, Algorithm Design, Graphical Analysis,String Search and Algorithm*

1 Introduction And Literature Survey

In mathematics and computer science, the process of seraching a string in a given set of string is known as String Searching.for example,suppose 7 randomly generated characters are b,i,i,a,l,o,n,m. Let us form a word of length 7. There will be 7!/2! ways to make words from above.We can see that it forms "Binomial" a meaningful word in dictionary. Thus we have a string "Binomial" and we have to find whether it is present in a Dictionary.Let Dictionary = ["in" , "at" , ... , "Multiply", "Factor", "Binomial" ,"Component" , ...] after searching ,the result will be the pointer to its location which in above case will be 3. In same way other words that can be formed from above set of characters are - "an" , "in" ,"man", "bin", "loan" etc. Thus this is the centre idea of our Assignment.

2 Algorithm Design

The first task in Algorithm design to be done is to produce the various permutations possible from the given letters. The algorithm description for the problem is as follows:

(1) procedure NextPermutation

1. Find largest index i such that arr[i-1]<arr[i]. (If no such i exists, then this is the last permutation).
2. Find largest index j such that $j \geq i$ and arr[j]>arr[i-1]
3. Swap arr[j] and arr[i-1]
4.Reverse the suffix starting at arr[i]

(2) procedure PossibleWord

1. We go through the loop from 1 to 2^7-1 and then for each number that comes in between, we have their binary representation consisting of only 0's and 1's.

2. Now for each number we go through their binary representation and check the position of bitset(that is the position where the binary digit is equal to 1) in the binary representation of the respective number.

3. Now the positions at which bit is set represents that, at present we are using those position characters from the randomly generated character string, for forming our different words using nextPermutation procedure.

4. As the words are generated, they are checked whether they are previously generated or not using the function check.If previously generated then we avoid it,Else we search it using search function.

2.1 Pseudo Code for producing the possible words

Algorithm 1 Produce Permutation

```
1: procedure NextPermutation(arr, l)
2:     i ← l
3:     while i > 0 and arr[i − 1] ≥ arr[i] do
4:         i ← i − 1
5:     end while
6:     if i ≤ 0 then return false
7:     end if
8:     j ← l
9:     while arr[j] ≤ arr[i − 1] do
10:        j ← j − 1
11:    end while
12:    temp ← arr[i − 1]
13:    arr[i − 1] ← arr[j]
14:    arr[j] ← temp
15:    j ← l
16:    while i < j do
17:        temp ← arr[i]
18:        arr[i] ← arr[j]
19:        arr[j] ← temp
20:        i ← i + 1
21:        j ← j − 1
22:    end while
23:    return true
24: end procedure
25: procedure PossibleWord(str, l)
26:    x ← 1 ≪ 7
27:    i ← 1
28:    while i < x do
29:        k ← 0
30:        j ← 0
31:        while j < 7 do
32:            if (i ≫ j)and1 then
33:                arr[k] ← str[j]
34:                k ← k + 1
35:            end if
36:            j ← j + 1
37:        end while
38:        arr[k] ← NULL
39:        l ← length of arr
40:        if l ≠ 1 then sort(arr)
41:            while nextp(arr,l) ≠ 0 do
42:                if check(arr) ≠ 1 then
43:                    if search(arr) ≠ 0
   then
44:                        Word Exist
45:                    else
46:                        Word do not exist
47:                    end if
48:                end if
49:            end while
50:        end if
51:    end while
52: end procedure
```

Algorithm 2 Check If Best

```
1: procedure checkIfBest(str, l)
2:     i ← 1
3:     count ← 0
4:     while i < l do
5:         if str[i] == str[i − 1] then
6:             count ← count + 1
7:         end if
8:     end while
9:     if count == l − 1 then
10:        No possible dictoinary Word
11:        end
12:    end if
13: PossibleWord(str, l)
14: end procedure
```

2.2 Pseudo code to Search for Dictionary Words

After we have produced all the possible permutations of the given letters, now we need to search for how many of these are dictionary words. For this we have used owlbot's Definition API and libcurl library to send and receive HTTP and JSON data to and fro the owlbot server.

Algorithm 3 Search

```
1: procedure Search(arr)
2:     curl ← curl_easy_init()
3:     str1 ← "https : //owlbot.info/api/v2/dictionary/"
4:     str2 ← malloc(strlen(s) + strlen(data) + 1)
5: str2[0] = NULL
6: strcat(str2,str1)
7: strcat(str2, arr)
8:     if curl == 1 then
9:         curl_easy_setopt(curl, CURLOPT_URL, str)
10:        curl_easy_setopt(curl, CURLOPT_FOLLOC, 1L)
11:        res ← curl_easy_perform(curl)
12:        if res ≠ CURLE_OK then
13:            stderr, curl_easy_perform()failed : curl_easy_strerror(res)
14:        end if
15:        curl_easy_cleanup(curl)
16:    end if
17: end procedure
```

3 Analysis and Discussions

3.1 Time Complexity

The time complexity for the above stated algorithm can be computed by adding units of time taken by each statement multiplied by the number of times the statement is executed.

Hence,

$T = \Sigma(cost(i) * no.of.times(i))$,for all statements

The running time will be dependent on the number of letters given(n) and also the repetition of individual letters. We found that

$$T_{best} \propto c_1 n + c_2$$

$T_{average} \propto c_1 n + c_2 2^n (n^2 + nlog(n) + (n/c_3)!(n)) + c_4$

$T_{worst} \propto c_1 n + c_2 2^n (n^2 + nlog(n) + (n)!(n)) + c_5$

Hence the order of growth for best case, average case and worst case are

$\Omega(n), \Theta(2^n.n!), O(2^n.n!)$ respectively.

Hence we observe that if all letters are same, it will result in best case and permutations will be produced in linear time. Whereas if all letters are not identical the average and worst cases are identified by number of repetitions of letters. In fact, for worst case none of the letters should be repeated.

For the dataset,

Table 1: Data for plotting running time graph

n	T_{best}	T_{avg}	T_{worst}
3	18	710	62952
4	24	4178	344175
5	30	127353	2403699
6	36	387298	28114280
7	42	11378805	756550762

We will get the following time complexity graph for n ranging from 3 to 5

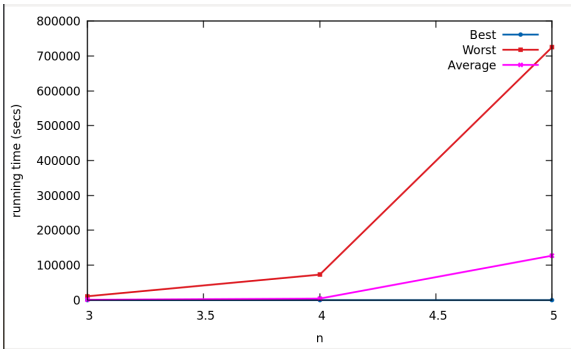

Figure 1: Running time graph.

3.2 Space Complexity

Suppose the number of alphabets randomly generated be n, then the space complexity for the algorithm is linear and is related as follows:

$$Space(n) \propto (2^n * n!)$$

4 Experimental Study

We may assume the following examples to study the various running times possible,

Best Case: The best case will be when all the randomly generated alphabets are same.In this case we do not need to generate all possible combination and permutation of these letters.Hence in this case we clearly return that **No possible word exist**.

Average Case: When some of the letters are repeated then we need to generate all the possible combination and permutation of those letters to generate different words and hence find out the validity of word in dictionary.For example a, a, a, b, l, l, o.

Worst Case:In this case no any letter is repeated and hence it takes the highest computation time to compute all the combinations and permutations of the letters and then search the word in dictionary to check whether the word is present in dictionary or not.For example suppose we take only 3 letters as a,b,c then all the posssible combination and permutation will be as follows:
ab,ba,ac,ca,bc,cb,abc,acb,bac,bca,cab,cba
Thus we can claim that the running time depends on the variation in the types of alphabets randomly generated. More distinct the aplhabets generated more will be the computation time to form every possible permutation.

5 Conclusion

In this paper we proposed an algorithm to generate all possible combination and permutation of given set of randomly generated alphabets and then find the word formed using those letters in the online dictionary using libcurl and hence to find whether how many words formed , are dictionary words in $\propto O(2^n * n!)$ and exponential space complexity. Here specifically the value of n is given equal to 7. However experimental study showed that the time complexity for best,average and worst case differs only on the variation in the types of alphabets randomly generated. More the distinct alphabets generated, more will be the computation time.

Hence practical implications are: Communication networks, cryptography and network security. Permutations are frequently used in communication networks and parallel and distributed systems.

References

[1] Introduction To Algorithms (Second Edition) by Thomas H.Cormen, Charles E.Leiserson , R.L.Rivest and Clifford Stein.

[2] How to Solve it by Computer by R.G.Dromey.

[3] The C Programming Language by Brian Kernighan and Dennis Ritchie.

[4] https://www.geeksforgeeks.org/

[5] https://www.stackoverflow.com/

[6] https://curl.haxx.se/libcurl/