CS301

2022-2023 Spring

Project Report

Group 105

Eren GÜNGÖR

1. Problem Description

Describe your problem both as intuitively and formally. If possible, talk about the applications (where this problem) might be used in practice.

State the hardness of your problem in the form of a theorem. For example, give a theorem claiming that your problem is NP-complete, or NP-hard.

For the proof of this theorem, you can simply cite (refer to) an appropriate source in the literature which gives this proof, or you can give an explicit proof in your report. In the case you give an explicit proof, and this proof is not a novel proof suggested by your group (which is ok for your report), you must still give the citation to the original paper/book from which you got this proof.

Intuitional: Given some strings, we need to determine whether it is possible to find the smallest superstring that contains all the strings as a substring.

Formal: Given a set of strings S = {s1, s2, …, sn}, Shortest Common Superstring problem aims to find the shortest possible string that contains all the strings in S as substrings.

Hardness: The Shortest Common Superstring problem is known to be NP-complete, which means that there is no known algorithm that can solve the problem in polynomial time.[1]

Applications: A few application examples would be:

1- It is used in **DNA Sequencing** to reconstruct the original DNA sequence from a set of short DNA fragments. It also has important applications in genetics and biotechnology such as genetic disorder diagnosis and treatment.

2- It is used in **Text Compression Algorithms**, which aims to compress large amounts of text into smaller representations. SCS facilitates this algorithm as it allows us to compress the input text without losing any information.

3- It is used in **Computer Networking** to optimize data transfer between different devices. With SCS, the amount of data that needs to be transmitted is minimized, and performance increases.

[1] Esko Ukkonen, "The Shortest Common Supersequence Problem over Binary Alphabet Is NP-Complete," Theoretical Computer Science, February 24, 2015, https://www.academia.edu/11055618/The_shortest_common_supersequence_problem_over_binary_alphabet_is_NP_complete, 189-191.

2. Algorithm Description

a. Brute Force Algorithm

Find a correct/exact/brute force algorithm for your problem and explain the algorithm in detail (Exponential/Factorial Time, Not Efficient). If you want, you can also design an algorithm yourself. Also give a pseudo-code of the algorithm. If the algorithm follows an algorithm design technique, e.g., divide-and-conquer, dynamic programming, etc., you need to mention this and explain why you think the algorithm is using this technique. (Guaranteed solution no matter the computational complexity.)

The **pseudo-code** for the **brute force** algorithm:

```
function find_shortest_common_superstring(strings):
        shortest_superstring = None
        for perm in permutations(strings):
                superstring = perm[0]
                for i in range(1, len(perm)):
        overlap = find_overlap(superstring, perm[i])
        superstring += perm[i][overlap:]
        if shortest_superstring is None or len(superstring) < len(shortest_superstring):
        shortest_superstring = superstring
        return shortest_superstring

        def permutations(strings):
          if len(strings) == 0:
            return [[]]
          result = []
          for i in range(len(strings)):
            rest = strings[:i] + strings[i+1:]
            for perm in permutations(rest):
              result.append([strings[i]] + perm)
          return result

        def find_overlap(s1, s2):
          max_overlap = min(len(s1), len(s2))
          for i in range(max_overlap, 0, -1):
            if s1[-i:] == s2[:i]:
              return i
          return 0
```

This algorithm simply generates all the permutations of substrings, then concatenates them together, and returns the resulting shortest superstring. Since it is a brute force algorithm, it does not follow any algorithm design techniques.

b. Heuristic Algorithm

Find an approximate/heuristic algorithm (Polynomial Time, Efficient) for your problem and explain the algorithm in detail. If you want, you can also design an algorithm yourself. Also give a pseudo-code of the algorithm. If the algorithm follows an algorithm design technique, e.g., divide-and-conquer, dynamic programming, etc., you need to mention this and explain why you think the algorithm is using this technique.

In addition, if the algorithm is an approximation algorithm, you need to state and show the proof of the ratio bound. The proof need not be a novel proof that your team put forward, although this is perfectly okay. You can just include a proof that already exists in the literature, by citing the appropriate source.

The algorithm above follows divide & conquer strategy as it is composed of 2 parts: preprocessing, and construction of H. It first solves the problem of creating an AC machine, then operates for finding the superstring.

ALGORITHM 1: PREPROCESSING

Input: Set R = {x1, …, xm} of strings, and the AC machine consisting of the goto function g and the failure function f for R.

Output: Depth d(s), list L(s), and link b(s) for each states of the AC machine; pointer B to the first state in the b-link chain; state F(i) representing string xi for each xi with the exception that if xi is a substring of another string in R then F(i) = 0.

Notation: Operator . denotes list catenation. An inverse of F is represented by E: if F(i) = s, then E(s) = i; initially E(s) = 0 for each states.

```
for i ← 1 to m do
        let xi = a1 … ak
        s ← 0
        for j ← 1 to k do
                s ← g(s, aj)
                L(s) = L(s) . {j}
```

```
if j = k then
        F(i) ← s
        E(s) ← i
        if s is not a leaf of the AC machine
                then F(i) ← 0
        fi
fi
    od
od
queue ← 0
d(0) ← 0
B ← 0
while queue != empty do
    let r be the next state in queue
    queue ← queue – r
    for each s such that g(r, a) = s for some a do
        queue ← queue . s
        d(s) ← d(r) + 1
        b(s) ← B
        B ← s
        F(E(f(s))) ← 0
    od
od
```
[2]

ALGORITHM 2: CONSTRUCTION OF H

Input: Augmented AC machine for R, as constructed by Algorithm 1.

Output: A Hamiltonian path H in the overlap graph of reduced R. As explained in Section 2, a common superstring for R can then be constructed by forming p(H).

Let initially each P(s) be empty.

```
for j ← 1 to m do
    if F(j) != 0 then P(f( F(j) ) ) *- P(f( F(j) ) ) . {j}
        FIRST(j) ← LAST(j) ← j
        else forbidden(j) ← true
    fi
od
s ← b(B)
while s != 0 do
```

2 Esko Ukkonen, "A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings – Algoritmica" (SpringerLink, 1990), 318-320.

```
        if P(s) is not empty then
                for each j in L(s) such that forbidden(j) = false do
                        i ← the first element of P(s)
                        if FIRST(i)=j then
                                if P(s) has only one element then goto next
                                else i ← the second element of P(s)
                        fi
                        H ← H . {(xi, xj)}
                        forbidden(j) ← true
                        P(s) ← P(s) ← {i}
                        FIRST(LAST(j)) ← FIRST(i)
                        LAST(FIRST(i)) ← LAST(j)
                next:
                od
                P(f(s)) ← n(f(s)) . n(s)
        fi
        s ← b(s)
od[3]
```

3. Algorithm Analysis

a. Brute Force Algorithm

- *Claim and show that the algorithm works correctly, possibly in the form of a theorem*
- *For the complexity analysis, drive the worst-case time complexity. Try not to give upper bounds which are too loose. If possible, try to give a tight upper bound by using **Θ**.*
- *Optionally, you can also consider the complexity of the algorithm for resources other than time, e.g., the space complexity.*

To prove the correctness of this algorithm, we can state the following theorem:

Theorem: The brute force algorithm for the shortest common superstring problem always returns the shortest common superstring of the input strings.

[3] Esko Ukkonen, "A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings – Algoritmica" (SpringerLink, 1990), 320-321.

Proof: Suppose there exists a set of n strings S = {s1, s2, …, sn} and let SC be the shortest common superstring of S.

Let p be the permutations of S such that the strings in p are concatenated in the order they appear in SC.

Since SC is the shortest common superstring, the length of SC is less than or equal to the length of any other common superstring of S. Therefore, the length of the superstring generated by concatenating the strings in p cannot be shorter than the length of SC. However, since the brute force algorithm generates all permutations of S, it generates the permutation p and concatenates the strings in p to form a superstring. Therefore, the brute force algorithm generates a superstring SC. Since this holds for all sets of input strings, we can conclude that the brute force algorithm always returns the shortest common superstring of the input strings.

Time-Complexity: The worst-case time complexity of this algorithm is O(n!), where n is the number of input strings. This is because finding all the permutations takes O(n!) time in worst-case, then finding overlaps is done and takes O(n) time for each pair of strings. Therefore, the overall complexity becomes O(n! * n), which is simply O(n!). Additionally, since all the permutations are generated in any case, lower bound is also Ω(n!). Thus, the time complexity of the algorithm is **ϴ**(n!) in general.

b. Heuristic Algorithm

- *Claim and show that the algorithm works correctly, possibly in the form of a theorem*
- *For the complexity analysis, drive the worst-case time complexity. Try not to give upper bounds which too loose. If possible, try to give a tight upper bound by using* ***ϴ****.*

- *Optionally, you can also consider the complexity of the algorithm for resources other than time, e.g., the space complexity.*

The algorithm above is a slightly modified version of Aho Corasick string-matching automaton, which is proved to be accurate. Further details regarding the correctness of this algorithm are given by Ukkonen[4].

The algorithm runs in O(n) for small alphabets, and O(n . min(log m, log|Σ|)) for arbitrary alphabets as indicated and proved by Ukkonen[5].

4. **Sample Generation (Random Instance Generator)**
   - *Implement/find a parametric (in terms of the size of the problem) algorithm to produce random sample inputs for your problem.*
   - *Put your implementation/pseudo codes and explanation of the algorithm to your reports.*

Input:

1) n: number of strings in the input set
2) min_len: the minimum length of each string
3) max_len: the maximum length of each string
4) alphabet: the set of characters that can appear in each string

Output: A set of n random strings.

Random Sample-Generation Algorithm:

1) Initialize an empty set S of n strings.
2) For i = 1 to n:
   a. Generate a random length l between min_len and max_len
   b. Initialize an empty string s of length l
   c. For j = 1 to l:
      i. Choose a random character from the alphabet

[4] Esko Ukkonen, “A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings – Algoritmica” (SpringerLink, 1990), 314-318.
[5] Esko Ukkonen, “A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings – Algoritmica” (SpringerLink, 1990), 321-322.

ii. Append c to s

d. Add s to S

3) Return the set S

This algorithm generates n random strings of random lengths between min_len and max_len with each character selected randomly from the given alphabet.

Code in Python for Random Sample-Generator:

```
import random
import string

def generate_random_input(n, min_length, max_length, alphabet):
  S = []
  for i in range(n):
    l = random.randint(min_length, max_length)
    s = ''
    for j in range(l):
      c = random.choice(alphabet)
      s += c
    S.append(s)
  return S
```

5. Algorithm Implementations

a. Brute Force Algorithm

Implement the brute force algorithm and perform an initial testing of the implementation by using 15-20 samples using the sample generator tool of Section 4. Note that, although you can implement this algorithm yourself (if you want to), it is also fine if you use a code that you find from the internet. However, you should be able to install and run it. Also, you need to get familiar with the source code to be able to answer any questions about the code.

Report the results of the initial testing by giving the number and the size of the instances tried. Report any failures and related fixes.

Code for Brute Force SCS Algorithm:

```
def find_shortest_common_superstring(strings):
```

```
    perms = permutations(strings)
    shortest = None
    for perm in perms:
        superstring = perm[0]
        for i in range(1, len(perm)):
            overlap = find_overlap(superstring, perm[i])
            superstring += perm[i][overlap:]
        if shortest is None or len(superstring) < len(shortest):
            shortest = superstring
    return shortest

def permutations(strings):
    if len(strings) == 1:
        return [strings]
    else:
        result = []
        for i in range(len(strings)):
            remaining = strings[:i] + strings[i+1:]
            perms = permutations(remaining)
            for perm in perms:
                result.append([strings[i]] + perm)
        return result

def find_overlap(s1, s2):
    max_overlap = 0
    for i in range(min(len(s1), len(s2))):
        if s1[-i:] == s2[:i]:
            max_overlap = i
    return max_overlap
```

SAMPLE TEST CASES:

#Main Function and Parameters:

```
For i in range(20):
        print("\nSample Test Case {}".format(i+1) )
        strings = generate_random_input(random.randint(1,5), 1, random.randint(5,10), string.ascii_lowercase)
        print("Randomly Generated Input Set of Strings: {}".format(strings))
        shortest_superstring = find_shortest_common_superstring(strings)
        print("The Shortest Common Superstring in trial {} is: {}\n". format(i+1, shortest_superstring))
```

#Samples:

Sample Test Case 1

Randomly Generated Input Set of Strings: ['drslgqxz']

The Shortest Common Superstring in trial 1 is: drslgqxz

Sample Test Case 2

Randomly Generated Input Set of Strings: ['pfrs', 'puuzy', 'syqzq']

The Shortest Common Superstring in trial 2 is: pfrsyqzqpuuzy

Sample Test Case 3

Randomly Generated Input Set of Strings: ['vducqqjt']

The Shortest Common Superstring in trial 3 is: vducqqjt

Sample Test Case 4

Randomly Generated Input Set of Strings: ['igj', 'lsnr', 't', 'qrjj']

The Shortest Common Superstring in trial 4 is: igjlsnrtqrjj

Sample Test Case 5

Randomly Generated Input Set of Strings: ['cjlnxxbvew', 'klrlu', 'nppwpss']

The Shortest Common Superstring in trial 5 is: cjlnxxbvewklrlunppwpss

Sample Test Case 6

Randomly Generated Input Set of Strings: ['q', 'kb', 'clsbl', 'i', 'ki']

The Shortest Common Superstring in trial 6 is: qkbclsbliki

Sample Test Case 7

Randomly Generated Input Set of Strings: ['zegm']

The Shortest Common Superstring in trial 7 is: zegm

Sample Test Case 8

Randomly Generated Input Set of Strings: ['pgbhyxpb', 'sewk', 'ocyrvnh', 'plwlfynvh']

The Shortest Common Superstring in trial 8 is: pgbhyxpbsewkocyrvnhplwlfynvh

Sample Test Case 9

Randomly Generated Input Set of Strings: ['eiclwwfz', 't']

The Shortest Common Superstring in trial 9 is: eiclwwfzt

Sample Test Case 10

Randomly Generated Input Set of Strings: ['m', 'wm', 'd', 'upot', 'lwjyij']

The Shortest Common Superstring in trial 10 is: mwmdupotlwjyij

Sample Test Case 11

Randomly Generated Input Set of Strings: ['yjio', 'wc']

The Shortest Common Superstring in trial 11 is: yjiowc

Sample Test Case 12

Randomly Generated Input Set of Strings: ['w', 'y']

The Shortest Common Superstring in trial 12 is: wy

Sample Test Case 13

Randomly Generated Input Set of Strings: ['z', 'mq', 'ai']

The Shortest Common Superstring in trial 13 is: zmqai

Sample Test Case 14

Randomly Generated Input Set of Strings: ['mn', 'ivp']

The Shortest Common Superstring in trial 14 is: mnivp

Sample Test Case 15

Randomly Generated Input Set of Strings: ['fgckyex', 'gni', 'rwbtyh', 'uqsfmttq']

The Shortest Common Superstring in trial 19 is: fgckyexgnirwbtyhuqsfmttq

Sample Test Case 20

Randomly Generated Input Set of Strings: ['qkhf', 'egpccc', 'caacno', 'acux', 'wnuhi']

The Shortest Common Superstring in trial 20 is: qkhfegpcccaacnoacuxwnuhi

b. Heuristic Algorithm

Implement the heuristic algorithm and perform an initial testing of the implementation by using 15-20 samples using the sample generator tool of Section 4.

Note that, you do not have to implement this algorithm. You can also find the source code from the internet. However, you should be able to install and run it. Also, you need to get familiar with the source code to be able to answer any questions about the code.

#Samples:

Sample Test Case 1

Randomly Generated Input Set of Strings: [“grqztox”]

The Shortest Common Superstring in trial 1 is: grqztox

Sample Test Case 2

Randomly Generated Input Set of Strings: ["rfzbuzc", "nzxsipf"]

The Shortest Common Superstring in trial 2 is: rfzbuzcnzxsipf

Sample Test Case 3

Randomly Generated Input Set of Strings: ["kzyqrnq", "pvvbfcn", "hjvrgrn"]

The Shortest Common Superstring in trial 3 is: kzyqrnqpvvbfcnhjvrgrn

Sample Test Case 4

Randomly Generated Input Set of Strings: ["yawzixw", "kliozve", "kbfqasx", "qiqteyp"]

The Shortest Common Superstring in trial 4 is: yawzixwkliozvekbfqasxqiqteyp

Sample Test Case 5

Randomly Generated Input Set of Strings: ["mrsogjv", "ukfqllx", "clvkgka", "jmtzokf", "sgcibrm"]

The Shortest Common Superstring in trial 5 is: ukfqllxclvkgkajmtzokfsgcibrmrsogjv

Sample Test Case 6

Randomly Generated Input Set of Strings: ["cgyqmkn", "cduhtfn", "ysyqput", "zxyjkjc", "wedpmmz", "ghpgorp"]

The Shortest Common Superstring in trial 6 is: cduhtfnysyqputwedpmmzxyjkjcgyqmknghpgorp

Sample Test Case 7

Randomly Generated Input Set of Strings: ["dfgqpof", "zicmwdo", "xhkcqju", "hizkvrp", "gltjqgz", "fewvboy", "bwfewbo"]

The Shortest Common Superstring in trial 7 is:

dfgqpofewvboyxhkcqjuhizkvrpgltjqgzicmwdobwfewbo

Sample Test Case 8

Randomly Generated Input Set of Strings: ["shwwgdf", "uapotls", "nryilgl", "twjdrel", "zvnfvch", "wcxrbby", "tgqnpye", "mqyjupg"]

The Shortest Common Superstring in trial 8 is:

uapotlshwwgdfnryilgltwjdrelzvnfvchwcxrbbytgqnpyemqyjupg

Sample Test Case 9

Randomly Generated Input Set of Strings: ["bsbmrbz", "vdbxmpx", "nkhvbfk", "lluuobt", "mmypvbs", "ewomnku", "qupkqrl", "pmjejov", "gpcsdde"]

The Shortest Common Superstring in trial 9 is:

nkhvbfkmmypvbsbmrbzqupkqrlluuobtpmjejovdbxmpxgpcsddewomnku

Sample Test Case 10

Randomly Generated Input Set of Strings: ["sclakzz", "vlkuymc", "uvblbii", "kdoepgf", "xarmhct", "rxcgqcm", "smftmma", "kawqexz", "kenhmng", "rfebkrs"]

The Shortest Common Superstring in trial 10 is:

vlkuymcuvblbiikdoepgfxarmhctrxcgqcmsmftmmakawqexzkenhmngrfebkrsclakzz

Sample Test Case 11

Randomly Generated Input Set of Strings: [“givbtro", "hmuxuvu", "wpsaqdu", "yypwxsk", "eutkpht", "odnvtdu", "vnoyruo", "gwtjvwz", "tdmotub", "nlqfkbw", "atafdqr”]

The Shortest Common Superstring in trial 11 is:

givbtrodnvtduhmuxuvuyypwxskeutkphtdmotubvnoyruogwtjvwznlqfkbwpsaqduatafdqr

Sample Test Case 12

Randomly Generated Input Set of Strings: ["nchpnro", "jegkbyt", "kzxbepz", "qiqdvkf", "anzuelh", "nmozljg", "uickrnv", "pzukfio", "mtozrov", "jwbcbmp", "hfqnbaq", "gvyquox"]

The Shortest Common Superstring in trial 12 is:
nchpnrojegkbytkzxbepzukfioanzuelhfqnbaqiqdvkfnmozljgvyquoxuickrnvmtozrovjwbcbmp

Sample Test Case 13
Randomly Generated Input Set of Strings: ["jfqccfl", "azlemag", "hhxjnwp", "lhmxwna", "xxqznuj", "ypgavem", "mcbqlzi", "tdwussj", "wvqtmlk", "vjflirz", "npjfcgc", "yacsxtx", "fzzxxxc"]
The Shortest Common Superstring in trial 13 is:
hhxjnwpypgavemcbqlzitdwussjwvqtmlkvjflirznpjfcgcyacsxtxxqznujfqccflhmxwnazlemagfzzxxxc

Sample Test Case 14
Randomly Generated Input Set of Strings: ["zzjkqxy", "gnldbqz", "kyhcxdb", "grvhfms", "uvlkmxv", "qcrdeux", "lpxneum", "mlogrel", "hohkhzl", "bnhlkdx", "vgrxwpm", "dplunzn", "pkkcpjv", "ezipaqg"]
The Shortest Common Superstring in trial 14 is:
kyhcxdbnhlkdxgrvhfmsuvlkmxvgrxwpmqcrdeuxhohkhzlpxneumlogreldplunznpkkcpjvezipaqgnldbqzz jkqxy

Sample Test Case 15
Randomly Generated Input Set of Strings: ["eprpvno", "zrprehr", "ifrcgpc", "hovwvmu", "bfswxdv", "gswblpa", "vgiymiv", "rpwbjww", "gmomnlc", "mrodrwy", "queithk", "jyblzty", "djwoqhf", "skfjhig", "gomdgyy"]
The Shortest Common Superstring in trial 15 is:
eprpvnozrprehrpwbjwwifrcgpchovwvmubfswxdvgiymivgmomnlcmrodrwyqueithkjyblztydjwoqhfskfj higswblpagomdgyy

Sample Test Case 16
Randomly Generated Input Set of Strings: ["lcxdpiy", "ixstftj", "aoguzdr", "vabolne", "axfyqnv", "afjaqvl", "ghzjhem", "ocxhgjv", "tmnouyy", "zwrdohp", "onycbvn", "fawilfi", "gumbfyo", "lpvtlyk", "yydnovv", "aohokoh"]
The Shortest Common Superstring in trial 16 is:
aoguzdraxfyqnvabolneafjaqvlcxdpiyghzjhemtmnouyydnovvzwrdohponycbvnfawilfixstftjgumbfyocxh gjvlpvtlykaohokoh

Sample Test Case 17

Randomly Generated Input Set of Strings: ["xzmkxry", "bbusmrs", "uytggxx", "zbifjay", "uhbovpo", "yrfhtar", "pmsjqmn", "ozrodwb", "wyqiqft", "rtzysrn", "izbqljp", "cvetemo", "krzqvye", "yxqlpxg", "wmmyvoo", "ciopqqt", "ojaeiro"]

The Shortest Common Superstring in trial 17 is:

uytggxxzmkxryrfhtartzysrnzbifjayxqlpxguhbovpozrodwbbusmrswyqiqftizbqljpmsjqmncvetemojaeiro krzqvyewmmyvoociopqqt

Sample Test Case 18

Randomly Generated Input Set of Strings: ["rztmtfe", "lkrajcq", "mtryyco", "gzvimot", "silzlfx", "vfjouer", "yrhwlpz", "okypgdl", "mbsdqah", "qsrexrl", "lwgsafj", "xqelums", "grwzaqu", "dtszbzw", "ttkmdzf", "hzjjevi", "afddhfl", "kxllkxp"]

The Shortest Common Superstring in trial 18 is:

mtryycokypgdlkrajcqsrexrlwgsafjgzvimottkmdzfsilzlfxqelumsvfjouerztmtfeyrhwlpzmbsdqahzjjevigrw zaqudtszbzwafddhflkxllkxp

Sample Test Case 19

Randomly Generated Input Set of Strings: ["tswgfou", "yxvvivk", "rfififj", "cvtwnwm", "mccqxcn", "iveecyi", "ceuecew", "zquschc", "jtfiktp", "cbfqtaw", "idcjhhj", "dmskhpd", "pwsbnhq", "vlodslk", "jdvxnrw", "jzribqe", "udvqwob", "htmejwp", "ytfbzhi"]

The Shortest Common Superstring in trial 19 is:

tswgfoudvqwobyxvvivkrfififjtfiktpwsbnhqceuecewzquschcvtwnwmccqxcncbfqtawdmskhpdvlodslkjzr ibqehtmejwpytfbzhiveecyidcjhhjdvxnrw

Sample Test Case 20

Randomly Generated Input Set of Strings: ["rckrdzz", "tdyxzwj", "xanvmca", "kbxazgo", "iogggme", "rfxlfio", "dckgbxv", "hrkwztd", "pqpxtij", "oymyaqv", "ugnbgzn", "fkywugt", "zxxvtuy", "cpnrlcz", "pfpbjhx", "eqqipvy", "dltejzq", "hnqsdkb", "fatuolp", "dfxsedv"]

The Shortest Common Superstring in trial 20 is:

rckrdzzxxvtuyrfxlfiogggmeqqipvydckgbxvhrkwztdyxzwjugnbgznfkywugtcpnrlczpfpbjhxanvmcadltejz qhnqsdkbxazgoymyaqvfatuolpqpxtijdfxsedv

6. Experimental Analysis of The Performance (Performance Testing)

*In the experimental analysis part of your project, you are expected to analyse the performance of the implementation of the algorithm (**in Section 2-b**) experimentally. The complexity results presented **in Section 3-b** are theoretical results. The worst-case time complexity of the algorithm may not be displayed in practice. You will assess the practical time complexity of the algorithm by performing performance tests. We expect you to use the statistical methods which you covered in the lectures for this part. Furthermore, we want you to visualize your results in graphs (for example: a running time graph). You need to fit a line for the measurement values.*

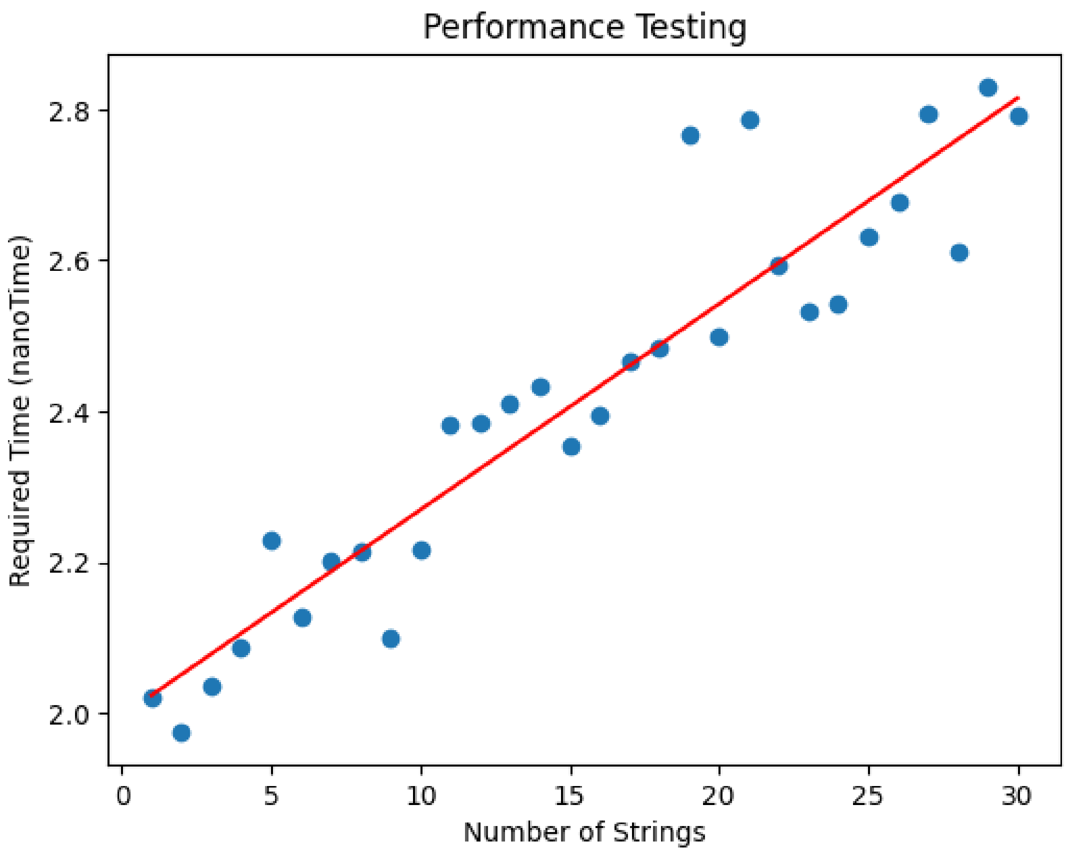
*In addition, you need to consider the solution quality of the heuristic algorithm in practice. Since the heuristic algorithms do not necessarily find the exact solutions, it is usually an issue how close they get the exact/correct answer. To be able to understand this, you will need to design and perform experiments, to see how the quality of the solutions of the heuristic algorithm changes with respect to the problem size. Note that, to carry out these tests you will also need to use an implementation of the brute force algorithm you implemented in **section 2-a.***

The algorithm is tested with each input size from 1 to 30. Length of strings are constant and is 7. The running time of the algorithm is tested (in nanoTime) 10 times for each input, then their average is taken. The resulting table is shown below:

```
df["Average"] = df.iloc[:, 1:10].mean(axis=1)/10000000
df
```

	numOfStr	num1	num2	num3	num4	num5	num6	num7	num8	num9	num10	Average
0	1	20095200	20179100	20192500	20204300	20215600	20233700	20245000	20256400	20266800	20277700	2.020984
1	2	19667400	19717800	19731100	19742900	19753900	19764300	19776000	19787300	19797900	19814900	1.974873
2	3	20271000	20320400	20333800	20345700	20357300	20369700	20381200	20392100	20414800	20426500	2.035400
3	4	20789600	20837600	20851300	20864400	20876000	20888600	20909500	20941700	20954300	20972500	2.087922
4	5	22182600	22222100	22241500	22258000	22303600	22330400	22351200	22367700	22383700	22400700	2.229342
5	6	21130200	21200400	21223100	21252400	21272900	21294000	21317900	21337800	21358300	21431900	2.126522
6	7	21884800	21940600	21961500	21981500	22002100	22029100	22050000	22070400	22090600	22109400	2.200118
7	8	21995700	22051200	22079300	22101400	22134600	22159300	22188200	22260100	22302400	22335200	2.214136
8	9	20878400	20938100	20960300	20981400	21003800	21030400	21053400	21077600	21100900	21121500	2.100270
9	10	22055100	22099500	22121100	22152500	22172900	22192700	22222400	22252400	22274300	22296100	2.217143
10	11	23698500	23744400	23768400	23792700	23825600	23850000	23882900	23905900	23937000	23960500	2.382282
11	12	23715900	23765300	23791400	23815700	23849000	23874000	23910800	23935000	23959300	23994400	2.384627
12	13	23954400	24002900	24043700	24069700	24101300	24134700	24161700	24186300	24212100	24245300	2.409631
13	14	24187800	24240000	24278100	24308000	24340200	24371200	24397900	24431900	24460200	24493300	2.433503
14	15	23369500	23448200	23492500	23522100	23556300	23583600	23611000	23646000	23678100	23720900	2.354526
15	16	23788800	23842700	23880600	23918100	23953300	23990900	24021500	24057600	24088500	24124700	2.394911
16	17	24478100	24538100	24569300	24611300	24660700	24697000	24735800	24773100	24893000	24955000	2.466182
17	18	24599700	24667900	24701200	24793300	24857200	24914000	24951600	25004000	25041900	25098400	2.483676
18	19	27424000	27487000	27522700	27576200	27610200	27770900	27820900	27870300	27972700	28017700	2.767277
19	20	24785100	24844200	24890100	24937200	24971500	25078400	25127700	25169300	25228400	25269700	2.500354
20	21	27649000	27715000	27763200	27812600	27848700	27914700	27977000	28025500	28074200	28115000	2.786443
21	22	25643800	25704600	25750900	25866000	25926700	26028600	26133100	26193200	26241900	26275900	2.594320
22	23	25056400	25114800	25257000	25303300	25362100	25401300	25444100	25485000	25517800	25556800	2.532687
23	24	25108100	25221800	25276600	25327300	25404300	25464000	25573600	25658700	25694300	25729000	2.541430
24	25	26071300	26134600	26201000	26251900	26343500	26403400	26457600	26495100	26540000	26578300	2.632204
25	26	26515700	26580800	26641300	26692800	26744800	26882300	26944200	26989600	27029900	27081300	2.678016
26	27	27647400	27716300	27788600	27860200	27991500	28057800	28118700	28160300	28214300	28266300	2.795057
27	28	25795400	25865500	25936400	26103600	26147300	26194300	26256100	26310400	26364900	26412200	2.610821
28	29	28013000	28096300	28196900	28278300	28332000	28380500	28429400	28481100	28528600	28566400	2.830401
29	30	27602100	27685400	27751400	27909900	27960400	28022900	28066000	28116600	28220300	28283400	2.792611

Also the resulting strings are tested by comparing the resulting strings of brute force algorithm. Due to the huge time requirement for the brute force algorithm, only the inputs with size 1 to 15 are tested, and they appeared to be exactly the same. As the input size gets bigger and bigger, the time required for brute force algorithm rocketed and the code could not exit for minutes whereas the heuristic algorithm continued to find the SCS in a few seconds. When the time/input size graph is plotted and the average line is fit, the resulting graph below is displayed:



As it can be seen here, a linear correlation is observed between required time and number of strings.

7. Experimental Analysis of the Correctness (Functional Testing)

The correctness results given in Section 3-b show that the algorithm is correct.

However, there can be errors introduced into the implementation. In other words, there can be coding errors. For this part of work, you will need to perform testing of the implementation to assess the correctness of the implementation of the algorithm. We want you to use some testing methods for your algorithm that is covered in the lectures for this part.

For Black Box testing, some of the extreme cases and their results are given below:

- If empty string input is given → throws exception since there is no string
- For 1 string only as input → output is the same string as expected

- For many strings as input → same output with what is given by the brute force algorithm (it is tested only up to 15 strings with length 7 since the rest takes too much time and sometimes fails to give the output)
- The same input string set but tried with the strings replaced with each other → output length is the same but the output string is different due to the order they are given

8. Discussion

Discuss your results in detail. Are there any defects in your algorithm? Is there any inconsistency between your theoretical analysis and experimental analysis?

The heuristic algorithm above seems to be a linear time solution for the test cases above, and it is accurate with the results of the brute force algorithm. However, the test cases shown in this study are not enough and needs improvement. Besides this inadequacy, the data collected shows that our theoretical analysis and experimental analysis are consistent with each other. The only detected defect of the algorithm is that it throws an exception if there are no strings in the input array or all the strings are empty.

<u>Submission</u>

On SUCourse+, for each group, only ONE person will submit the report.

For Progress Reports, we expect a report as PDF file. Your filename must be in the following format:

`CS301_Project_Progress_Report_Group_XXX.pdf`

where `XXX` is your group number.

For Final Reports, we expect a report as a PDF file, and also the codes and the data produced during the experiments. Please submit a single zip file that contains all your project files. Your zip file name must be in the following format:

`CS301_Project_Final_Report_Group_XXX.zip`

where `XXX` is your group number. In this zip file, please make sure that your final report is named as follows:

`CS301_Project_Final_Report_Group_XXX.pdf`

where `XXX` is your group number. In addition to the final report in PDF, this zip package all other project files as explained above.

REFERENCES:

Labs, Alfred V. Aho Bell, Alfred V. Aho, Bell Labs, Margaret J. Corasick Bell Labs, Margaret J. Corasick, Univ. of Chicago, and Other MetricsView Article Metrics. "Efficient String Matching: An Aid to Bibliographic Search: Communications of the ACM: Vol 18, No 6." Communications of the ACM, June 1, 1975. https://dl.acm.org/doi/10.1145/360825.360855.

M4rukku. "M4rukku/UKKONENS-Linear-Time-Shortest-Common-Superstring: An Implementation of Ukkonens 1990 Linear-Time Algorithm for Finding an Approximate Shortest Superstring in Java. Also Includes an Extendable Version of Aho Corasick's Efficient String Matcher." GitHub, 2021. https://github.com/M4rukku/Ukkonens-Linear-Time-Shortest-Common-Superstring.

Ukkonen, Esko. "A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings - Algorithmica." SpringerLink, 1990. https://link.springer.com/article/10.1007/BF01840391.

Ukkonen, Esko. "The Shortest Common Supersequence Problem over Binary Alphabet Is NP-Complete." Theoretical Computer Science, February 24, 2015. https://www.academia.edu/11055618/The_shortest_common_supersequence_problem_over_binary_alphabet_is_NP_complete.