WIA2005 ALGORITHM DESIGN AND ANALYSIS

LAB-2

Team 1, Occ 4

Name	Matric
Fadjar Soengkono	S2000808
Zahid Fajri Ramadhan	17216872
Muhammad Rafi Azhar Salman	S2003599
Huang Ruixin	S2011472
Samiul Hoque Sami	S2014442
Leiheng Qin	S2005744

Lab Viva - Sorting Algorithms and String Matching Algorithms

PART 1 – Implement the sorting algorithm

a) Counting Sort

Pseudocode:

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Source Code:

In [10]:

```
def countingSort(Array):
    maximum = int(max(Array))  # store maximum element of array in maximum
    minimum = int(min(Array))  # store minimum element of array in minimum
    element_range = maximum - minimum + 1
    length = len(Array)  # store length of array in length variable
    CountedArray = [0 for X in range(element_range)]  # set Counted_Array from 0 to element_range
    OutputArray = [0 for X in range(length)]  # set Output_Array from 0 to length
    for X in range(0, length):
        CountedArray[Array[X]-minimum] += 1  # increase Counted_Array[with particular indexing] by 1
    for X in range(1, len(CountedArray)):
        CountedArray[X] += CountedArray[X-1]  # store the cumulative sum into the counted_array
    for X in range(length-1, -1, -1):
        OutputArray[CountedArray[Array[X] - minimum] - 1] = Array[X]
        CountedArray[Array[X] - minimum] -= 1  # decrease Counted_Array[with particular indexing] by 1
    for X in range(0, length):
        Array[X] = OutputArray[X]  # store all the elements of Output_Array in Array
    return Array  # return the Sorted Array

# main program execute:
Array = [16, 30, 95, 51, 84, 23, 62, 44]
print("Sorted Array is: ", countingSort(Array))  # call the Counting_Sort function and print the return value o
```

```
Sorted Array is:  [16, 23, 30, 44, 51, 62, 84, 95]
```

Source Code (simpler version):

In [11]:

```
def count_sort(arr):
    min_element = int(min(arr))
    max_element = int(max(arr))
    range_of_element = max_element - min_element + 1
    count_arr = [0 for _ in range(range_of_element)]
    for value in arr:
        count_arr[value - min_element] += 1

    arrayIndex = 0
    for i in range(0, range_of_element, 1):
        while (count_arr[i] > 0):
            arr[arrayIndex] = i + min_element
            count_arr[i] -= 1
            arrayIndex += 1
    print(arr)


array = [5, 2, 7, 4, -2, 2]
count_sort(array)
```

```
[-2, 2, 2, 4, 5, 7]
```

Comprehension:

Counting Sort does not sort the input by comparing the input items, hence it is clearly not a comparison sort algorithm.

A counting sort is a sorting method that uses keys that fall inside a certain range. It operates by determining the number of items with unique key values (kind of hashing). The location of each object in the output sequence is then calculated using arithmetic.

Time Complexity:

It takes O(n+k) time to compute the array: It first iterates an input array of size n in O(n), then iterates the array in O(k) – making the total time O(n+k). It sorts the input by iterating the input array and performs a few rudimentary operations in each iteration after computing the array. As a result, the actual sort procedure takes O(n). Counting sort takes O(n+k) time to complete: O(n + k) + O(n) = O(2n + k) = O(n + k)

b) Radix Sort

Pseudocode:

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Source Code:

In [12]:

```
def maximumplaces(array):
    max_num = max(array)
#method to find maximum digit place for the maximum number and repeat the sorting
                                #iteration base on the digit place


    place = 1
    while max_num // place > 0:
        radixsort(array, place)                         #O(d)
        place *= 10                              #O(d) = value of d ,k is the maximum possible value, then d wo
#repeated for each iteration so for each iteration ( O(n) + O(k) +  0(n) +  0(n) )

def radixsort(array, place):
    sizenum = len(array)
    output = [0] * sizenum
    count = [0] * 10            #range of num in 0-9
    for i in range(0, sizenum):
        index = array[i] // place      #putting element into the places. We know that for this the
        count[index % 10] += 1              #end of this loop is n+1,therefore it TC is O(n)


    for i in range(1, 10):             #calculate the i + i-1
        count[i] += count[i - 1]            #O(k)


    i = sizenum - 1
    while i >= 0:                                     #0(n )

        if i == 0:
            print("")
            print("   new digit places sorting")


        index = array[i] // place
        output[count[index % 10] - 1] = array[i]            #sorting
        count[index % 10] -= 1
        i -= 1
        print(output)   #printing sorting for each number in an iteration


    for i in range(0, sizenum):
        array[i] = output[i]                #0(n)

#end of loop
#therefore the time complexity is O(d)x( O(n) + O(k) +  0(n) +  0(n) )   = O(d(n+k)))




data = [16, 30, 95, 51, 84, 23, 62, 44]

print(data)
print("")
print("     First sorting ")
maximumplaces(data)

print("Final result")
print(data)
```

```
[16, 30, 95, 51, 84, 23, 62, 44]

     First sorting
[0, 0, 0, 0, 0, 44, 0, 0]
[0, 0, 62, 0, 0, 44, 0, 0]
[0, 0, 62, 23, 0, 44, 0, 0]
[0, 0, 62, 23, 84, 44, 0, 0]
[0, 51, 62, 23, 84, 44, 0, 0]
[0, 51, 62, 23, 84, 44, 95, 0]
[30, 51, 62, 23, 84, 44, 95, 0]

   new digit places sorting
[30, 51, 62, 23, 84, 44, 95, 16]
[16, 0, 0, 0, 0, 0, 0, 0]
[16, 0, 0, 0, 0, 0, 0, 95]
[16, 0, 0, 44, 0, 0, 0, 95]
[16, 0, 0, 44, 0, 0, 84, 95]
[16, 23, 0, 44, 0, 0, 84, 95]
[16, 23, 0, 44, 0, 62, 84, 95]
[16, 23, 0, 44, 51, 62, 84, 95]

   new digit places sorting
[16, 23, 30, 44, 51, 62, 84, 95]
Final result
[16, 23, 30, 44, 51, 62, 84, 95]
```

Comprehension:

Radix sort is a number sorting algorithm that sorts integers by their digit locations. It works by using the digits' place value. It doesn't compare the integers, unlike most other sorting algorithms like Merge Sort, Insertion Sort, and Bubble Sort.

To sort the digits, Radix sort employs a subroutine that uses a reliable sorting technique. As a subroutine, we've utilized a counting sort version that sorts the digits in each location using the radix. Counting sort is a reliable sorting algorithm that performs well in the real world.

The Least Significant Digit (LSD) is sorted first, followed by the Most Significant Digit(MSD). Radix sort may also be used to process digits from MSD.

Time Complexity:

Radix Sort's performance is determined by the stable sorting algorithm used to sort the digits.

The Radix Sort was used to sort an array of n integers in base b. The basis in our situation is ten. The Counting Sort has been used c times, where c is the number of digits. Radix Sort's temporal complexity thus becomes O(c * (n + b)).

Because we utilized a variant of Counting Sort as a subroutine, the space complexity is O(n + b).

d) Shell Sort

Pseudocode:

```
shellSort(array, size)
  for interval i <- size/2n down to 1
    for each interval "i" in array
        sort all the elements at interval "i"
end shellSort
```

Source Code:

In [13]:

```
def shellSort(array, n):

    # Rearrange elements at each n/2, n/4, n/8, ... intervals
    interval = n // 2
    while interval > 0:
        for i in range(interval, n):
            temp = array[i]
            j = i
            while j >= interval and array[j - interval] > temp:
                array[j] = array[j - interval]
                j -= interval

            array[j] = temp
        interval //= 2


data = [16, 30, 95, 51, 84, 23, 62, 44]
size = len(data)
shellSort(data, size)
print('Sorted Array in Ascending Order:')
print(data)
```

```
Sorted Array in Ascending Order:
[16, 23, 30, 44, 51, 62, 84, 95]
```

Comprehension:

Shell sort is an extremely efficient sorting algorithm that is based on the Insertion sorting method. In general, the procedure divides a large set into smaller subsets, which are subsequently sorted using the Insertion sort algorithm.

However, it is not clear how it creates the subgroups. As one might imagine, it does not chose nearby items to build a subset. Shell sort, on the other hand, creates subsets using the interval or gap method. If we have the gap I, for example, it signifies that one subset will contain the components that are separated by I positions.

To begin with, the algorithm sorts the elements that are far apart. Then, when the distance narrows, closer items are compared. Some items that aren't in the right place can be repositioned faster this way than if we built subsets out of surrounding elements.

Time Complexity:

With medium-sized lists, the Shell sort algorithm is generally quite efficient. The time complexity fluctuates between O(N) and $O(N^2)$, but the difficulty is difficult to quantify since it depends so much on the gap sequence.

With O(1) auxiliary space, the worst-case space complexity is O(N).

PART 2 – Implement the String Matching Algorithm

a) Rabin-karp Algorithm

Pseudocode:

```
n = length of Text
m = length of Pattern
h1 = Pattern.hash
for s=0 to (n-m+1):
        h2 = Text[s:s + m].hash
        if h1 != h2:
            continue
        else:
            checkindex = 0
            for i =0 to m:
                if Text[s + i] != Pattern[i]:
                    break
                else:
                    checkindex + 1
            if checkindex = m:
                print(s)
```

Source Code:

In [14]:

```
def rabin_karp(Text, Pattern):
    n = len(Text)
    m = len(Pattern)
    r = 17  # set a number for carry
    def my_hash(text):
        hashnum = 0
        for i in range(len(text)):
            hashnum = hashnum * r + ord(text[i])
        # h = pow(r, len(text) - 1)  # the number for minus later
        return hashnum
    #get the hash of pattern and text
    patternhash = my_hash(Pattern)
    texthash = my_hash(Text[0:m])
    #search
    h = pow(r, m - 1)  # the number for minus later
    for index in range(0, n-m+1):#loop for search, where s is equal to the start point
        if patternhash != texthash:
            if (index+m)<=n:
                texthash=(texthash-ord(Text[index])*h)*r+ord(Text[index+m])#minus the first char's hash number,
            continue
        else:#the case that patternhash is equal to texthash
            #check the
            k = 0 #k is used for checking if pattern equal to text
            for i in range(0, m):
                if Text[index + i] != Pattern[i]:
                    break
                else:
                    k += 1
            if k == m:
                print("pattern is in index: ", index)
rabin_karp("algo-t4g1","g1")
```

```
pattern is in index:  7
```

Source Code (simpler version):

In [15]:

```
def rabinkarp(Text, Pattern):
    n = len(Text)
    m = len(Pattern)
    h1 = hash(Pattern)
    for s in range(0, n-m+1):#loop for search, where s is equal to the start point
        h2 = hash(Text[s:s + m])
        if h1 != h2:
            continue
        else:#the case of equal hash
            k = 0 #k is used for checking if pattern equal to text
            for i in range(0, m):
                if Text[s + i] != Pattern[i]:
                    break
                else:
                    k += 1
            if k == m:
                print(s)
rabinkarp("algo-t4g1","g1")
```

```
7
```

Comprehension:

The Rabin Karp algorithm is based on the notion of using hashing to discover a pattern in a text. We must generate a hash of the pattern at the start of the method, which will be utilized later in the algorithm. This is referred to as fingerprint computation.

Time Complexity:

The crucial point to remember about the pre-processing stage is that it has a time complexity of O(m), while iteration over the text takes O(n), giving the whole approach a time complexity of O(m+n).

In the worst-case situation, this algorithm's temporal complexity is O(m(n-m+1)). However, this approach has an average time complexity of O(n+m).

b) KMP Algorithm

Pseudocode:

```
 function KMPSearch() with parameter pattern and text
 Set m = len(pattern)
 Set n = len(text)
 Set i = 0
 Set j = 0
 Set lps = [0]
 Call function computeArray()
 while i < n:
 if pat[j] equal to txt[i]:
increment i by 1
increment j by 1
 if j equal to m:
set j = lps[j - 1]
 elif i < n and pat[j] not equal to txt[i]:
if j != 0:
set j = lps[j - 1]
else:
increment i by 1

Define function computeArray() with parameter pattern, m, and lps
 set i = 1
 set j = 0
 set lps[0] = 0
 while (i less than m){
if (pattern[i] equal to pattern[j]):
 set lps[i] = j
increment i by 1
 increment j by 1
else:
 if j is not equal to 0:
set j = lps[j-1]
 else:
set lps[i] =0
 increment i by 1
}
```

Source Code:

In [16]:

```
def kmp_matcher(string, pattern):
    n = len(string)
    m = len(pattern)
    lps = compute_prefix(pattern)
    print(lps)
    j = 0                                                        # number of characters matched
    for i in range(n):                                          # scan the text from left to right
        while j > 0 and string[i]!= pattern[j]:
            j = lps[j-1]                                         # next character does not match
        if string[i] == pattern[j]:
            j += 1                                               # next character matches
        if j == m:                                               # is all of pattern matched?
            print("Pattern found in index ", (i-(j-1)))
            j=lps[j-1]                                          # look for the next match


def compute_prefix(pattern):
    m = len(pattern)
    lps = [0]*m
    k = 0                                                        # number of character and suffix
    for i in range(1, m):
        while k > 0 and pattern[k] != pattern[i]:
            k = lps[k-1]                                         # find previous suffix
        if pattern[k] == pattern[i]:
            k += 1                                               # next character and lps value
        lps[i] = k                                               # store k in lps[]
    return lps

string = "algorisfunalgoisgreat"
pattern = "algo"
kmp_matcher(string, pattern)
```

```
[0, 0, 0, 0]
Pattern found in index  0
Pattern found in index  10
```

The idea of the Knuth-Morris-Pratt algorithm is the calculation of shift table which provides us with the information where we should search for our pattern candidates.

The KMP matching method takes use of the pattern's degenerating feature (patterns with the same sub-patterns appearing more than once in the pattern) to reduce the worst-case complexity to O. (n). The essential premise of KMP's method is that anytime we identify a mismatch (after a few matches), we already know some of the characters in the following window's content. We use this knowledge to prevent matching characters that we know will match in any case.

We used https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMPFail.html to visualise the algorithm (failure function).

KMP is efficient when we know that the pattern may have have an arguably a lot of suffix that is also a prefix, for example the the string "t4g1**t4g1rrrrrrrrt4g1*t4g1".

In [17]:

```
def kmp_matcher(string, pattern):
    n = len(string)
    m = len(pattern)
    lps = compute_prefix(pattern)
    print(lps)
    j = 0                                                        # number of characters matched
    for i in range(n):                                          # scan the text from left to right
        while j > 0 and string[i]!= pattern[j]:
            j = lps[j-1]                                         # next character does not match
        if string[i] == pattern[j]:
            j += 1                                               # next character matches
        if j == m:                                               # is all of pattern matched?
            print("Pattern found in index ", (i-(j-1)))
            j=lps[j-1]                                          # look for the next match


def compute_prefix(pattern):
    m = len(pattern)
    lps = [0]*m
    k = 0                                                        # number of character and suffix
    for i in range(1, m):
        while k > 0 and pattern[k] != pattern[i]:
            k = lps[k-1]                                         # find previous suffix
        if pattern[k] == pattern[i]:
            k += 1                                               # next character and lps value
        lps[i] = k                                               # store k in lps[]
    return lps

string = "algorisfunalgoisgreat"
pattern = "algo"
kmp_matcher(string, pattern)
```

```
[0, 0, 0, 0]
Pattern found in index  0
Pattern found in index  10
```

Comprehension:

The idea of the Knuth-Morris-Pratt algorithm is the calculation of shift table which provides us with the information where we should search for our pattern candidates.

The KMP matching method takes use of the pattern's degenerating feature (patterns with the same sub-patterns appearing more than once in the pattern) to reduce the worst-case complexity to O. (n). The essential premise of KMP's method is that anytime we identify a mismatch (after a few matches), we already know some of the characters in the following window's content. We use this knowledge to prevent matching characters that we know will match in any case.

We used https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMPFail.html to visualise the algorithm (failure function).

KMP is efficient when we know that the pattern may have have an arguably a lot of suffix that is also a prefix, for example the the string "t4g1**t4g1rrrrrrrrt4g1*t4g1".

Time Complexity:

In terms of input size, KMP is linear O(N). The failure function is computed linearly in the size of the pattern (the "needle"), and the search is linear in the size of the text (the "haystack").

The proof's fundamental point is that the algorithm maintains a state in each phase (which is essentially an index into the needle). The number of times the state is changed is obviously related to the running time. When we process a letter, we reduce the state a certain number of times (sometimes zero, sometimes a lot), and then increment it by one. Because the state never drops below zero, the total number of times it is increased by 1 is equal to the number of letters processed, and the total number of times it is decremented must be at most equal to the total number of times it is incremented. As a result, the total number of times the state changes is at most twice the number of times the state changes.

As a result, the total number of state changes is no more than twice the number of letters in the input.

c) TRIES Algorithm

Pseudocode:

```
void insert(String s)
{
    for(every char in string s)
    {
        if(child node belonging to current char is null)
        {
            child node=new Node();
        }
        current_node=child_node;
    }
}

boolean check(String s)
{
    for(every char in String s)
    {
        if(child node is null)
        {
            return false;
        }
    }
    return true;
}
```

Source Code:

In [18]:

```
def insert(word):
  for i in range(len(word)):       #loop for the step again after slice
      node = root
      for letter in word:          #loop for every letter
        if letter not in node:
          node[letter] = {}        #create new node
        node = node[letter]
        node[letter] = True        #set the node for every letter to true
      word=word[1:]                #slice the front letter by 1

def search(find):
  node = root
  for letter in find:              #loop for every letter
    if letter not in node:
      return False                 #check if the letter is not in trie
    node = node[letter]
  return True                      #the word is in trie

root = {}
word = "algorisfunalgoisgreat"
insert(word)

find = "algo"
if search(find) == bool(True):
  print(find, "is in the Trie.")
if search(find) == bool(False):
  print(find, "is not in the Trie.")
```

```
algo is in the Trie.
```

Comprehension:

Tries are a very unique and valuable data structure that is based on a string's prefix. They're used to symbolize data "retrieval," thus the term Trie.

A Trie is a unique data structure for storing texts that may be seen as a graph. It is made up of nodes and edges. Each node has a maximum of 26 children, with edges connecting each parent node to its offspring. Each of the 26 letters of the English alphabet is represented by one of these 26 points. For each edge, a distinct edge is maintained.

Time Complexity:

The complexity of making a trie is O(W*L), where W is the number of words and L is the average length of a word: for each of the W words in the set, you must execute L lookups on average.

The same is true when searching up words later: for each of the W words, you do L steps.

Hash insertions and lookups have the same difficulty: for the total complexity of O(W*L), you must check equality for each word, which takes O(L).