

LAB VIVA REPORT

WIA2005 ALGORITHM DESIGN & ANALYSIS

SEMESTER 2, 2021/2022

Tutorial Group : Tutorial 1

Group Name : Group 6

Group Members :

| **No.** | **Name** | **Matric ID** |
| --- | --- | --- |
| 1. | AMIRAH BINTI ABD NASIR | 17203688/2 |
| 2. | ATHIRAH BINTI AHMAD FAUZI | U2005247/1 |
| 3. | FARISAH ATHIRA BT MD ZAMRI | 17206915/2 |
| 4. | NURSYAZWANI BINTI MOHAMMAD | 17206746/2 |
| 5. | NUHA BINTI MOHAMED YAZID | 17206682/2 |
| 6. | NURUL AIN BT KHAIRUL ANWAR | U2005370/1 |

**TABLE OF CONTENTS**

[**QUESTIONS**](#_heading=h.ffw8nyop2ek6) **3**

[PART 1 – Implement the sorting algorithm](#_heading=h.fq2joy17noax) 3

[PART 2 – Implement the String Matching Algorithm](#_heading=h.5d9eh4myop4r) 3

[**ANSWERS**](#_heading=h.xg2170kj6ino) **4**

[PART 1: (a) Counting Sort](#_heading=h.xlm24bf8devy) 4

[PART 1: (b) Radix Sort](#_heading=h.unocv1jj5tfz) 8

[PART 1: (c) Shell Sort](#_heading=h.jzwnihyqhcqn) 12

[PART 2: (a) Rabin-Karp Algorithm](#_heading=h.3mgvskm75cih) 16

[PART 2: (b) KMP Algorithm](#_heading=h.bxxvmfmg5frk) 20

[PART 2: (c) TRIES](#_heading=h.v34kpcylupi9) 25

# 

# QUESTIONS

## PART 1 – Implement the sorting algorithm

Given the following array A = [16, 30, 95, 51, 84, 23, 62, 44], implement a program to sort

the array using the following algorithm:

1. Counting Sort
2. Radix Sort
3. Shell Sort

Provide the pseudocode, implement the algorithm using Python, explain the codes/algorithm,

and explain running time complexity of each algorithm.

## PART 2 – Implement the String Matching Algorithm

Given a String “algorisfunalgoisgreat”, search for the word “fun” and “algo” in the String by

applying the following algorithms:

a) Rabin-karp Algorithm

b) KMP Algorithm

c) TRIES

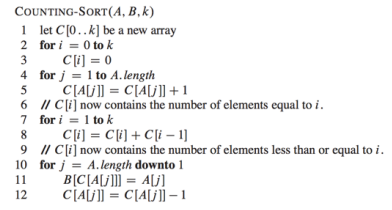
Provide the pseudocode, implement the algorithm using Python, explain the codes/algorithm,

and explain running time complexity of each algorithm.

# ANSWERS

## PART 1: (a) Counting Sort

**Pseudocode**



**Time Complexity**

O(n+k) where n is the number of elements in the input array and k is the range of input. In all

the cases whether it be the best case, worst case or average case the time complexity of the

algorithm is the same because the time complexity of the counting sort algorithm is not

dependent on how many elements we store in the array. therefore, the time complexity of the

counting sort algorithm is O(n+k).

**Code**

## 

**Explanation**

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

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

1. Find out the maximum element (let it be max) from the given array
2. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.

| index | 0... | 16 | ... | 23 | 30 | 44 | ... | 51 | 62 | ... | 84 | ... | 95 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

1. Store the count of each element at their respective index in count array

**counting instances**

| index | 0... | 16 | ... | 23 | 30 | 44 | ... | 51 | 62 | ... | 84 | ... | 95 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

1. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array

**modify the count (count\_arr)**

| index | 0... | 16 | ... | 23 | 30 | 44 | ... | 51 | 62 | ... | 84 | ... | 95 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| modified count | 0 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 | 6 | 7 | 7 | 8 |

1. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.
2. After placing each element at its correct position, decrease its count by one.

**Place 1 2 3 4 5 6 7 8**

| **16** |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** |  | **30** |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** |  | **30** |  |  |  |  | **95** |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** |  | **30** |  | **51** |  |  | **95** |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** |  | **30** |  | **51** |  | **84** | **95** |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** | **23** | **30** |  | **51** |  | **84** | **95** |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** | **23** | **30** |  | **51** | **62** | **84** | **95** |
| --- | --- | --- | --- | --- | --- | --- | --- |

| **16** | **23** | **30** | **44** | **51** | **62** | **84** | **95** |
| --- | --- | --- | --- | --- | --- | --- | --- |

## 

**Output**

## 

## PART 1: (b) Radix Sort

**Pseudocode**

**countRadix(inputArray, B):**

let C[0..9] be a new array

**for** i < inputSize:

Temp = (inputArray[i] / B) % 10

C[Temp]++

**for** i = 1 to 10:

C[i] += C[i-1]

**while** i= inputSize-1 downto 0

current= inputArray[i]

Temp= (inputArray[i] / B) % 10

countArray[Temp]-= 1

//put the current value inside outputArray

**return** outputArray

**radixSort(inputArray):**

B = 1

maxEl = max(inputArray)

**while** maxEL > 0:

//find the number of digit in maxEl and store it in int D

output = inputArray

**while** D > 0:

output = **countRadix**(output, B)

B \*= 10

D - -

**return** output

**Time Complexity**

**O(d\*(n+k))**

**Code**

**def countRadix(inputArray, placeValue):**

**countArray = [0] \* 10**

**inputSize = len(inputArray)**

**for i in range(inputSize):**

**Temp = (inputArray[i] // placeValue) % 10**

**countArray[Temp] += 1**

**for i in range(1, 10):**

**countArray[i] += countArray[i-1]**

**outputArray = [0] \* inputSize**

**i = inputSize - 1**

**while i>= 0:**

**current= inputArray[i]**

**Temp= (inputArray[i] // placeValue) % 10**

**countArray[Temp]-= 1**

**newPosition= countArray[Temp]**

**outputArray[newPosition]= current**

**i -= 1**

**return outputArray**

**def radixSort(inputArray):**

**placeVal = 1**

**maxEl = max(inputArray)**

**D = 0**

**while maxEl > 0:**

**maxEl //= 10**

**D += 1**

**outputArray = inputArray**

**while D > 0: # D**

**outputArray = countRadix(outputArray, placeVal) #(n+k)**

**placeVal \*= 10**

**D -= 1**

**return outputArray**

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

**print("Input array =" , A)**

**sortedArray= radixSort(A)**

**print("Result:", sortedArray)**

**Explanation**

**radixSort:**

First, we need to find the maximum element of the input given. Then, find the number of

digits for the maximum element and store it in D.

Next, we use the while loop to loop D times. In this loop, we will use the countRadix

function (where count sort is performed here) to arrange the input according to their least

significant values. placeValue would multiple 10 times in this loop. Below are the example of

outputArray of each loop:

| 16 | 30 | 95 | 51 | 84 | 23 | 62 | 44 |
| --- | --- | --- | --- | --- | --- | --- | --- |

D=2

| 30 | 51 | 62 | 23 | 84 | 44 | 95 | 16 |
| --- | --- | --- | --- | --- | --- | --- | --- |

D=1

| 16 | 23 | 30 | 44 | 51 | 62 | 84 | 95 |
| --- | --- | --- | --- | --- | --- | --- | --- |

**countRadix:**

First, create an empty array called countArray size 10. Then, the first for loop is to store the frequency of the least significant(to the most significant depends on the place value) value inside the countArray. The second for loop, the program would sum up the number inside the countArray using formula:

***countArray[i] = countArray[i] + countArray[i-1]*** (with i start with 1)

As for the last loop, we are using a counting sort method and the value would be put inside outputArray.

inputArray:

| 16 | 30 | 95 | 51 | 84 | 23 | 62 | 44 |
| --- | --- | --- | --- | --- | --- | --- | --- |

**0 1 2 3 4 5 6 7**

countArray:

| 1 | 1 | 1 | 1 | 2 | 1 | 1 | 0 | 0 | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

**0 1 2 3 4 5 6 7 8 9**

| 1 | 2 | 3 | 4 | 6 | 7 | 8 | 8 | 8 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

**0 1 2 3 4 5 6 7 8 9**

| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 8 | 8 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

**0 1 2 3 4 5 6 7 8 9**

outputArray:

|  |  |  |  |  | 44 |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |

**0 1 2 3 4 5 6 7**

…

| 30 | 51 | 62 | 23 | 84 | 44 | 95 | 16 |
| --- | --- | --- | --- | --- | --- | --- | --- |

**0 1 2 3 4 5 6 7**

## 

**Output**

## 

## PART 1: (c) Shell Sort

**Pseudocode:**

1 **ShellSort(array):**

2 Initialize gap = inputSize/2

3 **While** gap >0:

4 **For** each i <= gap & i = inputSize:

5 Set temporary value = arr[i]

6 **Set** j = i

7 **While** j>= gap & array [j-gap] > temp:

8 Set arr[j] = arr[j-gap]

9 Calculate new **j = j - gap** for next iteration

10 Set arr[j] = temp

11 Calculate new **gap = gap/2** for the next iteration

**Running time complexity:**

**Best case complexity: O(n\*log n)**

The best case running time complexity for shell sort is O(n\*log n). The best case for shell sort happens when the array is already sorted. This makes the inner while statement a constant time operation (there is a hardbound amount of time each operation will take to perform). The gap sequence used and the given input used in the program gives O(n\*log n) as the best case running time complexity.

**Worst case complexity: O(n^2)**

The worst case complexity for shell sort is O(n^2). This is because the gap sequences used in the program are reduced for each iteration which makes the number of swapping increases. The number of swaps of the inner while loop is <= n^2/g. By adding the upper-bound of the number of swaps for each gap together (n^2 + n^2/2 + n^2/4 + ... <= 2n^2 ∊ O(n^2)), it will match the worst-case complexity for the gaps that has been used which is O(n^2).

**Code:**

def ShellSort(arr):

inputSize = len(arr) #length of the array, size = n

gap = inputSize//2 #initialize the gap

while gap > 0:

for i in range(gap, inputSize):

temp = arr[i]

j=i

#compare element from the right to the left side

while j>=gap and arr[j-gap] > temp: #insertion sort

arr[j] = arr[j-gap]

j-= gap

arr[j] = temp

gap = gap // 2

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

print("Input array: ", A)

ShellSort(A)

print("Sorted Array:", A)

**Output:**

**Explanation:**

Shell sort is an optimization version of insertion sort algorithm. The sorting starts by comparing the elements that are far apart from each other. The gap between the elements must be less than the length of elements (value gap < n) and the gap sequence will reduce based on the sequence used after each iteration.

Based on the codes, the gap is initialized first by dividing the length of the array into 2 which will create a sub-array. Then, the while loop (while gap > 0:) is used to keep the loop until the gap is equal to 1. The gap will reduce by dividing the current gap into 2 in each iteration.

For each index (i) between gap size and input size, the value of index (i) will be set to the temporary value. Then, the values between the gaps will be compared from the right side (arr[j]) to the left size (arr[j-gap]). If the left side is bigger than the right side, then the values will swap using insertion sort. The iteration will keep on looping until the value in the left side is smaller than the right side.

i = 4

|  |  |  |  | j |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| arr[j-gap] |  |  |  | temp |  |  |  |
| 16 | 30 | 95 | 51 | 84 | 23 | 62 | 44 |

temp > arr[j-gap], so the value remain the same

i = 5

|  |  |  |  |  | j |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  | arr[j-gap] |  |  |  | temp |  |  |
| 16 | 30 | 95 | 51 | 84 | 23 | 62 | 44 |

Swap the value because temp < arr[j-gap]

| 16 | 23 | 95 | 51 | 84 | 30 | 62 | 44 |
| --- | --- | --- | --- | --- | --- | --- | --- |

i = 6

|  |  |  |  |  |  | j |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  | arr[j-gap] |  |  |  | temp |  |
| 16 | 23 | 95 | 51 | 84 | 30 | 62 | 44 |

Swap the value because temp < arr[j-gap]

| 16 | 23 | 62 | 51 | 84 | 30 | 95 | 44 |
| --- | --- | --- | --- | --- | --- | --- | --- |

i = 7

|  |  |  |  |  |  |  | j |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  | arr[j-gap] |  |  |  | temp |
| 16 | 23 | 62 | 51 | 84 | 30 | 95 | 44 |

Swap the value because temp < arr[j-gap]

| 16 | 23 | 62 | 44 | 84 | 30 | 95 | 51 |
| --- | --- | --- | --- | --- | --- | --- | --- |

When all the values have been compared and sorted between the sub-array for gap = 4, the gap will be reduced by dividing the current gap into 2. So, the new gap sequences will be used in the next iteration for the shell sort. it will loop until the gap = 1 and will stop as all the values have been sorted in ascending order.

## 

## PART 2: (a) Rabin-Karp Algorithm

**Pseudocode**

RabinKarpMatcher(text, pattern, q, d)

1 n = text.length

2 m = pattern.length

3 h = dm-1 mod q

4 p = 0

5 t0 = 0

6 **for** i = 1 **to** m //preprocessing

7 p = (dp + pattern[i]) mod q

8 t0 = (dt0 + text[i]) mod q

9 **for** s = 0 **to** n - m //matching

10 **if** p == ts

11 **if** pattern[1.....m] == text[s + 1..... s + m]

12 print "Pattern is found at position" s

13 **if** s < n-m

14 ts + 1 = (d (ts - text[s + 1]h) + text[s + m + 1]) mod q

**Run Time Complexity**

**Best case: O(n+m)**

The best case complexity for this algorithm is when no spurious hit occurs throughout the

searching process. Spurious hit happens when the hash value of the pattern in the window is

equal to the hash value of the pattern searched, however the pattern searched and the pattern

in the text window is not matched. Hence, for all cases, the program does not need to check

each letter of the pattern in the window one by one to determine whether both patterns are the

same.

**Worst case: O(mn)**

The worst case happens when spurious hit occurs in all text windows, where the hash value

of all the windows are equal to the hash value of the pattern searched, however the pattern

searched and the pattern in the text window is not matched. In order to minimize these

instances from occurring, modulus is used in the algorithm.

**Code**

def searchPattern(text, pattern, q, d):

m = len(pattern)

n = len(text)

p = 0 # hash value for pattern

t = 0 # hash value for text

h = 1

i = 0

j = 0

# The value of h would be "pow(d, m-1)%q"

for i in range(m-1):

h = (h\*d) % q

# Calculate hash value for pattern and text

for i in range(m):

p = (d\*p + ord(pattern[i])) % q

t = (d\*t + ord(text[i])) % q

# Find the match

for i in range(n-m+1):

# If the hash values match, check for characters one by one

if p == t:

for j in range(m):

if text[i+j] != pattern[j]:

break

else:

j += 1

# If p == t AND pattern[0...m-1] = text[i, i+1, ...i+m-1]

if j == m:

print("Pattern " + pattern + " is found at position: " + str(i))

# Hash rolling -> Calculate hash value for next window of text

if i < n-m:

t = (d\*(t-ord(text[i])\*h) + ord(text[i+m])) % q

# If t value is negative, convert it to positive by adding t with modulo

if t < 0:

t = t+q

q = 13

d = 26

text = "algorisfunalgoisgreat"

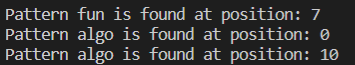
pattern1 = "fun"

searchPattern(text,pattern1,q,d)

pattern2 = "algo"

searchPattern(text,pattern2,q,d)

**Output**



**Explanation**

1. Initialize **m** as the length of the pattern and **n** as the length of the text.
2. Then, initialize the d value. **d** value is a set of numbers, which in this case is the total number of letters in the English alphabet, which is 26.
3. **q** is a prime number which acts as a modulo.
4. The **h** value is used for shifting left the high order digit. The formula used:

h = (h \* d) % q

1. Calculate hash value for pattern and first window of text. Hash value is derived from converting the character into its unicode value and times by **d** and modulus by **q**.
2. Match the value of the pattern and the text window in the range of (n - m + 1).
3. If the hash values match, check the characters one by one to rule out spurious hit.
4. If all characters match, then print the position of the pattern in the text.
5. Then, move to the next window of the text by using this formula:

(d (ts - text[s + 1]h) + text[s + m + 1]) mod q

1. If the hash value of the text is negative, convert it to positive by adding **t** and **q**:

t = t + q

1. The program runs until all text windows have been fully checked.

**Calculation Example**

text: algorisfunalgoisgreat

pattern: algo

**q** = 13, **d** = 26, **m** = 4, **n** = 21

**h** = (1 \* 26) % 13 = 0

* Calculate hash value for pattern and text

**Window 1**

text: |**algo**|risfunalgoisgreat

Iter 1 : **p** = (26 \* 0 + 97) % 13 = 6

**t** = (26 \* 0 + 97) % 13 = 6

Iter 2 : **p** = (26 \* 6 + 108) % 13 = 4

**t** = (26 \* 6 + 108) % 13 = 4

Iter 3 : **p** = (26 \* 4 + 103) % 13 = 12

**t** = (26 \* 4 + 103) % 13 = 12

Iter 4 : **p** = (26 \* 12 + 111) % 13 = 7

**t** = (26 \* 12 + 111) % 13 = 7

**p** = 7, **t** = 7

* Hash value for pattern and text matched
* Check for characters one by one to rule out spurious hit
* All characters matched
* Thus, print Pattern algo is found at position 0
* Calculate hash value for next window of text

**Window 2**

text: a|**lgor**|isfunalgoisgreat

**n** = 21, **m** = 4, **q** = 13, **d** = 26, **h** = 0, **p** = 7, **t** = 7

**t** = (26 \* (7 - 97 \* 0) + 114) % 13

= 10

* Hash value for pattern and text does not matched
* Calculate hash value for next window of text.
* The process ends when all the windows have been checked.

## 

## PART 2: (b) KMP Algorithm

**Pseudocode**

**KMP\_Matcher(T, P)**

1 n ← length [T]

2 m ← length [P]

4 lps = computePrefixFunction(P)

5 i = 0

6 j = 0

7 **while** i < n // scan S from left to right

8 **If** Pattern[j] == Text[i]

9 i = i + 1

10 j = j + 1

11 **If** j == m

12 print “Found pattern at index” i - j

13 j = lps[j - 1]

14 **Else** if i<n and P[j] != P[i]

15 **If** j != 0

16 j = lps[j - 1]

17 **Else** i += 1

**COMPUTE\_PREFIX\_FUNCTION (P)**

1 m ← length [P]

2 let lps[1..m] be a new array

3 i ← 1

4 j ← 0

5 **while** i < m

6 **If** P [i] == P [j]

7 j = j + 1

8 lps[i] = j

9 i = i + 1

10 **Else**

11 **if** j != 0

12 j = lps[j - 1]

13 **Else**

14 i = i + 1

15 return lps

**Run Time Complexity**

The KMP algorithm runs in O(m + n) time where n is the length of the string, and m is the length of the pattern.

**Code**

def KMP\_Matcher(T, P):

n = len(T)

m = len(P)

lps = compute\_Prefix\_Function(P)

i = 0

j = 0

# iterating the whole text

while i < n:

if P[j] == T[i]:

i += 1

j += 1

# condition when j is at the end of pattern (Found the pattern)

if j == m:

print("Found pattern at index", str(i - j))

j = lps[j - 1]

# mismatch after j matches

elif i < n and P[j] != T[i]:

if j != 0:

j = lps[j - 1]

else:

i += 1

def compute\_Prefix\_Function(P):

m = len(P)

lps = [0] \* m # creating list of length m with all zero value

i = 1

j = 0

# iterating the whole pattern

while i < m:

if P[i] == P[j]:

j += 1

lps[i] = j

i += 1

else:

# if j is in the middle, set j as the value of previous lps[]

if j != 0:

j = lps[j - 1]

else:

i += 1

return lps

txt = "algorisfunalgoisgreat"

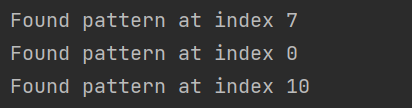
pat = "fun"

KMP\_Matcher(txt, pat)

pat = "algo"

KMP\_Matcher(txt, pat)

**Output**

****

**Explanation**

Knuth Morris Pratt algorithm is the string matching algorithm that runs in linear time. It is used to find a "Pattern" in a "Text". This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "Prefix Table'' to skip characters comparison while matching.

**compute\_Prefix\_Function(P)**

P = “f u n”

i = 1

j = 0

\*Round 1: P[i] != P[j] & j == 0

**j i**

| **index** | 0 | 1 | 2 |
| --- | --- | --- | --- |
| **P** | f | u | n |
| **lps** | 0 | 0 | 0 |

\*Round 2: P[i] != P[j] & j == 0

**j i**

| **index** | 0 | 1 | 2 |
| --- | --- | --- | --- |
| **P** | f | u | n |
| **lps** | 0 | 0 | 0 |

\*Currently, i == 3 (i >= m). So, while loop is terminated and the array of lps is returned.

**Another example (if P[i] == P[j])**

**j i**

| **index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **P** | a | b | a | b | a | c | a |
| **lps** | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

P[i] == P[j]

**j i**

| **index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **P** | a | b | a | b | a | c | a |
| **lps** | 0 | 0 | 1 | 2 | 0 | 0 | 0 |

P[i] == P[j]

**KMP\_Matcher(T, P)**

n = 21 (length of text)

m = 3 (length of pattern)

i = 0 (index for text)

j = 0 (index for pattern)

T = "algorisfunalgoisgreat"

P = "fun"

lps[] = {0, 0, 0}

i = 0, j = 0

T = "algorisfunalgoisgreat"

P = "fun"

\*not match → i++

i = 1, j = 0

T = "algorisfunalgoisgreat"

P = "fun"

\*not match → i++

i = 2, j = 0

T = "algorisfunalgoisgreat"

P = "fun"

\*not match → i++

.

.

i = 7, j = 0

T = "algorisfunalgoisgreat"

P = "fun"

\*Matched → i++ & j++

i = 8, j = 1

T = "algorisfunalgoisgreat"

P = "fun"

\*Matched → i++ & j++

i = 9, j = 2

T = "algorisfunalgoisgreat"

P = "fun"

\*Matched → i++ & j++

j == m

print("Found pattern at index", str(10-3))

## 

## PART 2: (c) TRIES

**Pseudocode**

1. **class** Node:
2. **def** \_\_init\_\_(self):
3. self.children = {}
4. self.last\_letter equal False
5. **class** Trie:
6. **def** \_\_init\_\_(self):
7. create root
9. **def** insert(self, word):
10. current as root
11. **for** ch **in** word:
12. **if** ch **not** **in** cur.children:
13. cur.children[ch] = Node()
14. cur = cur.children[ch]
15. cur.last\_letter equal True
17. **def** search(self, word):
18. current as root
19. **for** ch **in** word:
20. **if** ch **not** **in** cur.children:
21. **return** False
22. cur = cur.children[ch]
24. **if** cur.last\_letter:
25. **return** True
26. **else**:
27. **return** False

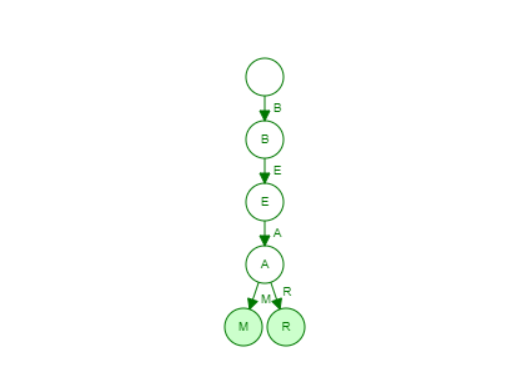
**Run Time Complexity**

**Searching and inserting in trie**

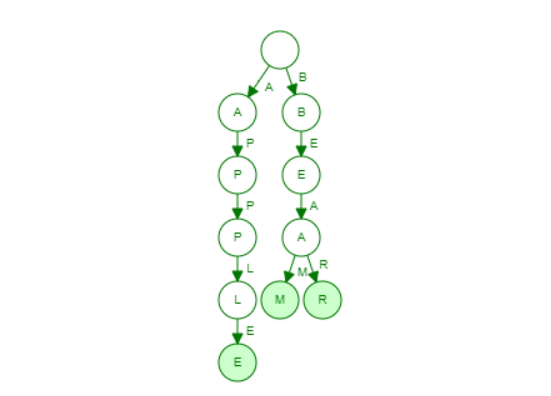
O(dm) where d = size of alphabets and m = size of the string parameter of the operation

**Insertion**

The best case insertion is when adding a word that have existent prefix in the trie.All words share the same prefix.For example adding word ‘bear’ where string ‘b’,’e’ and ‘a’ already present in trie and they share same prefix which is ‘b’. The time complexity is O(n)



The worst case insertion is when inserting a string ‘apple’.Since the link to ‘a’ is not available in the first current node itself,this will lead to creating a new node to store this word into trie as a new branch.The time complexity is O(n).



**Searching**

The best case search is when the prefix that needs to be searched is not present in the first current node then no need to traverse down further.The time complexity is O(1).

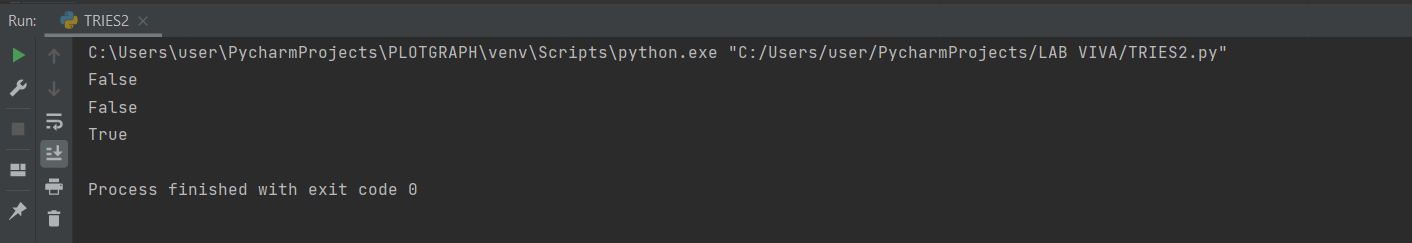
The worst case search is when you need to traverse the length if the word to find is in the trie and it depends on the height of the trie.The time complexity is O(n).

**Code**

1. **class** Node:
2. **def** \_\_init\_\_(self):
3. self.children = {}
4. self.last\_letter = False

7. **class** Trie:
9. **def** \_\_init\_\_(self):
10. self.root = Node()
12. **def** insert(self, word):
13. cur = self.root
15. **for** ch **in** word:
16. **if** ch **not** **in** cur.children:
17. cur.children[ch] = Node()
18. cur = cur.children[ch]
19. cur.last\_letter = True
21. **def** search(self, word):
22. cur = self.root
23. **for** ch **in** word:
24. **if** ch **not** **in** cur.children:
25. **return** False
26. cur = cur.children[ch]
28. **if** cur.last\_letter:
29. **return** True
30. **else**:
31. **return** False
33. **def** startwithPrefix(self, prefix):
34. cur = self.root
35. **for** ch **in** prefix:
36. **if** ch **not** **in** cur.children:
37. **return** False
38. cur = cur.children[ch]
39. **return** True
41. T = Trie()
42. T.insert("algorisfunalgoisgreat")
43. **print**(T.search("fun"))
44. **print**(T.search("algo"))
45. **print**(T.startwithPrefix("algo"))

**Output**



**Explanation**

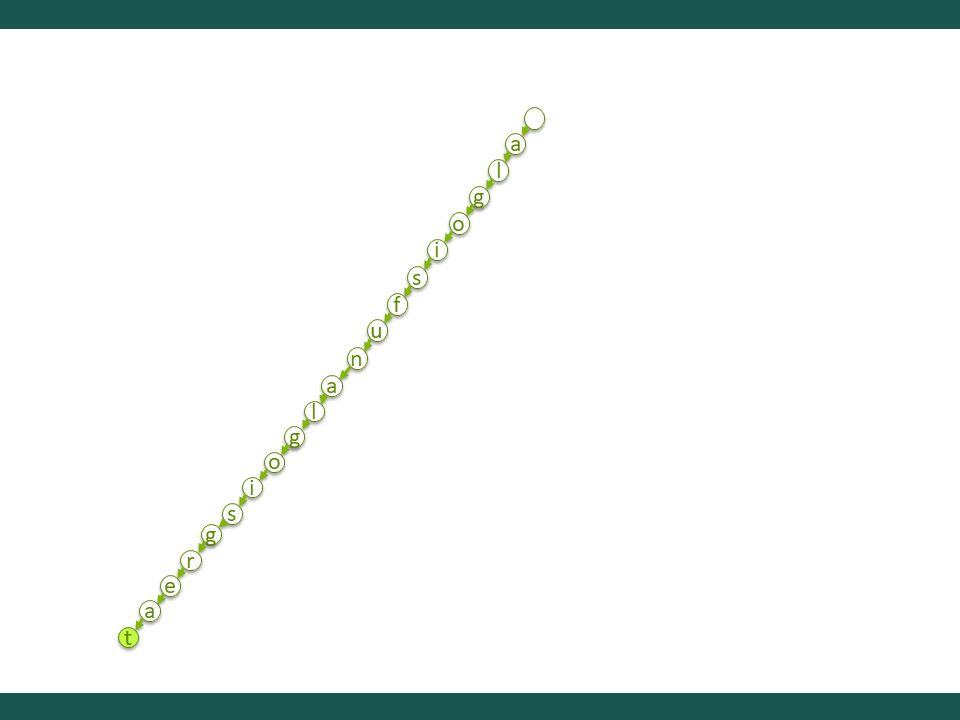
An implementation of this algorithm has been done in python. Basically, this is how the codes run:

1. A child node needs to be defined using a dictionary to store the first letter in words. So a class named Node has been created. Furthermore, in this class, self.last\_letter is defined as False to check the last letter of the word.

2.A class Trie has been created. Basically in this class, there are three important functions which are initial, insert and search and one optional function which is startwithPrefix .

**Initial Function (*Line 9*)** : A root node has been created but it does not store any character.

**Insert Function *(Line 12 )***: Insert function works to insert or add each of the characters in a child node using a *for* loop. If the current character is not present yet, it will add the character in a new node as that current child node. Once it is finished adding the character, it will look like this:-



and it will mark the last letter which is ‘t’ as True.

**Search Function *(Line 21 )*** : It will accept a string of words and similarly looks like an insert function that uses a *for* loop. The search function works to search whether the string exist in the nodes.It will match the same pattern of the letter with the one that has been stored. It will search through the children of the ‘current node’ to find the node containing that letter. If not found then it will return False to indicate that the character does not exist. For if statement cur.last\_letter, it will check if the last character of the string that has been searched is equal to the last character of the word to prove that the word that has been searched is a complete word.

**startwithPrefix (*Line 33*)** : It is similar to the search function but it will turn True if the first prefix that has been searched is equal with the first prefix of the words in trie.

3.Output is displayed.