Contents

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:

1.1 Counting Sort

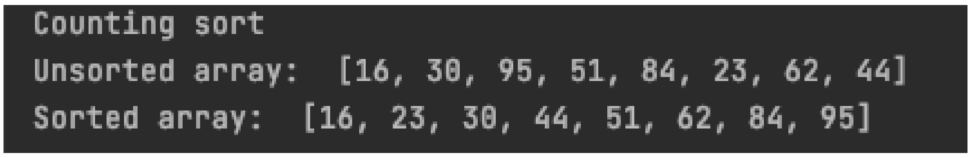
1.1.1 Pseudocode

```
let output[0...n] be array storing number sorted then
for i = 0 to n
    output[i] = 0
let count[0...k] be array storing occurrence
for i = 0 to k
    count[i] = 0

for i in array
    count [i] += 1
# count[i] = occurrence of number i in array

for j = 1 to k
    count[j] += count[j - 1]
for k = array.length down until 1
    output[count[array[k]] - 1]=array[k]
count[array[k] -= 1
```

1.1.2 Source Code

```python
def counting_sort(array):
    output = [0] * (len(array))
    count = [0] * (max(array) + 1)

    for i in array:
        count[i] += 1

    for j in range(1, max(array) + 1):
        count[j] += count[j - 1]

    for k in range(len(array) - 1, -1, -1):
        output[count[array[k]] - 1] = array[k]
        count[array[k]] -= 1

    return output
```

1.1.3 Output



```
Counting sort
Unsorted array:  [16, 30, 95, 51, 84, 23, 62, 44]
Sorted array:  [16, 23, 30, 44, 51, 62, 84, 95]
```

1.1.4 Explanation

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

First, *counting_sort* function takes in a list parameter. Next, we create a list to store an occurrence of values from 0 to the maximum number of the list to be sorted. This is done by using the *max* function. Then, we created a temporary list to store output to be sorted with size $len(array)$. For every unique value searched in the loop, we increase the corresponding count of the value in the list storing occurrence of value. We calculate cumulative counts by adding count of previous value to current value until loop reaches the end. Next, the program puts the number of the list from the end into a temporary list according to the index using cumulative counts of the value, decreasing the cumulative count of the respective value sorted in the temporary list. Lastly, we repeat the previous step until the first number in the list is reached.

1.1.5 Running Time Complexity

Type	**Code**	**Time Complexity**
max function	`count = [0] * (max(array)+1)`	$O(k)$
Multiplication with know range	`output = [0] * (len(array))`	$O(n)$
for loop	`for i in array:` `count[i] += 1`	$O(n)$
for loop	`for j in range(1, max(array) + 1):` `count[j] += count[j - 1]`	$O(k)$
for loop	`for k in range(len(array) - 1, -1, -1):` `output[count[array[k] - 1] = array[k]` `count[array[k]] -= 1`	$O(n)$

Thus, time complexity, $T(n) = O(k) + O(n) + O(n) + O(k) + O(n) = O(n + k)$ where k is maximum possible value in array and n is size of array.

1.2 Radix Sort

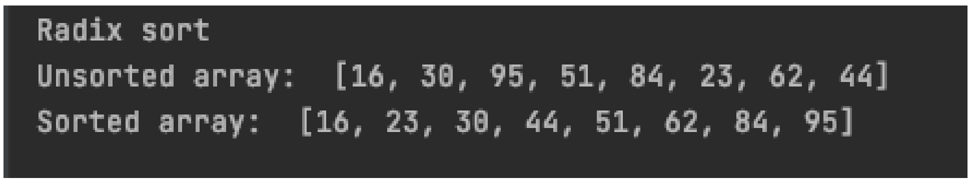
1.2.1 Pseudocode

```
mod_counting_sort(a,d)
    let output array length = number of list
    let count[0...size] be array storing occurrence
    for i = 0 to size
        index = arr[i] // d
        Count[index % 10] += 1
    let count array become cumulative count
        count[j] += count[j-1]

    let i = last element in arr
        Index = a[i] // d
        output[count[index % 10] - 1] = a[i]
        count[index % 10] -= 1
    convert the arr to output array

radix_sort(arr)
    Let maximum = max value in the array
    Let place = 1 start from first digit
        do when only digit is >0 while max/place >0:
            mod_counting_sort(arr,place)
            place*=10 for next tenth,hundredth
```

1.2.2 Source Code

```python
def radix_sort(arr):
    def mod_counting_sort(a, d: int):
        size: int = len(a)
        output = [0] * size
        count = [0] * (10)

        for i in range(0, size):
            index = arr[i] // d
            count[index % 10] += 1

        for j in range(1, 10):
            count[j] += count[j - 1]

        for i in range(size - 1, -1, -1):
            index = a[i] // d
            output[count[index % 10] - 1] = a[i]
            count[index % 10] -= 1
        # Deep copy array (for some reason output.copy() won't give the same assert
    result)
        for f in range(0, size):
            a[f] = output[f]

    maximum = max(arr)

    place = 1
    while maximum // place > 0:
        mod_counting_sort(arr, place)
        place *= 10

    return arr
```

1.2.3 Output



```
Radix sort
Unsorted array:  [16, 30, 95, 51, 84, 23, 62, 44]
Sorted array:  [16, 23, 30, 44, 51, 62, 84, 95]
```

1.2.4 Explanation

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

However, we need to know how many times we need to sort based on the biggest value. Therefore, we will require a function called $radix\_sort(arr)$ to obtain the maximum value in the array with $max_value = max(array)$. Then, we set it to start from the first digit which is $place = 1$. For the while loop, if the maximum value in the array divided by the place is less than zero, which means we have counted the digit to the biggest digit. After entering the while loop, we will need to apply counting sort by using the function $radix\_count\_sort$. We need to create temporary array which is $count[] * 10$ which is 10 slots for numbers 0-9. Then we need to create an output array $output[] * len(lst)$ based on the length of the array to place the output after sorted based on the digit. Next we will enter for loop for $i = 0$ to n to count the number of digits of the element and place it in the $count[]$ array. Then we make the $count[]$ array to become cumulative by $count[i] + = count[i - 1]$. Then we sort and place the value to the $output[]$ array.

1.2.5 Running Time Complexity

$mod\_counting\_sort(a, d)$ will be $O(n + k)$ whereas $radix\_sort$ includes counting sort which has a time complexity of $O(n + k)$. While it applies counting sort for every digit, therefore the time complexity for $radix\_sort$ is $T(n) = O(d(n + k))$ where $d =$ digit of maximum value.

1.3 Shell Sort

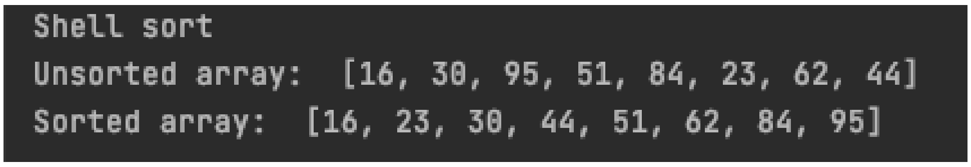
1.3.1 Pseudocode

```
let gap = length of array divided by 2
while gap is greater than 0
    let i = 0
let j = gap

while j less than size of arr
    if arr[i] greater than arr[j]
        swap the elements in arr[i] with arr[j]
    increment of i and j by 1

    let k = 1
    While k - gap greater than -1
        if arr[k - gap] greater than arr[k]
            swap the elements in arr[k - gap] with arr[k]
        decrement k by 1
    divide the gap by 2
return the output arr
```

1.3.2 Source Code

```python
def shellSort(arr):
    gap = len(arr) // 2  # initialize the gap

    while gap > 0:
        i = 0
        j = gap

        # check the array in from left to right
        # till the last possible index of j
        while j < len(arr):

            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]

            i += 1
            j += 1


            # now, we look back from ith index to the left
            # we swap the values which are not in the right order.
            k = i
            while k - gap > -1:

                if arr[k - gap] > arr[k]:
                    arr[k - gap], arr[k] = arr[k], arr[k - gap]
                k -= 1

        gap //= 2
    return arr
```

1.3.3 Output



```
Shell sort
Unsorted array:  [16, 30, 95, 51, 84, 23, 62, 44]
Sorted array:  [16, 23, 30, 44, 51, 62, 84, 95]
```

1.3.4 Explanation

Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted.

In the first while loop basically divides the gap by 2 for each iteration until the gap is less than 0. For each gap iteration, will enter the second while loop which will start comparing the array element index starting from the gap with the array element index starting from 0, for each iteration the comparing element index will move by 1 index to the right. For example, the 0^{th} element is compared with the 4^{th} element.

If the 0^{th} element is greater than the 4^{th} one then, the 4^{th} element is first stored in temp variable and the 0^{th} element (ie. greater element) is stored in the 4^{th} position and the element stored in temp is stored in the 0^{th} position. For the third while loop, look back from the index to the left and swap the elements which are not in correct order. All the processes continue until the gap is less than 1.

1.3.5 Running Time Complexity

Worst Case Complexity: less than or equal to $O(n^2)$
Worst case complexity for shell sort is always less than or equal to $O(n^2)$.
According to Poonen Theorem, worst case complexity for shell sort is $\Theta(\frac{nlogn)^2}{(loglogn)^2})$ or $\Theta(nlogn)^2/loglogn)$ or $\Theta(n(logn)^2)$or something in between.

Best Case Complexity: $O(n\ log\ n)$
When the array is already sorted, the total number of comparisons for each interval (or increment) is equal to the size of the array.

Average Case Complexity: $O(n\ log\ n)$
It is around $O(n^{1.25})$.

2 Implement the String-Matching Algorithm

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

2.1 Rabin-karp algorithm

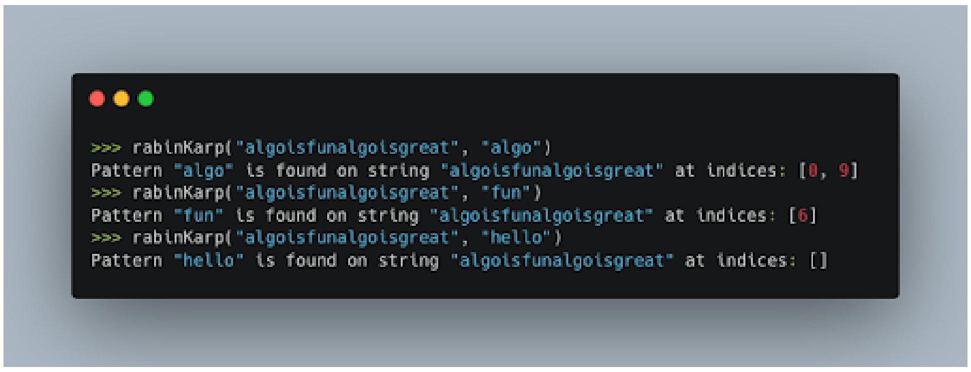
2.1.1 Pseudocode

```
Let N = length of text, M = length of pattern
Compute h = d^(M-1) mod q using for loop
Iterate M characters to calculate the hash value for p and t for hash value for pattern
    and text respectively.
Loop i from 0 to N-M
    If p equals t
        Check if each characters are the same using for loop
        If same, append i into indices
    Update t value by removing current ith character from t and add (i+M)th character to
     t (using hash)

Output indices which are matching with the pattern
```

2.1.2 Source Code

```python
# Rabin Karp Algorithm

def rabinKarp(text, pattern):
    """
        @param text:    text to search on
        @type  text:    str
        @param pattern: pattern to search
        @type  pattern: str
    """

    N = len(text)
    M = len(pattern)

    p = 0
    t = 0
    d = 256
    q = 1000000007
    h = 1

    for i in range(M-1):
        h = (h*d)%q

    for i in range(M):
        p = (d*p + ord(pattern[i]))%q
        t = (d*t + ord(text[i]))%q

    indices = []

    for i in range(N-M+1):
        if p==t:

            same = True
            for j in range(M):
                if text[i+j] != pattern[j]:
                    same = False
                    break

            if same:
                indices.append(i)

        if i < N-M:
            t = ((d*(t-ord(text[i])*h) + ord(text[i+M]))%q + q)%q

    print("Pattern \""+pattern+"\" is found on string \""+text+"\" at indices: "+str(
    indices))
```

2.1.3 Output



```
>>> rabinKarp("algoisfunalgoisgreat", "algo")
Pattern "algo" is found on string "algoisfunalgoisgreat" at indices: [0, 9]
>>> rabinKarp("algoisfunalgoisgreat", "fun")
Pattern "fun" is found on string "algoisfunalgoisgreat" at indices: [6]
>>> rabinKarp("algoisfunalgoisgreat", "hello")
Pattern "hello" is found on string "algoisfunalgoisgreat" at indices: []
```

2.1.4 Explanation

First of all the rabinKarp function accepts two parameters: text and pattern. N is the length of text while M is the length of pattern. We initialise p and t variables as 0, they are the rolling hash values for both pattern and text respectively. d is a random number as long as it is coprime with q and q is a prime number which is used for modulus. h is the variable is $d^{(M-1)}$ modulo q. Next, we iterate M times and calculate the hash value of pattern and text. We made an assumption here that the length of text is always greater than or equal to the length of the pattern.

We proceed by initializing indices to an empty list. Then we iterate from 0 to $N - M$, we compare the hash values of both p and t first, if they are the same, we check if all the characters are the same, if they are the starting index is added into indices. Regardless of whether p equals to t, we still update our t value by removing the current i^{th} character and adding the $(i+M)^{th}$ character. Lastly, we output the values of indices which are the starting index which match with the pattern.

2.1.5 Running Time Complexity

The running time complexity of Rabin Karp algorithm is $O(n+m)$ where n is the length of the text and m is the length of the pattern and its worst time complexity is $O(nm)$, which happens when spurious hits occur a number for all the windows.

2.2 KMP Algorithm

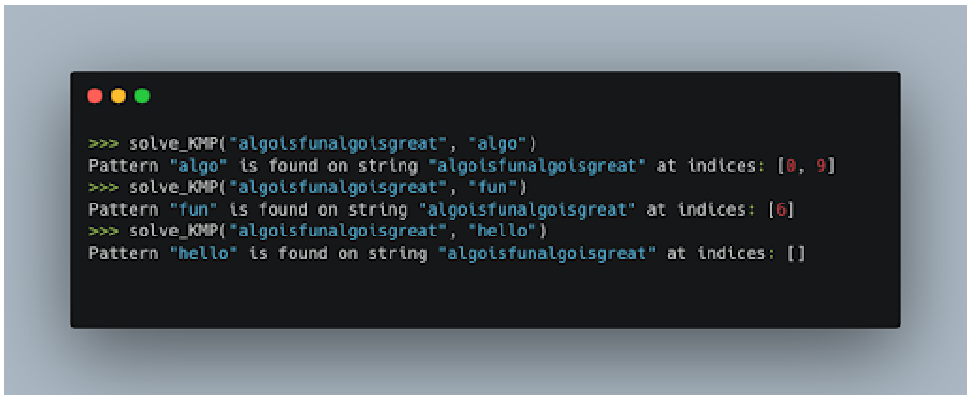
2.2.1 Pseudocode

```
Concatenate pattern, '!' and text as tmp

Perform kmp on tmp (compute prefix array aka pi)
S = length of tmp
Loop i from 1 to S-1
    Let j = pi[i-1]
    While j > 0 and ith character not equals to jth character
        Update j = pi[j-1]
    If ith character equals to jth character, increment j by 1
    assign pi[i] = j
Return pi

Let len_pattern = length of pattern
Loop i from M+1 to S-1 // loop text in tmp
    If pi[i] = M, index i-(M+1) is added into indices

Output indices matched
```

2.2.2 Source Code

```python
# Knuth-Morris-Pratt algorithm

def KMP(s):
    S = len(s)
    pi = [0] * S
    for i in range(1,S):
        j = pi[i-1]
        while j > 0 and s[i] != s[j]:
            j = pi[j-1]
        if s[i]==s[j]:
            j += 1
        pi[i] = j
    return pi

def solve_KMP(text, pattern):
    tmp = pattern + '!' + text
    pi = KMP(tmp)

    len_pattern = len(pattern)
    indices = []

    for i in range(len_pattern+1, len(pi)):
        if pi[i] == len_pattern:
            index = i - (len_pattern + 1) # remove the prefix length we made
            index -= len_pattern - 1      # start of index which matches
            indices.append(index)

    print("Pattern \""+pattern+"\" is found on string \""+text+"\" at indices: "+str(
    indices))
```

2.2.3 Output



```
>>> solve_KMP("algoisfunalgoisgreat", "algo")
Pattern "algo" is found on string "algoisfunalgoisgreat" at indices: [0, 9]
>>> solve_KMP("algoisfunalgoisgreat", "fun")
Pattern "fun" is found on string "algoisfunalgoisgreat" at indices: [6]
>>> solve_KMP("algoisfunalgoisgreat", "hello")
Pattern "hello" is found on string "algoisfunalgoisgreat" at indices: []
```

2.2.4 Explanation

KMP is known as Knuth-Morris-Pratt algorithm, we have two functions here, first one is KMP function itself and the second one is a solver function which uses KMP function.

KMP function accepts string s and returns an array, we called it pi in our code, where $pi[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$, which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. For example, prefix function of string “abcabcd” is [0,0,0,1,2,3,0], and prefix function of string “aabaaab” is [0,1,0,1,2,2,3].

First of all, we have a prefix array and we initialize the array with its size of the length of the string accepted S with all zeroes. We iterate from index, i, 1 to index $S - 1$, then j (the value to be assigned to $pi[i]$) is assigned to the value of $pi[j - 1]$, we have a while loop to check if current j is greater than 0 and the i^{th} character of s is not equal to j^{th} character of s, if it is true, j is assigned to $pi[j - 1]$ which is the previous index $j - 1$'s prefix value. If the ith character of the string s is equal to the j^{th} character of the string s, j is incremented by 1. Then the value $pi[i]$ is assigned to j. After iterating, the entire prefix array is returned.

Next function is the solver functions, it accepts two strings, the text and the pattern to find on the text. First of all, we create a temporary string, *tmp*, which is a concatenation of *pattern*, ‘!’ and *text*. Then we called the KMP function to know its prefix array. The reason behind the concatenation is to find the pattern easily on the temporary string. We made an assumption that both pattern and text do not have ‘!’ character, which then ‘!’ serves as the divider of the two strings. Since we have the prefix array of the string, we iterate from length of $pattern + 1$, which is the starting index of the *text* in *tmp*. We just check if the $pi[i]$ is equal to the length of the pattern. If it does, a pattern is matched from $(currentindex - lengthpattern - 1)$ to the current index. Then we append the index which is the start of the matching to the indices array. We then output the indices which match with the pattern (only starting indices).

2.2.5 Running Time Complexity

Denote n is the length of the text and m is the length of the pattern. The overall time complexity is $O(n + m)$. The concatenation of the *tmp* string takes $O(n + m)$. The KMP function iterates the entire *tmp* string from index 1 to index $S - 1$. The while loop takes $O(1)$ time since $j = pi[j - 1]$ helps to compress. Last iteration from the for loop in the solver function also takes $O(n + m)$ time.

2.3 TRIES

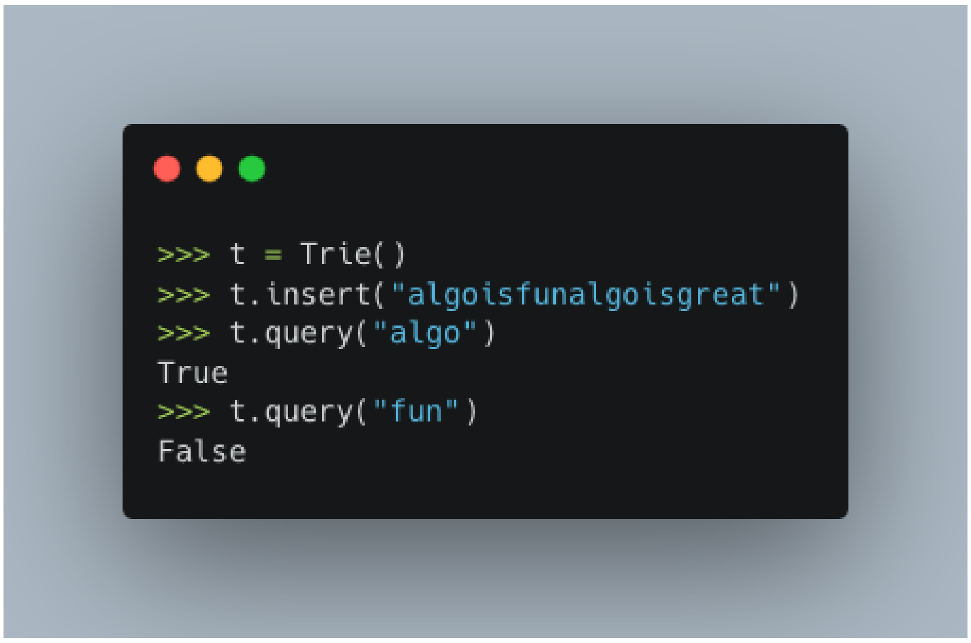
2.3.1 Pseudocode

```
Insert method
    Let node = root (TrieNode)
    Loop character from the text
        If character is not found in node.children:
            Create new TrieNode with character
            Add new TrieNode into node.children
        Update node = node.children[character]

Query method
    Let node = root (TrieNode)
    Loop character from the text
        If character is not found in node.children:
            Return False
        Update node = node.children[character]
    Return True
```

2.3.2 Source Code

```python
# TRIES

class TrieNode:
    """A node in the trie structure"""

    def __init__(self, char):
        # the character stored in this node
        self.char = char

        # whether this can be the end of a word
        self.is_end = False

        # a dictionary of child nodes
        # keys are characters, values are nodes
        self.children = {}

class Trie(object):
    """The trie object"""

    def __init__(self):
        """
        The trie has at least the root node.
        The root node does not store any character
        """
        self.root = TrieNode("")

    def insert(self, word):
        """Insert a word into the trie"""
        node = self.root

        # Loop through each character in the word
        # Check if there is no child containing the character, create a new child for
    the current node
        for char in word:
            if char not in node.children:
                # If a character is not found,
                # create a new node in the trie
                new_node = TrieNode(char)
                node.children[char] = new_node
            node = node.children[char]

        # Mark the end of a word
        node.is_end = True

    def query(self, word):
        node = self.root

        # Check if the prefix is in the trie
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

2.3.3 Output



```
>>> t = Trie()
>>> t.insert("algoisfunalgoisgreat")
>>> t.query("algo")
True
>>> t.query("fun")
False
```

The reason that “fun” is not found in “algoisfunalgoisgreat” is because Trie uses prefix to match. Hence, “fun” is not found as it does not share the same prefix as “algoisfunalgoisgreat”.

2.3.4 Explanation

First of all, we created a TrieNode structure, it has its character first, then it has a boolean to check if current TrieNode is the end of the word, it has its own children which is in dictionary form, which are the next characters of TrieNode.

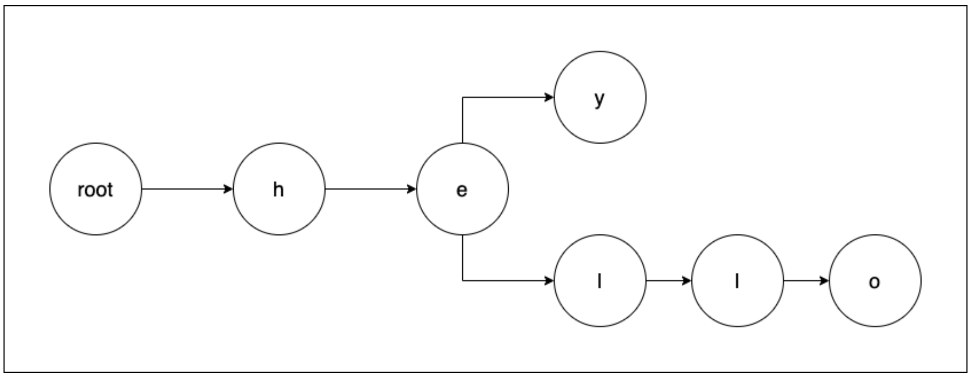
Next, we created a Trie data structure, for the *init* function, we initialize a TrieNode which is called root. We also create an insert function, we initialize node as the root and then iterate each character in the word. If the character is not found in the node children, we create a new TrieNode and place it in the current node’s children. Then we will move to its children node using $node = node.chilren[char]$. After iterating all the characters, we eventually come to the end of the TrieNode and we mark is_end as true.

The next function is query function, this is to query the word we are looking for whether it is added in the Trie. We initialise node as the Trie’s root. Next, we iterate through characters, along with assign our node to the current node’s children with tallies with the current character. If the character is not found in the node’s children means, the word queried does not exist, then we return false. Or else, we keep iterating until the end of the word and return true.

2.3.4.1 Justification

Trie is not suitable to do pattern matching, the only thing it could match is prefixes. If it requires to do matching, we could add all the suffixes of the text into the trie before checking the pattern. The reason why we should add all the suffixes is because trie is comparing prefixes. If we really do this method, it is not utilising trie data structure as the time complexity reaches $O(n^2)$ given n is the length of the pattern, this will be the same approach compared to the naive brute force method which compares each substring matches with the pattern.

Trie is suitable to find a word instead of pattern matching. Strings which shared the same prefixes will go through the same path / node initially, in which has utilised the space need. A good use case of trie is dictionary. For an example, we have 10’000 words to insert in our dictionary. Given the words, 'hello' and 'hey' share the same prefix – 'he'. Both words utilise the space from the root node to node 'e'.



2.3.5 Running Time Complexity

The time complexity is $O(n)$, given that n is the length of the word to insert or query.