ECE 522 Assignment 5

Zhaoyi Wang 1689747

Question 1

For Question a)

In one sentence, this program finds a given word appears in the which sentence and also finds where it appears in that sentence.

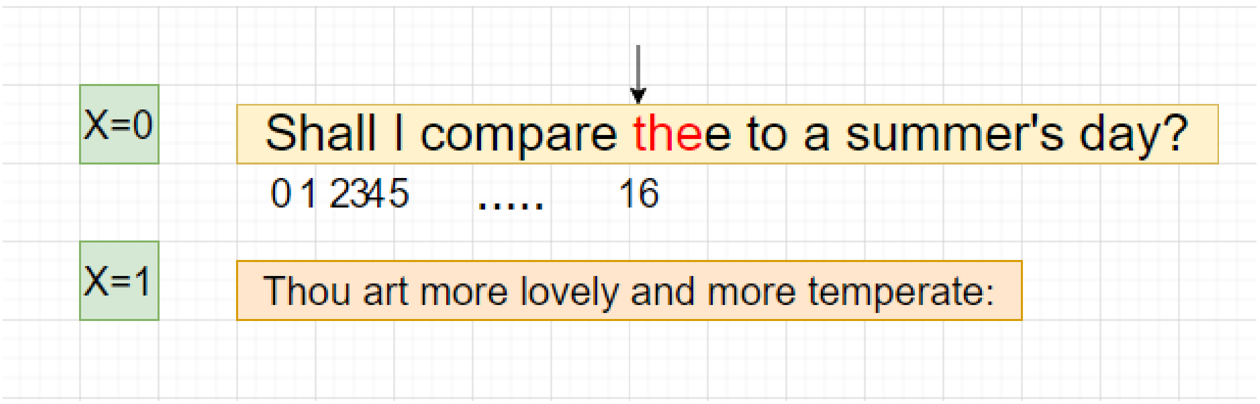
> For example: "Hi, everyone!" `Hi` appears in `sentence [0]`, the exact location is `index=0` in `sentence [0]`.

To be specific, First, this `r` variable is used to save the results. We can think of the `x` variable as the index of each sentence, for example, the index of the first sentence is 0. We iterate through all sentences by the first loop, and, at the same time, through each word in each sentence by the second loop. If the corresponding word is found in the sentence, we will save its position.

For Question b)

```
coder@ubuntu-s-1vcpu-2gb-tor1-01:~/personalProj/rusttest/Assign5/A5T1$ cargo run
   Compiling A5T1 v0.1.0 (/home/coder/personalProj/rusttest/Assign5/A5T1)
    Finished dev [unoptimized + debuginfo] target(s) in 2.14s
     Running `target/debug/A5T1`
x : 0, y : 16
x : 2, y : 21
x : 4, y : 18
x : 12, y : 23
x : 13, y : 42
```

> Take `x:0, y:16` as an example


X=0
Shall I compare thee to a summer's day?
0 1 2345 16
X=1
Thou art more lovely and more temperate:

Question 2

```rust
pub struct L {
    x: usize,
    y: usize,
}

pub fn foo(text: &str, string: &str) -> Vec<L> {
    text.lines()
        .enumerate()
        .flat_map(|(x, line)| {
            line.match_indices(string).map(move |(y, _)| { L { x, y } })
        })
        .collect()
}
```

For the output:

```
coder@ubuntu-s-1vcpu-2gb-tor1-01:~/personalProj/rusttest/Assign5/A5T1$ cargo run
   Compiling A5T1 v0.1.0 (/home/coder/personalProj/rusttest/Assign5/A5T1)
    Finished dev [unoptimized + debuginfo] target(s) in 1.89s
     Running `target/debug/A5T1`
x : 0, y : 16
x : 2, y : 21
x : 4, y : 18
x : 12, y : 23
x : 13, y : 42
```

Question 3

```rust
use std::borrow::{Borrow, BorrowMut};
use std::collections::HashMap;
use std::ops::Deref;

#[derive(Debug)]
struct TrieNode {
    chs: HashMap<char, TrieNode>,
    value: Option<i32>,
}

impl TrieNode {
    // Get the length
    fn length(&self) -> usize {
        let mut length: usize = 0;
        match &self.chs.is_empty() {
            // if this node is not empty, length add 1
            false => {
                length = length + 1;
            }
            _ => (),
        };

        for (_, trie_node) in &self.chs {
            length += trie_node.length();
        }
        length
    }

    // Returns an iterator
    fn iter(&self) -> Vec<(char, Option<i32>)> {
        let mut iter_vec = Vec::new();
        for (char, node) in &self.chs {
            match node.value {
                Some(val) => iter_vec.push((*char, Some(val))),
                None => iter_vec.push((*char, None)),
            }
            iter_vec.append(&mut node.iter())
        }
        iter_vec
    }

    // Search the trie for a given key
    fn find(&self, key: &String) -> Option<&TrieNode> {
        let mut current_node = self;
        for c in key.chars() {
            match current_node.chs.get(&c) {
                Some(node) => current_node = node,
                None => return None,
            }
        }
        Some(current_node)
    }
}

#[derive(Debug)]
struct Trie {
    root: TrieNode,
}

impl Trie {
    fn new() -> Trie {
        Trie {
            root: TrieNode {
                chs: HashMap::new(),
                value: None,
            },
        }
    }
    fn add_string(&mut self, string: String, value: i32) {
        let mut current_node = &mut self.root;
        for c in string.chars() {
            current_node = current_node.chs
                .entry(c)
                .or_insert(TrieNode {
                    chs: HashMap::new(),
                    value: None,
                });
        }
        current_node.value = Some(value);
    }

    // Remove a key
    fn delete(&mut self, key: &String) -> Option<i32> {
        if key.is_empty() {
            // if key is empty, no need to delete
            return None;
        }
        let mut current_node = &mut self.root;
        for (ind, ch) in key.chars().enumerate() {
            if ind < key.len() - 1 {
                match current_node.chs.get_mut(&ch) {
                    Some(node) => {
                        current_node = node;
                    }
                    None => return None,
                }
            }
        }
        // here current_node is actually the previous node of the deleted node
        let temp = current_node.chs.remove(&key.chars().last().unwrap());
        match temp {
            Some(node) => node.value,
            None => None,
        }
    }
}

fn main() {
    let mut trie = Trie::new();
    trie.add_string("B".to_string(), 1);
    trie.add_string("Bar".to_string(), 2);
    println!("This Trie: {:?}", trie);
    println!("Length of Trie: {:?}", trie.root.length());

    let iter = trie.root.iter();
    println!("Iter: {:?}", iter);

    println!("Find: {:?}", trie.root.find(&String::from("B")));

    let removed = trie.delete(&String::from("B"));
    println!("Remove: {:?}", removed);
}
```

For the output:

```
This Trie: Trie { root: TrieNode { chs: {'B': TrieNode { chs: {'a': TrieNode { chs: {'r': TrieNode { chs: {}, value: Some(2) }}, value: None }}, value: Some(1) }}, value: None } }

Length of Trie: 3

Iter: [('B', Some(1)), ('a', None), ('r', Some(2))]

Find: Some(TrieNode { chs: {'a': TrieNode { chs: {'r': TrieNode { chs: {}, value: Some(2) }}, value: None }}, value: Some(1) })

Remove: Some(1)
```