PA1 Report

Shane Drafahl

29 October,2017

War With Array

```
// S: the set of k length sub strings
function compute2k(S[], k) {
    size = 0
    2K[]
    x = 0
    for(x in range(s)) {
        y = 0
        for(y in range(s)) {
            concat = concat(s[x] + s[y]);
            if(validString(S, concat)) {
                2K[size] = concat
                size++
            }
        }
    }
    return 2K
}

function validString(S[], string) { // Checks if the string given is a po
    t = 1
    subSets = s.length - k // number of substrings to check
    for(t in range(subSets)) {
        substring = sub(string, t, t+k) // gets the substring of string b
        if(!isInDictionary(S, substring)) {
            return false
        }
```

```
    }
    return true
}

isInDictionary(S[], substring) {
    b = 0
    for(b in range(S)) {
        if(S[b] == substring) {
            return true
        }
    }
    return false
}
```

This algorithm concats each 1k substring with every other 1k substring and then checks if the string is valid with an array. There are $O(n^2)$ 2k substrings. Each string has k-1 substrings to be checked. To validate if those substrings are a substring must check n substrings and the comparison takes k time because string comparison is linear. So therefore the runtime of this algorithm is $O(n^3(k^2 - k))$.

War With Binary Tree

```
function buildTree(S[], int k)
{
    Node n // new node
    n.value = s
    T // new binary tree
    T.root = n
    x = 0
    for(x in range(s)) {
```

```
        Node newNode
        newNode.value = S[x]
        Node temp = T.root
        while(true) {
            if(temp.value < newNode.value)) {
                newNode.parent = temp;
                if(temp.right == null) {
                    temp.right = newNode;
                    break;
                } else {
                    temp = temp.right;
                }
            } else {
                if(temp.left == null) {
                    temp.left = newNode;
                    break;
                } else {
                    temp = temp.left;
                }
            }
        }
    }
    return T
}

function contains(T, string) { // BST method
    if(T.root == null) {
        return false;
    } else {
        Node temp = T.root;
        while(true) {
            if(temp.valu == string) {
                return true;
            } else {
                if(temp.value < string) {
                    if(temp.right == null) {
                        return false;
                    } else {
                        temp = temp.right;
                    }
```

```
                } else {
                    if(temp.left == null) {
                        return false;
                    } else {
                        temp = temp.left;
                    }
                }
            }
        }
    }
}

// S: the set of k length sub strings
// T: binary tree with k length substrings
function compute2k(S[], T, k) {
    size = 0
    2K[]
    x = 0
    for(x in range(s)) {
        y = 0
        for(y in range(s)) {
            concat = concat(s[x] + s[y]);
            if(validString(S, T, concat)) {
                2K[size] = concat
                size++
            }
        }
    }
    return 2K
}

function validString(S[], T, string) { // Checks if the string given
    t = 1
    subSets = s.length - k // number of substrings to check
    for(t in range(subSets)) {
        substring = sub(string, t, t+k) // gets the substring of stri
        if(!contains(T, substring)) {
            return false
        }
    }
```

```
        return true
    }
```

This algorithm is similar to war with array but on average it will take the binary tree $O(log(n))$ to find the string. Therefore for compute2k it takes $O(log(n)n^2(k^2 - k))$. To construct a binary tree it will take $O(n * log(n))$.

War With Hash

This algorith uses a hashset as a hash set as a dictionary algorithm.

```
function setsUpHash(S[], H) {
    x = 0
    for(x in range(S)) {
        insert(s[x])
    }
}

// S: the set of k length sub strings
// H hashset that contains k length substrings
function compute2k(S[], H, k) {
    size = 0
    2K[]
    x = 0
    for(x in range(s)) {
        y = 0
        for(y in range(s)) {
            concat = concat(s[x] + s[y]);
            if(validString(S, H, concat)) {
                2K[size] = concat
                size++
```

```
            }
        }
    }
    return 2K
}

// S set of k length substrings
function validString(S[], H, string, k) { // Checks if the string given i
    t = 1
    for(t in range(k-1)) {
        substring = sub(string, t, t+k) // gets the substring of string b
        if(!contains(H, substring)) { // contains() hashes the substring
            return false
        }
    }
    return true
}
```

On averge it should take constant time to search this algorithm so therefore the runtime of this algorithm is $O(n^2(k^2 - k))$. The function to add all the k length substrings to the hashset should take $O(n)$ time.

```
function createHash(S[], k)
{
    H[] // hash values
    x = 0
    for(x in range(S)) {
        H[x] = S[x]
    }
}

// S: the set of k length sub strings
function compute2k(S[], k) {
```

```
    size = 0
    2K[]
    x = 0
    for(x in range(s)) {
        y = 0
        for(y in range(s)) {
            concat = concat(s[x] + s[y]);
            if(validString(S, concat)) {
                2K[size] = concat
                size++
            }
        }
    }
    return 2K
}

// H the hash in an array of the k length substrings
function checkString(S[], H[], string, PRIME, k) {
    index0 = 1
    index2 = k
    String substring = substring(string, index0, index2)
    sum = hashString(substring)
    match = true
    while(true) {
        match = false;
        x = 0
        M[] //possible matches
        size = 0 //size of M
        for(x in range(H)) {
            if(sum == H[x]) {
                M[size] = S[x]
                size++
            }
        }
        x = 0
        if(contains(M, substring)) { // checks if substring is locatated
            match = true
        }
        if(match == false) {
            return false;
```

```
            } else if(index2 == length(string) - k + 1) {
                return true;
            } else {
                sum -= substring[0]
                sum /= PRIME
                sum += string[index2] * (PRIME^(k-1));
                substring = substring(substring, 1, length(substring))
                substring = concat(substring, string[index2])
                index0++
                index2++
            }
        }
    }

    function hashString(string, PRIME) {
        sum = 0
        x = 0
        for(x in range(string)) {
            character = string[x]
            sum = sum + (character *  PRIME^x)
        }
        return sum
    }
```

This algorithm is the most different in that it attempts to optomize the process of checking the substrings in a given string. This algorithm first like the others concats all the k length substrings but whats different is it creates a array to store the hashvalues for the k length strings in a array. This means when it is checking the k-1 substring if the hash value is different it does not need to do any string comparison. Worst case scenario is its comparing a valid substring and has to do string comparison every time. So therefore the runtime is $O(n^3(k^2 - k))$.