ENHANCE MCCREIGHT ALGORITHM TO BUILD WORDLIST FOR HASHCAT

BY JUSTIN PABON

BACKGROUND

- Passwords are commonly stored as hashed values on server
- Leaked Hashed Passwords can be attacked by Hashcat
- Hashcat's attack mode:
    - Dictionary attack
    - Combinator attack
    - Brute-force attack and Mask attack
    - Hybrid attack
    - Rule-based attack
    - Toggle-case attack

BACKGROUND (CONT'D)

- In previous research, our group used Hashcat to attack 1,000,000+ leaked passwords, about 89% of them were cracked. It has been difficult to increase the ratio.
- In the previous research, we find most cracked passwords use some words from wordlist; brute-force cracked very few passwords.
- We wonder:
  - Would an enhanced wordlist increase the cracking ratio?
- Ideally, this wordlist should be
  - Comprehensive: General enough to be applied to as many passwords as possible
  - Tight: We don't want a wordlist that is too exhaustive to go through.

OPTIONS FOR WORDLIST

Mac dictionary

- 90,000 out of 235,886 used

Rockyou list

- 6 millions out of 13 millions used

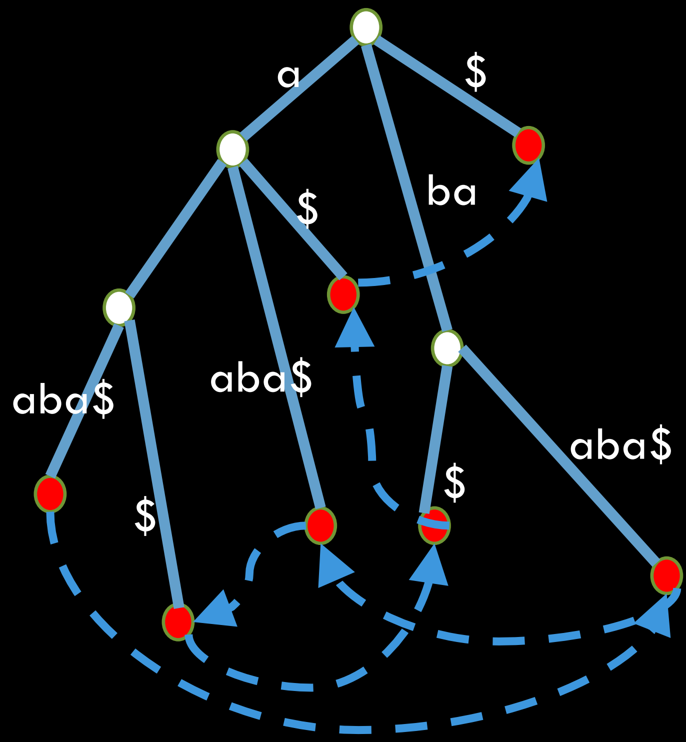
Traditional dictionaries

- Do not include commonly used strings.
- E.g. “qwerty”, “iloveyou”, “hahaha”, “test1234”

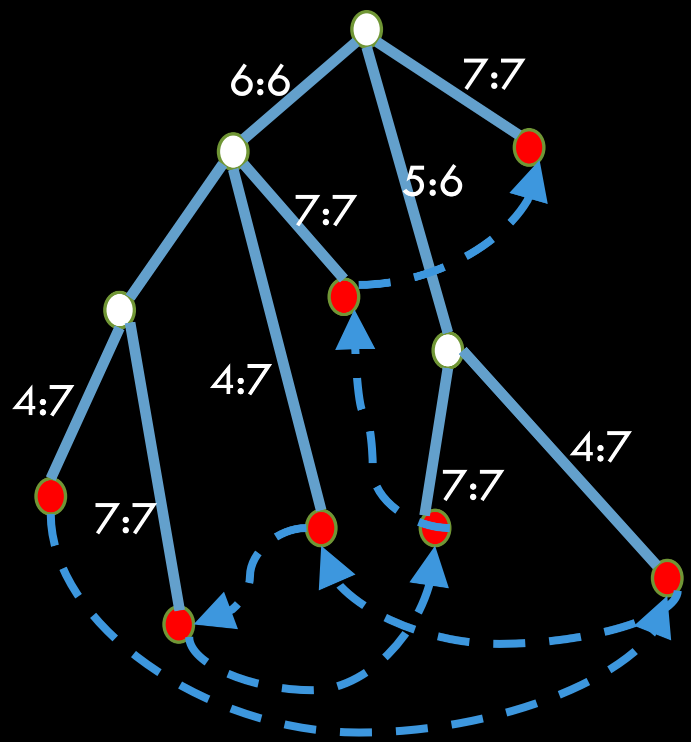
HOW DO WE FIND THESE COMMON SUBSTRINGS?

- Suffix Array
  - The tree data structure made to store every suffix of a given string.
- Generalized Suffix Tree
  - An implementation of a suffix tree that supports multiple strings.
  - Can be built using McCreight's Algorithm.

SUFFIX TREES

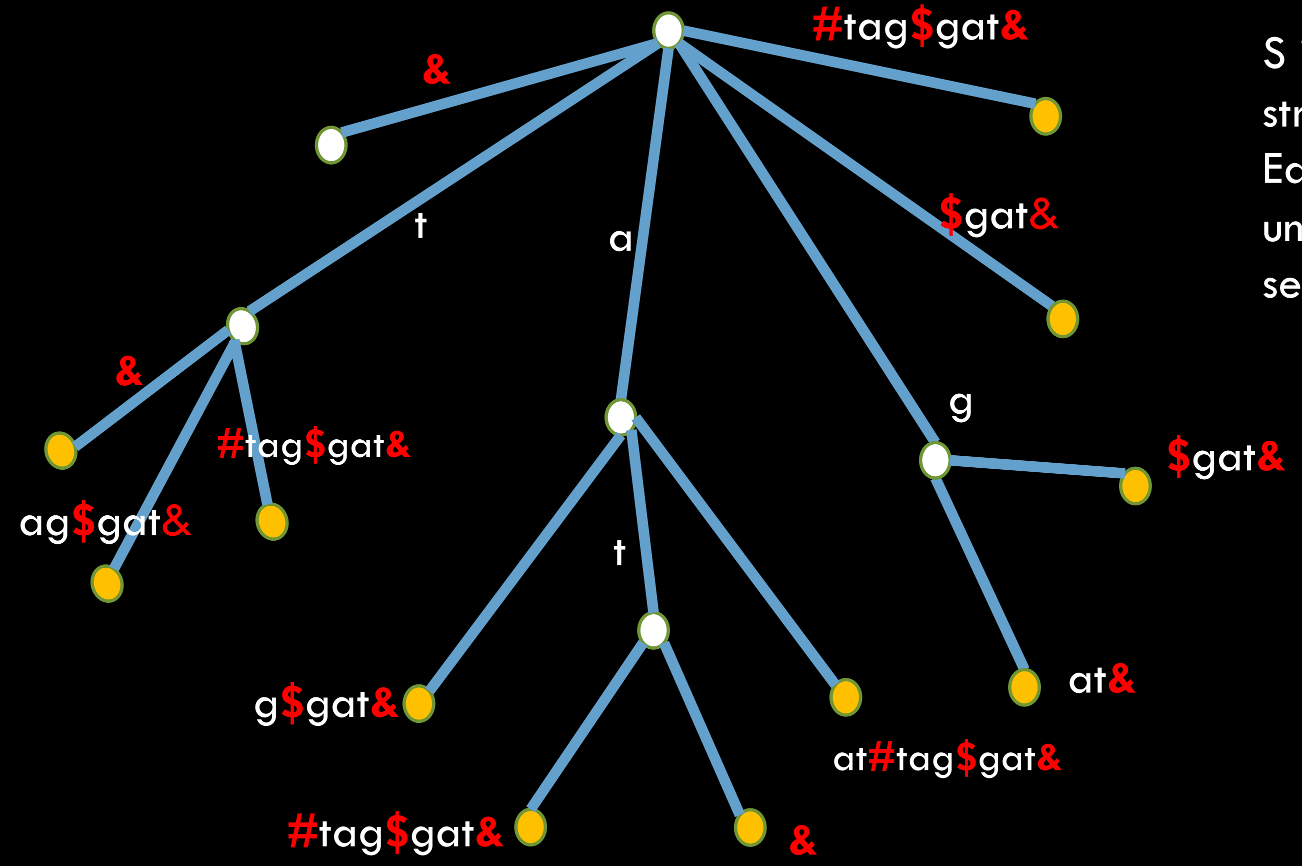


S = abaaba$
1234567



S = abaaba$
1234567

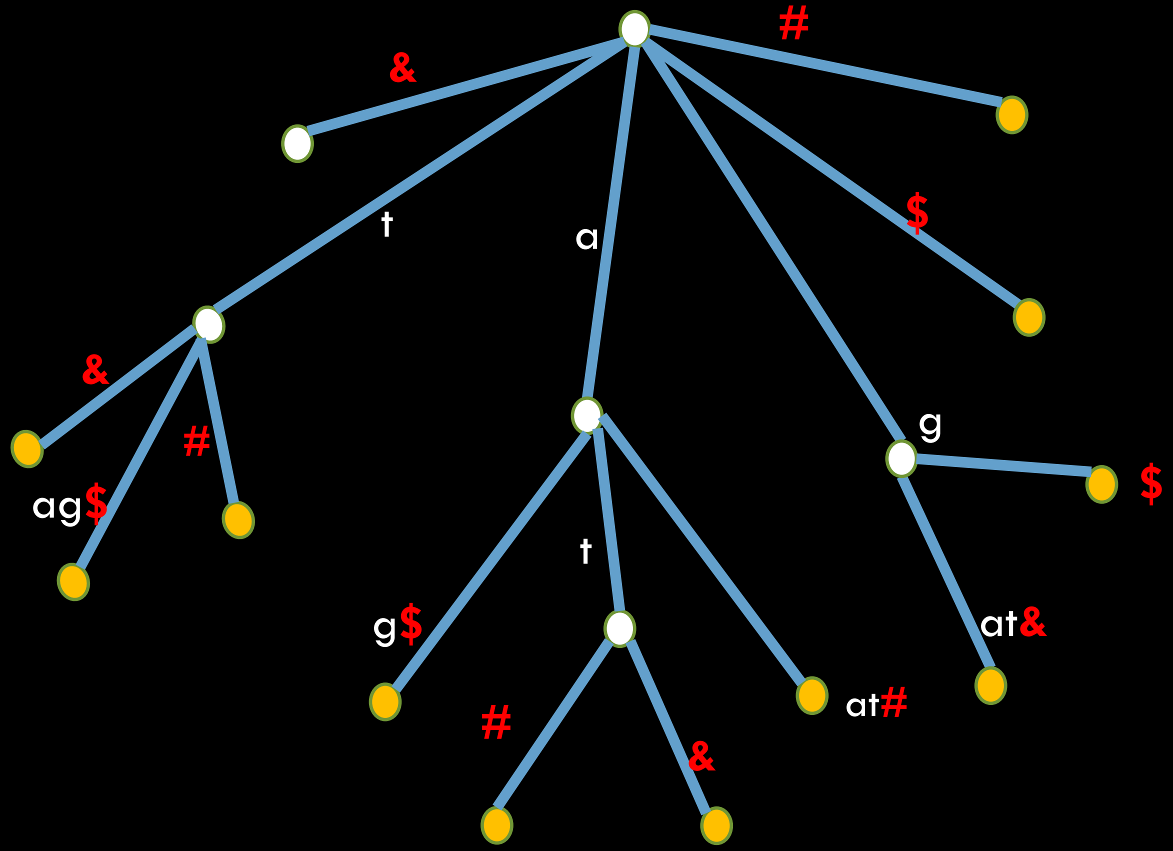
GENERALIZED SUFFIX TREES



S is the concatenation of three strings: att, tag and gat.
Each string is appended with a unique terminating character to separate the strings.

S = att#tag$gat&

GENERALIZED SUFFIX TREES (CONT'D)



We remove suffixes after every terminating character. This way, each path in the tree represents the suffix of a specific string.

S = att#tag$gat&

GOALS

- Modify McCreight's Algorithm to support millions of passwords.
- Collect information from our tree by traversing through it and creating a new wordlist of the most common substrings.
- Attack passwords using our new wordlist with the hope of increasing the ratio.

FIRST MODIFICATION: TERMINATING CHARACTER

- The original code generates a unique terminating character for each string using Unicode.
- Unicode cannot create enough unique terminating characters for millions of passwords.
- Modification: Return the same Unicode character for every password. We'll change them later to distinguish each one.

```python
    def _terminalSymbolsGenerator(self):
        """Generator of unique terminal symbols used for building the Generalized Suffix Tree.
        Unicode Private Use Area U+E000..U+F8FF is used to ensure that terminal symbols
        are not part of the input string.
        """
        # UPPAs = list(list(range(0xE000, 0xF8FF+1)) +          #ORIGINAL
        #              list(range(0xF0000, 0xFFFFD+1)) + list(range(0x100000, 0x10FFFD+1)))     #ORIGINAL


        # for i in UPPAs:     #ORIGINAL
        #     yield (chr(i))  #ORIGINAL

        return chr(0xE000) #MODIFICATION

        raise ValueError("To many input strings.")
```

MCCREIGHT ALGORITHM (PSEUDOCODE)

- Input: Text T[0…n], T[n] = $
- Output: suffix tree of T: root, child, parent, depth, start, slink

1. create new node root; depth(root) ← 0; slink(root) ← root
2. u ← root; d ← 0 // (u, d) is the active locus
3. for i ← 0 to n do // insert suffix Ti
4. while d = depth(u) and child(u, T[i + d]) 6= ⊥ do
5. u ← child(u, T[i + d]); d ← d + 1
6. while d < depth(u) and T[start(u) + d] = T[i + d] do d ← d + 1
7. if d < depth(u) then // (u, d) is in the middle of an edge
8. u ← CreateNode(u, d)
9. CreateLeaf(i, u)
10. if slink(u) = ⊥ then ComputeSlink(u)
11. u ← slink(u); d ← d − 1

Time Complexity: O(m)

MCCREIGHT'S ALGORITHM (THE CODE)

```python
    def _build_McCreight(self, x):
        """Builds a Suffix tree using McCreight O(n) algorithm.

        Algorithm based on:
        McCreight, Edward M. "A space-economical suffix tree construction algorithm." - ACM, 1976.
        Implementation based on:
        UH CS - 58093 String Processing Algorithms Lecture Notes
        """
        u = self.root
        d = 0
        for i in range(len(x)):
            while u.depth == d and x[i + d] != chr(0xE000) and u._has_transition(x[d + i]): #MODIFICATION
            # while u.depth == d and u._has_transition(x[d + i]): #ORIGINAL
                u = u._get_transition_link(x[d + i])
                d = d + 1
                while x[u.idx + d] != chr(0xE000) and x[i + d] != chr(0xE000) and d < u.depth and x[u.idx + d] == x[i + d]:   #GOOD MODIFICATION
                # while d < u.depth and x[u.idx + d] == x[i + d]:      #ORIGINAL
                    d = d + 1
            if d < u.depth:
                u = self._create_node(x, u, d)
            self._create_leaf(x, i, u, d)
            if not u._get_suffix_link():
                self._compute_slink(x, u)
            u = u._get_suffix_link()
            d = d - 1
            if d < 0:
                d = 0
```

CREATE LEAF (MODIFIED)

- All child nodes are stored in a parent node's set as transition links.
- Each entry in the set uses a character as a unique key to access a child node.
- To make each terminating character unique, we append the character's index value (idx) to the terminating character.

```
    def _create_leaf(self, x, i, u, d):
        w = _SNode()
        w.idx = i
        w.depth = len(x) - i
        # u._add_transition_link(w, x[i + d])   #ORIGINAL

        if x[i + d] == chr(0xE000):           #MODIFICATION
            item = x[i + d] + str(w.idx)     #MODIFICATION
            u._add_transition_link(w, item) #MODIFICATION
        else:                                 #MODIFICATION
            u._add_transition_link(w, x[i + d]) #MODIFICATION

        w.parent = u
        return w
```

SECOND MODIFICATION: NON-RECURSIVE TRAVERSAL

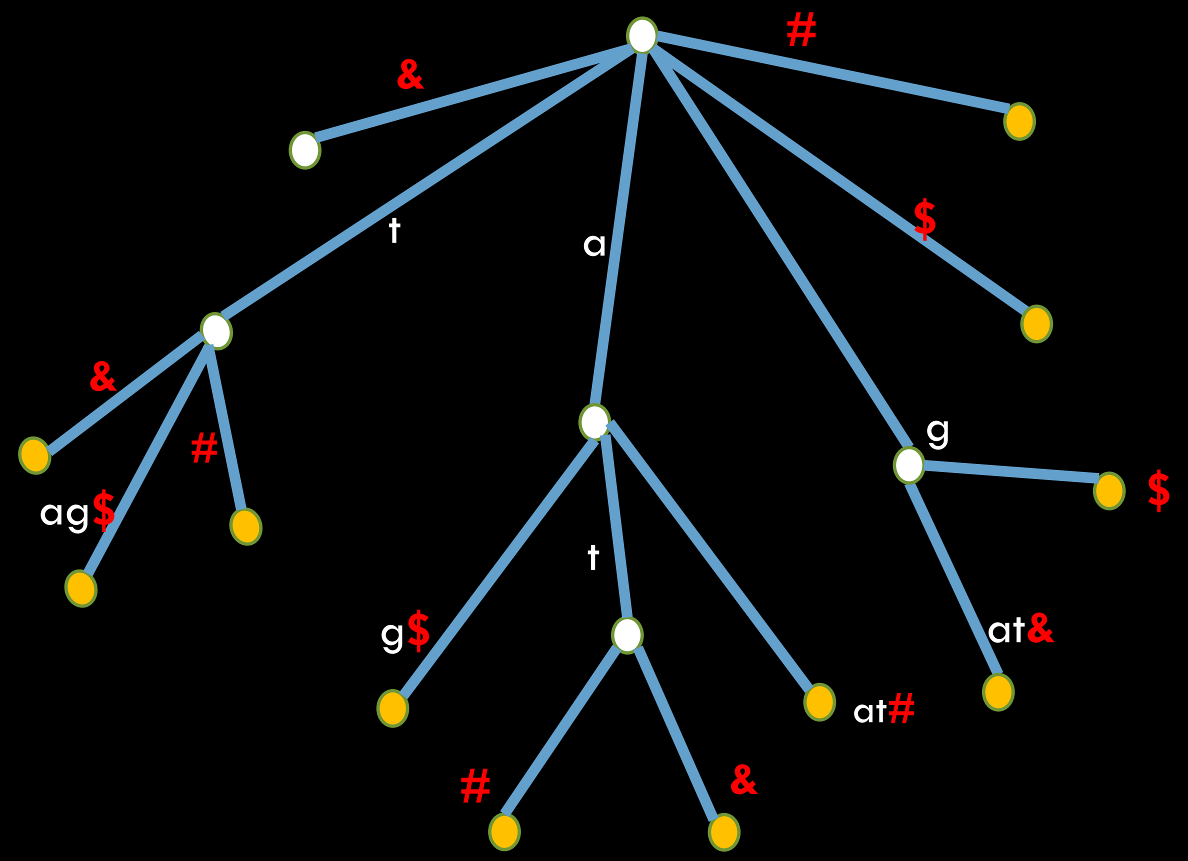
- Each node has a set called: “unvisited_links”. It is initialized to be empty when the node is created.
- The function starts from the root and then traverses down until reaching a leaf. When it moves down into a new node, we copy its transition_links set to unvisited_links set.
- If a node has an empty unvisited_links set, then it must be a leaf or every child of this node is already visited.
- Pop the node out of its parent’s univisited_links set. Then move up to the parent node and add the last visited node’s number of leaves to the parent node’s.
- Repeat this process until the unvisited_links set of root is empty.

TRAVERSAL CODE

```
def traversal(self):
    node=self.root

    file1 = open("myfile.txt","w")
    file1.write("Node   #ofLeaves" + '\n')

    self.root.unvisited_links=self.root.transition_links.copy()

    while len(self.root.unvisited_links)!=0:
        if len(node.unvisited_links)!=0:
            node = node.unvisited_links[next(iter(node.unvisited_links))]
            node.unvisited_links = node.transition_links.copy()
        else:
            if len(node.transition_links)==0:
                node.nL = 1 # Node is a leaf; set number of leaves, nL, to 1

            else: # Internal node; Write its info into txt file if edge >= 3 chars
                edge = self._edgeLabel(node, node.parent)
                if len(edge) >= 3:
                    file1.write(edge + "   " + str(node.nL) + '\n')

            node.unvisited_links.clear()
            leafNum = node.nL
            node = node.parent
            node.nL = node.nL + leafNum # Add the visited node's number of leaves to the parent node's.
            node.unvisited_links.pop(next(iter(node.unvisited_links))) # Pop the visited node out of parents unvisited_links
    file1.close()
```

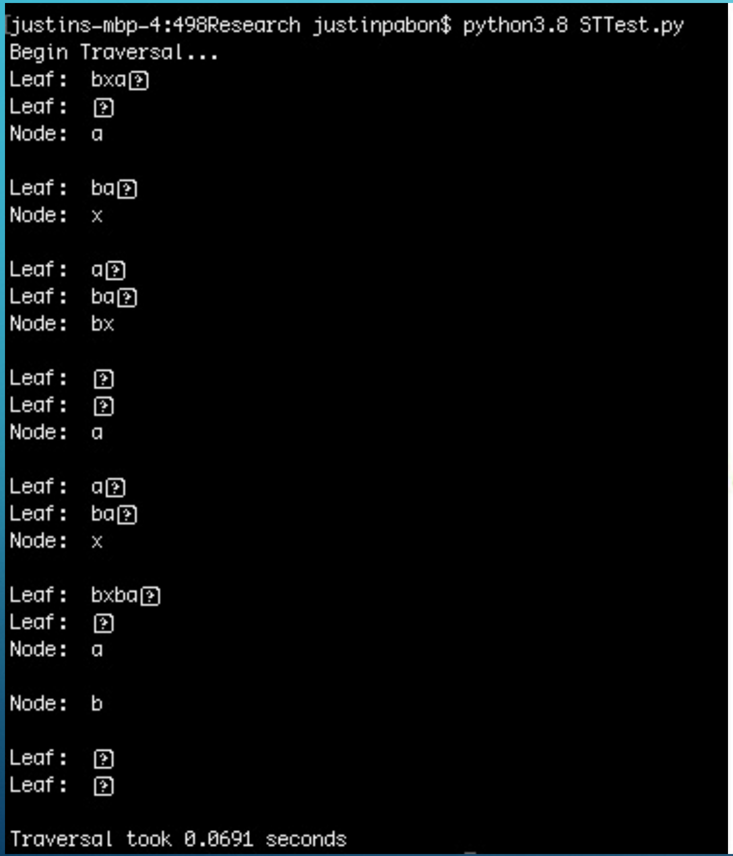
TEST CASE W/ A SMALL DATA SET (1/2)

```
justins-mbp-4:498Research justinpabon$ python3.8 STTest.py
String Array:  ['aat', 'tag', 'gat']

Begin Traversal...
Leaf:  at⍰
Leaf:  ⍰
Leaf:  ⍰
Node:  t

Leaf:  g⍰
Node:  a

Leaf:  ⍰
Leaf:  ag⍰
Leaf:  ⍰
Node:  t

Leaf:  ⍰
Leaf:  ⍰
Leaf:  at⍰
Node:  g

Leaf:  ⍰
Leaf:  ⍰

Traversal took 0.0007 seconds
```

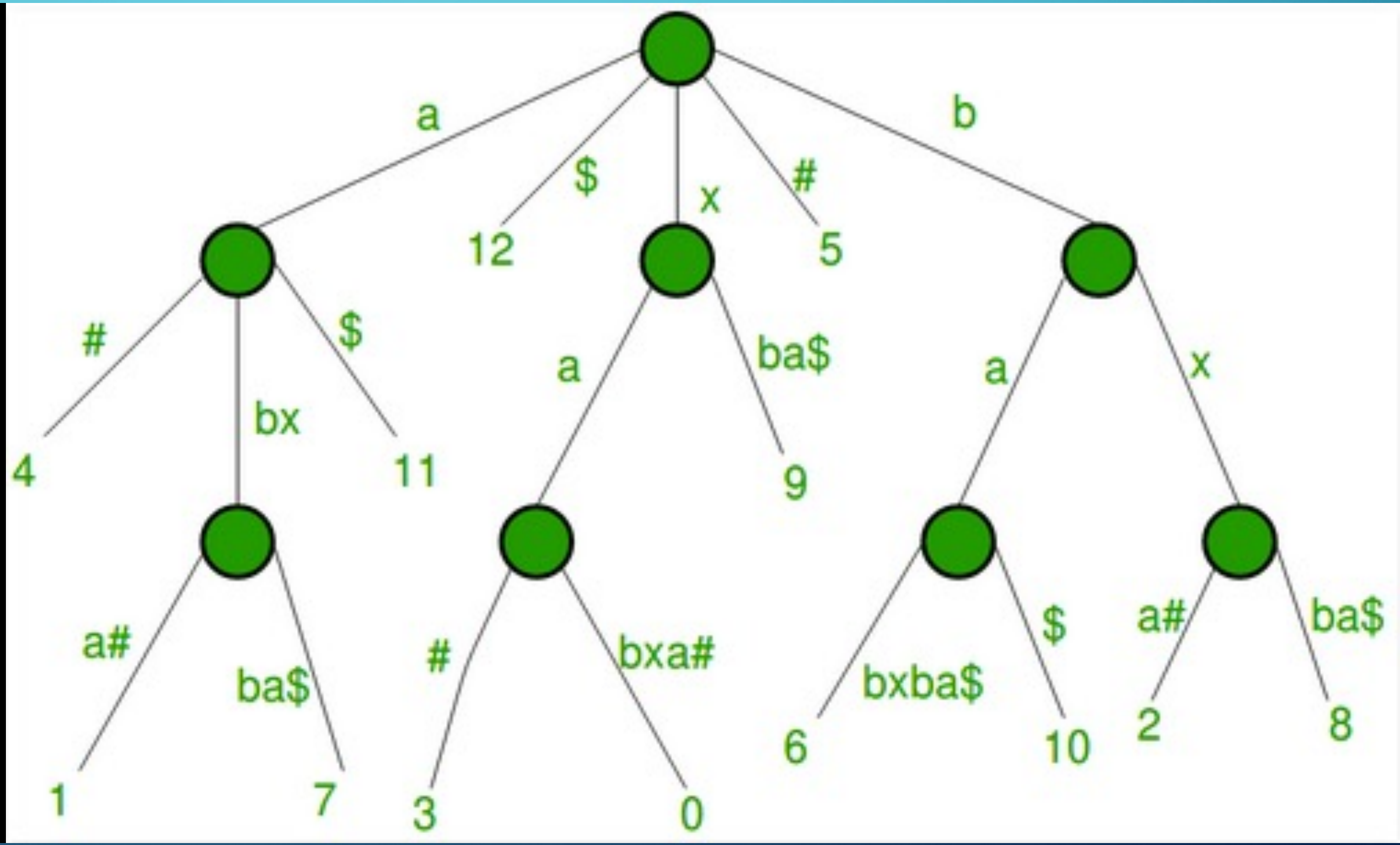


S = att#tag$gat&

TEST CASE W/ A SMALL DATA SET (2/2)



```
justins-mbp-4:498Research justinpabon$ python3.8 STTest.py
Begin Traversal...
Leaf:  bxa⍰
Leaf:  ⍰
Node:  a

Leaf:  ba⍰
Node:  x

Leaf:  a⍰
Leaf:  ba⍰
Node:  bx

Leaf:  ⍰
Leaf:  ⍰
Node:  a

Leaf:  a⍰
Leaf:  ba⍰
Node:  x

Leaf:  bxba⍰
Leaf:  ⍰
Node:  a

Node:  b

Leaf:  ⍰
Leaf:  ⍰

Traversal took 0.0691 seconds
```



S = xabxa#babxba$

RESULTS

1 Million Passwords:

- Traversal Time: 14 minutes 20.987 seconds
- Top 3 Substrings:
  - "123"; 65,936 leaves
  - "234"; 22,479 leaves
  - "198"; 22,245 leaves
- File Size: 14.1MB, 1,254,353 lines

3 Million Passwords:

- Traversal Time: 4 hours 44 minutes 47.5391 seconds
- Top 3 Substrings:
  - "123"; 160,740 leaves
  - "200"; 61,349 leaves
  - "198"; 60,246 leaves
- File Size: 41MB, 3,591,122 lines

5 Million Passwords:

- Traversal Time: 15 hours 43 minutes 44.3862 seconds
- Top 3 Substrings:
  - "123"; 242,532 leaves
  - "198"; 101,108 leaves
  - "200"; 100,899 leaves
- File Size: 68.5MB, 5,987,239 lines

FUTURE WORK

- Further data processing will need to be done on these results.
- How many passwords have a given substring?
  - Two child leaves could be suffixes for the same password or different passwords.
  - More passwords with the same suffix → higher rank in the dictionary.

QUESTIONS?

- **References:**
  - https://www.cs.helsinki.fi/u/tpkarkka/opetus/13s/spa/lecture10-2x4.pdf
  - *McCreight, Edward M. "A space-economical suffix tree construction algorithm." - ACM, 1976. http://libeccio.di.unisa.it/TdP/suffix.pdf*
  - https://pypi.org/project/suffix-trees/
  - https://www.geeksforgeeks.org/generalized-suffix-tree-1/