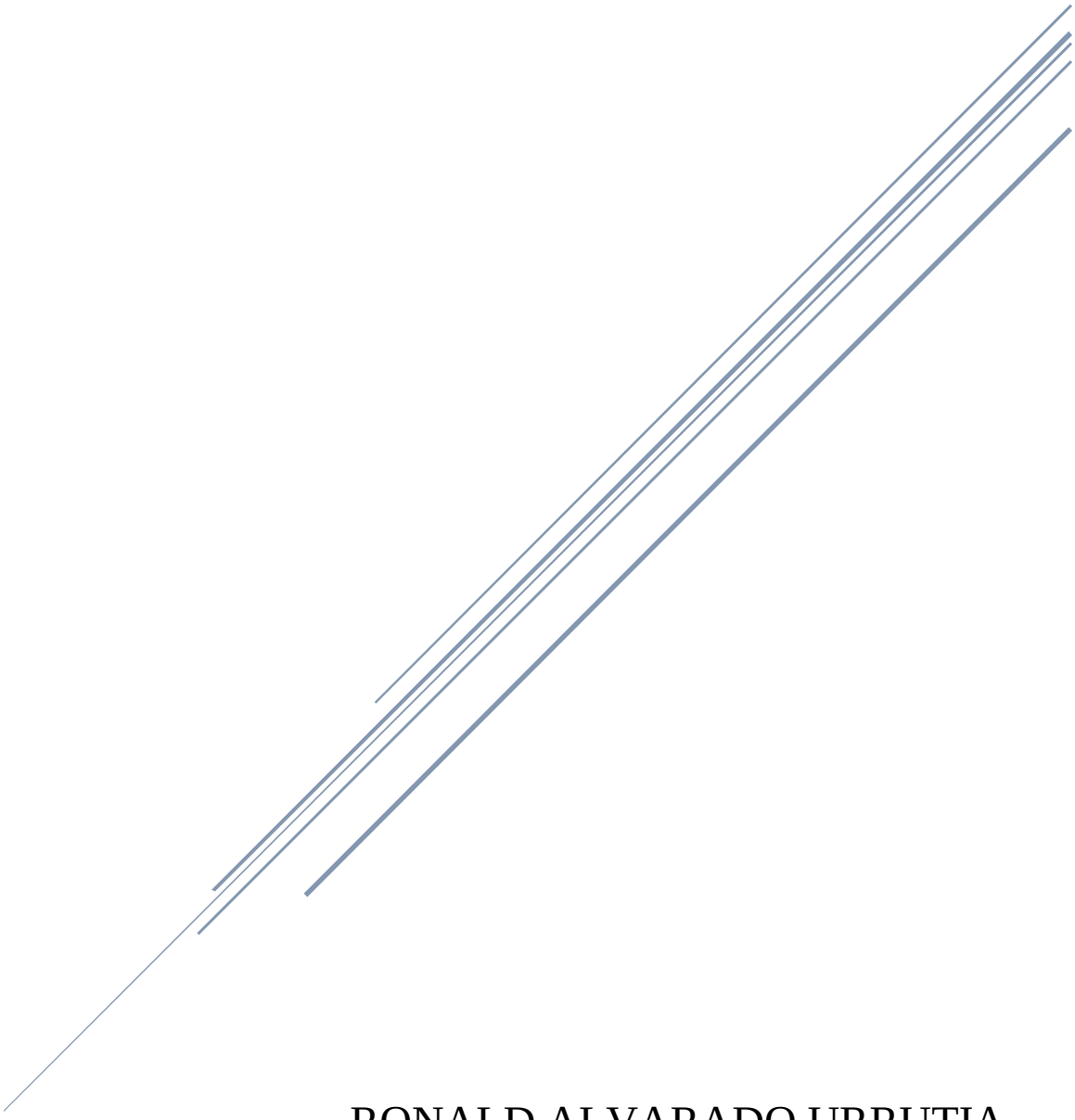
SUFFIX ARRAY – SUFFIX TREE

PROYECTO DE AULA



RONALD ALVARADO URRUTIA
POLITECNICO GRANCOLMBIANO

TABLA DE CONTENIDO

1. INTRODUCCIÓN.

En el presente documento vamos a abordar dos temas importantes que son suffix tree que es una estructura de datos que sirve para almacenar una cadena de caracteres con "información pre-procesada" y suffix array que es una tabla de sufijos se utiliza en la estructura de datos de ordenador y para finalizar terminaremos diseñaremos un esquema de experimentación para detectar cómo se comportan los algoritmos de construcción que se mencionaran mediente este documento.

2. BREVE EXPLICACIÓN DE QUE SON LOS SUFFIX TREES Y QUE SON LOS SUFFIX ARRAYS.

SUFFIX TREE

Suffix Tree, conocido en español como "Árbol de Sufijos", es una estructura de datos que sirve para almacenar una cadena de caracteres con "información pre-procesada" sobre su estructura interna.

Esa información es útil, por ejemplo, para resolver el problema de la subcadena en tiempo lineal:

- Sea un texto S de longitud m
- Se pre-procesa (se construye el árbol) en tiempo $O(m)$
- Para buscar una subcadena P de longitud n basta con $O(n)$. Esta cota no la alcanza ni el KMP ni el BM (requieren $O(m)$)

Sirve además para otros muchos problemas más complejos, como, por ejemplo:

- Dado un conjunto de textos $\{S_i\}$ ver si P es subcadena de algún S_i
- Reconocimiento inexacto de patrones.

SUFFIX ARRAYS

Suffix Arrays, conocido en español como "Arreglo de Sufijos"; Una tabla de sufijos se utiliza en la estructura de datos de ordenador, y en particular en la combinatoria de palabras y la bioinformática. Para una palabra dada, la tabla contiene una lista de números enteros que corresponden a las posiciones de inicio de sufijos de palabras.

La tabla de sufijos se utiliza como un índice para la coincidencia de patrones en el texto. La búsqueda de un patrón en un texto es equivalente al patrón de búsqueda como un prefijo de los sufijos de texto

3. ALGORITMOS DE CONSTRUCCIÓN

SUFFIX TREES:

- UKKONEN'S ALGORITHM

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package algorithm.suffixtrees;
import java.util.Random;
/**
 *
 * @author yogomezo10
 */
public class Ukkonen {
    static final String ALPHABET = "abcdefghijklmnopqrstuvwxyz0123456789\1\2";

  public static class Node {
    int begin;
    int end;
    int depth; // distance in characters from root to this node
    Node parent;
    Node[] children;
    Node suffixLink;

    Node(int begin, int end, int depth, Node parent) {
      this.begin = begin;
      this.end = end;
      this.parent = parent;
      this.depth = depth;
      children = new Node[ALPHABET.length()];
    }
  }
```

```
  public static Node buildSuffixTree(CharSequence s) {
    int n = s.length();
    byte[] a = new byte[n];
    for (int i = 0; i < n; i++) a[i] = (byte) ALPHABET.indexOf(s.charAt(i));
    Node root = new Node(0, 0, 0, null);
    Node node = root;
    for (int i = 0, tail = 0; i < n; i++, tail++) {
      Node last = null;
      while (tail >= 0) {
        Node ch = node.children[a[i - tail]];
        while (ch != null && tail >= ch.end - ch.begin) {
          tail -= ch.end - ch.begin;
          node = ch;
          ch = ch.children[a[i - tail]];
        }
        if (ch == null) {
          node.children[a[i]] = new Node(i, n, node.depth + node.end - node.begin, node);
          if (last != null) last.suffixLink = node;
          last = null;
        } else {
          byte t = a[ch.begin + tail];
          if (t == a[i]) {
            if (last != null) last.suffixLink = node;
            break;
          } else {
            Node splitNode = new Node(ch.begin, ch.begin + tail, node.depth + node.end - node.begin, node);
            splitNode.children[a[i]] = new Node(i, n, ch.depth + tail, splitNode);
            splitNode.children[t] = ch;
            ch.begin += tail;
            ch.depth += tail;
            ch.parent = splitNode;
            node.children[a[i - tail]] = splitNode;
            if (last != null) last.suffixLink = splitNode;
            last = splitNode;
          }
        }
        if (node == root) {
          --tail;
        } else {
          node = node.suffixLink;
        }
      }
    }
    return root;
  }

  // random test
  public static void main(String[] args) {
    Random rnd = new Random(1);
    for (int step = 0; step < 100_000; step++) {
      int n1 = rnd.nextInt(10);
      int n2 = rnd.nextInt(10);
      String s1 = getRandomString(n1, rnd);
      String s2 = getRandomString(n2, rnd);
      // build generalized suffix tree
      String s = s1 + '\1' + s2 + '\2';
      Node tree = buildSuffixTree(s);
      lcsLength = 0;
      lcsBeginIndex = 0;
      // find longest common substring
      lcs(tree, s1.length(), s1.length() + s2.length() + 1);
      int res2 = slowLcs(s1, s2);
      if (lcsLength != res2) {
        System.err.println(s.substring(lcsBeginIndex - 1, lcsBeginIndex + lcsLength - 1));
        System.err.println(s1);
        System.err.println(s2);
        System.err.println(lcsLength + " " + res2);
        throw new RuntimeException();
      }
    }
  }
```

```
  static int slowLcs(String a, String b) {
    int[][] lcs = new int[a.length()][b.length()];
    int res = 0;
    for (int i = 0; i < a.length(); i++) {
      for (int j = 0; j < b.length(); j++) {
        if (a.charAt(i) == b.charAt(j))
          lcs[i][j] = 1 + (i > 0 && j > 0 ? lcs[i - 1][j - 1] : 0);
        res = Math.max(res, lcs[i][j]);
      }
    }
    return res;
  }

  static String getRandomString(int n, Random rnd) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
      sb.append((char) ('a' + rnd.nextInt(3)));
    }
    return sb.toString();
  }
}
```

```
static int lcsLength;
static int lcsBeginIndex;

// traverse suffix tree to find longest common substring
public static int lcs(Node node, int i1, int i2) {
  if (node.begin <= i1 && i1 < node.end) {
    return 1;
  }
  if (node.begin <= i2 && i2 < node.end) {
    return 2;
  }
  int mask = 0;
  for (char f = 0; f < ALPHABET.length(); f++) {
    if (node.children[f] != null) {
      mask |= lcs(node.children[f], i1, i2);
    }
  }
  if (mask == 3) {
    int curLength = node.depth + node.end - node.begin;
    if (lcsLength < curLength) {
      lcsLength = curLength;
      lcsBeginIndex = node.begin;
    }
  }
  return mask;
}
```

- CHAR SEQUENCE SUFFIXTREES ALGORITHM

```java
import static com.globalmentor.collections.SuffixTrees.*;

import com.globalmentor.collections.CharSequenceSuffixTree.*;
import com.globalmentor.model.ObjectHolder;

/**
 * Utilities for working with suffix trees of sequences of characters.
 *
 * @author Garret Wilson
 */
public class CharSequenceSuffixTrees {

	/**
	 * Determines the longest subsequence that is repeated in the given subsequence.
	 * <p>
	 * This implementation walks the tree and finds the non-leaf node that is farthest in terms of characters from the root of the tree. The repeated subsequence
	 * is the sequence of characters from the root to that node.
	 * </p>
	 * @param charSequence The character sequence to check.
	 * @return The longest repeated subsequence in the given character sequence, or <code>null</code> if no subsequence is repeated.
	 * @throws NullPointerException if the given character sequence is <code>null</code>.
	 */
	public static CharSequence getLongestRepeatedSubsequence(final CharSequence charSequence) {
		final CharSequenceSuffixTree suffixTree = CharSequenceSuffixTree.create(charSequence); //create a suffix tree
		final ObjectHolder<String> result = new ObjectHolder<String>(); //create an object to hold the resulting string
		visit(suffixTree, new AbstractCharSequenceVisitor() {

			int maxLength = 0; //keep track of the longest length

			@Override
			public boolean visit(final SuffixTree suffixTree, final CharSequenceNode node, final CharSequenceEdge parentEdge, final CharSequence charSequence) {
				if(!node.isLeaf()) { //ignore leaf nodes---they aren't repeated sequences
					if(charSequence.length() > maxLength) { //if this depth is farther than any before
						maxLength = charSequence.length(); //update our max length
						result.setObject(charSequence.toString()); //make a copy and keep track of the resulting string
					}
				}
				return true;
			}
		});
		return result.getObject(); //return the result, if any
	}
```

```java
        /**
         * An abstract implementation of a visitor for character sequences. This implementation keeps track of the current sequence being visited for each node.
         *
         * @author Garret Wilson
         */
        public static abstract class AbstractCharSequenceVisitor implements Visitor<CharSequenceNode, CharSequenceEdge> {

                /** The string builder to keep track of the current sequence. */
                final StringBuilder sequenceBuilder;

                /** Default constructor starting an empty sequence. */
                public AbstractCharSequenceVisitor() {
                        this("");
                }

                /**
                 * Character sequence constructor. This constructor is useful for creating a visitor that will begin on a non-root node.
                 * @param charSequence The initial character sequence.
                 * @throws NullPointerException if the given character sequence is <code>null</code>.
                 */
                public AbstractCharSequenceVisitor(final CharSequence charSequence) {
                        sequenceBuilder = new StringBuilder(charSequence);
                }

                @Override
                public final boolean visit(final SuffixTree suffixTree, final CharSequenceNode node, final CharSequenceEdge parentEdge, final int length) {
                        if(parentEdge != null) { //if this isn't the root node
                                sequenceBuilder.replace(length - parentEdge.getLength(), sequenceBuilder.length(), parentEdge.getSubSequence().toString()); //append this edge's subseque
                        }
                        return visit(suffixTree, node, parentEdge, sequenceBuilder); //visit the node with the current sequence
                }

        /**
         * Determines the longest subsequence that is repeated in the given subsequence.
         * <p>
         * This implementation walks the tree and, for every non-leaf node (which indicates a repeated sequence as the sequence of characters from the root to that
         * node), determines if there exists the exact sequence <em>starting</em> with the node just found (which indicates that the repeated sequence is followed by
         * an identical sequence). This process continues until the longest of these sequences is determined.
         * </p>
         * @param charSequence The character sequence to check.
         * @return The longest repeated subsequence in the given character sequence, or <code>null</code> if no subsequence is repeated.
         * @throws NullPointerException if the given character sequence is <code>null</code>.
         */
        public static CharSequence getLongestSequentialRepeatedSubsequence(final CharSequence charSequence) {
                final CharSequenceSuffixTree suffixTree = CharSequenceSuffixTree.create(charSequence); //create a suffix tree
                final ObjectHolder<String> result = new ObjectHolder<String>(); //create an object to hold the resulting string
                visit(suffixTree, new AbstractCharSequenceVisitor() {

                        int maxLength = 0; //keep track of the longest length

                        @Override
                        public boolean visit(final SuffixTree suffixTree, final CharSequenceNode node, final CharSequenceEdge parentEdge, final CharSequence charSequence) {
                                if(!node.isLeaf()) { //ignore leaf nodes---they aren't repeated sequences
                                        if(charSequence.length() > maxLength) { //if this depth is farther than any before, see if the repeat sequence is sequential
                                                if(parentEdge.getChildNode().startsWith(charSequence)) { //if the same sequence appears starting with the edge's child node
                                                        maxLength = charSequence.length(); //update our max length
                                                        result.setObject(charSequence.toString()); //make a copy and keep track of the sequentially repeated sequence
                                                }
                                        }
                                }
                                return true;
                        }
                });
                return result.getObject(); //return the result, if any
        }
                /**
                 * Visits the given node. The provided sequence will be modified on further visits; if it is desired that the sequence should be stored, a copy of it should
                 * first be made.
                 * @param suffixTree The suffix tree being visited.
                 * @param node The node being visited.
                 * @param parentEdge The parent edge of the node being visited, or <code>null</code> if the node has no parent.
                 * @param charSequence The current sequence from the root to the node being visited.
                 * @return <code>true</code> if visiting should continue to other nodes.
                 */
                public abstract boolean visit(final SuffixTree suffixTree, final CharSequenceNode node, final CharSequenceEdge parentEdge, final CharSequence charSequence);
        }

}
```

SUFFIX ARRAYS:

- SUFFIX ARRAY ALGORITHM

```cpp
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return  suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt,  n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

- MANBER ALGORITHM

```
/*************************************************************************
 *  Compilation:  javac Manber.java
 *  Execution:    java Manber < text.txt
 *  Dependencies: StdIn.java
 *
 *  Reads a text corpus from stdin and sorts the suffixes
 *  in subquadratic time using a variant of Manber's algorithm.
 *
 *************************************************************************/

public class Manber {
    private int n;                // length of input string
    private String text;          // input text
    private int[] index;          // offset of ith string in order
    private int[] rank;           // rank of ith string
    private int[] newrank;        // rank of ith string (temporary)
    private int offset;

    public Manber(String s) {
        n = s.length();
        text = s;
        index   = new int[n+1];
        rank    = new int[n+1];
        newrank = new int[n+1];

        // sentinels
        index[n] = n;
        rank[n] = -1;

        msd();
        doit();
    }

    /**
     * Returns the length of the input string.
     * @return the length of the input string
     */
    public int length() {
        return n;
    }



   /**
    * Returns the index into the original string of the <em>i</em>th smallest suffix.
    * That is, {@code text.substring(sa.index(i), n)}
    * is the <em>i</em>th smallest suffix.
    * @param i an integer between 0 and <em>n</em>-1
    * @return the index into the original string of the <em>i</em>th smallest suffix
    * @throws java.lang.IndexOutOfBoundsException unless 0 <= <em>i</em> < <em>n</em>
    */
   public int index(int i) {
       if (i < 0 || i >= n) throw new IndexOutOfBoundsException();
       return index[i];
   }

   /**
    * Returns the <em>i</em>th smallest suffix as a string.
    * @param i the index
    * @return the <em>i</em> smallest suffix as a string
    * @throws java.lang.IndexOutOfBoundsException unless 0 <= <em>i</em> < <em>n</em>
    */
   public String select(int i) {
       if (i < 0 || i >= n) throw new IndexOutOfBoundsException();
       return text.substring(index[i]);
   }
```

```
    // do one pass of msd sorting by rank at given offset
    private void doit() {
        for (offset = 1; offset < n; offset += offset) {
            int count = 0;
            for (int i = 1; i <= n; i++) {
                if (rank[index[i]] == rank[index[i-1]]) count++;
                else if (count > 0) {
                    // sort
                    int left = i-1-count;
                    int right = i-1;
                    quicksort(left, right);

                    // now fix up ranks
                    int r = rank[index[left]];
                    for (int j = left + 1; j <= right; j++) {
                        if (less(index[j-1], index[j]))  {
                            r = rank[index[left]] + j - left;
                        }
                        newrank[index[j]] = r;
                    }

                    // copy back - note can't update rank too eagerly
                    for (int j = left + 1; j <= right; j++) {
                        rank[index[j]] = newrank[index[j]];
                    }

                    count = 0;
                }
            }
        }
    }

/**************************************************************************
 *  Quicksort code from Sedgewick 7.1, 7.2.
 **************************************************************************/

    // swap pointer sort indices
    private void exch(int i, int j) {
        int swap = index[i];
        index[i] = index[j];
        index[j] = swap;
    }


    // SUGGEST REPLACING WITH 3-WAY QUICKSORT SINCE ELEMENTS ARE
    // RANKS AND THERE MAY BE DUPLICATES
    private void quicksort(int lo, int hi) {
        if (hi <= lo) return;
        int i = partition(lo, hi);
        quicksort(lo, i-1);
        quicksort(i+1, hi);
    }
```

```
   // sort by leading char, assumes extended ASCII (256 values)
   private void msd() {
       final int R = 256;

       // calculate frequencies
       int[] freq = new int[R];
       for (int i = 0; i < n; i++)
           freq[text.charAt(i)]++;

       // calculate cumulative frequencies
       int[] cumm = new int[R];
       for (int i = 1; i < R; i++)
           cumm[i] = cumm[i-1] + freq[i-1];

       // compute ranks
       for (int i = 0; i < n; i++)
           rank[i] = cumm[text.charAt(i)];

       // sort by first char
       for (int i = 0; i < n; i++)
           index[cumm[text.charAt(i)]++] = i;
   }


'***************************************************************************
*  Helper functions for comparing suffixes.
***************************************************************************/

 /*************************************************************************
  * Is the substring text[v..n] lexicographically less than the
  * substring text[w..n] ?
  *************************************************************************/
   private boolean less(int v, int w) {
       return rank[v + offset] < rank[w + offset];
   }



    private int partition(int lo, int hi) {
        int i = lo-1, j = hi;
        int v = index[hi];

        while (true) {

            // find item on left to swap
            while (less(index[++i], v))
                if (i == hi) break;   // redundant

            // find item on right to swap
            while (less(v, index[--j]))
                if (j == lo) break;

            // check if pointers cross
            if (i >= j) break;

            exch(i, j);
        }

        // swap with partition element
        exch(i, hi);

        return i;
    }
```

```
    /**
     * Unit tests the {@code Manber} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        String s = StdIn.readAll().replaceAll("\\s+", " ").trim();
        Manber suffix = new Manber(s);

        StdOut.println("   i  ind     select");
        StdOut.println("-----------------------");

        for (int i = 0; i < s.length(); i++) {
            int index = suffix.index(i);
            String ith = "\"" + s.substring(index, Math.min(index + 50, s.length())) + "\"";
            StdOut.printf("%4d %4d  %s\n", i, index, ith);
        }
    }

}
```

4. IDEA PRINCIPAL ALGORITMOS DE SUFFIX ARRAY Y SUFFIX TREE.

SUFFIX TREES:

- UKKONEN'S ALGORITHM

  El algoritmo de Ukkonen es un algoritmo on-line, con tiempo de computación lineal, para construir un árbol de sufijos de una cadena S. Este algoritmo fue propuesto por Esko Ukkonen en 1995. Anteriormente existían dos algoritmos capaces de construir el árbol de sufijos de una cadena S en tiempo lineal, estos son el algoritmo de Weiner (1973) y el algoritmo de McCreight (1976). Pero el algoritmo de Ukkonen se destaca por ser más sencillo y por tener la característica de ser **on-line**.

- CHAR SEQUENCE SUFFIXTREES ALGORITHM

  Encuentra la subcadena mas larga repetida de forma secuencial. El mas repetido de la subcadena se pueda encontrar mediante la busquedad del nodo no hoja que se encuentre mas alejado de la raiz (en terminus de caracteres).

SUFFIX ARRAY

- NAIVE ALGORITHM

  Un algoritmo Naive suele ser la solución más obvia cuando uno se pregunta a un problema. Puede que no sea un algoritmo inteligente, pero probablemente conseguirá el trabajo hecho.

  P.ej. Tratando de buscar un elemento en una matriz ordenada. Un algoritmo ingenuo sería utilizar una búsqueda lineal. Una solución no tan ingenua sería utilizar la búsqueda binaria.

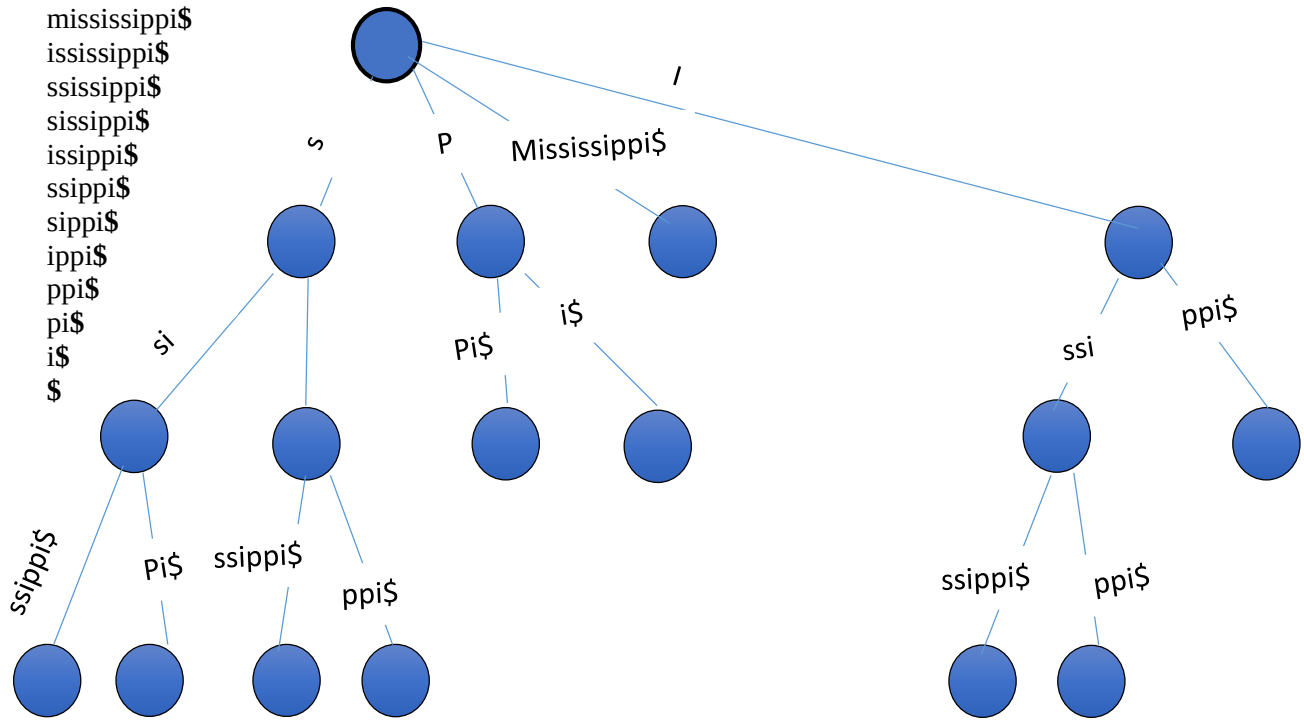
- MANBER ALGORITHM

  Lee un corpus de texto de la entrada estándar y clasifica los sufijos en tiempo subquadratic usando una variante del algoritmo de Manber.

5. EXPLICACIÓN GRÁFICA DE QUE ES Y COMO SE VE ESQUEMÁTICAMENTE UN SUFFIX ARRAY Y SUFFIX TREE.

SUFFIX TREE

Para la explicación grafica cogeremos la cadena **R = mississippi $** y realizaremos el árbol de sufijos R es un árbol compirmido de todos los sufijos de la cadena **R = mississippi $**



SUFFIX ARRAY

Para la explicación grafica cogeremos la cadena **R = mississippi$** y realizaremos el array de sufijos R de todos los sufijos de la cadena **R = mississippi $**

Nuestra cadena debe terminar con el character expecial $, el cual debe ser unico dentro de esta y lo mas importante debe ser lexicograficamente mas pequeño que cualquier otro carater

M	I	S	S	I	S	S	I	P	P	I	$
1	2	3	4	5	6	7	8	9	10	11	12

SUFFIX	**i**

mississippi$	1
ississippi$	2
ssissippi$	3
sissippi$	4
issippi$	5
ssippi$	6
sippi$	7
ippi$	8
ppi$	9
pi$	10
i$	11
$	12

Ahora debemos ordenar lexicograficamente

SUFFIX	**i**
$	12
i$	11
ippi$	8
issippi$	5
ississippi$	2
mississippi$	1
pi$	10
ppi$	9
sippi$	7
sissippi$	4
ssippi$	6
ssissippi$	3

Una vez ordenado el arreglo ya hemos contriuido nuestro arreglo de sufijos el cual queda de la siguiente manera:

I	1	2	3	4	5	6	7	8	9	10	11	12
I[i]	12	11	8	5	2	1	10	9	7	4	6	3

Para nuestro ejemplo A[6] contiene el valor 1 y por lo tanto se refiere al sufijo que empieza en la posición 1 dentro de R, el cual es el sufijo **mississippi$.**

6. PRINCIPALES APLICACIONES DE SUFFIX ARRAY Y SUFFIX TREE.

Aunque hemos dicho que los suffix array y suffix trees se pueden aplicar a varios problemas en comun, es valido aclarar que en algunas situaciones uno funciona mejor que otro, y recordamos que el suffix tree mantiene un mayor espacio.

SUFFIX TREES:

- Mayor subcadena repetida:

Para encontrar la subcadena mas larga de una cadena que se produce al menos dos veces. Este problema puede ser resuelto en el tiempo lineal y el espacio mediante la construccion de un arbol de sufijos para la cadena y encontrar el nodo interno mas profundo del arbol. La cadena de delereado por los bordes de la raiz a un nodo de este tipo es una subcadena mas larga repetida.

- Reconocimiento exacto de un conjunto de patrones:
  Existe un algoritmo de Aho y Corasick (no trivial, sección 3.4 del libro de D. Gusfield) para encontrar todas (las k) apariciones de un conjunto de patrones de longitud total m en un texto de longitud n con coste O(n+m+k)

  Con un árbol de sufijos se obtiene exactamente la misma cota.

- Un árbol de sufijos generalizado:
  Sirve para guardar los sufijos de un conjunto de textos.

- Comprobar subcadena:
  Busquedad de una subcadena, pat [1..m], en txt[1..n], puede ser resuelto en tiempo O(m) despues de que el arbol de sufijos para txt se ha construido en un tiempo O(n).

- Mayor subcadena palindromico:
  Detectar todos los palindromes (text que se puede leer igualmente de izquierda a derecho o derecho a izquierda) en una cadena dada.

SUFFIX ARRAY:

Sabemos las ventajas que ofrece suffix array frente a suffix tree, a continuacion se relacionan algunas de las mayors aplicaciones que tiene suffix array

- Encontrar los palindromos de una cadena
  Encontrar todos los palindromes (palabras que se pueden leer igualmente de izquierda a derecho o de derecho a izquierda) de una cadena.

- Encontrar la subcadena mas larga repetida
  Consiste en recorrer un array para encontrar la cadena mas larga

- Encontrar la subcadena comun mas larga
  Dadas 2 subcadenas, encontrar la subcadena que es comun para t1 y t2 y entre estas encontrar la mas larga.

7. BIBLIOGRAFÍA.

Hackerearth Página Web
Article
En el texto: (Suffix Tree, Suffix Array,2014)
Bibliografia: anonimo
Available At: **https://www.hackerearth.com/notes/trie-suffix-tree-suffix-array/**

cs.helsinki.fi PDF
Helsinki
En el texto: (Suffix Array, 2015)
Bibliografia: anonimo
Available At: **https://www.cs.helsinki.fi/u/tpkarkka/publications/icalp03.pdf**

Stackoverflow Página Web
Documentation
En el texto: (Suffix Tree vs Suffix Array,2014)
Bibliografia: anonimo
Available At: **http://stackoverflow.com/questions/2487576/trie-vs-suffix-tree-vs-suffix-array**

cs.umd Pdf
Suffix Arrays
En el texto: (cs.umd, 2015)
Bibliografia: anonimo
Available At: **https://www.cs.umd.edu/class/fall2011/cmsc858s/SuffixArrays.pdf**

cs.umd Pdf
Suffix Trees
En el texto: (cs.umd, 2015)
Bibliografia: anonimo
Available At: **https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf**

8. ESQUEMA DE EXPERIMIENTACION CORRECTNESS

Para este esquema inicialmente vamos a asumir que en cada caso el algoritmo de Knuth-Morris-Pratt(KMP) es correcto, se realizara la creación de un método de comparación para cada uno de los manejadores de sufijos (Suffix array, Suffix Tree), verificando que el resultado de la construcción del arreglo de sufijos o el de árbol de sufijos del algoritmo más elaborado de cada uno de estos nos arroje el mismo resultado que el algoritmo de Knuth-Morris-Pratt(KMP) para ello voy a realizar 200 pruebas con cadenas con tamaños distintos que variaran entre 800, 1600 y 2400 por cada tamaño a probar se van a realizar 5 pruebas con el mismo tamaño contando los errores que se van presentando en cada caso, al terminar de ejecutar nuestro esquema deberá mostrar los errores que fueron encontrados en cada uno de los algoritmos.

9. ESQUEMA DE EXPERIMENTACIÓN COMPLEXITY

Para este esquema se piensa realizar un par de casos de prueba para ser exactos 2, lo que voy a hacer es hacer 20 pruebas mediante el cual se ira variando los textos enviando de 250 en 250 y por cada tamaño se realizara una iteración de 10, en cada una de estas iteraciones se ira tomando el tiempo y almacenando en una lista global (Tamaño vs Tiempo), al finalizar este esquema se visualizar mejor estos resultados en una gráfica ya sea en eEcel o cualquier otro programa.

- **Clase Utilitaria:**
  https://gist.github.com/roalur123/e78413d6b45bd2123335729c3da9202e