1 Cabecera y Programa Principal

```
/*
  Codigo para primer acercamiento a Suffix Arrays.
  Solo es necesario compilar (con c++11 almenos) y correr en consola.

  Suffix Array O(n*lg(n)^2) - Manber, Udi; Myers, Gene (1990).
  Longest Common Prefix Array O(n) - Kasai, T.; Lee, G.; Arimura, H.; Arikawa,
      ↪ S.; Park, K. (2001).
 */
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm> // Para algoritmo de ordenamiento entre cadenas
#include <cstdlib> // Para generador
#include <ctime> // Para generador

// Constantes para la clase...
const bool print_details = true;
const bool check_stress  = false;
const bool output_test   = false;

// Prototipos importantes
std::vector<size_t> naiveSA(const std::string& s);
std::vector<size_t> naiveLCP(const std::string& s, const std::vector<size_t>&
    ↪ suffix_array);
std::vector<size_t> suffixArray(const std::string& s);
std::vector<size_t> lcp(const std::string& s, const std::vector<size_t>&
    ↪ suffi_array);

// Prototipos de funciones auxiliares
template <typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& v);
template <typename T>
bool operator==(const std::vector<T>& a, const std::vector<T>& b);
std::string generateString(unsigned int length);

int main(){
  using namespace std;

  vector< string > tests{
    "bobocel",
      "banana",
      "",
      "a",
      "aa",
      "another␣test␣string..."
      };

  if( print_details ){

    for(string test: tests){
      auto suffix_array = suffixArray(test);
      auto lcp_array    = lcp(test, suffix_array);

      cout << "String:␣"       << test << endl;
```

```
      cout << "Suffix Array: " << suffix_array << endl;
      // Detalles sobre el Suffix Array.
      for(size_t index: suffix_array){
        cout << setw(4) << index << ": " << test.substr(index) << endl;
      }

      cout << "LCP Array:     " << lcp_array << endl;
      // Detalles sobre el LCP Array.
      cout << setw(4) << 0 << ": -" << endl;
      for(size_t index = 1; index < lcp_array.size(); index++){
        size_t length = lcp_array[index];
        size_t str_index = suffix_array[index];
        cout << setw(4) << length << ": " << test.substr(str_index, length) <<
            ↪ endl;
      }

      cout << endl;
    }
  }// Fin de detalles

  if( check_stress ){
    string str = generateString(1e5);
    auto suffix_array  = suffixArray(str);
    auto lcp_array     = lcp(str, suffix_array);
    // Para una cadena de 100 000 caracteres fueron
    // ~0.7s para Suffix Array y
    // ~0.9s para LCP.

  }// Fin de stress

  if( output_test ){
    string test_string = generateString(1e5);
    cout << test_string;
  }

  return 0;
}
```

2 Construcción del Suffix Array en $O(n^2 \log n)$

```cpp
// Algoritmo O(n^2*lg(n)) para la construccion de un Suffix Array.
// Simplemente se generera un vector de pares de subcadenas (de la cadena principal) e indices,
// se ordena y retorna solo los indices.
// Se aprobecha std::sort de <algorithm> para ordenar los pares,
// cuyo ordenamiento es inicialmente sobre el primer elemento (la subcadena)
// y luego sobre el segundo elemento (el indice).
std::vector<size_t> naiveSA(const std::string& s){
  // Solo utilizamos std dentro de la funcion
  using namespace std;

  // Definicion del vector de pares
  vector< pair<string, size_t> > strings = vector< pair<string, size_t> >();

  // Iterando sobre la cadena principal utilizando iteradores
  auto iter = s.begin();
  while( iter != s.end() ){
    size_t index = iter - s.begin(); // Toma los valores 0, 1, 2, 3, ...
    string substring = s.substr(index); // Se obtiene la subcadena s[index, ...].
    auto element = make_pair(substring, index); // Se construye el par
    strings.push_back( element ); // Se almacena en el vector de pares
    iter++;
  }

  // Se ordena
  sort(strings.begin(), strings.end());

  // Y obtenemos los indices
  vector<size_t> indexes = vector<size_t>();
  for(auto element: strings){
    indexes.push_back(element.second);
  }

  return indexes;
}
```

3 Construcción del Suffix Array en $O(n \log^2 n)$

```cpp
// Algoritmo O(n*lg(n)^2) para la construccion del Suffix Array
std::vector<size_t> suffixArray(const std::string& s){
  using namespace std;

  // Se define una 3-tupla compuesta de
  // sa: correspondiente a los valores del SA.
  // msd: el digito mas significativo (msd) para el ordenamiento.
  // lsd: el digito menos significativo (lsd) para el ordenamiento
  // Se define los metodos de orden e igualdad.
  struct Triple{
    size_t sa;
    int msd;
    int lsd;
    // Constructor
    Triple(size_t sa, int msd, int lsd): sa(sa), msd(msd), lsd(lsd){}
    // Funcion a utilizarse por std::sort de <algorithm>
    bool operator<(const Triple& t) const{
      if( msd == t.msd ){
        if( lsd == t.lsd )
          return false;
        else
          return (lsd < t.lsd);
      }else
        return (msd < t.msd);
    }
    // Funcion de igualdad
    bool operator==(const Triple& t) const{
      return (msd == t.msd && lsd == t.lsd);
    }
  };

  // Se inicializa los valores
  auto v = vector<Triple>();
  for(size_t index = 0; index < s.size(); index++){
    size_t sa = index;
    // Se define la relacion entre el primer caracter y el msd
    int msd = (int)(s[index]);
    // El segundo caracter con lsd y si no existe, se indica el menor valor
        ↪ posible
    int lsd = (index < s.size() - 1)?(int)(s[index+1]):-1;
    v.push_back( Triple(sa, msd, lsd) );
  }

  // Se repite lg(n) veces
  for(int gap = 2; gap < 2*s.size(); gap *= 2){

    // Se ordena segun msd y lsd
    sort(v.begin(), v.end());

    // Se redefinen los msd manteniendo el orden
    int msd = 0;
    for(size_t index = 0; index < v.size() - 1; index++){
      if( !(v[index] == v[index+1]) ){
        v[index].msd = msd;
```

```
        msd++;
      }else{
        v[index].msd = msd;
      }
    }
    v[v.size()-1].msd = msd;

    // Se inicializa el vector auxiliar
    // Tal que es el inverso del SA
    // aux[ sa[i] ] = i
    auto aux = vector<size_t>(v.size());
    for(size_t index = 0; index < aux.size(); index++)
      aux[ v[index].sa ] = index;

    // Se calculan los valores de los lsd segun el invariante
    // (La parte complicada!!)
    for(size_t index = 0; index < v.size(); index++){
      size_t value_index = v[index].sa + gap;
      if( value_index < aux.size() )
        v[index].lsd = v[ aux[value_index] ].msd;
      else
        v[index].lsd = -1;
    }


  }

  // Se prepara el SA y se retorna
  auto suffix_array = vector<size_t>();
  for(Triple t: v)
    suffix_array.push_back( t.sa );

  return suffix_array;
}
```

4 Construcción del LCP Array en $O(n^2)$

```cpp
// Algoritmo O(n^2) para la construccion del Longest Common Prefix Array.
// Utilizando el suffix array se obtienen las subcadenas
// y se itera sobre ellas para obtener la longitud del prefijo comun.
// lcp[i] es la longitud del prefijo entre las subcadenas definidas en sa[i-1]
    ↪ y sa[i].
// Por ejemplo: S = "banana", sa[] = {5, 3, 1, 0, 4, 2} entonces
// lcp[1] = 1 porque como sa[0] = 5 -> "a" y sa[1] = 3 -> "ana", entonces el
    ↪ prefijo es "a".
// lcp[2] = 3 porque como sa[1] = 3 -> "ana" y sa[2] = 1 -> "anana", entonces
    ↪ el prefijo es "ana".
std::vector<size_t> naiveLCP(const std::string& s, const std::vector<size_t>&
    ↪ suffix_array){
  using namespace std;

  // Se inicializa el vector con el tamao de suffix array y sus elementos en 0.
  // lcp[0] no tiene significado, no deberia de accederse durante su uso.
  vector<size_t> lcp = vector<size_t>(suffix_array.size(), 0);

  // Se itera sobre el suffix array
  for(size_t index = 1; index < suffix_array.size(); index++){
    // Se obtienen las cadenas a comparar
    string a = s.substr(suffix_array[index-1]);
    string b = s.substr(suffix_array[index]);
    // Se calcula la longitud del prefijo entre las cadenas
    size_t prefix_length = 0;
    while( a[prefix_length] == b[prefix_length] ) prefix_length++;
    // y se almacena
    lcp[index] = prefix_length;
  }

  return lcp;
}
```

5 Construcción del LCP Array en $O(n)$

```cpp
// Algoritmo O(n) para la contruccion del LCP Array
std::vector<size_t> lcp(const std::string& s, const std::vector<size_t>&
    ↪ suffix_array){
  using namespace std;

  // Se reserva espacio suficiente para el LCP Array
  auto lcp_array = vector<size_t>(s.size());

  // Se inicializa el vector auxiliar
  // Es el inverso de Suffix Array
  auto aux = vector<size_t>(suffix_array.size());
  for(size_t index = 0; index < suffix_array.size(); index++)
    aux[ suffix_array[index] ] = index;

  // Se inicializa la longitud del prefijo a cero.
  size_t length = 0;
  for(size_t index = 0; index < aux.size(); index++){
    // Se recorren las subcadenas el orden en que se encuentran en la cadena
        ↪ principal
    size_t lcp_index = aux[index];
    // Se reduce en uno la longitud del prefijo,
    // asi no se realizan comparaciones innecesarias
    // Pero siempre manteniendolo no negativo
    // (Parte complicada de explicar POR QUE!!!)
    if( lcp_index != 0 ){ // lcp_index == 0 implica que no existe un antecesor.
      if( length > 0 )
        length--;

      // Se comparan los caracteres de
      // la subcadena A,
      // y la subcadena B, la antecesor de A.
      size_t index_substr_a = index;
      // Con aux[index] obtenemos el orden de la subcadena A
      // Restamos 1 para obtener el orden de la subcadena B (su antecesor)
      // Con suffix_array[aux[index]-1] obtenemos el indice del inicio de B
      // (Parte complicada de entender el codigo)
      size_t index_substr_b = suffix_array[aux[index] - 1];
      while( index_substr_a < s.size() &&
             index_substr_b < s.size() &&
             s[index_substr_a + length] == s[index_substr_b + length]
             ){
        // Incrementamos en uno, mientras se incremente el prefijo
        length++;
      }

      // Al finalizar, guardamos el resultado
      lcp_array[lcp_index] = length;
    }
  }

  return lcp_array;
}
```

6 Funciones Auxiliares

```
// Generador de cadenas de pruebas
std::string generateString(unsigned int length){
  std::string s = std::string();

  std::srand( std::time(NULL) );
  while( length-- > 0 ){
    char c = 'a' + std::rand() % (int)('z' - 'a');
    s.push_back(c);
  }

  return s;
}

// Template para imprimir vectores con std::cout
template <typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& v){
  os << '[';

  if( !v.empty() ){
    typename std::vector<T>::const_iterator iter = v.begin();
    os << *iter++;
    while( iter != v.end() )
      os << ", " << *iter++;
  }

  os << ']';
  return os;
}

// Template para comparar vectores
template <typename T>
bool operator==(const std::vector<T>& a, const std::vector<T>& b){
  if( a.size() == b.size() ){
    size_t index = 0;
    while( index < a.size() && a[index] == b[index] ) index++;
    return index == a.size();
  }else
    return false;
}
```