Suffix arrays Efficient algorithms for string processing play a critical role in commercial applications and in scientific computing. From the countless strings that define web pages that are searched by billions of users to the extensive genomic databases that scientists are studying to unlock the secret of life, computing applications of the 21st century are increasingly string-based. As usual, some classic algorithms are effective, but remarkable new algorithms are being developed. Next, we describe a data structure and an API that support some of these algorithms. We begin by describing a typical (and a classic) string-processing problem.

Longest repeated substring. What is the longest substring that appears at least twice in a given string? For example, the longest repeated substring in the string `"to be or not to be"` is the string `"to be"`. Think briefly about how you might solve it. Could you find the longest repeated substring in a string that has millions of characters? This problem is simple to state and has many important applications, including data compression, cryptography, and computer-assisted music analysis. For example, a standard technique used in the development of large software systems is *refactoring code*. Programmers often put together new programs by cutting and pasting code from old programs. In a large program built over a long period of time, replacing duplicate code by function calls to a single copy of the code can make the program much easier to understand and maintain. This improvement can be accomplished by finding long repeated substrings in the program. Another application is found in computational biology. Are substantial identical fragments to be found within a given genome? Again, the basic computational problem underlying this question is to find the longest repeated substring in a string. Scientists are typically interested in more detailed questions (indeed, the nature of the repeated substrings is precisely what scientists seek to understand), but such questions are certainly no easier to answer than the basic question of finding the longest repeated substring.

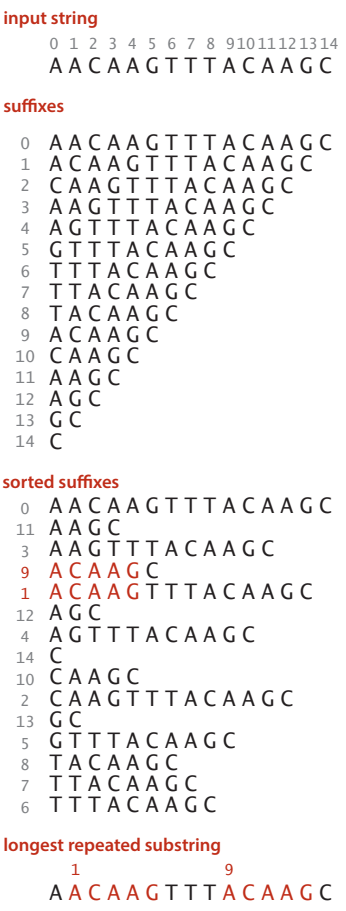
Brute-force solution. As a warmup, consider the following simple task: given two strings, find their longest common *prefix* (the longest substring that is a prefix of both strings). For example, the longest common prefix of `acctgttaac` and `accgttaa` is `acc`. The code at right is a useful starting point for addressing more complicated tasks: it takes time proportional to the length of the match. Now, how do we find the longest repeated substring in a given string? With `lcp()`, the following

```
private static int lcp(String s, String t)
{
   int n = Math.min(s.length(), t.length());
   for (int i = 0; i < n; i++)
      if (s.charAt(i) != t.charAt(i)) return i;
   return n;
}
```

Longest common prefix of two strings

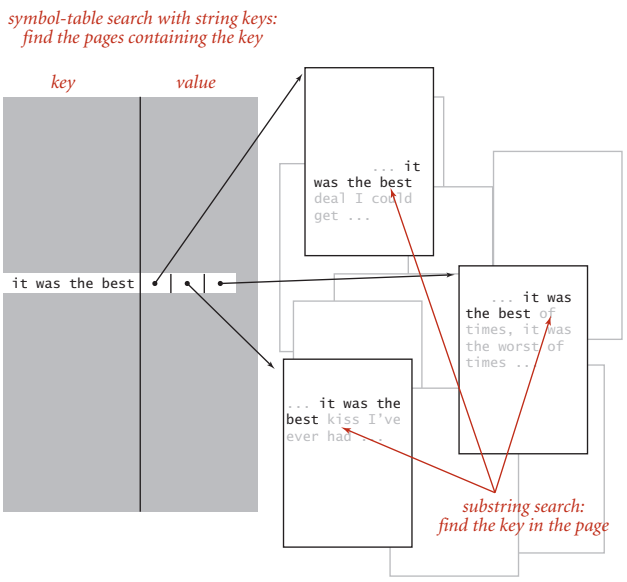
brute-force solution immediately suggests itself: we compare the substring starting at each string position `i` with the substring starting at each other starting position `j`, keeping track of the longest match found. This code is not useful for long strings, because its running time is at least *quadratic* in the length of the string: the number of different pairs `i` and `j` is $n\,(n-1)\,/\,2$, so the number of calls on `lcp()` for this approach would be $\sim n^2/2$. Using this solution for a genomic sequence with millions of characters would require trillions of `lcp()` calls, which is infeasible.

Suffix sort solution. The following clever approach, which takes advantage of sorting in an unexpected way, is an effective way to find the longest repeated substring, even in a huge string: we make an array of the n suffixes of `s` (the substrings starting at each position and going to the end), and then we sort this array. The key to the algorithm's correctness is that every substring appears somewhere as a prefix of one of the suffixes in the array. After sorting, the longest repeated substrings will appear in adjacent positions in the array. Thus, we can make a single pass through the sorted array, keeping track of the longest matching prefixes between adjacent strings. The key to the algorithm's efficiency is to form the n suffixes implicitly (storing only the original string and the index of the first character in each suffix) instead of explicitly (since that would require quadratic time and space). This suffix sorting approach is significantly more efficient than the brute-force method, but before implementing and analyzing it, we consider another application of suffix sorting.



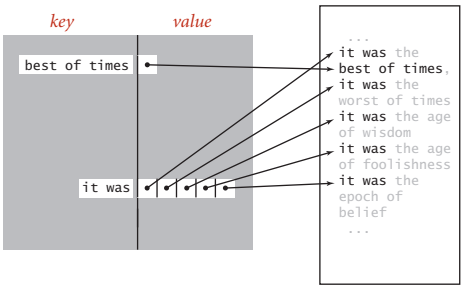
Computing the LRS by sorting suffixes

Indexing a string. When you are trying to find a particular substring within a large text—for example, while working in a text editor or within a page you are viewing with a browser—you are doing a *substring search*, the problem we considered in SECTION 5.3. For that problem, we assume the text to be relatively large and focus on preprocessing the *substring*, with the goal of being able to efficiently find that substring in any given text. When you type search keys into your web browser, you are doing a *search with string keys*, the subject of SECTION 5.2. Your search engine must precompute an index, since it cannot afford to scan all the pages in the web for your keys. As we discussed in SECTION 3.5 (see `FileIndex` on page 501), this would ideally be an inverted index associating each possible search string with all web pages that contain it—a symbol table where each entry is a string key and each value is a set of pointers (each pointer giving the information necessary to locate an occurrence of the key on the web—perhaps a URL that names a web page and an integer offset within that page). In practice, such a symbol table would be far too big, so your search engine uses various sophisticated algorithms to reduce its size. One approach is to rank web pages by importance (perhaps using an algorithm like the PageRank algorithm that we discussed on page 502) and work only with highly-ranked pages, not all pages. Another approach to cutting down on the size of a symbol table to support search with string keys is to associate URLs with *words* (substrings delimited by whitespace) as keys in the precomputed index. Then, when you search for a word, the search engine can use the index to find the (important) pages containing your search keys (words) and then use substring search within each page to find them. But with this approach, if the text were to contain `"everything"` and you were to search for `"thing"`, you would not find it. For some applications, it *is* worthwhile to build an index to help find *any substring* within a given text. Doing so might be justified for a linguistic study of an important piece of literature, for a genomic sequence that might be an object of study for many scientists, or just for a widely accessed web page. Again, ideally,

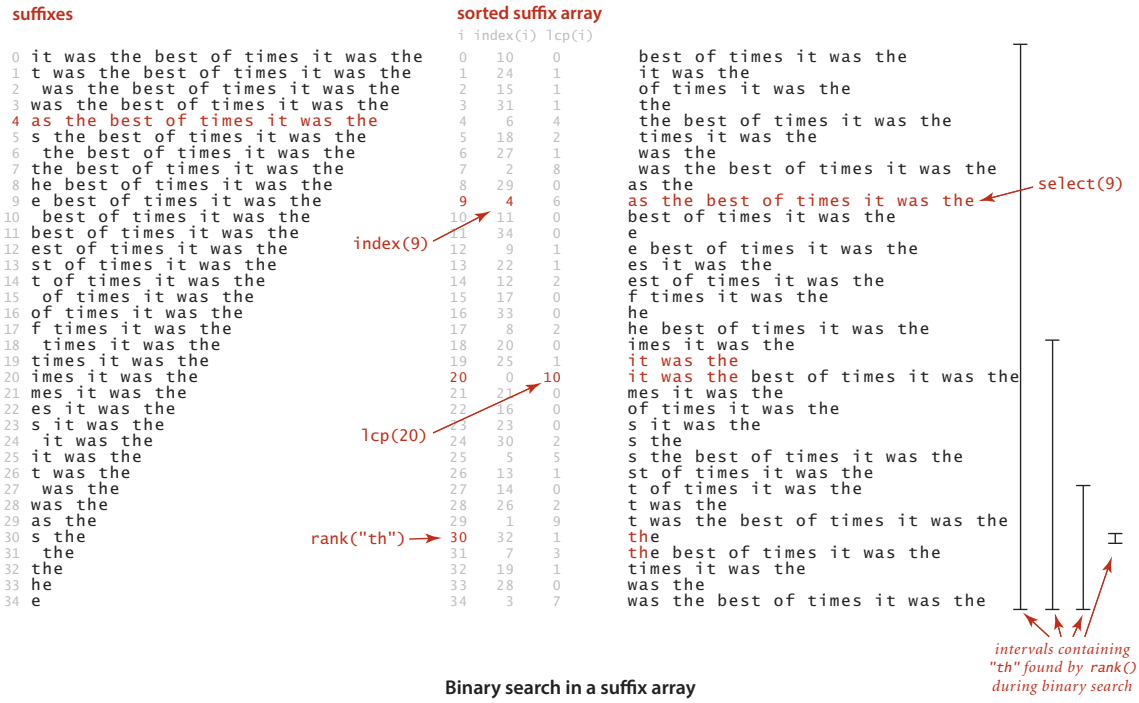


Idealized view of a typical web search

the index would associate all possible substrings of the text string with each position where it occurs in the text string, as depicted at right. The basic problem with this ideal is that the number of possible substrings is too large to have a symbol-table entry for each of them (an n-character text has $n(n+1)/2$ substrings). The table for the example at right would need entries for `b`, `be`, `bes`, `best`, `best o`, `best of`, `e`, `es`, `est`, `est o`, `est of`, `s`, `st`, `st o`, `st of`, `t`, `t o`, `t of`, `o`, `of`, and many, many other substrings. Again, we can use a suffix sort to address this problem in a manner analogous to our first symbol-table implementation using binary search, in SECTION 3.1. We consider each of the n suffixes to be keys, create a sorted array of our keys (the suffixes), and use binary search to search in that array, comparing the search key with each suffix.



Idealized view of a text-string index



Binary search in a suffix array

API and client code. To support client code to solve these two problems, we articulate the API shown below. It includes a constructor; a `length()` method; methods `select()` and `index()`, which give the string and index of the suffix of a given rank in the sorted list of suffixes; a method `lcp()` that gives the length of the longest common prefix of each suffix and the one preceding it in the sorted list; and a method `rank()` that gives the number of suffixes less than the given key (just as we have been using since we first examined binary search in CHAPTER 1). We use the term *suffix array* to describe the abstraction of a sorted list of suffix strings, without committing to use an array of strings as the underlying data structure.

`public class SuffixArray`	
`SuffixArray(String text)`	*build suffix array for `text`*
`int length()`	*length of `text`*
`String select(int i)`	*`i`th in the suffix array (`i` between `0` and `n-1`)*
`int index(int i)`	*index of `select(i)` (`i` between `0` and `n-1`)*
`int lcp(int i)`	*length of longest common prefix of `select(i)` and `select(i-1)` (`i` between `1` and `n-1`)*
`int rank(String key)`	*number of suffixes less than `key`*

Suffix array API

In the example on the facing page, `select(9)` is `"as the best of times..."`, `index(9)` is 4, `lcp(20)` is 10 because `"it was the best of times..."` and `"it was the"` have the common prefix `"it was the"` which is of length 10, and `rank("th")` is 30. Note also that the `select(rank(key))` is the first possible suffix in the sorted suffix list that has `key` as prefix and that all other occurrences of `key` in the text immediately follow (see the figure on the opposite page). With this API, the client code on the next two pages is immediate. `LongestRepeatedSubstring` (page 880) finds the longest repeated substring in the text on standard input by building a suffix array and then scanning through the sorted suffixes to find the maximum `lcp()` value. `KWIC` (page 881) builds a suffix array for the text named as command-line argument, takes queries from standard input, and prints all occurrences of each query in the text (including a specified number of characters before and after to give context). The name KWIC stands for *keyword-in-context* search, a term dating at least to the 1960s. The simplicity and efficiency of this client code for these typical string-processing applications is remarkable, and testimony to the importance of careful API design (and the power of a simple but ingenious idea).

```
public class LongestRepeatedSubstring
{

   public static String lrs(String text)
   {
      int n = text.length();
      SuffixArray sa = new SuffixArray(text);
      String lrs = "";
      for (int i = 1; i < n; i++)
      {
         int length = sa.lcp(i);
         if (length > lrs.length())
            lrs = text.substring(sa.index(i), sa.index(i) + length);
      }
      return lrs;
   }

   public static void main(String[] args)
   {
      String text = StdIn.readAll().replaceAll("\\s+", " ");
      StdOut.println("'" + lrs(text) + "'");
   }
}
```

Longest repeated substring client

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair

% java LongestRepeatedSubstring < tinyTale.txt
'st of times it was the '

% java LongestRepeatedSubstring < mobyDick.txt
',- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th'
```

```
public class KWIC
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int context = Integer.parseInt(args[1]);

      String text = in.readAll().replaceAll("\\s+", " ");
      int n = text.length();
      SuffixArray sa = new SuffixArray(text);

      while (StdIn.hasNextLine())
      {
         String query = StdIn.readLine();
         for (int i = sa.rank(query); i < n; i++)
         {
            // Check if sorted suffix i is a match.
            int from1 = sa.index(i);
            int to1   = Math.min(n, sa.index(i) + query.length());
            if (!query.equals(text.substring(from1, to1))) break;

            // Print context surrounding sorted suffix i.
            int from2 = Math.max(0, sa.index(i) - context);
            int to2   = Math.min(n, sa.index(i) + context + query.length());
            StdOut.println(text.substring(from2, to2));
         }
         StdOut.println();
      }
   }
}
```

Keyword-in-context indexing client

```
% java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
 dispersing in search of other carri
n that bed and search the straw hold


better thing
t is a far far better thing that i do than
 some sense of better things else forgotte
was capable of better things mr carton ent
```

Implementation. The code on the facing page is an elementary implementation of the `SuffixArray` API. The key to the implementation is a nested class `Suffix` that represents a suffix of a text string. A `Suffix` has two instance variables: a `String` reference to the text string and an `int` index of its first character. It provides four utility methods: `length()` returns the length of the suffix; `charAt(i)` returns the `i`th character in the suffix; `toString()` returns a string representation of the suffix; and `compareTo()` compares two suffixes, for use in sorting. Using this nested class, it is straightforward to complete the implementation. The constructor builds an array of `Suffix` objects and sorts them, so `index(i)` just returns the index associated with `suffixes[i]`. The implementations of `length()` and `select()` are also one-liners. The implementation of `lcp()` is similar to the `lcp()` on page 875, and `rank()` is virtually the same as our implementation of binary search for symbol tables, on page 381. Again, the simplicity and elegance of this implementation should not mask the fact that it is a sophisticated algorithm that enables the solution of important problems like the longest repeated substring problem that would otherwise seem to be infeasible.

Performance. The efficiency of our suffix sorting implementation depends on the fact that we form the suffixes implicitly—each suffix is represented by a reference to the text string and the index of its first character. Thus, the space to store the array of suffixes is linear in the length of the text string. This point is a bit counterintuitive because the total number of characters in the n suffixes is $\sim n^2/2$, a quadratic function of the length of the string. Moreover, that quadratic factor gives one pause when considering the cost of sorting the array of suffixes. It is very important to bear in mind that this approach is effective for long strings because of our implicit representation for suffixes: when we exchange two suffixes, we are exchanging only references, not the whole suffixes. Now, the cost of *comparing* two suffixes may be proportional to the length of the suffixes in the case when their common prefix is very long, but most comparisons in typical applications involve only a few characters. If so, the running time of the suffix sort is linearithmic.

```
public int compareTo(Suffix that)
{
   if (this == that) return 0;
   int n = Math.min(this.length(), that.length());
   for (int i = 0; i < n; i++)
   {
      if (this.charAt(i) < that.charAt(i)) return -1;
      if (this.charAt(i) > that.charAt(i)) return +1;
   }
   return this.length() - that.length();

}
```

Comparing two suffixes

ALGORITHM 6.2 Suffix array (elementary implementation)

```
import java.util.Arrays;

public class SuffixArray
{
   private Suffix[] suffixes;   // array of suffixes

   public SuffixArray(String text)
   {
      int n = text.length();
      this.suffixes = new Suffix[n];
      for (int i = 0; i < n; i++)
         suffixes[i] = new Suffix(text, i);
      Arrays.sort(suffixes);
   }

   private static class Suffix implements Comparable<Suffix>
   {
      private final String text; // reference to text string
      private final int index;   // index of suffix's first character

      private Suffix(String text, int index)
      {
         this.text = text;
         this.index = index;
      }

      private int length()       {  return text.length() - index;   }
      private char charAt(int i) {  return text.charAt(index + i);  }
      public String toString()   {  return text.substring(index);   }
      public int compareTo(Suffix that)  // See page 882.
   }

   public int index(int i)     {  return suffixes[i].index;       }
   public int length()         {  return suffixes.length;         }
   public String select(int i) {  return suffixes[i].toString();  }

   public int lcp(int i)        // See Exercise 6.28.
   public int rank(String key)  // See Exercise 6.28.
}
```

This implementation of our `SuffixArray` API depends for its efficiency on the fact that the suffixes are represented implicitly (see text), using the nested class `Suffix`.

For example, in many applications, it is reasonable to use a random string model:

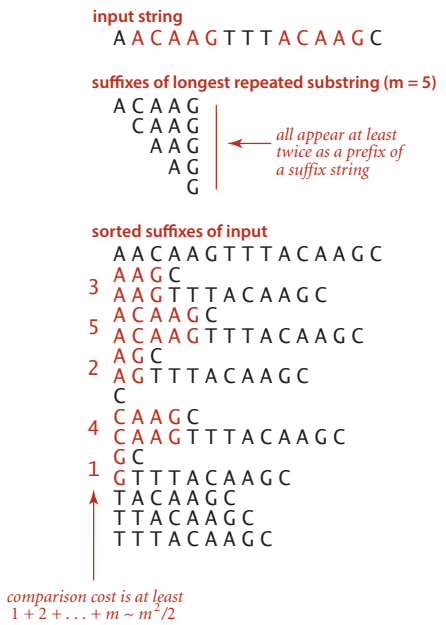
Proposition C. Using 3-way string quicksort, we can build a suffix array from a random string of length n with space proportional to n and $\sim 2n \ln n$ character compares, on the average.

Discussion: The space bound is immediate, but the time bound follows from a detailed and difficult research result by P. Jacquet and W. Szpankowski, which implies that the cost of sorting the suffixes is asymptotically the same as the cost of sorting n random strings (see PROPOSITION E on page 723).

Improved implementations. Our elementary implementation of `SuffixArray` (ALGORITHM 6.2) has poor worst-case performance. For example, if all the characters are equal, the sort examines every character in each suffix and thus takes *quadratic* time. For strings of the type we have been using as examples, such as genomic sequences or natural-language text, this is not likely to be problematic, but the algorithm can be slow for texts with long runs of identical characters. Another way of looking at the problem is to observe that the cost of finding the longest repeated substring is (at least) *quadratic in the length of the longest repeated substring* because all of the prefixes of the repeat need to be checked (see the diagram at right). This is not a problem for a text such as *A Tale of Two Cities*, where the longest repeated substring

```
"s dropped because it would have
  been a bad thing for me in a
  worldly point of view i"
```

has just 84 characters, but it is a serious problem for genomic data, where long repeated substrings are not unusual. How can this quadratic behavior for repeat searching be avoided? Remarkably, research by P. Weiner in 1973 showed that *it is possible to solve the longest repeated substring problem in guaranteed linear time*. Weiner's algorithm was based on building a suffix tree data structure (essentially a



LRS cost is quadratic in repeat length

trie for suffixes). With multiple pointers per character, suffix trees consume too much space for many practical problems, which led to the development of suffix arrays. In the 1990s, U. Manber and E. Myers presented a linearithmic algorithm for building suffix arrays directly and a method that does preprocessing at the same time as the suffix sort to support *constant-time* `lcp()`. Several linear-time suffix sorting algorithms have been developed since. With a bit more work, the Manber–Meyers implementation can also support a two-argument `lcp()` that finds the longest common prefix of two given suffixes that are not necessarily adjacent in guaranteed constant time, again a remarkable improvement over the straightforward implementation. These results are quite surprising, as they achieve efficiencies quite beyond what you might have expected.

Proposition D. With suffix arrays, we can solve both the suffix sorting and longest repeated substring problems in linear time.

Proof: The remarkable algorithms for these tasks are just beyond our scope, but you can find on the booksite code that implements the `SuffixArray` constructor in linear time and `lcp()` queries in constant time.

A `SuffixArray` implementation based on these ideas supports efficient solutions of numerous string-processing problems, with simple client code, as in our `LongestRepeatedSubstring` and `KWIC` examples.

SUFFIX ARRAYS ARE THE CULMINATION of decades of research that began with the development of tries for KWIC indices in the 1960s. The algorithms that we have discussed were worked out by many researchers over several decades in the context of solving practical problems ranging from putting the *Oxford English Dictionary* online to the development of the first web search engines to sequencing the human genome. This story certainly helps put the importance of algorithm design and analysis in context.