Week 8 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, psuedocode may be provided where it illustrates a particular useful concept.

Assessed Preparation

Problem 1. Write the suffix array for the string `ACACIA$` and compute its Burrows-Wheeler transform.

Solution

i	SA[i]	Suffix
0	6	$
1	5	A$
2	0	ACACIA$
3	2	ACIA$
4	1	CACIA$
5	3	CIA$
6	4	IA$

Source: `https://visualgo.net/bn/suffixarray`

The sorted rotations of `ACACIA$` are shown below. The BWT is obtained by reading the characters down the right hand side, shown in red.

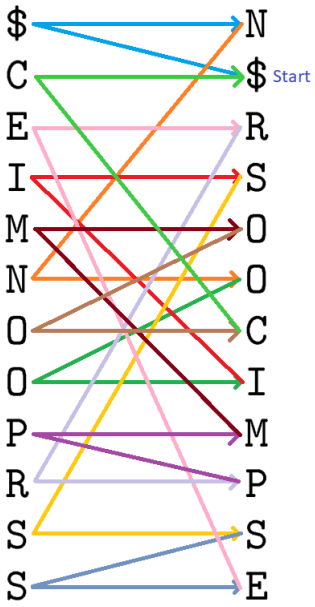
```
$ACACIA
A$ACACI
ACACIA$
ACIA$AC
CACIA$A
CIA$ACA
IA$ACAC
```

The BWT of the string is therefore `AI$CAAC`.

Problem 2. Show the steps in the inverse Burrows-Wheeler transform for the string `N$RSOOCIMPSE`.

Solution

To invert BWT we use the L-F mapping.



Which yields the string `$NOISSERPMOC`. Reversing this gives us the original string, `COMPRESSION$`

Tutorial Problems

Problem 3. Write the suffix array for the string `GATTACA$` and compute its Burrows-Wheeler transform.

Solution

i	SA[i]	Suffix
0	7	$
1	6	A$
2	4	ACA$
3	1	ATTACA$
4	5	CA$
5	0	GATTACA$
6	3	TACA$
7	2	TTACA$

Source: `https://visualgo.net/bn/suffixarray`

The sorted rotations of ACACIA$ are shown below. The BWT is obtained by reading the characters down the right hand side, shown in red.

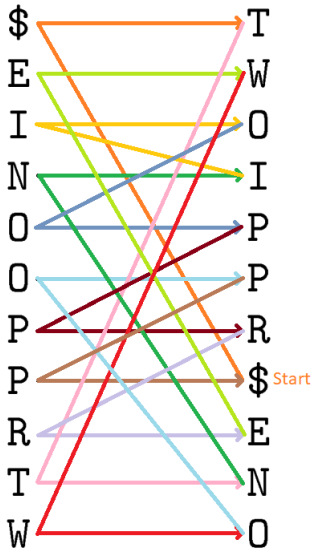
```
$GATTACA
A$GATTAC
ACA$GATT
ATTACA$G
CA$GATTA
GATTACA$
TACA$GAT
TTACA$GA
```

The BWT of the string is therefore ACTGA$TA.

Problem 4. Show the steps in the inverse Burrows-Wheeler transform for the string TWOIPPR$ENO.

Solution

To invert BWT we use the L-F mapping.



Which yields the string $TNIOPREWOP. Reversing this gives us the original string, POWERPOINT$

Problem 5. Compute the Burrows-Wheeler transform of the string woolloomooloo$, and show the steps taken by the pattern matching algorithm for the following patterns:

(a) olo

(b) oll

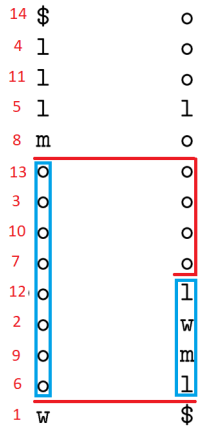
(c) oo

(d) wol

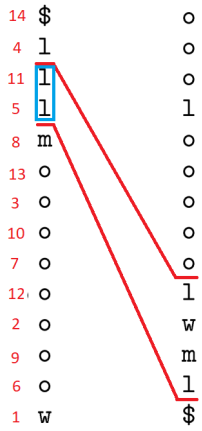
Solution

The BWT of `woolloomooloo$` is `ooolooooolwml$`. If the location of a substring is also to be determined, during the pre-processing step, in addition to the first and last column, we also maintain the ID of cyclic rotation (i.e., suffix ID) with each row – this is the same as a suffix array (see the figures below where the IDs are shown in red). During the query processing, these IDs are used to determine the location as explained later. When searching for patterns we start at the back of the pattern, and search in reverse, using the LF mapping to guide us to occurrences of the previous character each time. Diagrams for each search follow.

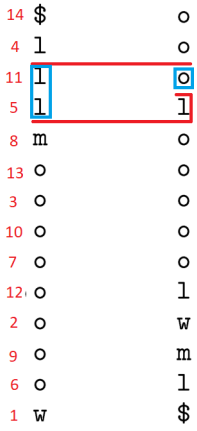
(a) Searching for `olo`



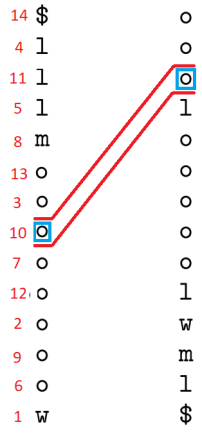
Starting at "o", we see there are 2 "l"s within the range in the last column, so we contract the range



Using the L-F mapping, our range now contains the cyclic shifts which have "lo" as a prefix.

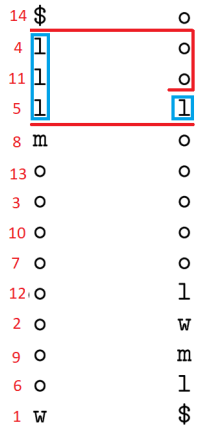


In the last column there is only one "o" in our range, so there is one instance of the substring `olo`

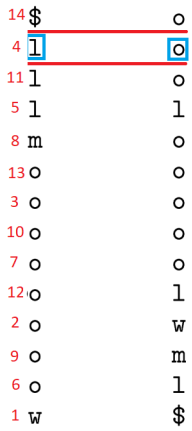


We use L-F mapping to get the location of "o" in the first column and determine its location using suffix ID, e.g., the location of `olo` in the string is 10.

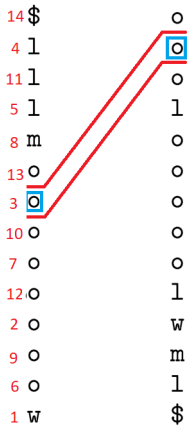
(b) Searching for `oll`



Starting at "`l`", we see there is one "`l`" within the range in the last column, so we contract the range

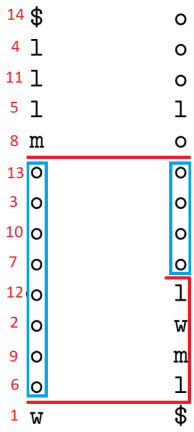


After applying the L-F mapping, we see one "`o`" in our range, we know there is one "`oll`" substring.

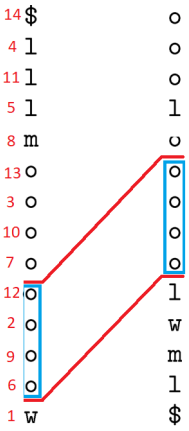


Again using the L-F mapping, we can now look up the location of our substring which is 3.

(c) Searching for oo

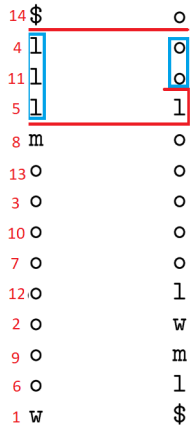


Starting with "o", we can see 4 "o"s in the range, so there are 4 "oo" substrings.

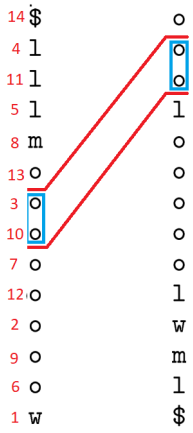


Using the L-F mapping, we can now look up the locations of our substrings in the suffix array (12, 2, 9 and 6)

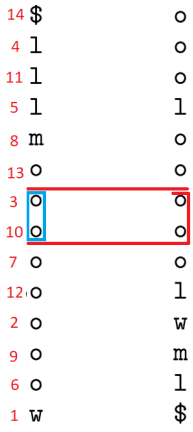
(d) Searching for `wol`



Starting at "`l`", we see there are 2 "`o`"s within the range in the last column, so we contract the range



Apply the L-F mapping. We are tracking `ol` substrings.



We observe that the range contracts to 0 (there are no "`l`"s in the range) so there are no `wol` substrings.

Problem 6. Consider the prefix doubling algorithm applied to computing the suffix array of `JARARAKA$`. Write the partially-sorted suffix array after the length two prefixes have been sorted. Write the corresponding rank array and perform the next iteration of prefix doubling, showing the partially-sorted suffix array for the length four prefixes.

Solution

After sorting the length 2 prefixes, the suffixes are in the following order, so we have the following suffix array and rank array.

Index	Suffix	Rank
9	$	1
8	A$	2
6	AKA$	3
2	ARARAKA$	4
4	ARAKA$	4
1	JARARAKA$	5
7	KA$	6
3	RARAKA$	7
5	RAKA$	7

The order of the suffixes

Index	1	2	3	4	5	6	7	8	9
Suffix Array	9	8	6	2	4	1	7	3	5
Rank	5	4	7	4	7	3	6	2	1

The corresponding suffix array and rank array

After performing one more iteration and sorting the length 4 prefixes, the suffixes will be in the following order, yielding the following suffix array and rank array.

Index	Suffix	Rank
9	$	1
8	A$	2
6	AKA$	3
4	ARAKA$	4
2	ARARAKA$	5
1	JARARAKA$	6
7	KA$	7
5	RAKA$	8
3	RARAKA$	9

The order of the suffixes

Index	1	2	3	4	5	6	7	8	9
Suffix Array	9	8	6	4	2	1	7	5	3
Rank	6	5	9	4	8	3	7	2	1

The corresponding suffix array and rank array

Since all of the suffixes have unique ranks, we can tell that the suffixes are fully sorted and therefore the suffix array is the final suffix array.

Problem 7. The *minimum lexicographical rotation* problem is the problem of finding, for a given string, its cyclic rotation that is lexicographically least. For example, given the string `banana`, its cyclic rotations are `banana`, `ananab`, `nanaba`, `anaban`, `nabana`, `abanan`. The lexicographically (alphabetically) least one is `abanan`. Describe how to solve the minimum lexicographical rotation problem using a suffix array.

Solution

A cyclic rotation of a string is a suffix of that string followed by a prefix of that string. Since a suffix array stores the suffixes in sorted order, the lexicographically least suffix will be the first element of the arrray (ignoring the "$" which will always be first.) However, we can run into problems such as the string `XAA$`:

i	SA[i]	Suffix
0	3	$
1	2	a$
2	1	aa$
3	0	xaa$

Source: `https://visualgo.net/bn/suffixarray`

Here, the suffix "`A$`" would be followed by `XA` in its cyclic rotation, giving `AXA`, wheras the suffix `AA$` would be followed by `X`, giving `AAX`, which is lexicographically earlier than `AXA`. So it is not enough to just construct the suffix array, we need to account for the elements of the corresponding prefix.

We do this by first appending a copy of the string in question to itself, and then constructing the suffix array. This gaurantees that the corresponding prefix for each suffix is taken into account in the order of the suffix array. Note that we should only consider suffixes which are strictly longer than the original string, since only these suffixes contain characters from the appended copy.

i	SA[i]	Suffix
0	6	$
1	5	a$
2	4	aa$
3	1	aaxaa$
4	2	axaa$
5	3	xaa$
6	0	xaaxaa$

Source: `https://visualgo.net/bn/suffixarray`

Now `AAXAA$X` is the first such string in the suffix array, and hence `AAX` is the lexicographically least rotation.

Supplementary Problems

Problem 8. A string S of length n is called *k-periodic* if S consists of n/k repeats of the string $S[1..k]$. For example, the string `abababab` is two-periodic since it is made up of four copies of `ab`. Of course, all strings are n-periodic (they are made of one copy of themselves!) The *period* of a string is the minimum value of k such that it is k-periodic. Describe an efficient algorithm for determining the period of a string using a suffix array.

Solution

To solve this problem, let's relate the period of the string to the concept of cyclic rotations. If a string is k-periodic, then this means that it is equal to its k^{th} cyclic rotation. For example, if we take the string `abababab`, then its cyclic rotations are `babababa`, then `abababab`, which is the original string again. We

can therefore use an idea similar to Problem 7. Let's append a copy of the string S to itself to obtain the string SS. Now we observe that if a string is k-periodic, then the string S will appear in position $k+1$ in SS. For example, take `abababab`, and append it to itself to obtain `ab`**`abababab`**`ababab`. Then we notice that the substring at position 3 is precisely the original string. Thus, after suffix array on the string SS is constructed, we can do a substring search for S in the suffix array to find all suffixes that contain the substring S. Among these suffixes, the suffix with the smallest ID (ignoring the first suffix) can be used to determine the value of k. For example, we search `abababab` in the suffix array of `abababababababab` which matches with suffixes 1, 3, 5, 7, 9. The smallest suffix (except first suffix) that matches is 3. Therefore, the value of k is 2.

Building the suffix array takes $O(n \log^2 n)$ time, and substring search takes $O(n \log(n))$.

Problem 9. Write psuedocode for an algorithm that converts the suffix tree of a given string into the suffix array in $O(n)$ time.

Solution

A suffix array is just the suffixes in sorted order. Since the suffix tree contains all of the suffixes as leaves, we just need to traverse it in lexicographic order and note down the order in which we find the leaves, and this will give us the suffix array. Whenever we reach a leaf, we can figure out which suffix it belongs to by looking at its length. A suffix of length k must be the suffix from position $n-k+1$

```
1:  function TO_SUFFIX_ARRAY()
2:      SA[1..n] = null
3:      counter = 1     // Tracks which suffix we are up to
4:      CONVERT(root, 0)
5:      return SA[1..n]
6:  end function
```

```
1:  function CONVERT(node, length)
2:      if node is a leaf then
3:          SA[counter] = n - length + 1
4:          counter = counter + 1
5:      else
6:          for each child of node, in lexicographic order do
7:              CONVERT(child, length + child.num_chars)
8:          end for
9:      end if
10: end function
```

Problem 10. (Advanced) Recall the pattern-matching algorithm that uses the Burrows-Wheeler transform of the text string. One downside of this algorithm is the large memory requirement if we decide to store the occurrences $\text{Occ}(c, i)$ explicitly for every position i and every character c. In this problem we will explore some space-time tradeoffs using *milestones*. Suppose that instead of storing $\text{Occ}(c, i)$ for all values of i, we decide to store it for every k^{th} position only, i.e. we store $\text{Occ}(c, 0)$, $\text{Occ}(c, k)$, $\text{Occ}(c, 2k)$, $\text{Occ}(c, 3k)$, ... for all characters c.

(a) What is the space complexity of storing the preprocessed statistics in this case?

(b) What is the time complexity of performing the preprocessing?

To compute $\text{Occ}(c, i)$, we will take the value of $\text{Occ}(c, j)$ where j is the nearest multiple of k up to i and then manually count the rest of the occurrences in $S[j+1..i]$

(c) What is the time complexity of performing a query for a pattern of length m?

(d) Describe how bitwise operations can be exploited to reduce the space complexity of the preprocessed statistics to $O\left(\frac{|\Sigma|n}{w}\right)$ where w is the number of bits in a machine word, while retaining the ability to perform pattern searches in $O(m)$.

Solution

(a) Since we will store the value of $\text{Occ}(c, k)$ for all multiples of k up to n, the amount of space required will be

$$O\left(\frac{|\Sigma|n}{k}\right).$$

(b) We perform a linear sweep over the text string, maintaining the counts for all characters. When we hit a multiple of k, we save the current state of the counts. In total, this takes

$$O\left(|\Sigma| + n + \frac{|\Sigma|n}{k}\right)$$

time.

(c) For a given position, we can compute the nearest multiple of k in constant time, hence the only work required is to count the number of occurrences between the milestone and the query position. In the worst case, we may have to count $k-1$ elements, hence this takes $O(k)$ time per character. Therefore to search a pattern of length m takes $O(mk)$ time in the worst case.

(d) In order to achieve $O(m)$ time pattern searches, we need to be able to compute $\text{Occ}(c, i)$ in constant time. To do so, we make milestones of size $k = w$, giving us the value of Occ at every w^{th} position. This gives us the space complexity desired. Our goal is to speed up the computation of counting the remaining occurrences between the milestones.

For each milestone position j, for each character $c \in A$, let's store a w-bit integer whose b^{th} bit is 1 if the character at position $j + b$ is c, or a 0 otherwise. To compute the number of occurences of c in the interval $[j+1..i]$ then corresponds to counting the number of 1 bits in the first $i-j$ positions of this integer. This can be achieved by masking off the bottom bits by taking its bitwise AND with $2^{i-j} - 1$ and then using a *popcount* operation, which counts the number of 1 bits in an integer. Assuming that our machine model supports popcount in $O(1)$ time, we can now compute $\text{Occ}(c, i)$ in $O(1)$ time. Since we stored just one integer at each milestone for each character, the amount of extra space used is $O\left(\frac{|\Sigma|n}{w}\right)$, in addition to the $O\left(\frac{|\Sigma|n}{w}\right)$ space used for the milestones, making the total space complexity as required.

Problem 11. (Advanced) A given suffix array could have come from many strings. For example, the suffix array 7, 6, 4, 2, 1, 5, 3 could have come from `banana$`, or from dbxbxa$, or many other possible strings.

(a) Devise an algorithm that given a suffix array, determines any string that would produce that suffix array. You can assume that you have as large of an alphabet as you need.

(b) Devise an algorithm that given a suffix array, determines the lexicographically least string that would produce that suffix array. You can assume that you have as large of an alphabet as required. Your algorithm should run in $O(n)$ time

Solution

(a) To construct any string that has a given suffix array is quite simple. The first suffix will be the empty suffix so we always have $ as the final character. From there, the next suffix should begin with `a`, the next with `b`, the next with `c`, and so on. This means that we place `a` at position SA[2], `b` at position SA[3], `c` at position SA[4] and so on. For example, given the suffix array above, this algorithm would produce the string `dcfbea$`.

(b) To produce the lexicographically earliest string is a bit trickier. What we would like to do is reuse the same character as many times as possible, starting several suffixes with `a`, then several with `b`, and so on if we can. But we need to be careful to ensure that we keep the order of the suffixes correct. Suppose we place some letter c at the position SA[i]. How do we determine whether it is safe to also place c at position SA[$i+1$]? Well, the suffix at position SA[$i+1$] must be greater since it comes later in the suffix array, so we need to ensure that

$$c + S[(\text{SA}[i]+1)..n] < c + S[(\text{SA}[i+1]+1)..n],$$

which means that we need $S[(\text{SA}[i]+1)..n] < S[(\text{SA}[i+1]+1)..n]$. In other words, we just need to compare the two suffixes at positions SA[i]+1 and SA[$i+1$]+1. To do so quickly, let's compute the ranks of the suffixes. These are easy to compute since we have the full suffix array. Deciding whether we can place character c then comes down to checking whether rank[SA[i]+1] < rank[SA[$i+1$]+1], which can be done in constant time. If it is not safe to reuse character c, we move on to the next character in the alphabet. In total, this algorithm takes $O(n)$ time since computing the ranks can be done in $O(n)$, and each suffix comparison takes $O(1)$ with the rank array. This algorithm will produce `bacaca$` as the lexicographically least string with the same suffix array as `banana$`.

Problem 12. (Advanced) A useful companion to the suffix array is the *longest common prefix* array. It contains for each suffix in sorted order, the length of the longest prefix that is shared by that suffix and the one lexicographically preceding it. Formally, it contains:

$$\text{LCP}[i] = \{\text{The maximum value of } k \text{ such that } S[\text{SA}[i]..(\text{SA}[i]+k)] = S[\text{SA}[i-1]..(\text{SA}[i-1]+k)]\}$$

For example, consider our favourite string `banana$`. The common prefixes are highlighted in the figure below.

```
$
a$
ana$
anana$
banana$
na$
nana$
```

The suffixes `a` and `ana` share the prefix `a`, the suffixes `ana` and `anana` share the prefix `ana`, and the suffixes `na` and `nana` share the prefix `na`. The longest common prefix array would therefore be $0,0,1,3,0,0,2$.

(a) Describe an $O(n^2)$ algorithm to compute the LCP array (this should be easy)

(b) Recall that rank[i] denotes the order of suffix i in the suffix array. Explain why

$$\text{LCP}[\text{rank}[i]] \geq \text{LCP}[\text{rank}[i-1]] - 1.$$

(c) Use the fact given in (b) to write an $O(n)$ algorithm to compute the LCP array.[1]

(d) Describe how the LCP array can be used to solve the longest repeated substring problem

(e) Describe how the LCP array can be used to count the number of distinct substrings of a string

Solution

(a) With $O(n^2)$ time we can simply do the naive algorithm, which involves checking every pair of adjacent suffixes in the suffix array and seeing how many characters they have in common.

(b) LCP[rank[i]] refers to the LCP of the suffix beginning at position i. Similarly, LCP[rank[$i-1$]]-1

[1] This is known as Kasai's algorithm.

refers to the LCP of the suffix at position $i-1$. Observe that the suffix at position i overlaps with the suffix at position $i-1$ in all but the first character, since it is just the very next suffix in the string. This means that if there is a suffix that shares k characters with the suffix at position $i-1$, then there is a suffix that shares at least $k-1$ characters with the suffix at position i, since they overlap in all of those characters except the first.

(c) We will use the fact from (b) to compute the LCPs quickly. Instead of iterating through the suffixes in the order that they appear in the suffix array, we'll iterate in the order that they appear in the string. Once we compute LCP[rank[i]], we then compute LCP[rank[$i+1$]], whose value is at least the previous value minus one. This means that we do not have to recompare all of the characters that we just compared, but can skip over the ones that we already know are equal. By doing so, this ensures that each character only ever gets compared once, and hence the total time complexity is $O(n)$. Here is some pseudocode implementing this algorithm.

```
1:  function LCP(S[1..n], SA[1..n])
2:      Set len = 0
3:      Set rank[1..n] = 0
4:      Set LCP[1..n] = 0
5:      for i = 1 to n do
6:          rank[SA[i]] = i     // Compute ranks
7:      end for
8:      for i = 1 to n do
9:          if rank[i] > 1 then
10:             Set j = SA[rank[i]-1]     // j is the lexicographically previous suffix
11:             while i + len ≤ n and j + len ≤ n and S[i+len] = S[j+len] do
12:                 len = len + 1
13:             end while
14:             LCP[rank[i]] = len
15:         end if
16:         if len > 0 then len = len - 1
17:     end for
18:     return LCP[1..n]
19: end function
```

(d) Since suffixes that share characters end up adjacent in the suffix array, the suffixes containing the longest repeated substrings will end up next to eachother. Since the LCP array counts the number of characters that adjacent suffixes have in common, the length of the longest repeated substring will just be the largest number in the LCP array.

(e) The total number of substrings in a string is $\binom{n+1}{2}$ since we can choose any start and end point (the plus one accounts for the fact that a substring can start and end at the same place). We can use the LCP array to count the number of duplicates. Observe that for any particular suffix, if it overlaps with k characters from its preceding suffix, then the first k prefixes of that suffix are duplicated in the predecessor. This means that the number of duplicated substrings is just the sum of the values in the LCP array. So the total number of distinct substrings is

$$\binom{n+1}{2} - \sum_{i=1}^{n} \text{LCP}[i].$$