Solution to Homework-2

Sanath Kumar Ramesh, A50054305

13-April-2011

1 Problem-1

The solution to this problem follows from Last First Property. According to last first property, the i^{th} occurrence of a character in last column and the i^{th} occurrence of the same character in the first column, correspond to the same character in the original string. Let our original string be represented as $u\epsilon\sigma v$ where u, v are prefix and suffix of the string. ϵ, σ are single characters. FindPrev(B,i) must return ϵ if $B[i] = \sigma$. By last first property, if $B[i]$ is the p^{th} occurrence of σ in the last column, then the last character in the row that has the p^{th} occurrence σ in the first column gives ϵ.

FindPrev(B[], Pos[], Occ[], i)
Output: previous character
begin
  $\sigma = B[i]$
  $p = Occ[\sigma, i]$
  $rowNum = Pos[\sigma] + p$
  return $B[rowNum]$
end

Example: Let the original string be $S = agtct$. Let $\epsilon =$ t and $\sigma =$ c. Therefore, $u =$ ag and $v =$ t. In computing the BW transform, we rotate the string circularly as shown in Table 1. ϵ is represented by a bold letter and σ by a circled letter. Table 2 shows the strings after sorting them.

For $i = 5$, $B[5] = c$. c is the 1st occurrence in B[i]. The first occurrence of c in first column happens at row 3. Therefore the last character in row-3 ie. $B[3]$ gives our result. $B[3] = t$, therefore **t** is our result.

Table 1: Circularly rotated string

$	a	g	**t**	ⓒ	t
t	$	a	g	**t**	ⓒ
ⓒ	t	$	a	g	**t**
t	ⓒ	t	$	a	g
g	**t**	ⓒ	t	$	a
a	g	**t**	ⓒ	t	$

Table 2: Sorted Suffix Strings

$	a	g	**t**	ⓒ	t
a	g	**t**	ⓒ	t	$
ⓒ	t	$	a	g	**t**
g	**t**	ⓒ	t	$	a
t	$	a	g	**t**	ⓒ
t	ⓒ	t	$	a	g

This algorithm takes constant time to find the previous character and constant extra space.

2 Problem-2

The solution to this problem follows from the property of how the original string was rotated. After constructing the BW transform, in any row of the transform, the first character and the last character in that row are always next to each other in the original string. Since the strings have been sorted in ascending order, given the last column ie. B[i], one can compute the first column of the transform. With the first column and last column, one can computer the pairwise ordering of characters in the string. These pairwise orderings can

Table 3: Circularly rotated string

$	b	t	a	t	c
c	$	b	t	a	t
t	c	$	b	t	a
a	t	c	$	b	t
t	a	t	c	$	b
b	t	a	t	c	$

Table 4: Sorted Suffix Strings

$	b	t	a	t	c
a	t	c	$	b	t
b	t	a	t	c	$
c	$	b	t	a	t
t	a	t	c	$	b
t	c	$	b	t	a

be combined to get back the original string. The procedure is explained with an example: Let the original string be *btatc*. Table 3 gives the cirularly rotated string and Table 4 gives the sorted suffix strings.

The last column of this transform is given to us as $B[i]$ which is:

t
$
t
b
a

The algorithm will go through the array $B[i]$ and find the number of occurrences of $\$, a, c, g, t$. In this example, $\$ = 1, a = 1, b = 1, c = 1, t = 2$. This count of the number of occurrences is the first column which we represent by the *count*[] array. For this example, $count[0] = 1, count[1] = 1, count[2] = 1, count[3] = 1, count[4] = 2$. We duplicate this array and name them as $countFirst$ and $countLast$ holding the counts for first column and $B[i]$ which is the last column in the transform.

Now go about constructing the pairwise ordering using $B[i]$ and first column which we have it in the form of $countFirst$. The first column is created by scanning through each each element in the $columnFirst$ array and creating as many alphabets as the count in that position. For above example, we'll create one $, one a, one b, one c and two t in order.

For the example, the last column along with the first column is

Last column	1st column
c	$
t	a
$	b
t	c
b	t
a	t

The pairwise ordering is that for some row i, the character in the $last\_column[i]$ precedes the character in the $first\_column[i]$ in the original string. With a set of pairwise orderings, to construct the original string one needs to guarantee that all the elements in the ordering are unique. Since here we can have repeating characters, we make them unique by attaching an index along with character. This index is the equal to the occurrence of that character in that column when scanning the column from bottom-up. This index is assigned using the count arrays $countLast$ and $countFirst$. $countLast$ keeps track of the indices for last column and $countFirst$ keeps track of the indices for the first column. For the example, the indices are as follows:

countLast	Last column	1st column	countFirst
$=1, a=1, b=1, c=1, t=2	c_1	$\$_1$	$=1, a=1, b=1, c=1, t=2
$=1, a=1, b=1, c=0, t=2	t_2	a_1	$=0, a=1, b=1, c=1, t=2
$=1, a=1, b=1, c=0, t=1	$\$_1$	b_1	$=0, a=0 ,b=1 ,c=1, t=2
$=0, a=1, b=1, c=0, t=1	t_1	c_1	$=0, a=0, b=0, c=1, t=2
$=0, a=1, b=1, c=0, t=0	b_1	t_2	$=0, a=0, b=0, c=0, t=2
$=0, a=1, b=0, c=0, t=0	a_1	t_1	$=0, a=0, b=0, c=0, t=1
$=0, a=0, b=0, c=0, t=0			$=0, a=0, b=0, c=0, t=0

Now the pairwise ordering is as follows:

$$c_1 \longrightarrow \$_1$$

$$t_2 \longrightarrow a_1$$

$$\$_1 \longrightarrow b_1$$

$$t_1 \longrightarrow c_1$$

$$b_1 \longrightarrow t_2$$

$$a_1 \longrightarrow t_1$$

Now join the orderings using transitive relationship. This will yield the following complete ordering.

$$\$_1 \longrightarrow b_1 \longrightarrow t_2 \longrightarrow a_1 \longrightarrow t_1 \longrightarrow c_1$$

Dropping the indices and \$ gives the original string - *btatc*.

```
Algorithm construct-string(B)
Output: original string
begin
  countFirst[ ] = countLast[ ] = findCounts(B)
  ordering = {}
  first_col_ptr = 1                                  // Tracks the non-zero count character countFirst
  for iinB do
      x = character_at(first_col_ptr)
      y = B[i]
      X =< x, countFirst[first_col_ptr] >
      Y =< y, count_at(countLast, y)) >
      ordering = ordering ∪ X, Y
      countFirst[first_col_ptr] = countFirst[first_col_ptr] − 1
      if countFirst[first_col_ptr] == 0 then
        first_col_ptr = first_col_ptr + 1
      fi
      decrement_count(countLast, y)
  od
end
```

```
function character_at(pos)
Output: character denoted by pos
begin
  if pos == 0 then return '$'
else  if pos == 1 then return 'a'
else  if pos == 2 then return 'c'
else  if pos == 3 then return 'g'
else  return 't'
```

```
  fi
end
```

```
function decrement_count(countArray, char)
Output: none
begin
  if char == '$' then countArray[0] = countArray[0] − 1
    else if char == 'a' then countArray[1] = countArray[1] − 1
    else if char == 'c' then countArray[2] = countArray[2] − 1
    else if char == 'g' then countArray[3] = countArray[3] − 1
    else countArray[4] = countArray[4] − 1
  fi
end
```

```
function count_at(countArray, char)
Output: value of countArray corresponding to char
begin
  if char == '$' then return countArray[0]
    else if char == 'a' then return countArray[1]
    else if char == 'c' then return countArray[2]
    else if char == 'g' then return countArray[3]
    else return countArray[4]
  fi
end
```

3 Problem-3

This may not be a good idea for compression because, BW transform of a string is going to be very different from the original string. In case the original string had many recurring patterns, then BW transformed string might break those patterns making it less amenable to compression.

4 Problem-4

The pseudo-code for my program is as follows:

```
Algorithm FindRepeat(DNA)
begin
  KeywordTree = {}
  OutputList = {}
  for  each substring s of length 16 from DNA do
      pos =  position of beginning of s
      posList = Add(KeywordTree, s, pos)
      if OutputList.exists(posList) == false then
        OutputList.add(posList)
                                                    /* posList is added as a reference to OutputList
                                          Changes made on the posList through KeywordTree will reflect here */
      fi
  od
  print OutputList
end
```

```
function Add(KeywordTree, s, pos)
begin
  current_node = KeywordTree.root
  for i = 1to16 do
      if current_node[ s[i] ] == NULL then
        node = create_newnode()
        current_node[ s[i] ] = node
      fi
      current_node = current_node[ s[i] ]
  od
  comment: Last node in KeywordTree will have pointer to position-list
  if current_node.posListExists == false then
    posList = create_poslist()
    posList = (s, pos)
    current_node.postList = posList
    return posList
else
    posList = current_node.posList
    posList.add(pos)
    return posList
  fi
end
```

Explaining in words, split DNA sequence into substrings of 16characters. Using these strings, construct a Keyword Tree. The leaf node of the keyword tree will point to a list called position-list. This list stores the starting position of substring in DNA sequence. When another occurrence of the same string is encountered, an existing path will be traversed by the algorithm to reach the final node. In the position-list attached to the final node, add the starting position of this string. Thus, position-list at each leaf node will have a list of starting positions of the substring in DNA sequence. To make the algorithm faster, another list called OutputList keeps record of the position-lists attached to the leaf nodes. Thus after the whole DNA sequence has been traversed, the OutputList will hold the position-list corresponding to each and every substring in the DNA. In a KeywordTree, since each substring has a unique path from root to the leaf node, the repetition counts of all substrings in the DNA can be found in just one traversal of the DNA sequence. Thus this algorithm has $O(n)$ time and $O(n)$ space complexity.

4.1 Expected number of repeats

Since each base is equally likely, each has a probability of 1/4. For a DNA sequence, each letter in the sequence has 4 possibilities. For a sequence of length 16, there are 4^{16} possible combinations of bases. For DNA sequence of length N, there are $N - 15$ substrings of length 16. For large N, $N - 15 \approx N$. Therefor, there are N different substrings of length 16 possible from a DNA sequence of length N.

Since there are 4^{16} possible substrings of length 16, the probability of getting one particular substring is $1/4^{16}$. Probability of finding two same substrings then becomes $(1/4^{16}) * (1/4^{16})$. Modeling this process as a Binomial Distribution, the expected number of repeats with N substrings is $N * (1/4^{16}) * (1/4^{16})$

For $N = 4^{12}$, the expected value is $1/4^{20}$!! This number is really small. It is clearly shown by the number of repeats observed. This is shown in Figure 1. The raw data provides more clear picture of the trend. It is shown in Table 5.

Such dramatically low number for 2-repeats and 3-repeats is because of the low probablity of finding one substring as described above.

The Figure 2 shows the running time as a function of data size. The algorithm scales really well with data size. This is because along with the data structure construction, the algorithm finds the position of repeats.

Table 5: Frequency of repeats seen

	1 repeats	2 repeats	3 repeats
4^7	163690	0	0
4^8	655202	4	0
4^9	2621170	60	0
4^{10}	10483110	1250	0
4^{11}	41902166	20359	2

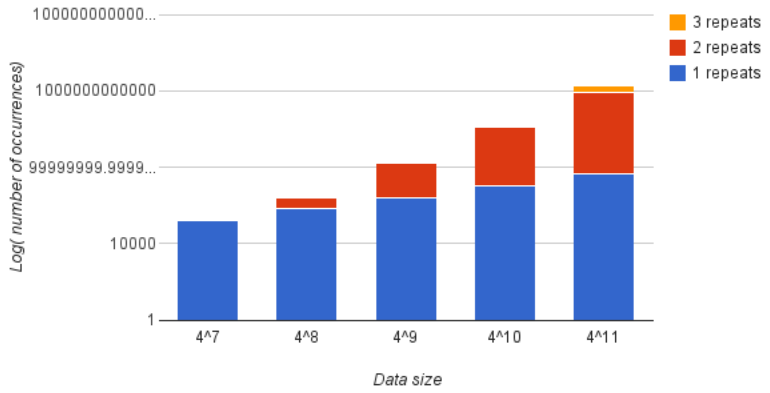

Figure 1: Number of occurrences of repeats in log scale. I was unable to get data for 4^{12} size

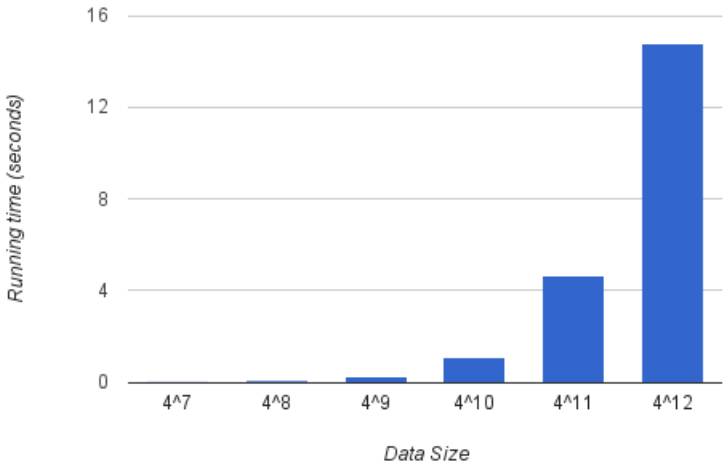

Figure 2: Running time of code on various data sizes