# Comp 555 Problem Set 4

Elliott Hauser

November 20, 2012

## Question 1

### a

While both Hash-table and Suffix trees would correctly solve this problem, they may be preferred over one another depending on the circumstances. Suffix trees have a run time of $O(m)$, making them preferable *ceteris paribus.* But there is computational and engineering overhead to generate the suffix tree. Hash tables are implemented natively in many programming languages (such as Python's `dict` structure), making them easier to implement in some situations.

That said, for our purposes computational and engineering overhead are negligible when compared to runtime savings, especially given that $n$ is likely to be very long. So, the suffix tree structure is most appropriate for our purposes here.

### b

Assuming the suffix table for $m$ is already constructed as a graph, an algorithm to find the number of occurrences of $m$ in $n$ is given in Figure 1

### c

The time complexity is $O(m)$.

```
Preprocess n into suffix tree s_n in O(n).

SuffixSearch(m, s_n)
position = 0
for edge in childEdge(s_n) == m[position:len(edge_label)]
   if descendent vertex has child edge == m[position +1:len(
       child_edge_label)]
      SuffixSearch(m[position+1:len(m)], descendants(edge)
   else
      matches = count(descendants(edge))
      return matches
```

Figure 1: An algorithm to find a string $m$ in a longer string $n$.

# Question 2

A hash table scheme for providing the student id, given a score $x$, is to sort student IDs into a Hash table based on some feature of the scores themselves. For this problem, we've been told that the scores can range from -10 to 20 and that the hash table should have at most 5 rows. There are many solutions to this, but an optimal solution would divide both possible and actual values as evenly as possible amongst the 5 rows to reduce collisions. The best solution I've found is to separate scores based on the sum of their digits. Table 2 illustrates where each possible score would be stored. This scheme achieves the goal of equally distributing possible values of scores across the rows. In addition, as shown below in b, the actual values of scores are distributed evenly, $+/-1$ score. Within rows, values should be ordered by ascending score.-

| Digit sums | Corresponding scores |
|---|---|
| $1, 2$ | $-2, -1, 1, 2, 10, 11$ |
| $3, 4$ | $-4, -3, 3, 4, 12, 13$ |
| $5, 6$ | $-6, -5, 5, 6, 14, 15$ |
| $7, 8$ | $-8, -7, 7, 8, 16, 17$ |
| $9, 10, 0$ | $-9, 0, 9, 10, 18, 19$ |

Table 1: A hash table scheme for storing student IDs and scores based on the sum of the digits of the score. Each row has six possible scores.

## a

The hash function in Figure 2 shows a hash function to construct and populate the table described in Table 2.

## b

The actual hash table for this question is shown in Table 2. As noted above, the scheme achieves a relatively even distribution of scores, minimizing collisions. Even though Row 2 has three students in it, and Row 1 only has one, there is no way to reduce collisions further in a hash table this small.

## c

The script in Figure 3 would return the two students with a score of four.

```
MakeTable()
# Makes a blank table
hash_table = 2 columns x 5 rows

# Add digit counts
for row, count in hash_table, range(1,9,2)
   row[0] ← [count, count+1]

# Add zero
hash_table[5][0] ← 0

DigitHash(score)
# Adds the digits of a score
for digit in score
   sum += abs(digit)
return sum

AddStudent(student,x)
# Adds student and score to table in correct spot
sum = DigitHash(x)

for row in hash_table where row[0] == sum
   # if there's already a student with that score, add this student
   if row[1][0] == sum
      row[1] ← student
   # otherwise, make a new entry with score and student.
   else
      add \{sum, student\} to row[1]
```

Figure 2: A digit sum hash function.

| Digit sums | Corresponding scores |
|---|---|
| 1, 2 | {-1,A4} |
| 3, 4 | {4,A1,A2},{12,A8} |
| 5, 6 | {14,A5},{15,AA} |
| 7, 8 | {7,A9},{17,A7} |
| 9,10,0 | {9,A3},{19,A6} |

Table 2: The actual hash table for Question 2.

```
index = DigitHash(4)              # =>  4

for row[0] in hash_table == index   # =>  2nd row
   Find 4 in entry in row[1]      # =>  first item
   return entry[1::]              # =>  [A1, A2]
```

Figure 3: Locating all students with score of 4 using my hash table.

# Question 3

An algorithm to find all palindromes of length 2 or greater is in Figure 4. This algorithm is $O(nm)$, where $n$ is the length of the string being searched and $m$ is the average length of palindromes found.

A more efficient method would be to simultaneously search two suffix trees, one of the string, and one of its reverse. This method would involve seeking mirror image patterns in the trees and then searching for the total pattern. A rough sketch of this approach is in Figure 5.

```
PalindromeSearch(text)
"""Find & return all palindrome strings in a text"""

palindromes = [ ]

for n in len(text)
   if nth char in text equals (n+1)^{th} char
      palindromes ← PalindromeBuild(n, n+1, text)
   if nth char in text equals (n+2)^{th} char
      palindromes ← PalindromeBuild(n, n+2, text)

return palindromes


PalindromeBuild(start, stop, text)
"""Takes the start and stop positions of a palindrome in a text.
   Extends palindrome found between start and stop, if possible, and
    returns longest palindrome found"""
if text[start-1] == text[stop+1]
   PalindromeBuild(start-1,stop-1)
else
   return text[start:stop]
```

Figure 4: An algorithm to find all palindromes in a text in O(n) on the length of the text.

```
# Assume we have suffix trees of both the string and its inverse

suffix = suffix tree of string
prefix = suffix tree of reverse of string

depth = 1
candidates = [ ]
path = [ ]

SuffixSearch(suffix, prefix)
RecurTree(suffix, prefix, depth)
   (returns candidates)

For path in candidates
   thread path through suffix
      when thread breaks, return path_traveled.

def RecurTree(suffix, prefix, depth, path)
   For each node at depth in suffix
      follow path through prefix
      if prefix has a node at end of path equal to suffix node
         path ← node
         depth +1
         RecurTree(suffix, prefix, depth, path)
      candidates ← path
```

Figure 5: A sketch of the suffix tree approach to finding palindromes in a string.

| | x | y |
|---|---|---|
| p1 | 2.2 | 3.3 |
| p2 | 1.6 | 4.5 |
| p3 | 6.7 | 3.3 |
| p4 | 4.7 | 1.2 |
| p5 | 8.6 | 7.5 |
| p6 | 5.3 | 5.6 |
| p7 | 7.6 | 6.5 |
| p8 | 2.5 | 4.6 |
| p9 | 5.5 | 4.5 |
| p10 | 1.1 | 3.8 |

| | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | p9 | p10 |
|---|---|---|---|---|---|---|---|---|---|---|
| p1 | 0.0 | 1.3 | 4.5 | 3.3 | 7.7 | 3.9 | 6.3 | 1.3 | 3.5 | 1.2 |
| p2 | 1.3 | 0.0 | 5.2 | 4.5 | 7.6 | 3.9 | 6.3 | 0.9 | 3.9 | 0.9 |
| p3 | 4.5 | 5.2 | 0.0 | 2.9 | 4.6 | 2.7 | 3.3 | 4.4 | 1.7 | 5.6 |
| p4 | 3.3 | 4.5 | 2.9 | 0.0 | 7.4 | 4.4 | 6.0 | 4.0 | 3.4 | 4.4 |
| p5 | 7.7 | 7.6 | 4.6 | 7.4 | 0.0 | 3.8 | 1.4 | 6.8 | 4.3 | 8.4 |
| p6 | 3.9 | 3.9 | 2.7 | 4.4 | 3.8 | 0.0 | 2.5 | 3.0 | 1.1 | 4.6 |
| p7 | 6.3 | 6.3 | 3.3 | 6.0 | 1.4 | 2.5 | 0.0 | 5.4 | 2.9 | 7.0 |
| p8 | 1.3 | 0.9 | 4.4 | 4.0 | 6.8 | 3.0 | 5.4 | 0.0 | 3.0 | 1.6 |
| p9 | 3.5 | 3.9 | 1.7 | 3.4 | 4.3 | 1.1 | 2.9 | 3.0 | 0.0 | 4.5 |
| p10 | 1.2 | 0.9 | 5.6 | 4.4 | 8.4 | 4.6 | 7.0 | 1.6 | 4.5 | 0.0 |

Table 3: A Euclidean distance matrix.

# Question 4

## a

A distance matrix is given in Table 3

## b & c

In constructing these trees, it seemed that the minimal distance pairs were much easier to construct from the distance matrix. That said, the actual tree was less expressive. As you can see at the points starred in Figure 6, there were several 'collisions', where the minimum distances were identical for more than one pair of clusters. This makes the tree less expressive than the maximum distance tree. On the other hand, the greatest distance tree seemed to penalize much more for outliers. It seems prudent to utilize centroids in many cases to mitigate these shortcomings.

## d & e

Figure 7 shows k-means clustering schemes for two different initial points. As you can see, both sets of initial cluster representatives yield the same clustering once equilibrium is reached. The k-means algorithm is robust to poor choices of cluster representatives, but picking sensible initial points could significantly reduce run times.

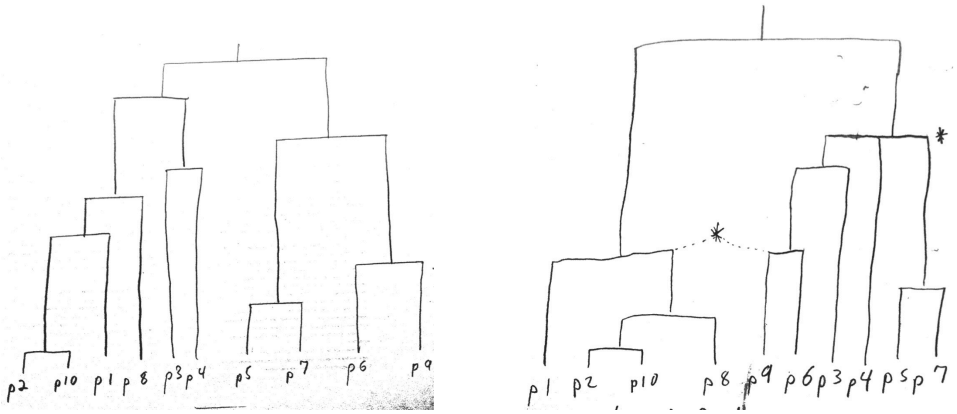


Figure 6: Maximum (right) and minimum (left) distance hierarchical agglomerative clusters.

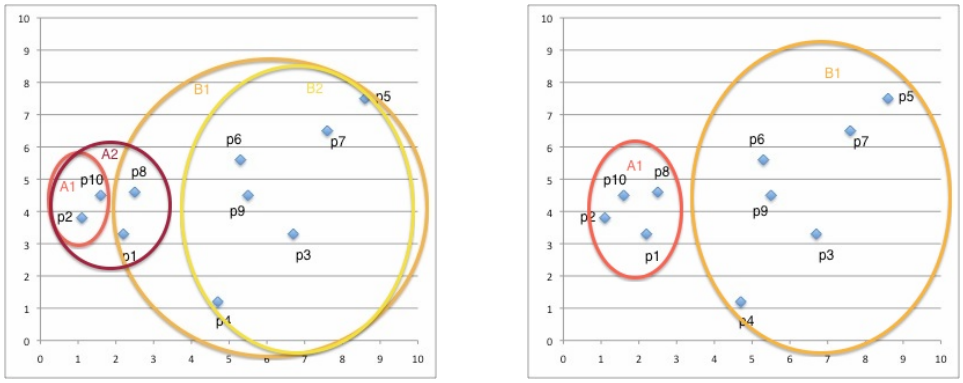


Figure 7: k-means clustering using p1 and p2 (left) and p9 and and p10 (right) as initial cluster representatives. Both sets of parameters converge into identical clusters A and B.

# Question 5

## 1

The substrings were taken from crYsubset.fa. I generated random 'slices' of this text of length 100 characters 500 times to generate the data below.

## 2

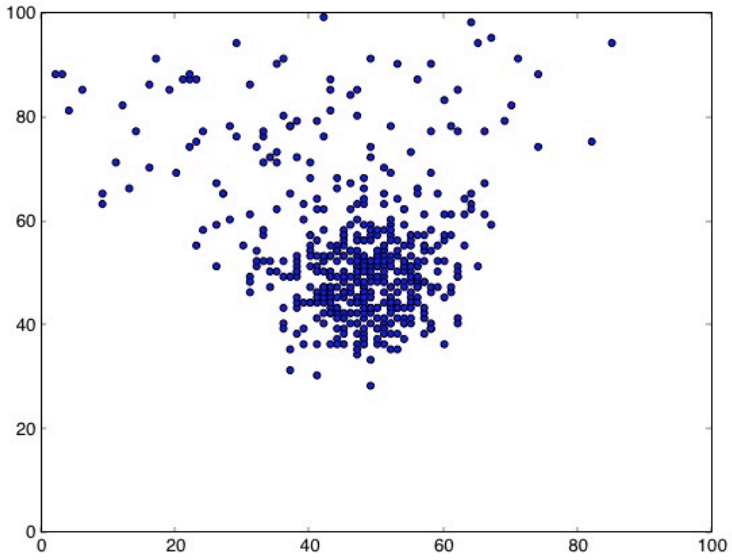


Figure 8: A scatter plot of runs in random 100 character subsequences, untransformed (x axis) and BW Transformed (y axis).

## 3

The scatter plot has a roughly normal distribution across the x axis (the untransformed sequence runs), centered around 50. This is to be expected, as runs of characters should roughly be a function of the length of the string. But the Y axis, the Burrows -Wheeler transformed string runs, has a distribution skewed upwards. So we know that this pattern is is very likely to generate large amounts of repeated characters, even in strings

that don't have many to begin with. This is most noticeable in the upper left hand portion of the graph. For these data points, the original string had close to no runs, while the BWT string had almost all runs. Additionally, the algorithm almost never decreased the number of runs in a string, as evidenced by the fact that almost all of the results lie above the identity line ($y = x$).

It is clear from these results that the BWT algorithm almost always increases runs. In other words, in the graph, Why might this be?

The Burrows-Wheeler Transform involves an alphabetization step in it. If the BWT string were taken from the first column of this alphabetized array, practically every string would be full of runs. Even though the algorithm takes the last column of these rotated strings, the last-first principle indicates that the number of runs will be much greater than otherwise would be the case. This is, of course, one of the strengths of the algorithm with regards to lossless compression.

# Question 6: Programming exercise

See separate submission.