Ayaana Patel Sikora

Bi 181: Problem Set 1

1) The time complexity of the Random Sort algorithm is O(n*n!). This is because, in the worst case, every possible ordering of the list must be checked until the correctly sorted one is found. This would be infinite run time for n = 10,000 in the worst possible case.

2) This sorting algorithm is the Bubble sort algorithm, which runs in O(n^2) time. This is because it needs to compare every number to every other number, which is n*(n-1).
A sorting algorithm that runs in O(nlogn) time is the Mergesort algorithm. The quicksort algorithm sorts when merging split arrays.

   The code for the merge sort algorithm is attached in the file 'mergeSort.py'

3)
```
samtools sort –T /tmp/P1Q3.sorted –o P1Q3.sorted.bam P1Q3.bam
samtools index P1Q3.sorted.bam
samtools view P1Q3.sorted.bam chr17:220-300
```

```
chr17_45_242_0:0:0_1:0:0_50f5f     147    chr17   193    60     50M    =      45
        -198    TCCAGCTTAACCTGCATCCCTAGAAGGGAAGGCACCGCCCAAAGACACGC
        22222222222222222222222222222222222222222222222222       XT:A:U NM:i:1
        SM:i:37         AM:i:37         X0:i:1  X1:i:0  XM:i:1  XO:i:0  XG:i:0  MD:Z:26T23
chr17_204_401_0:0:0_0:0:0_43ea1    99     chr17   204    60     50M    =      352
        198     CTGCATCCCTAGAAGTGAAGGCACCGCCCAAAGACACGCCCATGTCCAGC
        22222222222222222222222222222222222222222222222222       XT:A:U NM:i:0
        SM:i:23         AM:i:23         X0:i:1  X1:i:1  XM:i:0  XO:i:0  XG:i:0  MD:Z:50
chr17_224_415_1:0:0_1:0:0_a2d53    163    chr17   224    60     50M    =      366
        192     GCACCGCCCAAAGACACGCCCATGTCCAGCTTATTCTCCCCAGTTCCTCT
        22222222222222222222222222222222222222222222222222       XT:A:U NM:i:1
        SM:i:37         AM:i:37         X0:i:1  X1:i:0  XM:i:1  XO:i:0  XG:i:0  MD:Z:37G12
chr17_123_290_3:0:0_1:0:0_27b3b    83     chr17   241    60     50M    =      123
        -168    GCCCATGTCCAGCTTATTCTGCCCAGTTCCTCTCCAGATAGGCTGCATGG
        22222222222222222222222222222222222222222222222222       XT:A:U NM:i:1
        SM:i:37         AM:i:25         X0:i:1  X1:i:0  XM:i:1  XO:i:0  XG:i:0  MD:Z:38A11
```

4) Given the sequences:
    a. Expected Frequency = (¼)^4. Each k-mer has a (¼)^4 because each is composed of 4 bases, each which has a probability of ¼.

       The expected and observed frequency of each possible 4-mer is given below.

4-mer	Observed	Expected
AGTC	0.02	0.00390625
GTCG	0.02	0.00390625
TCGT	0.04	0.00390625
CGTA	0.02	0.00390625
GTAC	0.04	0.00390625
TACG	0.06	0.00390625
ACGT	0.1	0.00390625
CGTG	0.08	0.00390625
GTGA	0.08	0.00390625
TGAC	0.04	0.00390625
AGTA	0.04	0.00390625
GTAG	0.02	0.00390625
TAGA	0.02	0.00390625
AGAC	0.02	0.00390625
GACG	0.04	0.00390625
GTGC	0.02	0.00390625
TGCC	0.02	0.00390625
GCCG	0.02	0.00390625
TGAG	0.02	0.00390625
GAGA	0.02	0.00390625
AGAT	0.02	0.00390625
GATA	0.02	0.00390625
ATAC	0.02	0.00390625
GAAC	0.02	0.00390625
AACG	0.02	0.00390625
ACGG	0.04	0.00390625
CGGA	0.02	0.00390625
GGAG	0.02	0.00390625
GAGT	0.02	0.00390625
CGGT	0.02	0.00390625
GGTG	0.02	0.00390625
TGAT	0.02	0.00390625

+ 224 other possible 4-mers with an observed frequency of 0.0 and and expected frequency of 0.00390625 (which is $(¼)^4$).

b. The frequency of all possible 4-mers is $(¼)^4 * 50$. This is because there are 50 possible k-mers in the sequences (10 per sequence). Since we saw previously that each k-mer has a $(¼)^4$ probability, the frequency is this * 50.

c. See the below table for the number of observed occurrences and the expected number of occurrences:

4-mer	Observed	Expected
AGTC	1	0.1953125
GTCG	1	0.1953125
TCGT	2	0.1953125
CGTA	1	0.1953125
GTAC	2	0.1953125
TACG	3	0.1953125
ACGT	5	0.1953125
CGTG	4	0.1953125
GTGA	4	0.1953125
TGCC	1	0.1953125
GCCG	1	0.1953125
TGAG	1	0.1953125
GAGA	1	0.1953125
AGAT	1	0.1953125
GATA	1	0.1953125
ATAC	1	0.1953125
GAAC	1	0.1953125
AACG	1	0.1953125

TGAC	2	0.1953125	ACGG	2	0.1953125
AGTA	2	0.1953125	CGGA	1	0.1953125
GTAG	1	0.1953125	GGAG	1	0.1953125
TAGA	1	0.1953125	GAGT	1	0.1953125
AGAC	1	0.1953125	CGGT	1	0.1953125
GACG	2	0.1953125	GGTG	1	0.1953125
GTGC	1	0.1953125	TGAT	1	0.1953125

This table does not include the 224 other possible k-mers whose observed number is 0 and whose expected number is 0.1953125 (which is $(¼)^4$ * 50).

The 4-mers more common than expected are the ones whose observed number is greater than 2. These are TACG (observed 3), ACGT (observed 5), CGTG (observed 4), and GTGA (observed 4).

d. AGTCGTA**C*GTG***_A_C
AGTAGA**CGTG**CCG
A**C*GTG***_A_GATACGT
GAACGGAGTACGT
T**C*GTG***_A_CG*GTGA*T

TACG: Highlighted
ACGT: Underlined
CGTG: Bolded
GTGA: italicized

5) Hamming Distance:
   a. The Hamming Distance code can be found in the attached file entitled ‘hammingDist.py’

   b. TTGTAGG: 40
   GAGGACC: 43
   TATACGG: 44
   CCGCAGG: 28
   CAGCAGG: 11

   c. The pattern most likely to be the implanted motif is CAGCAGG. Since we implemented the minimum hamming distance algorithm, the most likely motif will have the smallest Hamming distance.

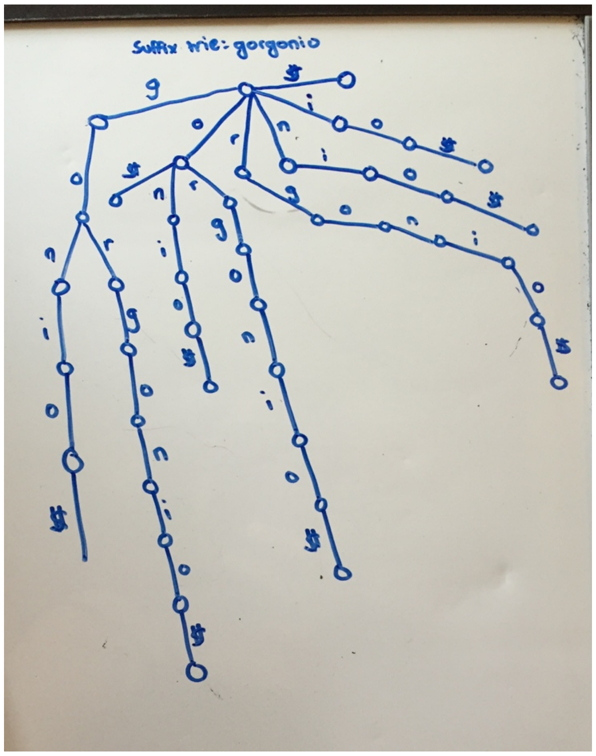
d. The entropy of the discovered motif is 0.607381590645. The code detailing how this was computed is also in the hammingDist.py

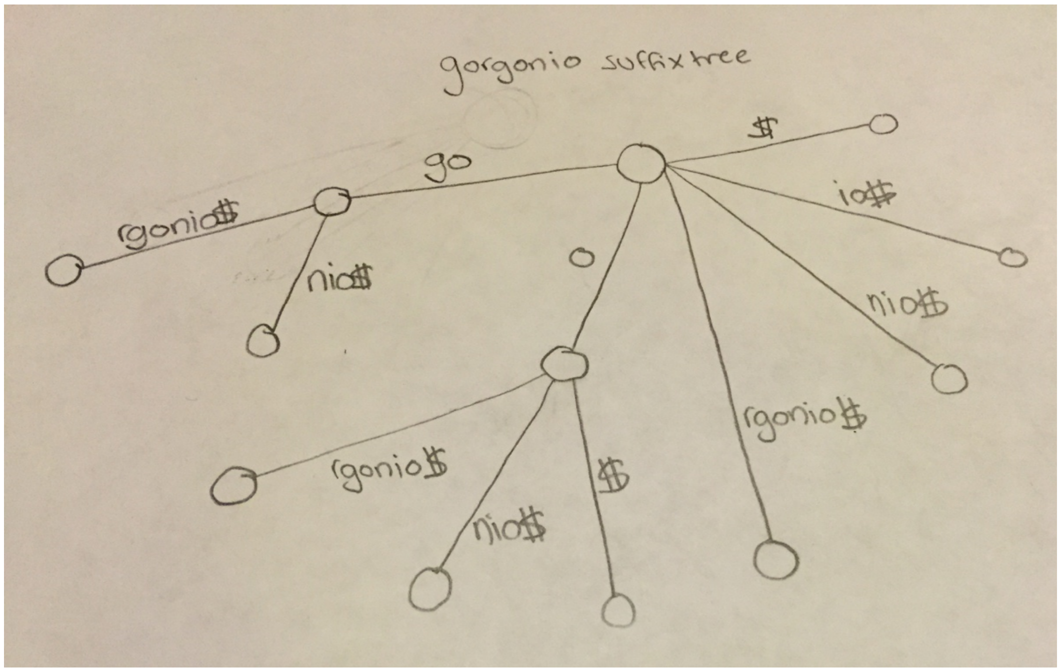
6) The probability that we will select the correct motif is $\frac{n*k}{x}$, where x is the number of base pairs, k is the length of the k-mer and n is the number of sequences. We would need an average of $1/(\frac{n*k}{x}) = \frac{x}{n*k}$.

7) The code is in the attached file entitled 'randomizedFinal.py'.
Running the algorithm once generates the following motif: CACTTGC. Every time the algorithm is run just once, a different motif (a random motif) is generated.
The most common motif gotten form running the algorithm 10 times is CAGCAGG. Sometimes other motifs are returned, but this is the most common one.
Running the algorithm 1000 times generates the following motif: CAGCAGG. Every time the algorithm is run 1000 times, the same motif is generated.

8) The code can be found in the attached file entitled 'gibbs.py'. Yes, the same motif (CAGCAGG) was generated. The code I decided on implements the Gibbs Sampling 150 times. After this many times, despite the excluded sequence, the same motif is decided on because it has the best score. This is why the same motif is arrived at.

9)





10) For parts a) and b) see attached file entitled 'suffixArray.py'
    a. One instance of 'atattaacaaagccaaaagtttcaaacttt' was found in the file. It was found at index 60964.
    b. 12 instances of 'aaaattat' was found in the file. It was found at the following indexes: [32966, 35291, 35797, 40130, 61342, 75613, 76368, 83602, 87732, 90817, 94126, 97925].

11) The entropy metric, because it measures level of uncertainty, tells us so much more about a motif. A simple difference score simply reports on how different the subsequence that we are considering is from the main sequence. Meanwhile, the entropy metric uses gathered data from previous comparisons to report on how certain it is that this subsequence is a motif. Consider the following example: (each numeric column represents the score for 1 subsequence).

A	.2	0
C	.6	.6
G	0	0
T	.2	.4

By the simple difference score, both subsequences are equally likely to be the motif since they have the same score. However, by the entropy metric, the first column has a score of 1.371 while the second has a score of 0.971. This means that it is more likely (certain) for the second subsequence to be the motif than the first (since a higher entropy metric score indicates more uncertainty). Thus, the entropy metric is more reliable than the simple difference score. This example was from the Lecture Slides (Lecture 2, Slide 26).