Homework 3 Answers

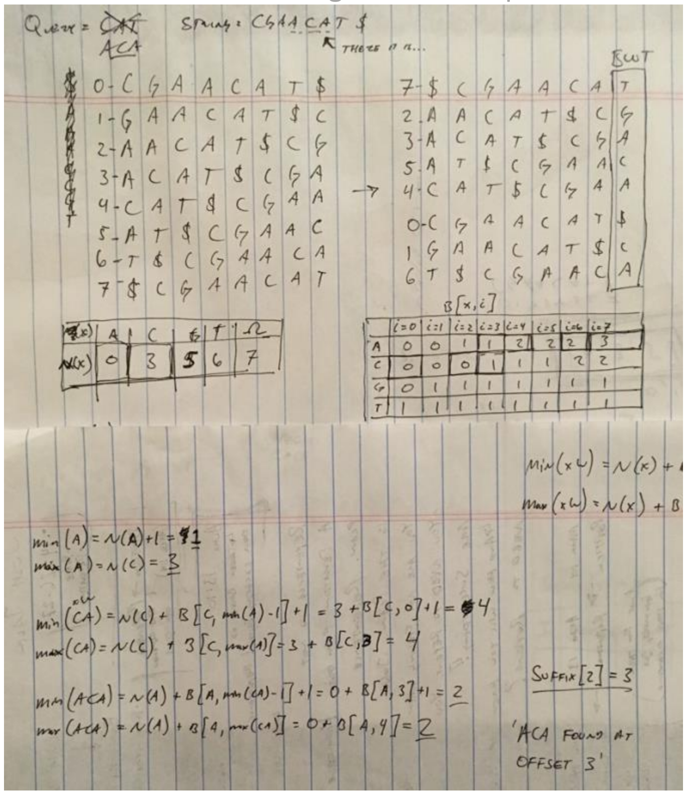
Download the attached files (database.txt, query.txt). Submit all code files, along with a word document with the answers to each question, including pictures if applicable. For question 1-5, you can use any packages you want. Part 6 should not rely on any pre-packaged programs. **Reminder: This homework is due Sunday night, before midnight.**

Part 1-5 will help demonstrate the utility of having gapped-pmers in comparison to simple k-mers. The database.txt file contains over 1,000,000 bp of genomic data. The second file, query.txt, contains the 138 bp pattern you are trying to locate.

1) Search the database for the substring consisting of the first 18 basepairs of your query string. (You can use your code from Homework 1 to do this, but as the purpose is not to redo homework 1, you may use any functions/packages, including regex). What are the offsets of this substring in the database? `locations for simple 18-mer: [`**`6048`**`, 331992, 845184, 1733616, 2043828, 2508588, 2597004, 3016344, 3055422, 4390632, 5210172, 5567712, 5754498, 6147888, 6201048, 6313488, `**`6602046`**`]`
2) Construct one gapped p-mer of your query string with the following specifications: p = 10, and d = 14 (positions searched = 10, total span = 136). Find all locations of this gapped p-mer in the database. (Again, any method for search is fine). What are the location offsets for your gapped p-mer? `locations for gapped p-mer: [`**`6048`**`, 78632, 1177787, 2634037, 3063146, 3452018, 3579681, 4630223, 4632050, 4843583, 5634644, 6049073, `**`6602046`**`]`
3) You can use gapped p-mers of different sizes to use a multiple filtration method. Find the location offsets in the database string of one gapped p-mer where p=10 and d=13 (positions searched = 10, span = 127). `locations for gapped p-mer (#2): [`**`6048`**`, 53985, 60527, 410697, 838073, 996776, 1071597, 1335996, 1761709, 1927836, 2332665, 4247745, 6417651, 6462985, 6493439, `**`6602046`**`]`

4) Using the offsets returned in 1-3, give every possible offset of your query pattern. 6048, 6602046
5) Normally, you would have indexed the genome in advance, from which you would have known the offsets of the substrings you found above (along with all other substrings given by your indexing method). What would you do next to locate the full query string? How does the gapped p-mer indexing approach change the efficiency of search? Why was it more or less efficient in this case? Starting from my two possible positions, I would search for the rest of the query string, the BM algorithm would work well here. The gapped p-mer index returned fewer locations with almost half the positions searched, which is a big increase in efficiency. When combined with multiple filtration, we reduced the possible sites to two. This means, when building an index, we would have made 4^10 * 2 indices for 2 possibilities, vs 4^18 indices for 17 possibilities). This is all because of the highly repeated nature of this DNA sequence (if you preview the database.txt, you'll notice strings of the motif 'TGAACA' repeated many, many times). This motif also appears in our query string. If the database and query were completely random, we would not see such a dramatic increase (although the multiple filtration would still help a lot)

6) Choose one of the following problems to complete, using the Burrows-Wheeler Transform.
   a) Using the BWT of the string "CGAACAT$", find the substring ACA using the method described in the readings (Note, you will need to construct the BWT and the two look-up tables) Show your work on paper.
   b) Implement an algorithm to search for any substring using the BWT of the string "CGAACAT$". You may hardcode the N(x) and B[x,i] tables, as well as the suffix array. Output the position of substring ACA, along with your tables and suffix array.
   c) Implement the full BWT search algorithm: given any string, compute the BWT, calculate the suffix array, the N(x) and B[x,i] tables, and find the location of any substring. Show your outputs (please print your arrays also) for the string "CGAACAT$"" and the substring ACA. (Hint, you should just use the O(n^2) algorithm of constructing the circular permutation table discussed in class.)


Query = ~~CAT~~ ACA   String: CGAACAT $
THERE IT IS...

```
0- C G A A C A T $          7- $ C G A A C A | T
1- G A A C A T $ C          2. A A C A T $ C | G
2- A A C A T $ C G          3- A C A T $ C G | A
3- A C A T $ C G A          5. A T $ C G A A | C
4- C A T $ C G A A    →     4- C A T $ C G A | A
5- A T $ C G A A C          0- C G A A C A T | $
6- T $ C G A A C A          1  G A A C A T $ | C
7- $ C G A A C A T          6  T $ C G A A C | A
```
BWT

x	A	C	G	T	Ω
N(x)	0	3	5	6	7

B[x,i]

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
A	0	0	1	1	2	2	2	3
C	0	0	0	1	1	1	2	2
G	0	1	1	1	1	1	1	1
T	1	1	1	1	1	1	1	1

$\min(x\omega) = N(x) +$

$\max(x\omega) = N(x) + B$

$\min(A) = N(A) + 1 = 1$
$\max(A) = N(C) = 3$

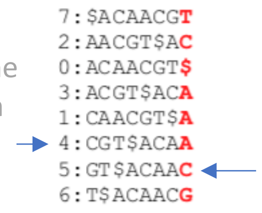
$\min(CA) = N(C) + B[C, \min(A)-1] + 1 = 3 + B[C,0] + 1 = 4$
$\max(CA) = N(C) + B[C, \max(A)] = 3 + B[C,3] = 4$

$\min(ACA) = N(A) + B[A, \min(CA)-1] + 1 = 0 + B[A,3] + 1 = 2$
$\max(ACA) = N(A) + B[A, \max(CA)] = 0 + B[A,4] = 2$

Suffix[2] = 3

'ACA FOUND AT OFFSET 3'

```
Code output:
Database String: CGAACAT
Search String: ACA
BWT: TGACA$CA
Suffix Array: [7, 2, 3, 5, 4, 0, 1, 6]
Sorted String: $AAACCGT
N_table: [0, 3, 5, 6, 7]
B_table:
 [[0, 0, 1, 1, 2, 2, 2, 3],
  [0, 0, 0, 1, 1, 1, 2, 2],
  [0, 1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1, 1]]
Min/Max: 2 2
String found at Offset: 3
```



```
7:$ACAACGT
2:AACGT$AC
0:ACAACGT$
3:ACGT$ACA
1:CAACGT$A
4:CGT$ACAA
5:GT$ACAAC
6:T$ACAACG
```

Extra question to consider, you may choose not to respond, but humor us and at least think about it: As you may have noticed, the BWT search relies on the fact that all same letters (e.g., C's) in the last column appear in the same order as they do in first column (in other words, the 2nd C in the last column is also the 2nd C in the first column, although their positions may change with respect to A's, G's, and T's.)
Why does this property hold? In other words, why don't the letters get out of order given how many times they are rearranged?
Maybe this was obvious to everyone, but it sort of blew my mind when I realized this. Because the letters in the first column are sorted in alphabetical order, and because the table is a circular permutation (meaning you could think of the first column as following the last column), letters are guaranteed to fall in the same order (only with respect to themselves).

So looking at just the C's:
A-------C
…
G-------C

But the table loops around:
A-------CA-------C
G-------CG-------C

And that central pair is what you see in the first column
A-------**CA**-------C
G-------**CG**-------C

CA--------
CG--------

So because the strings are sorted alphabetically, CA will obviously come before CG, and A------C will obviously come before G-----C. Which means that for any letter, all letters of the same type appear in the same order in the first column and the last column (note that this is only true within, not across letters. And it also isn't true for every column, only for the first and last, which is why the BWT can be used in this way.)