Task 1: BuildSA

What did you find to be the most challenging part of implementing the buildsa *program?*

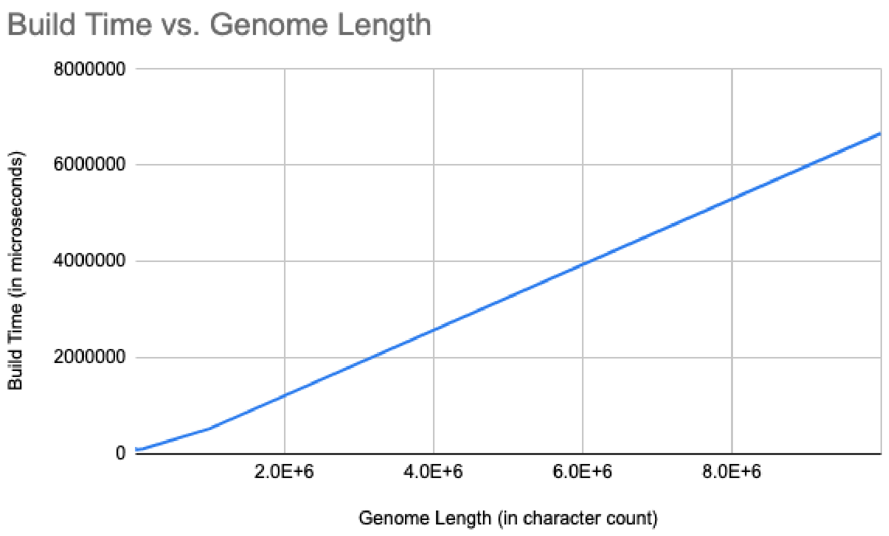
The most challenging part of *buildsa* was conceptually understanding how the prefix table worked as a speed-up. I was not aware that the preftable was distinct from the simple accelerant algorithm. Additionally, I was unsure at first how two binary searches could be used to find the range of suffix array indexes associated with a k-mer. It was only after I consulted several online sources (some of which seem to be former classes taught by Dr. Patro himself) did I realize that the two binary searches were bisect_left and bisect_right, not the classic binary search algorithm.

For references of various size:

*** View the data here ***

- *How long does building the suffix array take?*

  For this benchmark, we took references of length 100; 1,000; 10,000; 100,000; 1,000,000; and 10,000,000. We ran the *buildsa* program on these references without prefix table creation. We measured build time in microseconds and took special care not to include I/O in the timings.

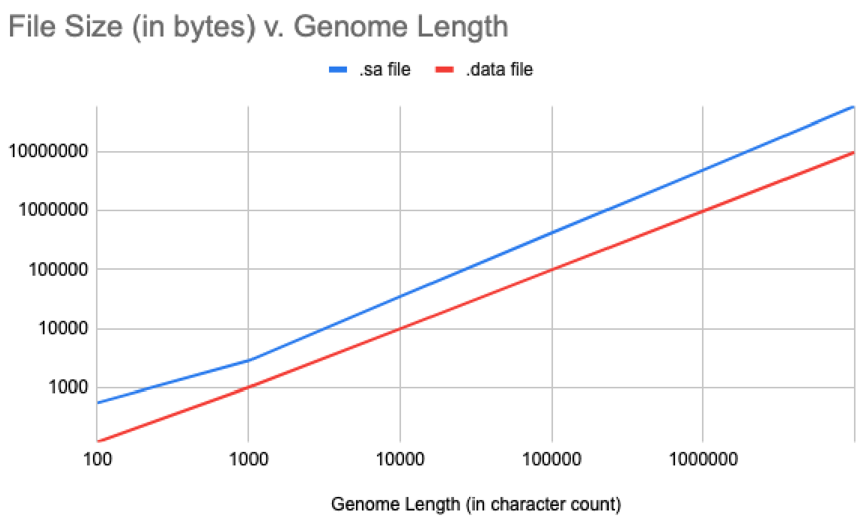


As the reference length increased, build time increased linearly. Interestingly, build times at very small reference lengths were longer than build times for moderate genome lengths. Specifically, the 100-length reference took 146 milliseconds; the 1,000-length reference took .103 millisecond; the 10,000-length reference took .081 milliseconds. Only after 10,000-length did the build time truly scale linearly with reference length.

We attribute the low length behavior to overhead associated with initializing data structures and IO.

- *How large is the resulting serialized file?*

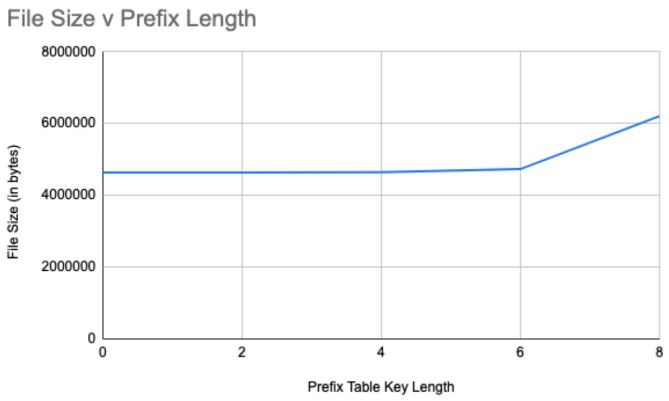
  For this benchmark, we took references of length 100; 1,000; 10,000; 100,000; 1,000,000; and 10,000,000. We ran the *buildsa* program on these references without prefix table creation. We measured build time in microseconds and took special care not to include I/O in the timings.

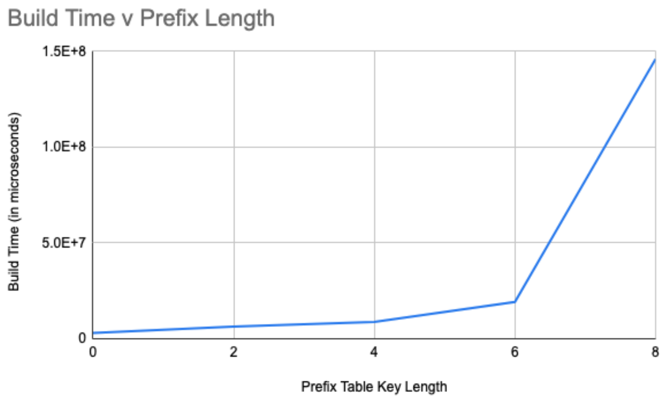


  As the reference length increased, file size in both the .sa (suffix array) binary file and the .data (genome string) binary file increased linearly.

1. *For the times and sizes above, how do they change if you use the --preftab option with some different values of k?*

   For this benchmark, we took the full genome for E. coli. We ran the *buildsa* program on this reference with varying values for the prefix table key length: 0, 2, 4, 6, 8. We measured build time in microseconds and file size in bytes, taking special care not to include I/O in the timings.





As the key length increased linearly, the build time grew exponentially. Meanwhile, the file size only increase marginally. The jump from k=6 to k=8 in file size is likely due to the hashmap reaching some critical threshold for overhead in cereal's BinaryOutputArchive code.

2. *Given the scaling above, how large of a genome do you think you could construct the suffix array for on a machine with 32GB of RAM, why?*

We will assume that the entire *buildsa* program must fit into memory, including the entire reference string. Therefore, we will not create a prefix table in an effort to maximize memory for the genome.

.sa file size scales linearly with the length of the input genome. When modeling the line with linear regression, the binary file scales around 6 bytes for every 1 byte (character) of the genome.

Therefore, given 32GB of RAM we need 7 bytes total per character of the genome (1 byte for the character, 6 for the suffix array). This allows room for 4.57 GB of characters for the genome.

Task 2: QuerySA

What did you find to be the most challenging part of implementing the querysa program?

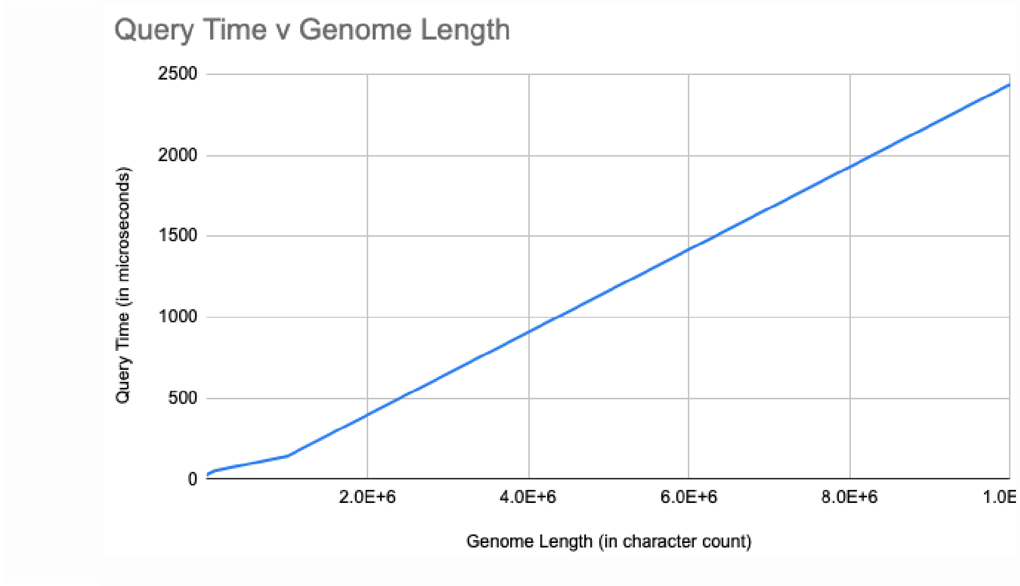
The most challenging part of *querysa* was understanding that the simple accelerant modification was independent from the prefix table optimization. I also did not realize the simple accelerant did not require any LCP data structures like the super accelerant. A final realization that helped me with this task was understanding that I had to return an interval range, just like for the prefix table in task 1.

For references of various size:

*** View the data here ***

1. *How long does querysa take on references of different size?*

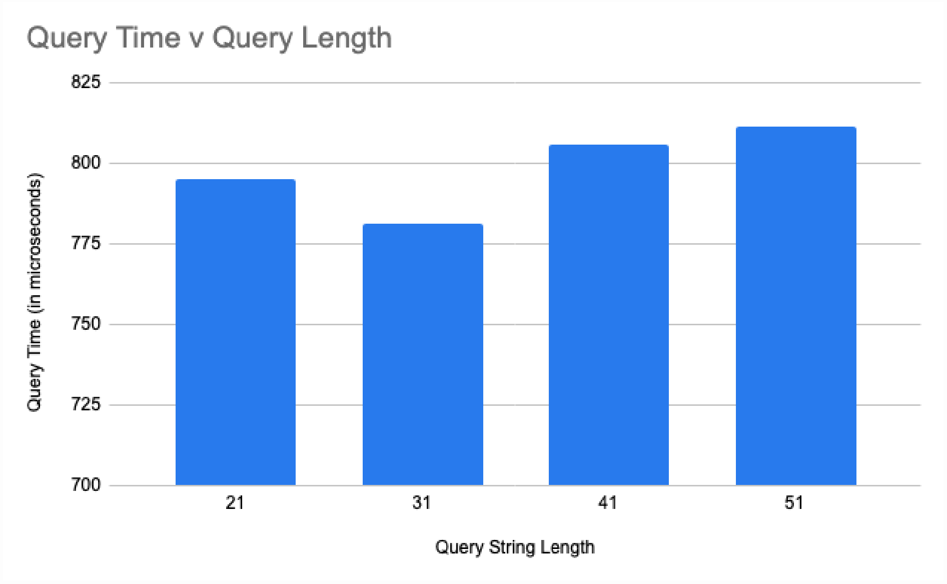
   For this benchmark, we took references of length 100; 1,000; 10,000; 100,000; 1,000,000; and 10,000,000. We fixed all queries to be 31-mers that were guaranteed to exist at least once within the reference. We ran *querysa* in naive mode. Query time was measured in microseconds.



   As genome length increased, query time increased linearly. At smaller reference lengths, this linear scaling was dampened by program overhead.

2. *How long does querysa take on queries of different length?*

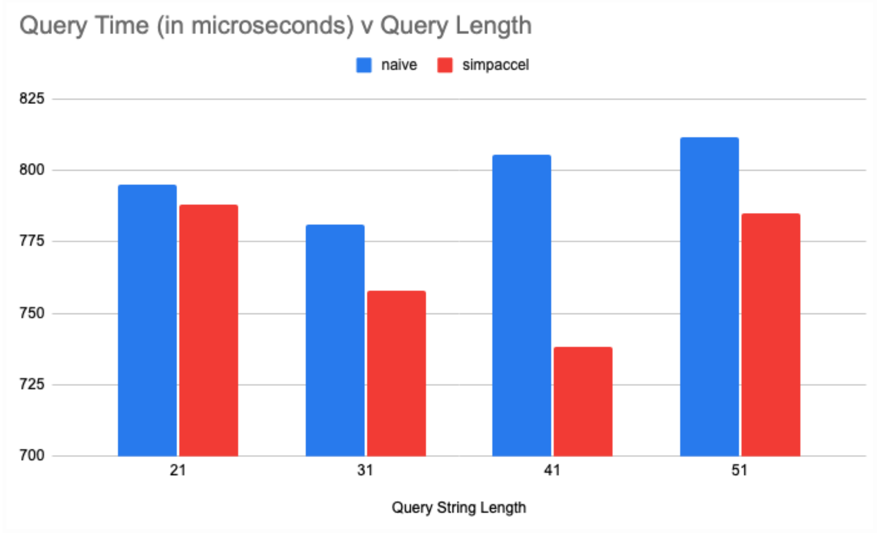
   For this benchmark, we took the full genome for E. coli. We ran the *querysa* program on this reference in naive mode on varying query lengths: 21-mers, 31-mers, 41-mers, and 51-mers. Query time was measured in microseconds.



At time measurements this precise, it is very difficult to see any pattern emerge. Across the board, query string length does not seem to correlate to an increase in query time. This may be due to the fact that comparing 21 characters takes around the same amount of time as comparing 51 characters for the CPU. Perhaps with extremely large query lengths, query time would increase.

3. *How does the speed of the naive lookup algorithm compare to the speed of the simpleaccel lookup algorithm?*

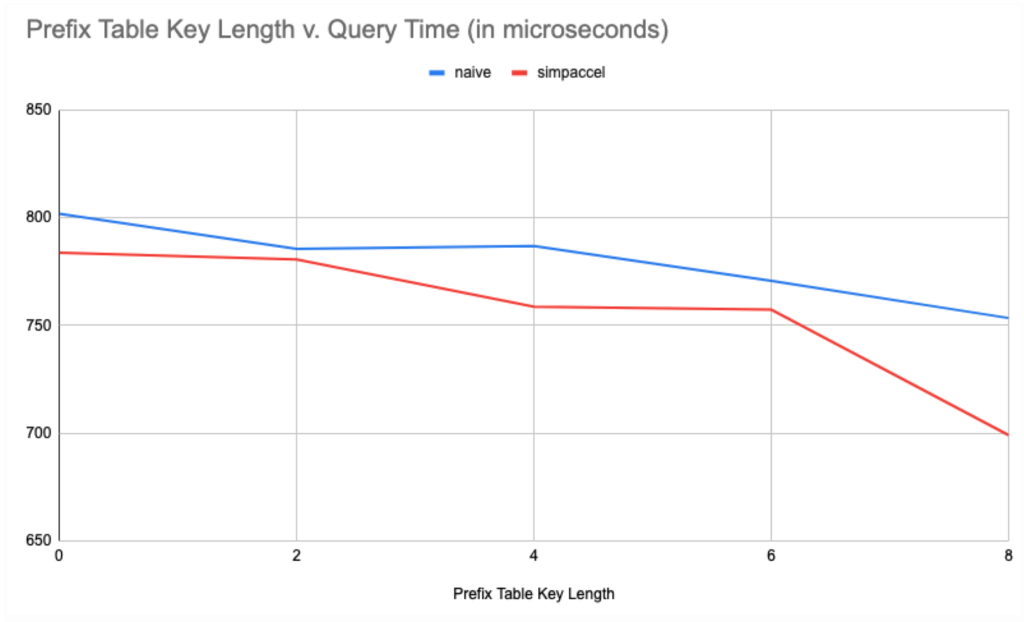
   For this benchmark, we run the same set-up as above but with *querysa* in simpaccel mode. We see that the simple accelerant look-up algorithm performs faster than the naive algorithm across the board.



4. *How does the speed further compare when not using a prefix lookup table versus using a prefix lookup table with different values of k?*

   For this benchmark, we took the full genome for E. coli. We fixed all queries to be 31-mers that were guaranteed to exist at least once within the reference. We ran *querysa* in naive and simpaccel mode with varying prefix table key lengths: 0, 2, 4, 6, and 8. Query time was measured in microseconds.

   We again see that simpaccel performs better than naive mode across the board. As the prefix table key length increases, the query time drops marginally.



5. *Given the scaling above, and the memory requirements of each type of index, what kind of tradeoff do you personally think makes sense in terms of using more memory in exchange for faster search.*

   Creating a prefix look-up table requires only a little bit more memory and allows for a slightly faster query time. The main issue is that long prefix table key lengths cause *buildsa*'s execution time to increase exponentially. In order to gain a marginally faster query speed, the build speed must blow up. Personally, I think that using more memory in exchange for a faster search is not worthwhile when considering total execution times between *buildsa* and *querysa.* If execution time did not need to be taken into consideration, then I believe that it is always worth it to use more memory in exchange for a faster search - especially if the prefix table key length is almost as large as the query string lengths.