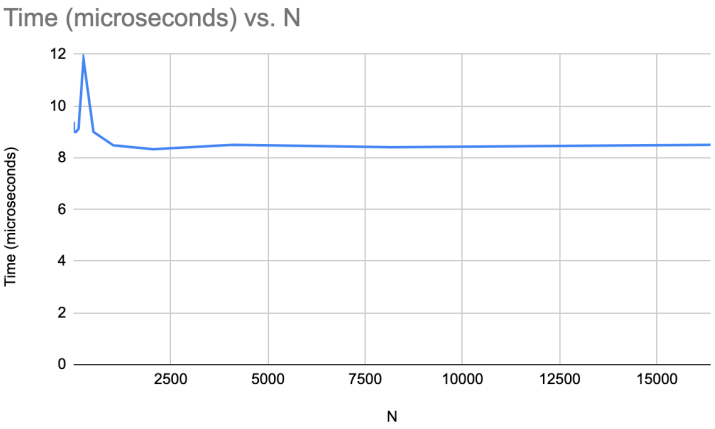
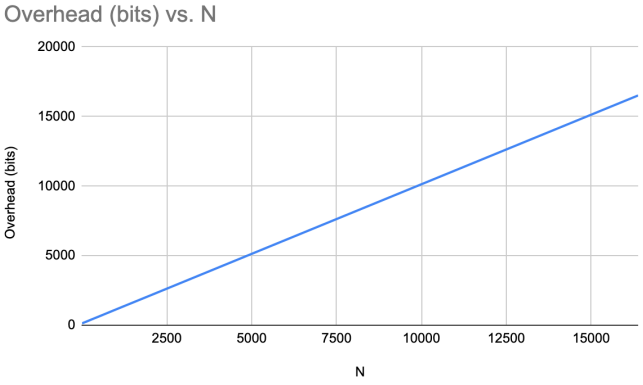
Code: https://github.com/shuai626/cmsc858d_hw1
Data: https://docs.google.com/spreadsheets/d/10LiIzQ0nOauExvcPVQ__GhyuU7fsaLFyFaYN0peYWjQ/edit?usp=sharing

***Writeup**: For this programming task, test your implementation by invoking it for bit vectors of various sizes, and plotting the bit-vector size (say N) versus the time required to do some fixed number of rank operations. Also, plot the bit-vector size (say N) versus the result of calling the `overhead()` function. Does your implementation match the expected theoretical bounds?*

Timings were gathered in experiments by applying 100 rank operations to a randomly initialized sdsl::bit_vector. Experiments were averaged over 100 trials. My implementation matches the expected theoretical bounds of constant loop up time and O(n) bit memory.





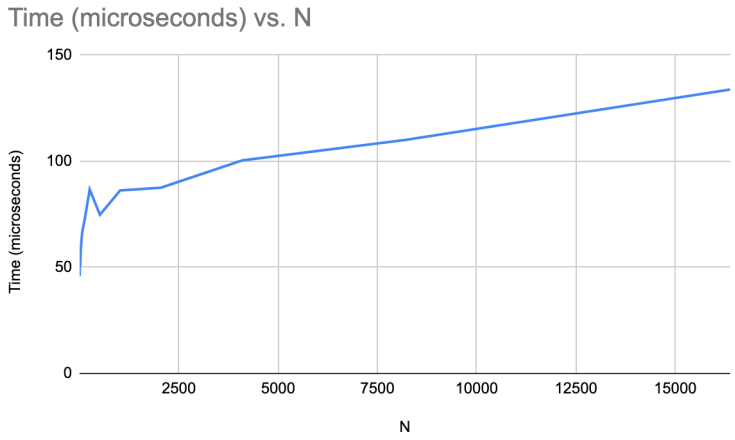
On initialization, my rank_support class iterates over the provided sdsl::bit_vector class. It then allocates space for a superblock array and a block array. Next, the constructor iterates over each bit of the bit_vector and updates each entry of the superblock and block array accordingly. The "Rp" table discussed in class is not created at construction. On execution of rank1(), sdsl::bit_vector::get_int function is used to read a portion of a block (up to a word of bits) in constant time. This block is then passed through std::pop_count to find the total number of bits in constant time. This alternative implementation of rank still guarantees constant time operation.

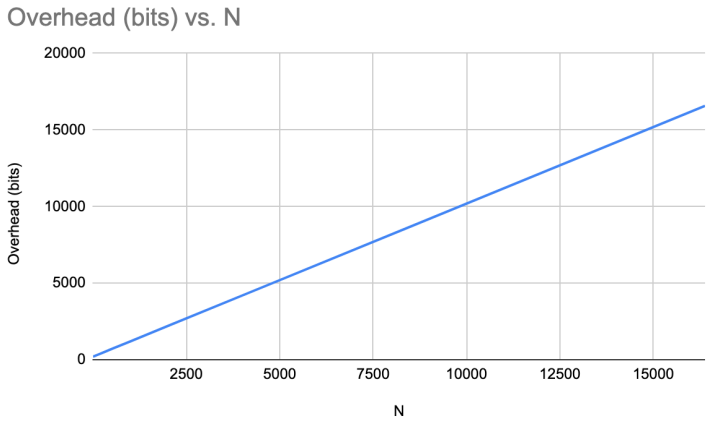
The most difficult part of implementing this task was getting std::pop_count and sdsl::bit_vector to work. I started off using std::bitset, but its functionality was too limited as it does not guarantee a packed data structure. Using bitset prevented rank1() from working in constant time. std::pop_count only works in C++20, which my machine had trouble

updating to at first. Additionally, sdsl was difficult to link to my code as I wasn't aware of the nuances of Makefile at first.

***Writeup**: For this programming task, test your implementation by invoking it for bit vectors of various sizes, and plotting the bit-vector size (say N) versus the time required to do some fixed number of select operations. Also, plot the bit-vector size (say N) versus the result of calling the `overhead()` function. Does your implementation match the expected theoretical bounds? If you feel ambitious, you can additionally implement a constant-time bit-vector select, though this is not required.*

Timings were gathered in experiments by applying 100 select operations to a randomly initialized sdsl::bit_vector passed to a rank_support class. Experiments were averaged over 100 trials. My implementation matches the expected theoretical bounds of O(log(n)) execution time and O(n) bit memory.





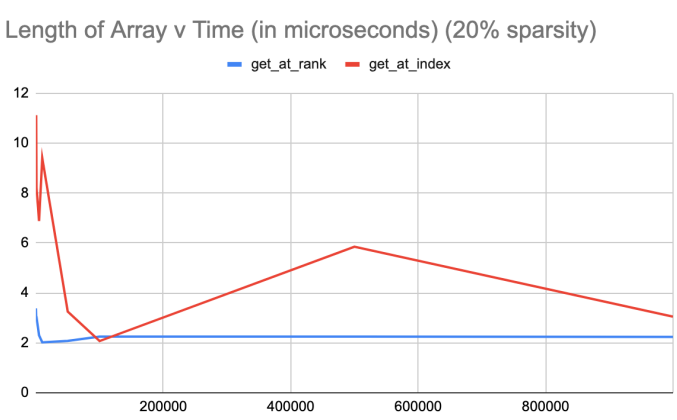
select_support does not perform any additional processing on top of the rank_support class that is provided at construction. When executing select1(), select_support conducts a binary search over the ranks of underlying indices in the bit_vector. Once select1() arrives at its desired rank, it will compare the rank of the current indexes immediate neighbors. If the left neighbor is the same rank, then binary search will continue looking leftward. Otherwise, we have found the smallest index with the provided rank and return the current index.

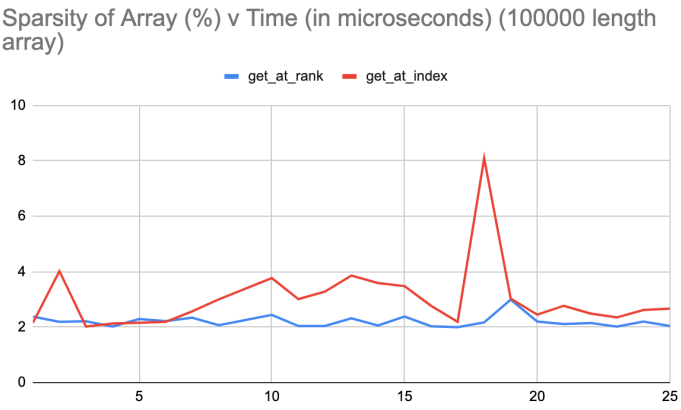
There were two difficult portions of implementing this class. First, the fact that more than one index in the bit vector could have the same rank forced me to think about modifications to the binary search algorithm that would still work. These constraints were not immediately

evident to me and took awhile to debug. Second, the load() function of the class had difficulties interfacing with IO correctly. This also took quite awhile to debug.

***Writeup**: For this programming task, test your implementation by generating sparse arrays of a few different lengths (e.g. 1000, 10000, 100000, 1000000) and having various sparsity (e.g. 1%, 5%, 10%). How does the speed of the different functions vary as a factor of the overall size? How about as a function of the overall sparsity? Finally, try and estimate how the size of your sparse array in memory compares to what the size would be if all of the 0 elements were instead explicitly stored as "empty" elements (e.g. as empty strings). How much space do you save? How do your savings depend on sparsity?*

Timings were gathered in experiments by applying 100 get_at_index or get_at_rank operations to a randomly initialized sparse_array. Both types of operations randomly access values over the possible range of inputs. Experiments were averaged over 100 trials. We fixed sparsity to 20% as we increased the length of the vector. We fixed the length to 1,000,00 as we increased the sparsity % of the vector.





get_at_rank is a constant time algorithm. Meanwhile, the runtime of get_at_index varies probabilistically. As the length of the sparse vector increases, the chance that get_at_index hits an index with a non-zero value decreases. As sparsity percentage increases, there are more elements in the array for get_at_index to hit.

In C++, each string needs at least 3 pointers worth of memory. This equates to 24 bytes == 192 bits per string. Even if strings were stored as the empty string, they would still need at

least 192 bits. Therefore, in a length of N strings with S% sparsity, a regular array would require 192N + <overhead of N array> bits to store. Meanwhile, our sparse_array only requires 192*sparsity*N + <overhead 192*sparsity*N array> of for all the non-empty strings, and N bits to maintain the whole bit vector. Savings depend on sparsity because the lower the value of S%, the less bits are needed to store the entire string list!

On construction, sparse_array creates an empty vector of strings and a bitvector. Appending to the sparse_array updates both vector<string> and bitvector accordingly. When needed, the bitvector is loaded into rank_support and select_support structures to calculate dynamic functions as such num_elems and get_rank_at_index.

The most difficult part about this task was creating an experimental benchmark to perform a grid search of sparsity% versus array length against. It was difficult to come up with a design that was simple to both code, test, and parse data from afterwards.