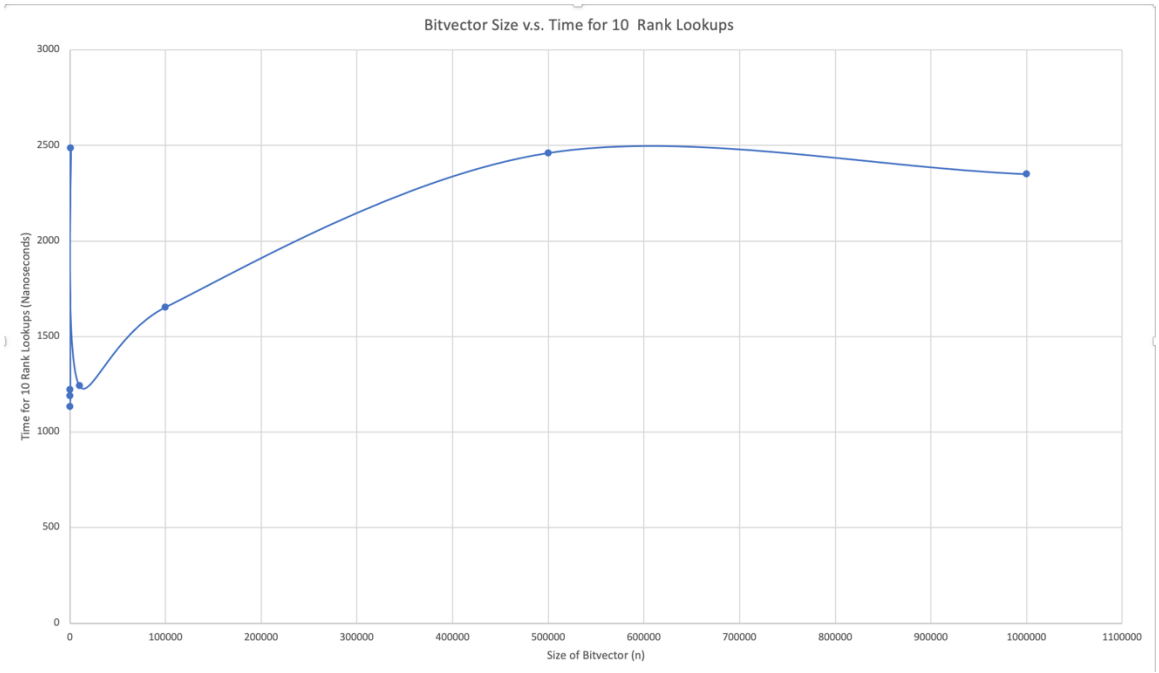
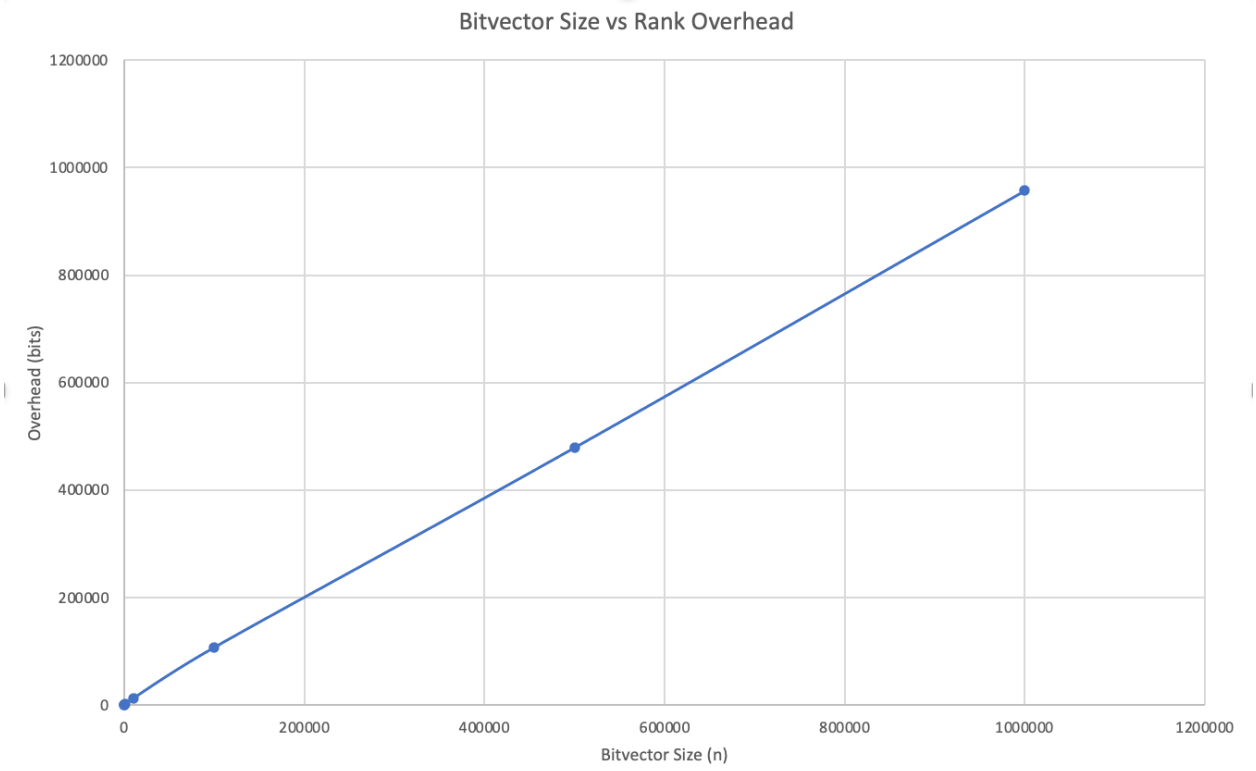
Link to Public Github Repo: https://github.com/stang10/CMSC701_HW2
Task 1:

1) My task, one implementation is based on Jacobson's rank. When a rank support object is initialized, it takes in a bitvector pointer, which it stores as a local variable. At this point a number of calculations are done. First, the size of the bitvector is calculated. Then, in order to create an int_vector object to store the relative cumulative rank of the chunks of a bitvector, calculations are done. Such calculations include the bits needed to represent a number for the value of the relative cumulative rank of each chunk, the length of a chunk, and the number of chunks needed for the expression of this bitvector. Similarly, calculations are done as well to create an int_vector object to store the relative cumulative rank of the subchunks of a bitvector. We need the bits needed to express the value of the relative cumulative rank of each subchunk, the number of subchunks per chunk, and the length of those subchunks. An int_vector is created for the chunks representation with size and bit length as calculated above, and similarly for an int_vector for subchunks. These values are created and stored as local variables. The rank function takes advantage of these vectors, and uses the value stored within each appropriate chunk and subchunk along with population count performed on and the get_int() function for the SDSL int_vectors to determine the rank. Overhead uses the SDSL size_in_bytes() function for the chunks and subchunks int_vectors, adds the two values, multiplies the values by 8 to get bits and returns it. Save prints the number of bits used by a vector, the size of that vector, and then each element of the vector on all separated by new lines for the bitvector, then the chunks vector, followed by the subchunks vector. The load function reads each of these values in and creates new int_vector objects to account for these values for the chunks and subchunks local variables.
2) For task one, the most difficult implementation was the rank1() function. This was due to the fact I had to determine the correct calculations for each chunk and sub chunk that an index would fall into. This also required that I did the correct building of the chunk and subchunk int_vector objects in the initialization of the rank support structure to make sure that I would get approximately O(1) time for determining the rank. Besides this, before I had switched to using the SDSL library from compact::vector objects, the load and save functions were particularly difficult, mostly because compact::vector objects did not allow for resizing a bit length.
3) The image below is a graph of the size of a bitvector versus the time needed to complete 10 rank1() function calls. Though there is some variation across sizes, it appears to me that the run time does not rely heavily on the size of the bitvector as bitvectors of exponentially larger sizes only appear to have runtimes very close to those of much smaller bitvectors. I would say, accounting to variations in my system, that there is an approximate O(1) runtime as we would expect.

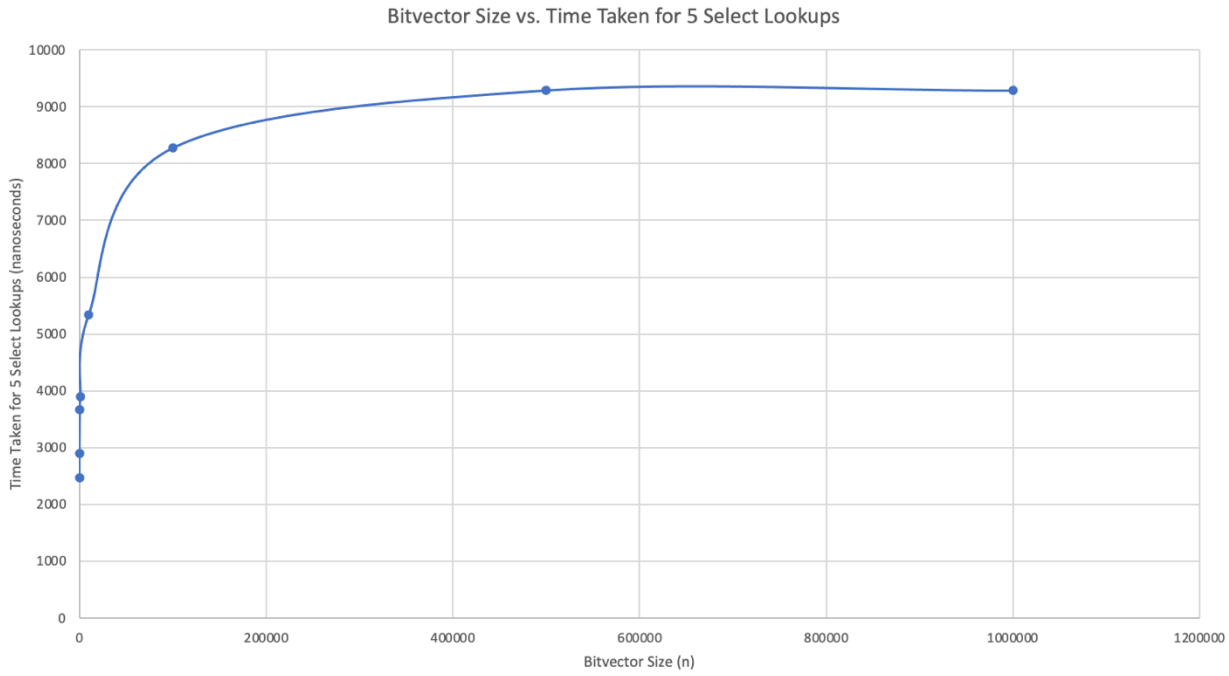


The image below is a graph of the size of a bitvector versus the overhead bits necessary to create a rank_support structure for the bitvector. As you can see, there very clearly appears to be a linear relationship between these two variables. As such, we see this is approximately O(n) space as expected.

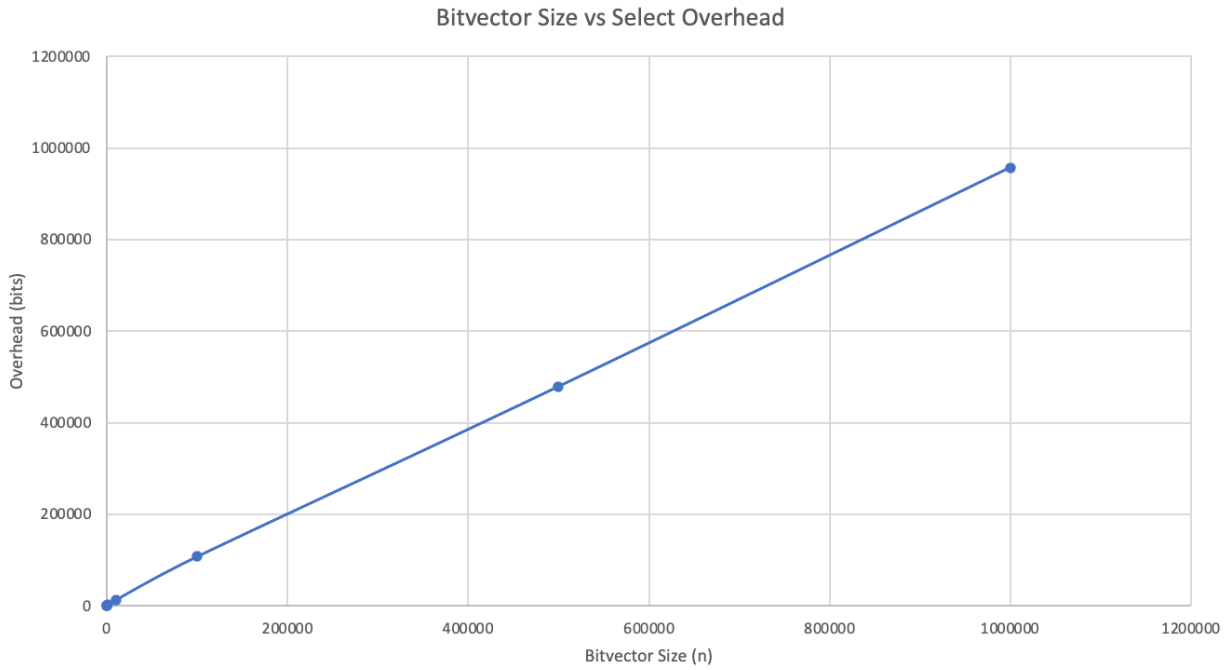


Task 2:

1) My task 2 implementation is heavily based upon the task 1 implementation. When a select support object is initialized, it takes in a pointer to a rank support object which contains the bitvector pointer, the chunks int_vector, and subchunks int_vector that were created previously. Upon initialization just the local variable for the rank support object pointer is set to the parameter passed in. The select1() function takes advantage of binary search, and the rank function. The binary search functionality is specialized so that it will continue to search for further left indices after an index is found that contains the correct rank being searched for. This leftmost value is then returned. The overhead function, since I chose not to implement Clark's select, simply returns the result of calling overhead on the rank support object pointed to by the local variable stored. Save and load simply call the save and load functions on the rank support object pointed to by the local variable stored.
2) For task two, the most difficult implementation was the select1() function. This difficulty lied within the binary search portion. For this part of the select function, I had to determine the leftmost index that carried the correct rank value. As such this required additional thought, unlike the other functions that I could make use of the previous definitions in the rank support class. Again, I still had some difficulties initially when using compact::vector for load and save as explained above. After switching to SDSL those became much easier to handle.
3) The image below is a graph of the size of a bitvector versus the time needed to complete 5 select1() function calls. As you can see, there very clearly appears to be a logarithmic relationship between these two variables. As such, we see this runs in approximately O(log(n)) time as expected of the non-Clark select implementation.



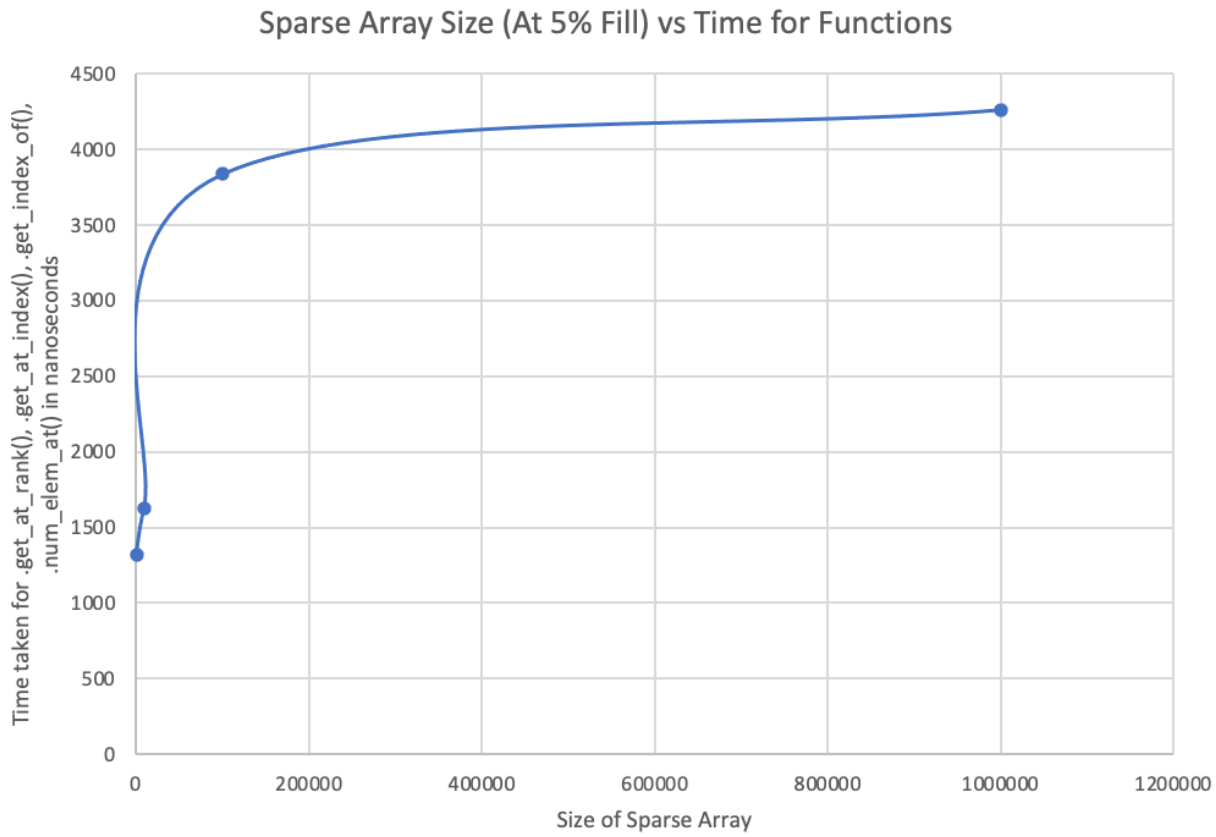
The image below is a graph of the size of a bitvector versus the overhead bits necessary to create a select_support structure for the bitvector which is simply the overhead bits necessary to create a rank_support structure for the bitvector. As you can see, there very clearly appears to be a linear relationship between these two variables. As such, we see this is approximately O(n) space as expected.



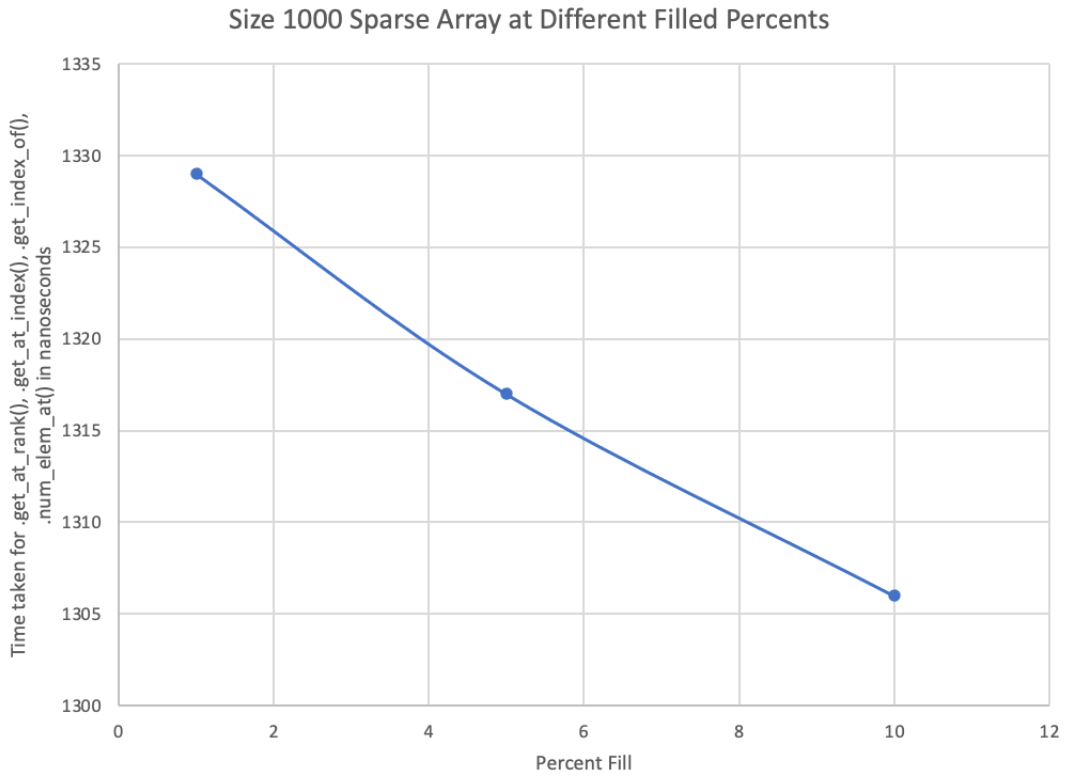
Task 3:

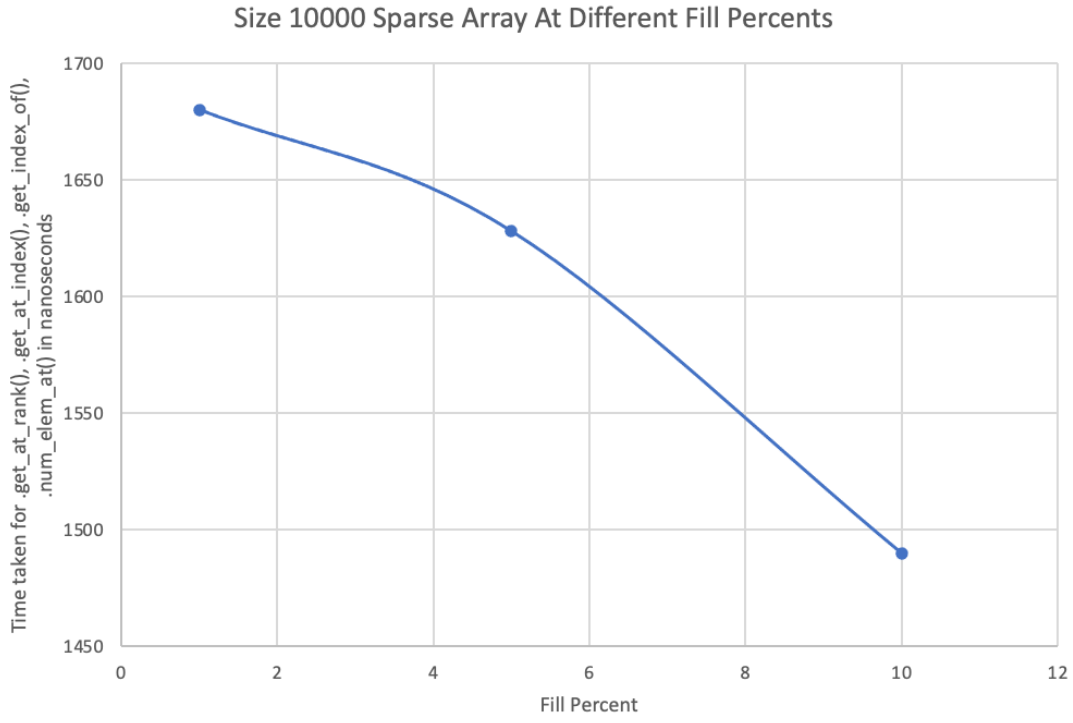
1) Within the sparse array class, there are a few local variables needed. There is a bitvector and a standard vector of strings created to hold the indices of the non-empty values for the sparse array and the values themselves, respectively. Additionally, a rank support and a select support structure are needed as well. There is also a Boolean value to mark whether a sparse array is finalized or not. Initialization of an object in this class establishes a dummy bitvector to be replaced later from the create() function. The rank support and select support structures are initialized using this dummy bitvector, again with the intention to be changed later. Also, the finalized variable is set to false. The create() function reinitializes the local bitvector variable to a bitvector object of the specified size. The append() function adds the specified string to the end of the strings vector and sets the specified index in the bitvector to 1. Each of get_at_rank(), get_at_index(), get_index_of(), and num_elem_at() leverage the results of the rank1() and select1() functions along with simple calculations to return the desired results. The size() function returns the size of the bitvector and the num_elem() function returns the size of the strings vector. The save and load functions simply run the select_support save and load functions (which run those of the rank_support class as specified above) with the addition of the value of the finalized Boolean value printed on a newline at the end of the file and read in at the end of the load function.

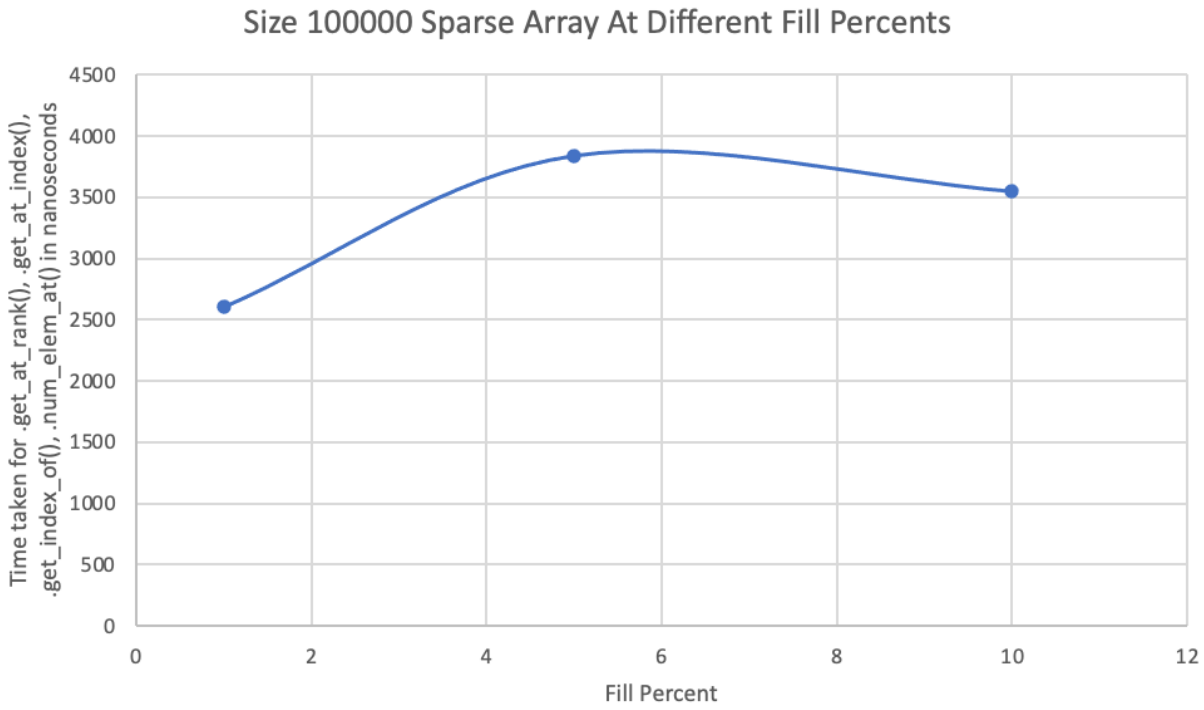
2) For task three, the implementations in my opinion were much easier to handle due to the completeness of tasks one and two. I did not struggle too much beyond dealing with the issue of inclusive versus exclusive rank values. Overall, there was not one particular function that I struggled with, more so, I needed extra effort to determine which rank I should be using, which index I should be using, etc. Again, due to the fact that the descriptions for the functions were written using inclusive rank and our tasks one and two were done with exclusive rank.
3) The image below is a graph of the size of a sparse array (that is 5% filled) versus the time needed to complete one call to each of .get_at_rank(), .get_at_index(), .get_index_of(), and .num_elem_at(). There appears to be an approximately logarithmic relationship between the size of the sparse array and the time needed to complete these calls.

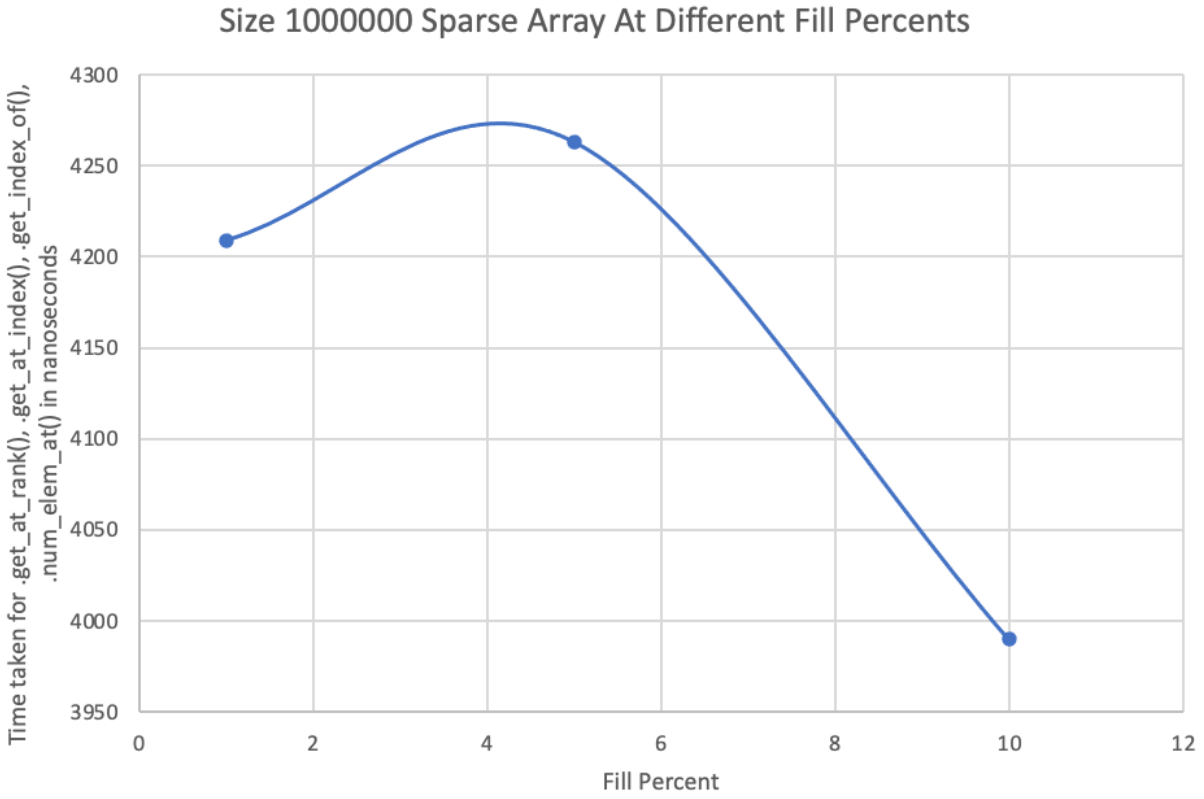


The next series of images shows the percent fill of a sparse array versus the time needed to complete one call to each of .get_at_rank(), .get_at_index(), .get_index_of(), and .num_elem_at() for various sizes of sparse arrays.


Size 1000 Sparse Array at Different Filled Percents
Time taken for .get_at_rank(), .get_at_index(), .get_index_of(), .num_elem_at() in nanoseconds
1335
1330
1325
1320
1315
1310
1305
1300
0
2
4
6
8
10
12
Percent Fill


Size 10000 Sparse Array At Different Fill Percents
Time taken for .get_at_rank(), .get_at_index(), .get_index_of(), .num_elem_at() in nanoseconds
1700
1650
1600
1550
1500
1450
0
2
4
6
8
10
12
Fill Percent





Though there appears to be some variation across the various sparse array sizes, there generally appears to be a decrease in time needed to carry out the function calls as the sparse array becomes fuller.

Assuming that each empty string that would have been contained in a sparse array of strings is approximately 1 byte (for the null character), we are saving thousands of bytes of memory using the bitvector representation. The number of bytes needed for a bitvector the size of each of the above sizes for sparse arrays are 136 bytes, 1264 bytes, 12512 bytes, and 125008 bytes respectively. An array the size of each of the sizes of sparse arrays above of only empty strings would require at least 1000, 10000, 100000, and 1000000 bytes respectively. As such, we are using a little more than a tenth of the space to create the bitvectors plus the space needed for the lengths of the non-empty strings we store in our string vector. However, as the sparse arrays become fuller, the savings decrease as the number of bytes needed to store the vector of non-empty strings will increase. As such, the fuller the sparse array, the smaller the savings.