

The writeups for all three tasks should be put into a single document (either a PDF or a Word Document). Each writeup should include the following 3 components:

1. A short (no more than a paragraph) prose description of your implementation.
2. A brief (again, no more than a paragraph) summary of what you found to be the most difficult part of implementing each task.
3. The plots as described in each task.

## **Task 1: bit-vector rank**

### *Implementation*

For the implementation of this data structure, I followed the strategy and technique learned in class and discussed over Piazza. The RankSupport structure consists of a `bit_vector`, an appropriately sized `int_vector` of superblocks, as based on the size of the input `bit_vector`, and an appropriately sized `int_vector` of blocks, as based on the size of the `bit_vector` and superblock. `rank1()` is implemented as a constant time operation, accessing the appropriate elements from the superblocks and blocks, and using popcount instead of a lookup table for `r_p`.

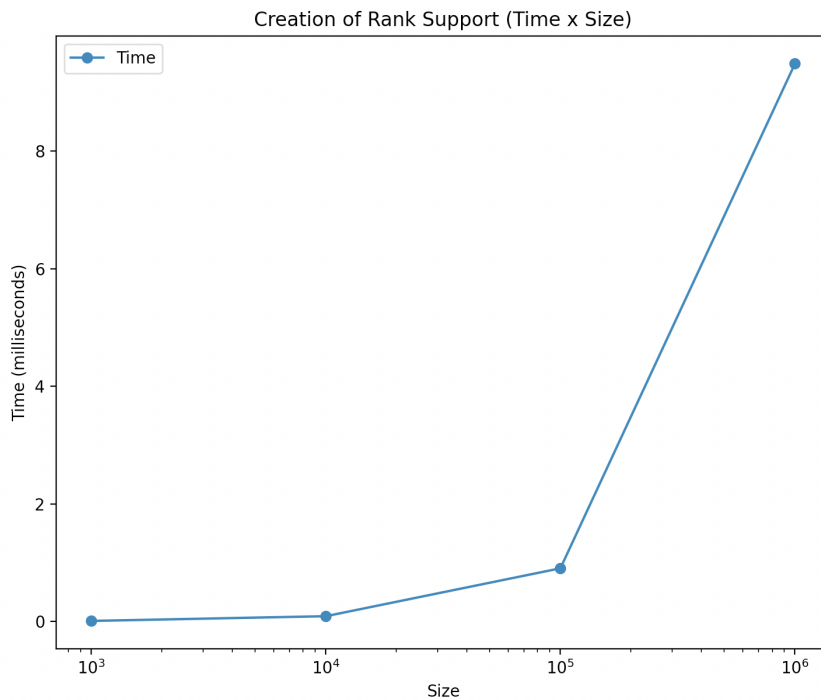
### *Challenges*

Handling the edge cases of sizing and confirming my understanding from class was the hardest part. I understood the general theory, but realized I had many little questions about sizing specificities or how things should align. MG's questions on Piazza significantly helped in covering exactly what I was unsure about.

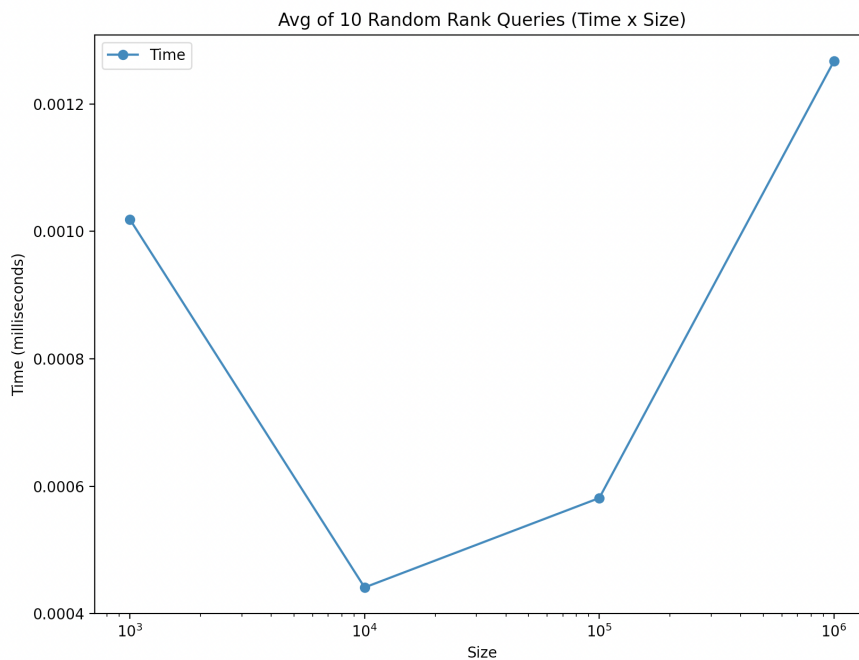
### *Experiments / Plots*

Provided are plots and descriptions on RankSupport structures built around input `bit_vectors` of the following sizes: 1000, 10,000, 100,000, 1,000,000 bits. Provided are charts that timed the average initialization of the RankSupport structure, average lookup speed of 10 random `rank1()` queries, and the average overhead for each Structure. Each test was repeated 10 times and timings were averaged together.

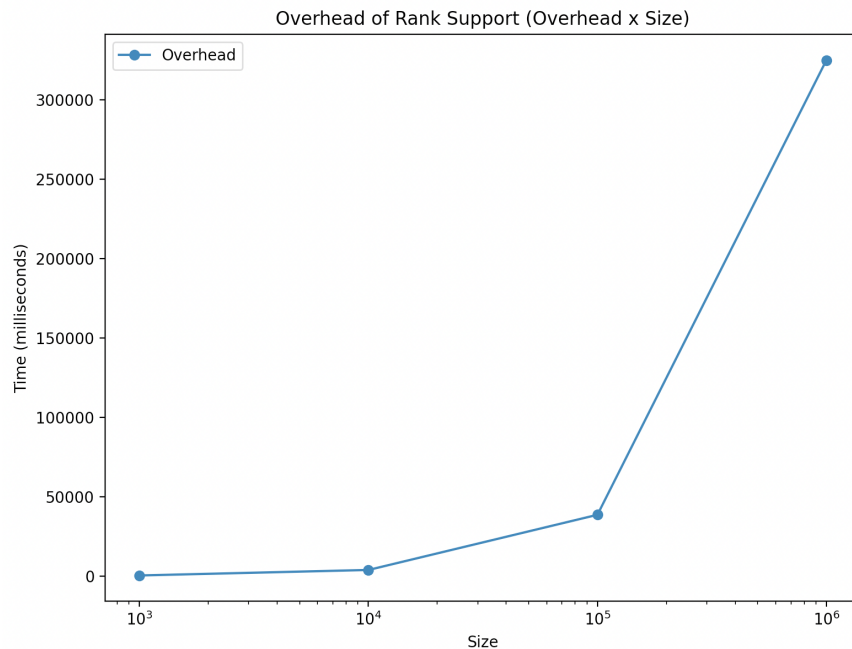
**All graphs shown have a linear y-scale and  $\log(n)$  x-scale.**



There appears to be a linear relationship between the size of the given bit vector and the time of creation (y scale is linear, x scale is  $\log(n)$ ). This is consistent with expectations. We only require one pass over the bit vector in order to populate the rest of the RankSupport structures.



There does seem to be a constant time relationship between performing random `rank1()` queries on `bit_vectors` of varying sizes. The drop and rise between size 1000 and 1000000 is a bit strange, however, I am not sure why the two sizes tested in between fall in such a manner.



There appears to be a linear relationship between the overhead of the RankSupport structure and the given bit vector. This is consistent with our expectations. ***(Once again noting the y-scale for ALL of this presentation is linear, while the x-scale is logarithmic).***

## Task 2: bit-vector select

### Implementation

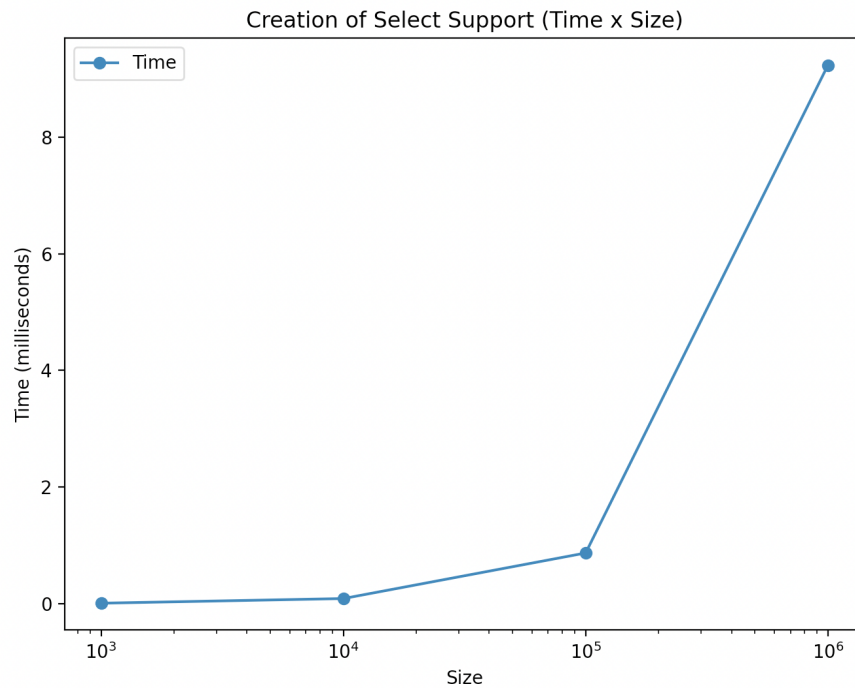
The SelectSupport structure simply contains a reference to a RankSupport structure within. select1() functionality was simply built on top of the existing RankSupport class.

### Challenges

No particular challenges with this one, although I did mess up my index changes in the binary search the first time.

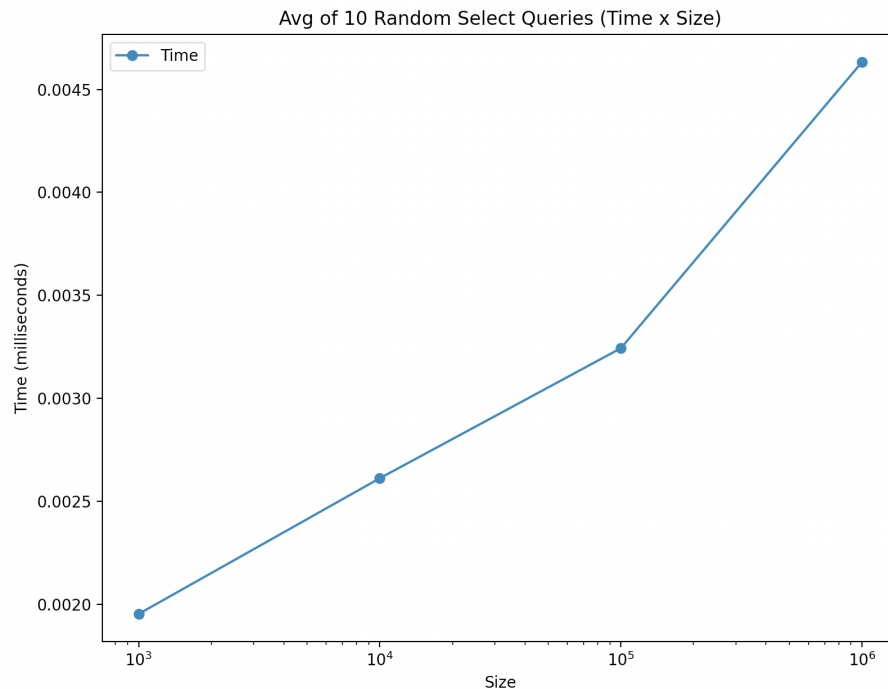
### Experiments / Plots

Just as with RankSupport, provided are plots and descriptions on SelectSupport structures built around input bit\_vectors of the following sizes: 1000, 10,000, 100,000, 1,000,000 bits. Provided are charts that timed the average initialization of the SelectSupport structure, average lookup speed of 10 random select1() queries, and the average overhead for each Structure. Each test was repeated 10 times and timings were averaged together.

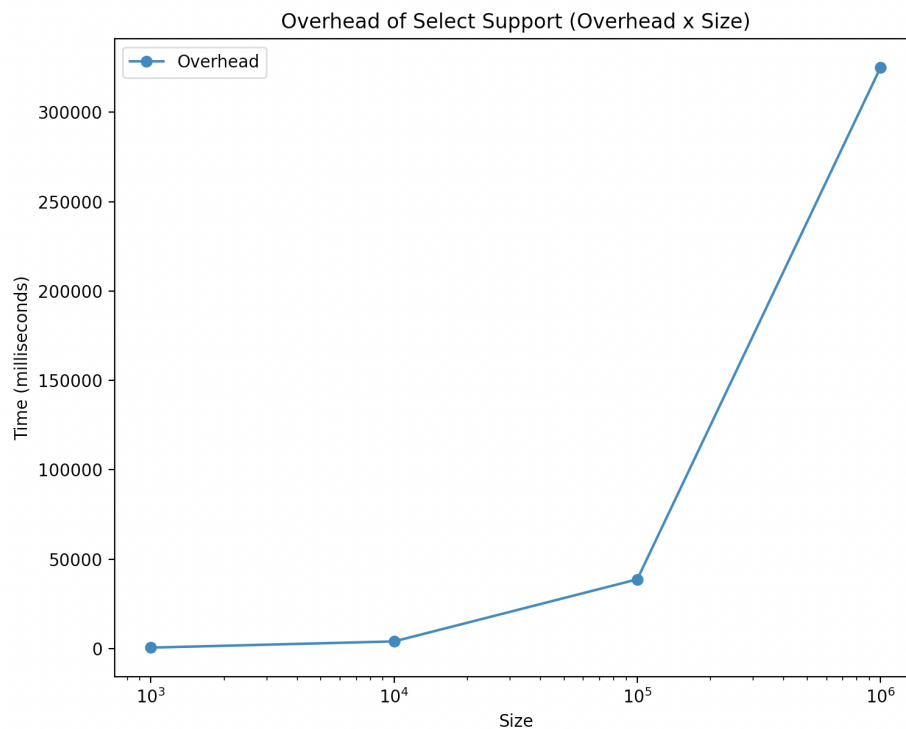


This chart is inevitably almost exactly the same as the one given for RankSupport, as there are no additional vector like structures on top of RankSupport, just some different methods and variables.

This chart is consistent with our expectations of a linear relationship. **(y-scale is linear, x-scale is logarithmic)**



Averaging over random select1() queries we begin to see a logarithmic relationship between the time and size of the bit vector. This is to be expected, as select1() as implemented is simply a binary search, running in  $O(\log_2 n)$  time.



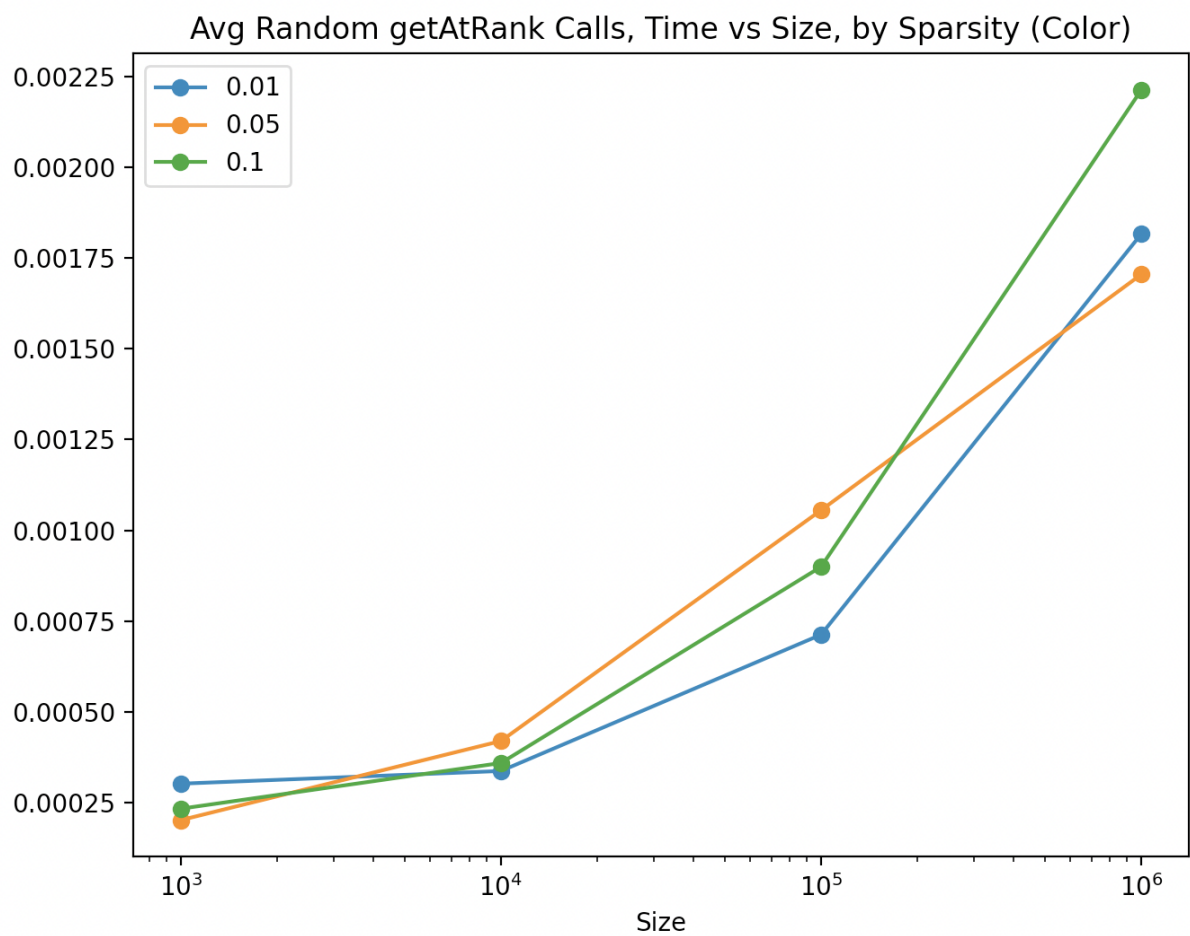
This chart is inevitably almost exactly the same as the one given for RankSupport, as there are no additional vector like structures on top of RankSupport, just some different methods and variables.

This chart is consistent with our expectations of a linear relationship.

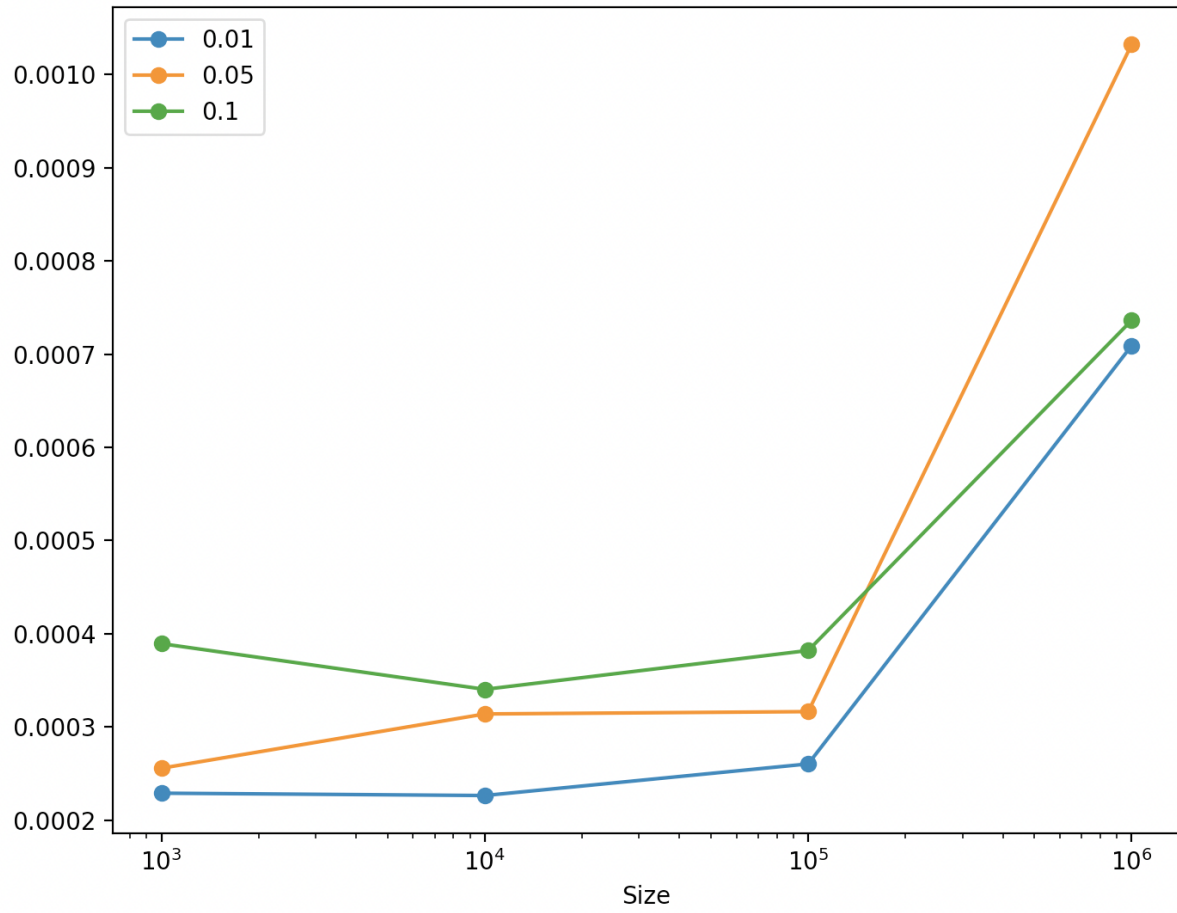
### Task 3: Sparse Array

#### *Implementation*

The SparseArray class contains a reference to a RankSupport structure within. Given the choice of storage of elements themselves (`std::vector`), it was all that was necessary for efficient querying. One can initialize the SparseArray with a size.



Avg Random getAtIndex Calls, Time vs Size, by Sparsity (Color)



Avg Random numElAt Calls, Time vs Size, by Sparsity (Color)

