CMSC 701 - Computational Genomics, Spring 2023
Homework 2

Name: Sazan Mahbub
UID: 118214443

Task1 – bit-vector rank:

Implementation:

For this task, I wrote a C++ class named *rank_support*, where implemented and experimented with two algorithms:

1. **Precompute and save all ranks:** This rank1 algorithm takes constant time (O(1)) and linear (O(*n*)) space, where n is the size of the input bit-vector. The original algorithm has O(*n**log(*m*)) space complexity (where m is the number of set bits); however, since the *rank1()* operation returns an *uint64_t* value, we can precompute and save the ranks as *uint64_t* in an O(*n*) list. This implementation sets an upper bound on *m* and converts the space complexity to O(n). The function named *rank1()* in my implementation uses this algorithm.
2. **Jacobson's rank algorithm**: First I computed the cumulative ranks of the chunks' starting positions and the sub-chunks' relative cumulative ranks, and stored them in a vector and a matrix respectively. During *rank1()* operation for an index *r,* I access the ranks of the chunk and the subchunk holding index *r*; and then compute the bit count from the starting of the subchunk's starting index to index r and add all three together. This way the space complexity becomes sub-linear *o(n)* instead of O(*n*) or O(*n**log(*m*)). The function named *rank1_jacobson()* in my implementation uses this algorithm.

The most difficult part I found while implementing Task1:

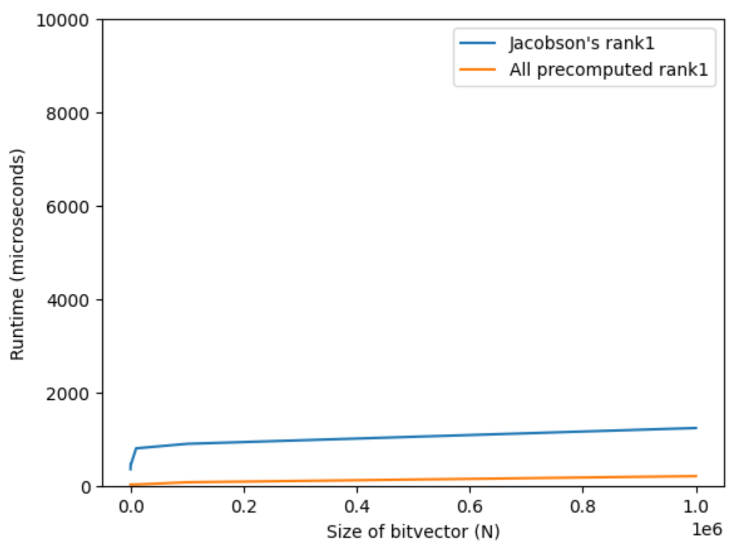
I found it difficult to make the runtime strictly O(1) while implementing Jacobson's rank algorithm. I started working with *std::vector<bool>*, which actually saves bits instead of booleans (ref: https://en.cppreference.com/w/cpp/container/vector_bool). Unfortunately, I could not find a *popcount* or similar method that works with this data structure despite my best effort. I eventually used the count() method, which I assumed might not be constant time. Then I experimented and compared its performance with my other implementation (precomputed and saved ranks), which is *certainly constant time* since it simply performs O(1) access to an array during *rank1()* operation. I found that the performance is not that worse comparatively, especially when the size of the bit-vector is larger, as we can see from the results in the next section. I assume this could probably be because of the runtime getting an asymptotic bound for this implementation as well.

Plots and Analysis:

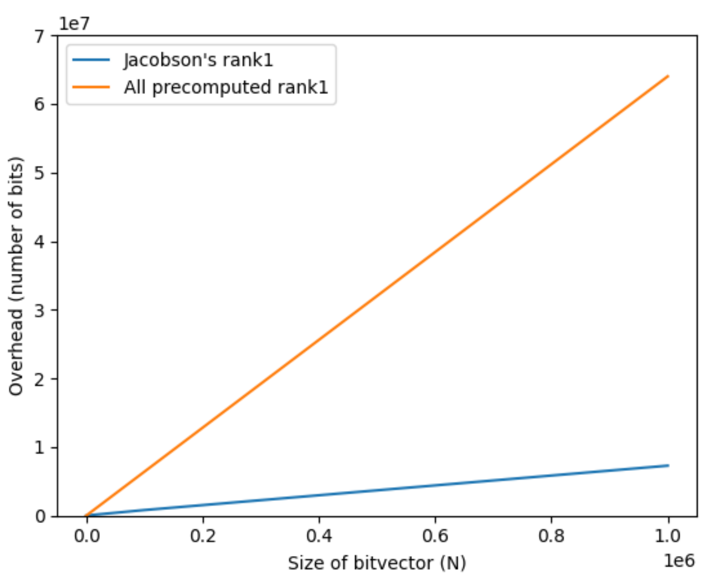
For this section, I computed the total runtime to perform 10,000 *rank1()* operations (shown along the y-axis in the first graph) for each bit-vector size (shown along the x-axis). The overhead is computed as the number of bits needed to store the extra data structures for these algorithms. The used bit-vector sizes are {100, 500, 1000, 10000, 100000, 1000000}.

To compare the results with the *select1()* operation (in Task2) we used the same y-axis range as well. The runtime seems to the asymptotically bounded. However, as expected, the overhead kept increasing. The *linear* and *sublinear* bounds on space complexity for the two algorithms (precomputed rank and Jacobson's rank respectively) are clearly reflected in the results.

***Runtime* vs *Size of bitvector*:**



***Overhead* vs *Size of bitvector*:**



Task2 – bit-vector select:

Implementation:

For this task, I wrote a C++ class named *select_support*, where implemented a *constant time* select algorithm, where I precompute and store the the selection operation outputs in an extra vector. During *select1()*, we can simply access this vector in O(1) time using the *rank* as the index of this vector. For overhead computation, I kept two functions – *overhead()* and *overhead_2()*.

1. *overhead():* the overall overhead to use the *select1()* operation. This computes the number of bits of the data structures needed for *rank1()+select1()* operations.
2. *overhead_2():* this computes the overhead/number of bits for only the extra vector I added for the constant time *select1()* operation.

The most difficult part I found while implementing Task2:

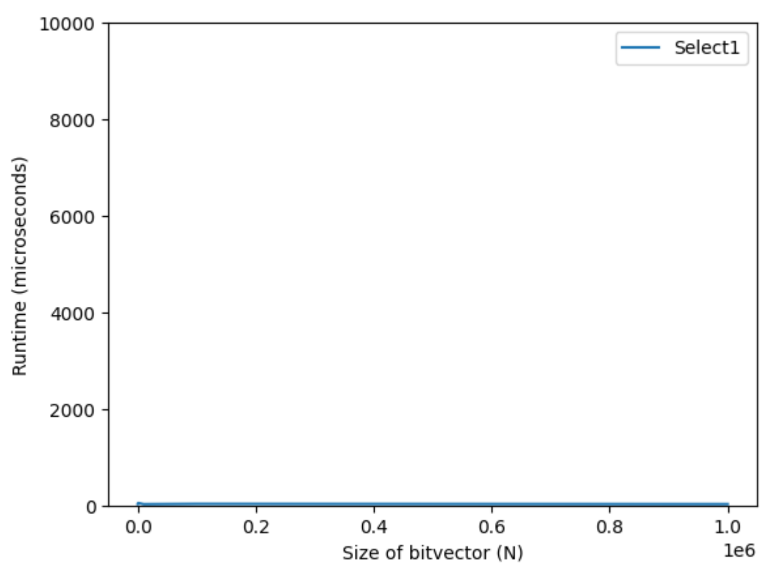
I was confused about whether I should store the data structures for *rank1()* in the this class, or should I store them outside. I finally decided on storing them outside separately.

Plots and Analysis:

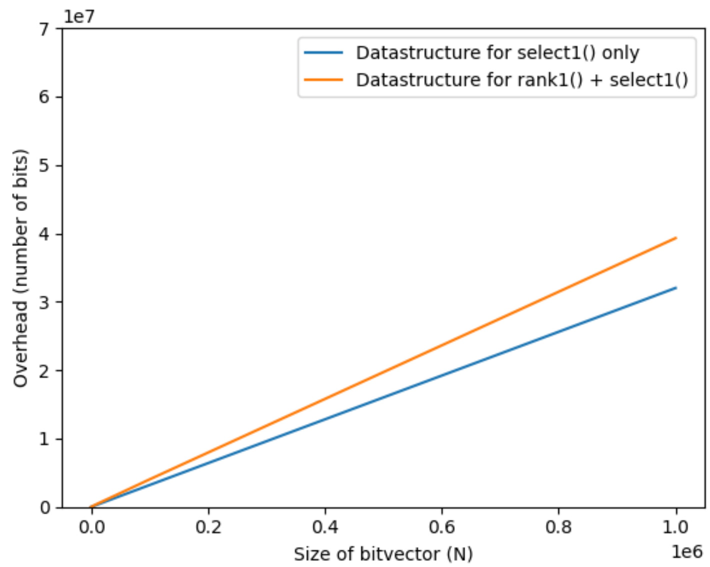
For this section, the runtimes and the overhead values are computed in the same way described above in the *Plots* section of *Task1*. For overhead, I am showing two line plots for *overhead()* and *overhead_2()*, as discussed in the implementation section.

As we used the same y-axis ranges, we can compare and see that 10,000 *select1()* operations take significantly shorter runtime than 10,000 *rank1()* operations. However, as we can see the vector for the constant time *select1()* operation results in much more overhead than *rank1()*.

***Runtime* vs *Size of bitvector*:**



***Overhead* vs *Size of bitvector*:**



Task3 – implementing a sparse array using bit-vector rank and select:

Implementation:

I wrote a C++ class named *sparse_array* in this part using the classes I wrote in the two previous sections. I store the bit-vector and the input elements as members of this class. Here, I create and store objects of rank_support and select_support classes that I use to implement our expected functions. I implemented this part for values of type string only.

The most difficult part I found while implementing Task3:

I was facing difficulties saving and loading the elements (a vector of strings). I spent much time finding an efficient approach to do this that would require less code and would be computationally less expensive. I finally implemented it in a rather straightforward way by iterating over the vector and saving/loading each string separately.

Plots and Analysis:

For this section, I computed the average time of running each function 100 times (shown along the y-axis). In these graphs, I demonstrate how the runtimes or speed (of the individual functions as well as running all the functions) – vary with respect to the following two variables:

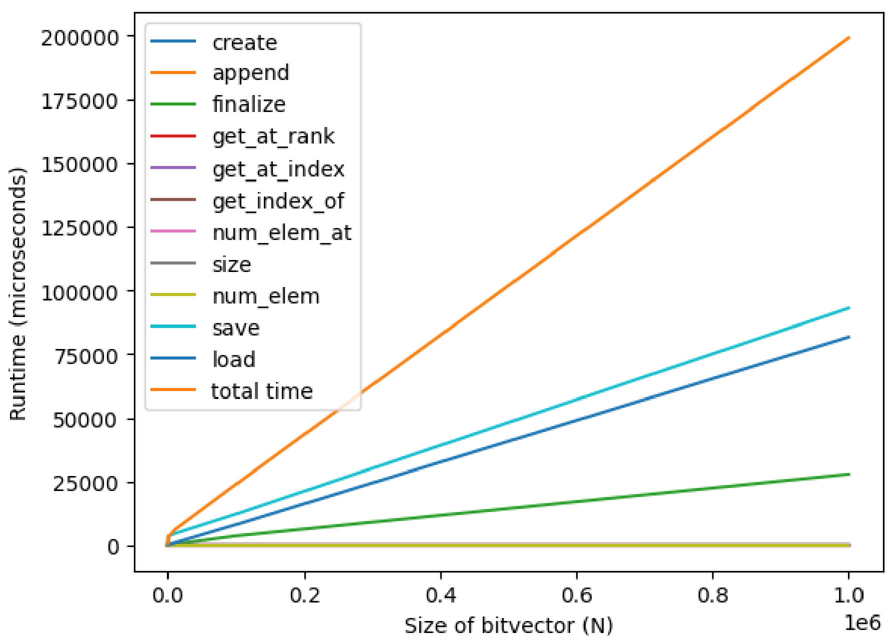
1. **Overall size (first graph):** The size of the bit-vector. Here, the used bit-vector sizes are {100, 500, 1000, 10000, 100000, 1000000}. For this experiment, I kept the sparsity level constant at 50%.
2. **Overall sparsity (second graph):** The fraction of 0 bits. Here, the used fractions are {0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 0.95, 0.99, 1.0}. For this experiment, I kept the overall size constant at 10,000.

The legends in the graphs represent the names of the major functions I implemented here. The lines labeled "total time" represent the average time of running all functions 100 times.

Here we can see that the functions *save()*, *load()*, and *finalize()* are the most computationally expensive ones. The *save()* and *load()* functions need to access the hard disk drive to complete their operations, which significantly increases their runtimes. The *finalize()* function creates the *rank_support* and *select_support* objects that are relatively expensive to compute. The rest of the functions operate in almost constant-time (shown here as the lines parallel to the x-axis overlapped with one another).

Moreover, when increasing the size of the bitvector, the runtime increases linearly for those three functions. When we increase the fraction of 0 bits (representing sparsity), the runtime of *load()* function decreases, but *finalize()* it remains almost constant. For *save()* function we see a a rather interesting pattern. The runtime first goes down in a non-linear fashion and then suddenly starts increasing at a high rate, especially after the percentage of 0 bits is more than 90%.

***Runtime* vs *Size of bitvector*:**



***Runtime* vs *Fraction of 0 bits*:**