Tobias Rubel **Homework 1** CMSC858D
March 9, 2022

For all parts of this document I looked at sparse arrays of size 1000, 10000, 100000, and 1000000 with sparsity of $1\%, 5\%$, and 10% respectively for a total of 12 arrays. I then used their rank support objects for part 1 and their select support objects for part 2. You can generate the underyling data for the plots by compiling and running main.cpp in the top level directory.

The naming scheme I'm using is that a1 is the 1000 length sparse array, a2 is the 10000 length sparse array and so on. Then s1 denotes a sparsity of 1%, s2, denotes a sparsity of 5%, and s3 denotes a sparsity of 10%. So for instance a4s2 is the sparse array with 1000000 positions of which 10% of them have values.

1 Part 1

1.1 Implementation

In order to build the data structure I compute the block-sizes of the Rb and Rs tables as $\lceil \log n \rceil$ and $\lceil \log n \rceil^2$ respectively. I set the width of the cells in Rb and Rs to be the ceiling of the log of the Rb and Rs block-sizes plus 1 (see Reflection for commentary on this). The number of Rs blocks then falls out as

$$\lceil n/\text{Rb\_blocksize} \rceil$$

, and similarly for Rs_blocksize. I then scan through the integer vector incrimenting the Rs[Rs_index+1] and Rb[Rb_index+1] when I see a 1 value. When I get to new Rs blocks then I set the current Rb block to 0 and also add the value of the prior Rs block to the current one. When I get to a new Rb block I add the value of the prior one. To compute rank support I use SDSL builtins to get the machine word and then do a little bitshifting plus a popcount. I use builtin serialization to save and load the object.

1.2 Reflection

I worked quite diligently on this assignment over the weeks that it was assigned, but had to step away for a few days to deal with other courses. When I returned to (I thought) do my writeup today, I found that rank support was mysteriously giving wrong values for big vectors, but worked for small vectors. This led to a frenzy of debugging — I ultimately believe the issue was that I was compacting the block widths to be too small and so my additions were overflowing. Otherwise I felt like things went pretty smoothly overall. MG's questions and your helpful answers helped me to catch some conditions I hadn't thought of (like mismatch of Rb and Rs on certain schemes of breaking things up).

1.3 Plots & Tables

In order to determine the runtime of rank1 I computed rank1 on 1000 different number for each sparse array. Since rank1 is an O(1) operation, it's not very interesting to actually plot out how the function grows with respect to the size or density of the vector. I hope it suffices to provide a table:

array	time
a1s1	0.000037
a1s2	0.000036
a1s3	0.000036
a2s1	0.000036
a2s2	0.000036
a2s3	0.000036
a3s1	0.000036
a3s2	0.000036
a3s3	0.000036
a4s1	0.000036
a4s2	0.000037
a4s3	0.000036

As can be seen, this plot is exactly as boring as we would hope. Whether a bit vector has only 1000 elements or 1000000, we can efficiently get the rank of an element.

The overhead looks good as well, scaling linearly in the size of the bit vector. This did not depend on the density of the vector (as expected), so I've just got the 4 points here. The y axis is in bits. This value was computed by summing the number of bits in the Rs and Rb tables. I didn't include a few integers which I keep track of because those don't grow with the size of the vector.

2 Part 2

2.1 Implementation

I wrote this one recursively. Basically the code is just good old fashion linear bisection. I compute the rank of the middle value. If the rank I'm after is less then I search in the middle of the bottom half, if it's larger then I search in the middle of the top half.

2.2 Reflection

The hardest part of this, embarrassingly, was importing the rank code into this. I'd never learned how header files work in c or c++ and when I was trying to link the files together for

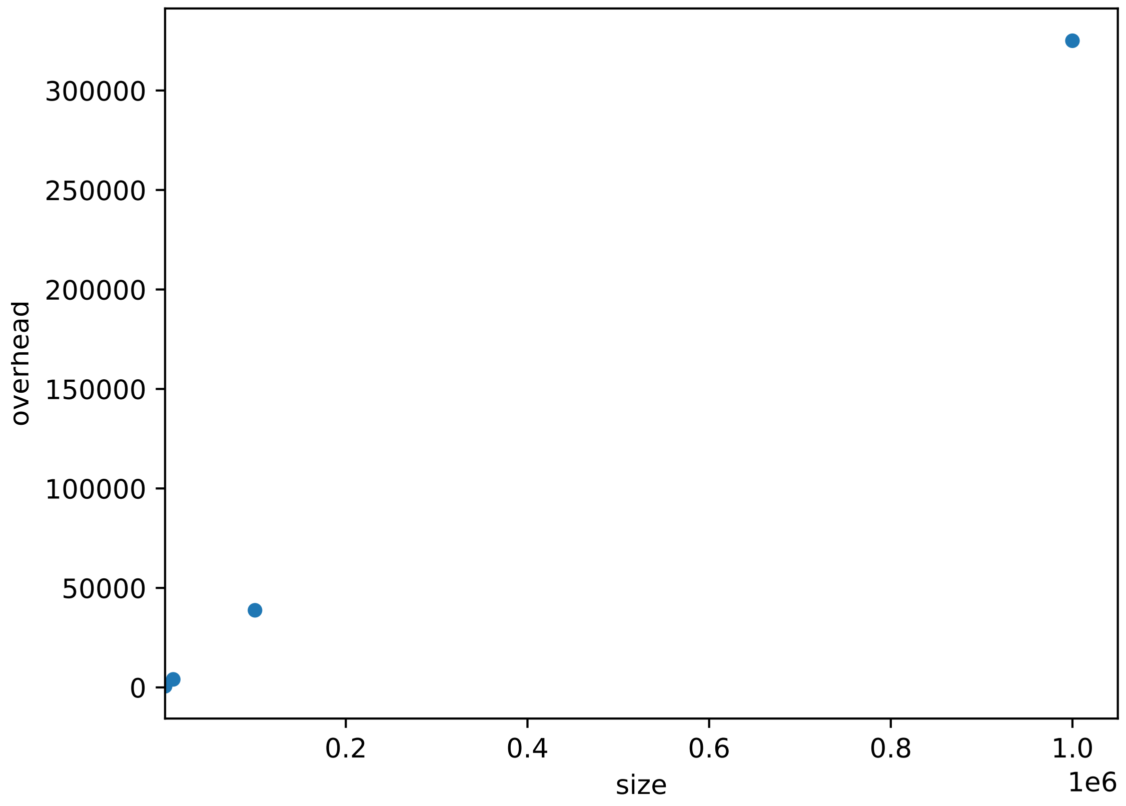

Figure 1: Rank Support Overhead

compiling I couldn't for the life of me get it to work. Eventually I found that making an hpp file and then just including it worked fine, but it took several hours of reading and trying different things out before I came to that conclusion. The other big issue was that I kept getting infinite loops today, but I believe that problem was caused by the issue described above.

2.3 Plots & Tables

The select support class has no additional overhead over rank support, since it just implements an additional method on top of rank. Thus Figure 1 does double duty for us as a plot of the select support class's overhead. Regarding the runtime of support select, in hind sight I ought to have plotted more points to really be sure, but the 3 points that I did plot (for 10000, 100000, 1000000 sized vectors), show a story consistent with logarithmic growth. I was having some trouble with occasionally hitting infinite loops with this code right up to the deadline so I didn't get to analyze it as much as I ought to have.

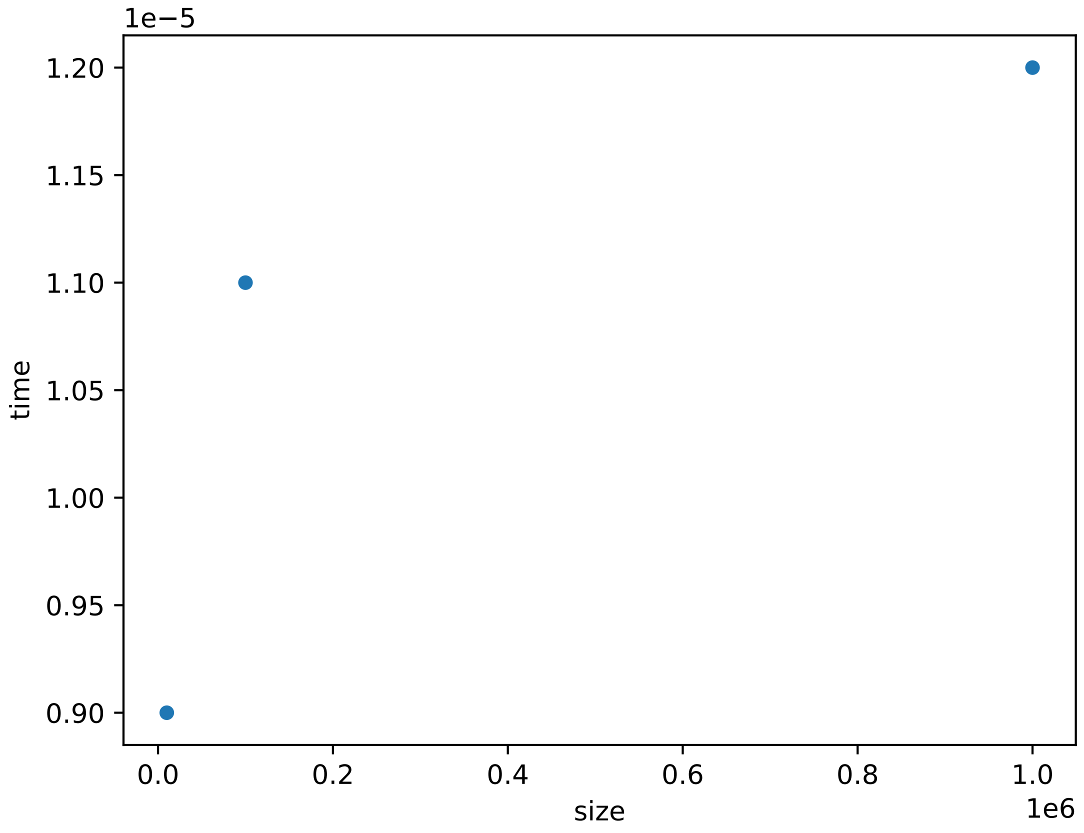

Figure 2: Support Select Runtime

3 Part 3

3.1 Implementation

The sparse array class is more or less just a wrapper around the rank support class. On initialization I create a bit vector of the allotted size, as well as an empty string vector. When I want to append something at a position I toggle the bit at that position and push_back on the vector. I did implement a convenient method which allows me to create vectors already filled with a certain number of values. This made generating this report easier.

3.2 Reflection

Once rank support is working this goes smoothly. The hardest part of this task was probably generating all of the data to analyze it.

3.3 Plots & Tables

For the sparse array class I looked at the performance of get_at_rank, get_at_index, and num_elem_at. Looking firstly at get_at_rank:

array	time
a1s1	0.000002
a1s2	0.000001
a1s3	0.000001
a2s1	0.000001
a2s2	0.000002
a2s3	0.000004
a3s1	0.000005
a3s2	0.000005
a3s3	0.000004
a4s1	0.000004
a4s2	0.000004
a4s3	0.000005

It can be seen plainly that sparsity is not having an effect on runtime while vector size seems to be. Looking next at get_at_index, we see that the exact opposite is true. Sparsity is the sole driver of runtime variation:

array	time
a1s1	0.000010
a1s2	0.000012
a1s3	0.000013
a2s1	0.000010
a2s2	0.000012
a2s3	0.000014
a3s1	0.000010
a3s2	0.000012
a3s3	0.000014
a4s1	0.000010
a4s2	0.000012
a4s3	0.000014

Finally num_elem_at has exactly the same runtime characterstics as rank1, since it *just is* rank1.

Assuming that an empty string is a single byte in size in cpp, then explicitly storing the whole sparse array requires approximately 8 times the number of bits we use to store the empty values in the bit vector. The rank structure requires some overhead of course, but for sparse arrays it only requires

$$\lceil \log(\log(n)) \rceil \cdot n / \lceil log(n) \rceil$$

bits to store Rb, and

$$\lceil \log(\log(n)^2) \rceil \cdot n / \lceil log(n)^2 \rceil$$

to store Rs. Thus the equation to compute for a vector of size n with m elements filled is:

$$8 * (n - m) - (\lceil \log(\log(n)^2) \rceil \cdot n / \lceil log(n)^2 \rceil) + (\lceil \log(\log(n)) \rceil \cdot n / \lceil log(n) \rceil)$$

If you get a positive number then it's worth using the sparse array. Otherwise just explicitly store the values and avoid the overhead.