Assignment 1

By Nikhil Pateel

Github link: https://github.com/npateel/sparse-array

This notebook does all of the analysis for the bitvector rank and select, and sparse array code.

Implementation details

Project structure and explanation

This project has severla folders to help with structure. `analysis` contains the data of experiments and corresponding jupyter notebook, `src` contains the source code, `build` stores all the object files, `bin` is where the executable is found. As an important note, this project relies on `sdsl` for creating compact integer vectors (and bitvectors). The code is written in C++, with `main.cpp` containing all of the code to run experiments (would have moved to another file barring more time). `rank.cpp` implements the `RankSupport` class, which takes in a `sdsl::bit_vector` and provides the tools needed to `rank` (in $O(1)$ time) the 1s in that bitvector. In addition to `rank`, there is also support for `size`, serialization using `load` and `save`, and an `overhead` function which provides the total number of bits used to store auxillary data (not the bitvector, theoretically $O(n)$ bits).

`SelectSupport` defined in `select.cpp` is a subclass of `RankSupport` and extends the class definition to include a `select` algorithm (which runs in $O(\lg n)$ time). Since `SelectSupport` uses no extra overhead, this singular function is the only difference between it and `RankSupport`

Finally, there is a `SparseArray` class in `sparse-array.cpp` that defines the Sparse array. It supports basic insertion, lookup, and serialization options. This object uses a `SelectBitset` under the hood.

Challenges

The biggest issue throughout this project was unrelated to the datastructures as a whole. For me, I struggled the most with understanding C++ syntax (which I've never used before), and the `sdsl` library. A considerable amount of time was spent looking through the documentation of both the standard library and `sdsl` with very slow progress. Once I learned these things, I struggled most with aligning the superblocks and block vectors properly in the `rank` class. Once I did this, I was able to implement `rank`, `select`, and the sparse array in a very short time.

In [1]:
```python
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np
```

In [2]:
```python
bitvector_data = pd.read_csv("rank.csv")
bitvector_data
```

Out[2]:

	n	rank_time	select_time	overhead
0	100	0.006930	0.044879	198
1	200	0.006860	0.050560	328
2	300	0.007110	0.057189	360
3	400	0.007140	0.060290	432
4	500	0.007550	0.059269	576
5	1000	0.007440	0.066570	872
6	1500	0.007720	0.070479	1216
7	2000	0.007570	0.073409	1568
8	2500	0.007340	0.076229	1800
9	5000	0.007600	0.085789	3192
10	7500	0.007430	0.087500	4752
11	10000	0.007700	0.092109	5896
12	12500	0.007850	0.092099	7240
13	25000	0.007940	0.099690	13512
14	37500	0.007210	0.100129	23592
15	50000	0.007280	0.101010	31432
16	62500	0.007110	0.102169	39272
17	125000	0.007770	0.116789	73682
18	187500	0.007840	0.117999	104292
19	250000	0.007609	0.120880	139032
20	312500	0.007680	0.126879	164612
21	625000	0.008170	0.135259	312672
22	937500	0.007880	0.133449	468872
23	1250000	0.007909	0.142470	595422
24	1562500	0.008030	0.142319	744312
25	3125000	0.008520	0.150799	1420612
26	4687500	0.008660	0.158569	2038332
27	6250000	0.008690	0.162269	2717522
28	7812500	0.008710	0.161729	3396942
29	15625000	0.008820	0.177109	6510552
30	23437500	0.008670	0.172799	9375072
31	31250000	0.008879	0.178059	12500072
32	39062500	0.010370	0.185819	15024172
33	78125000	0.024909	0.243389	28935432
34	117187500	0.013860	0.204409	43403112
35	156250000	0.024870	0.267578	55803792

Bitvector

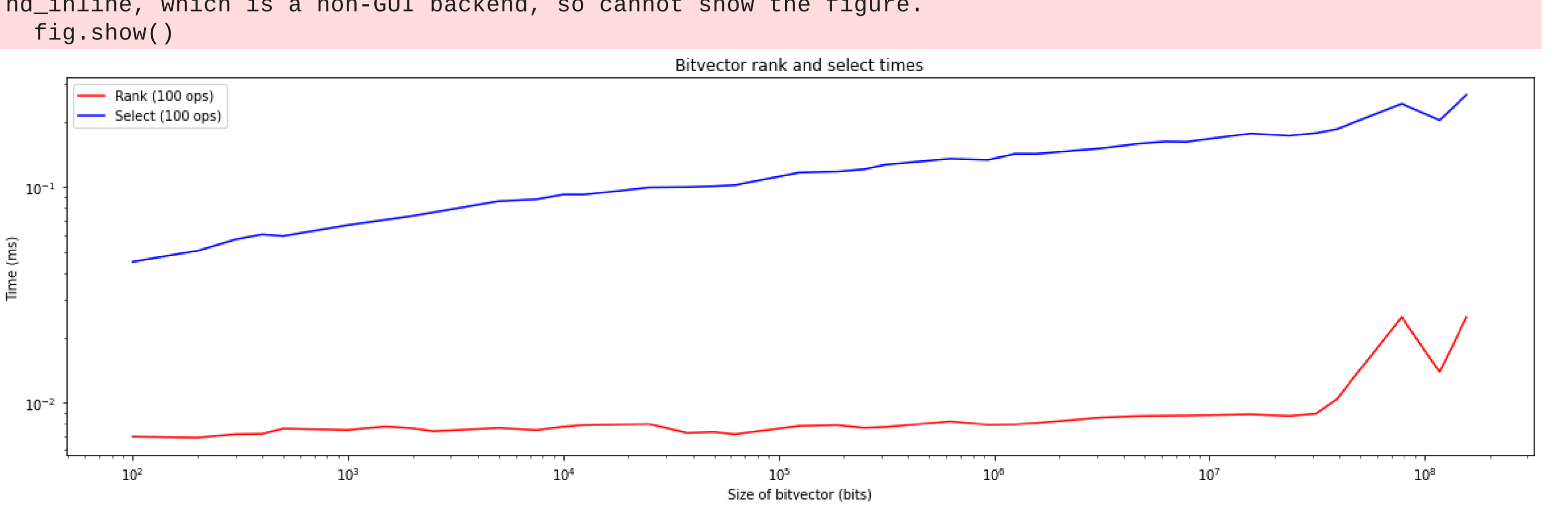
First, we test out bitvector. We can see from the following graphs that `rank` was able to reach near $O(n)$ `rank` runtime, and that `select` was near some form of sub-linear time. It likely checks out that it was $O(n \lg n)$ runtime, however, more testing would be needed for this

After the first graph, is another graph plotting the overhead of the rank datastructure against the size of the bitvector. We can see very clearly that we were able to achieve a $O(n)$ space efficiency for our `SelectSupport` and `RankSupport` data structures.

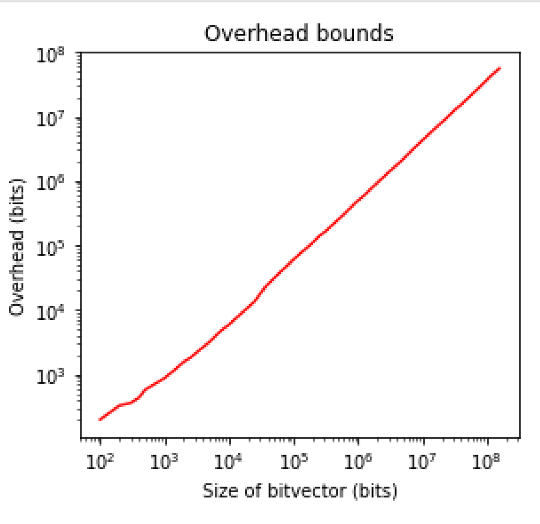
Interestingly, the time spikes after approximately 10^8 bits (or 1.25MB). I'm not entirely sure why this is the case, but suspect it might have to do with superblock sizes becoming increasingly large, or cache-related.

In [3]:
```python
fig, ax = plt.subplots(figsize=(20,20))
ax.set_xlabel('Size of bitvector (bits)')
ax.set_ylabel('Time (ms)')
ax.loglog(bitvector_data['n'], bitvector_data['rank_time'],color='red', label='Rank (100 ops)')
ax.set_title('Bitvector rank and select times')
#selectax = ax.twinx()
#ax.set_ylabel('Time for 100 select operations (ms)')
ax.loglog(bitvector_data['n'], bitvector_data['select_time'], color='blue', label='Select (100 ops)')
ax.set_aspect('equal', adjustable='box')
fig.show()
plt.legend(loc='upper left')
plt.savefig('bitvectors.png')
```

```
<ipython-input-3-d35f6e8d7503>:10: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backe
nd_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



In [4]:
```python
fig, ax = plt.subplots()
ax.set_xlabel('Size of bitvector (bits)')
ax.set_ylabel('Overhead (bits)')
ax.set_title('Overhead bounds')
ax.loglog(bitvector_data['n'], bitvector_data['overhead'],color='red')
ax.set_aspect('equal')
```



Sparse Array

Size comparison

First, we see that our implementation of the `SparseArray` produces near constant time `get_rank_at`, `get_index_at`, and `num_elem_at` operations until about 10^6 bits. After that size, each operation becomes a near `O(n)` operation. I'm unsure why this is the case, but it might have to do with how some `std::vector` operations are being done

In [5]:
```python
array = pd.read_csv('array.csv')
array['savings'] = array['size'] - array['size_bytes']
array
```

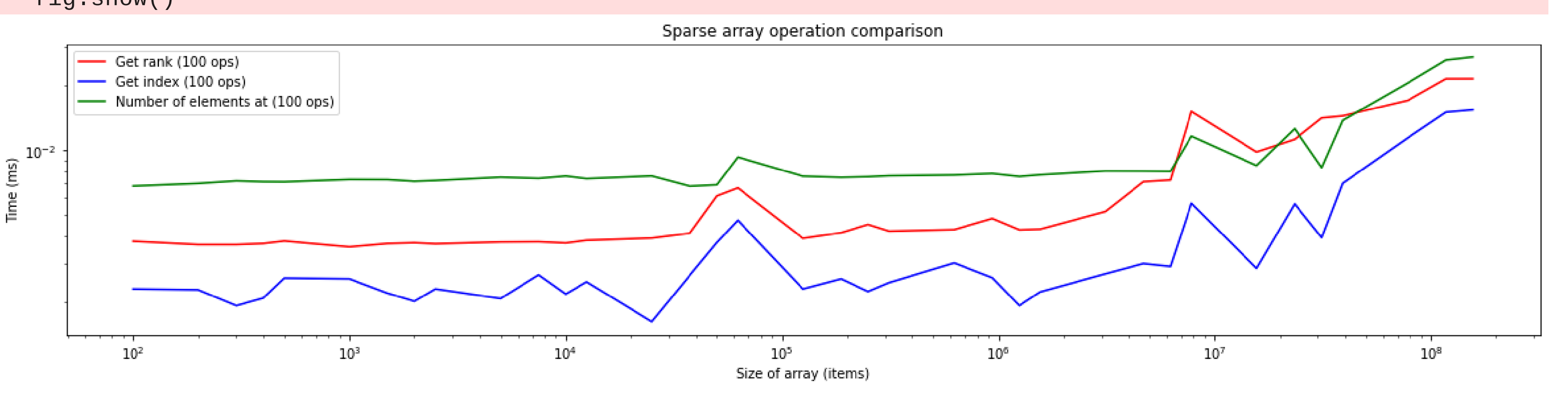
Out[5]:

	size	sparsity	build_index_time	get_at_rank_time	get_at_index_time	num_elem_at_time	size_bytes	overhead	savings
0	100	0.005	0.00148	0.00415	0.002040	0.00700	148	123	-48
1	100	0.010	0.00134	0.00379	0.001740	0.00683	148	123	-48
2	100	0.050	0.00140	0.00379	0.002280	0.00682	152	123	-52
3	100	0.100	0.00126	0.00366	0.002700	0.00692	157	123	-57
4	100	0.200	0.00129	0.00360	0.003380	0.00679	167	123	-67
...	...	...	...	...	...	...	...	...	...
175	156250000	0.005	272.20600	0.01874	0.009149	0.01707	27288085	26506811	128961915
176	156250000	0.010	273.41300	0.02342	0.015990	0.03679	28069335	26506811	128180665
177	156250000	0.050	273.72600	0.02137	0.015350	0.02691	34319335	26506811	121930665
178	156250000	0.100	272.59300	0.02238	0.019460	0.02857	42131835	26506811	114118165
179	156250000	0.200	270.25200	0.02470	0.024050	0.03080	57756835	26506811	98493165

180 rows × 9 columns

In [6]:
```python
array_five_percent = array.loc[array['sparsity'] == 0.050]
fig, ax = plt.subplots(figsize=(20,20))
ax.set_xlabel('Size of array (items)')
ax.set_ylabel('Time (ms)')
ax.set_title('Sparse array operation comparison')
ax.loglog(array_five_percent['size'], array_five_percent['get_at_rank_time'],color='red', label='Get rank (10
#selectax = ax.twinx()
#ax.set_ylabel('Time for 100 select operations (ms)')
ax.loglog(array_five_percent['size'], array_five_percent['get_at_index_time'], color='blue', label='Get index
ax.loglog(array_five_percent['size'], array_five_percent['num_elem_at_time'], color='green', label='Number of
#ax.loglog(array_five_percent['size'], array_five_percent['build_index_time'], color='purple', label='Index b
ax.set_aspect('equal', adjustable='box')
fig.show()
plt.legend(loc='upper left')
plt.savefig('arrayTimes.png')
```

```
<ipython-input-6-115e30d47a18>:13: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backe
nd_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



Sparsity Comparison

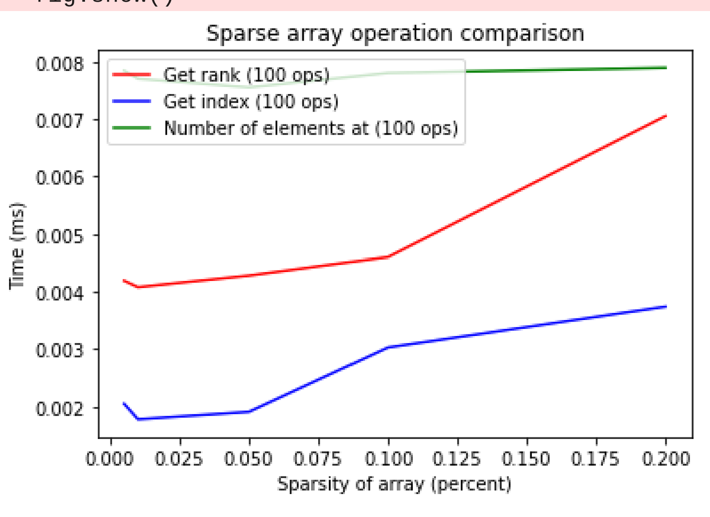
Next, we test the effects of sparsity on each of the operations. We see that there isn't much of an effect on `SparseArray` s operations.

This is likely because the underlying `rank` operation is $O(1)$ and therefore unaffected by the sparsity of the array.

Interestingly `get_rank_at` increases significantly with size, and that method only relies on `std::vector` operations, which leads me to believe that this is a `vector` problem

In [7]:
```python
array_12KB = array.loc[array['size'] == 1250000]
fig, ax = plt.subplots()
ax.set_xlabel('Sparsity of array (percent)')
ax.set_ylabel('Time (ms)')
ax.set_title('Sparse array operation comparison')
ax.plot(array_12KB['sparsity'], array_12KB['get_at_rank_time'],color='red', label='Get rank (100 ops)')
#selectax = ax.twinx()
#ax.set_ylabel('Time for 100 select operations (ms)')
ax.plot(array_12KB['sparsity'], array_12KB['get_at_index_time'], color='blue', label='Get index (100 ops)')
ax.plot(array_12KB['sparsity'], array_12KB['num_elem_at_time'], color='green', label='Number of elements at (
#ax.loglog(array_12KB['sparsity'], array_12KB['build_index_time'], color='purple', label='Index building')
#ax.set_aspect('equal', adjustable='box')
fig.show()
plt.legend(loc='upper left')
plt.savefig('arraySparsity.png')
```

```
<ipython-input-7-5bb19a4ea33e>:13: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backe
nd_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



Space savings

Finally, we calculate our space savings of using the `SparseArray`. We measure this using a specially designed `size_bytes` function in our `SparseArray` class, and compare it to `size`. Since each of the elements in a non-sparse array would be 1 or 2 bytes long, we can assume that the non sparse array would have a size of at least `size`. We subtract `size_bytes` from `size` to get our `savings`. What we find is that for low n, our `SparseArray` actually is bigger than a standard vector. However, when n is not much larger, we get very large savings on the size of our array, sometimes even saving an order of magnitude of data. The first graph is a 2d graph showing how sparsity and size affects our theoretical savings, and the second graph is a more traditional graph of savings as the size of the array gets larger (with a 5% sparse array). The savings are $O(n)$

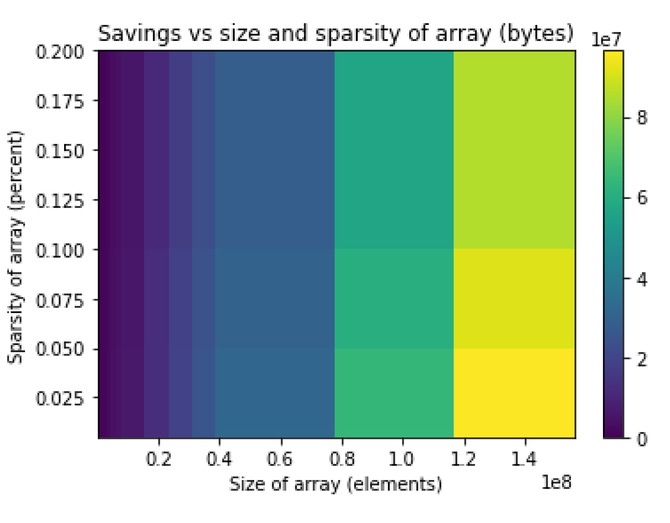
In [8]:
```python
array[['size', 'sparsity', 'savings']].head(180)
array['savings'] = array['size'] - array['size_bytes']
```

In [9]:
```python
fig, ax = plt.subplots()
ax.set_ylabel('Sparsity of array (percent)')
ax.set_xlabel('Size of array (elements)')
ax.set_title('Savings vs size and sparsity of array (bytes)')
x = array['size'].values.reshape(36,5)
y = array['sparsity'].values.reshape(36,5)
z = array['savings'].values.reshape(36,5)

X, Y = np.meshgrid(x[:,0], y[0])
X
c = ax.pcolor(X,Y,z.transpose())
fig.colorbar(c, ax=ax)
```

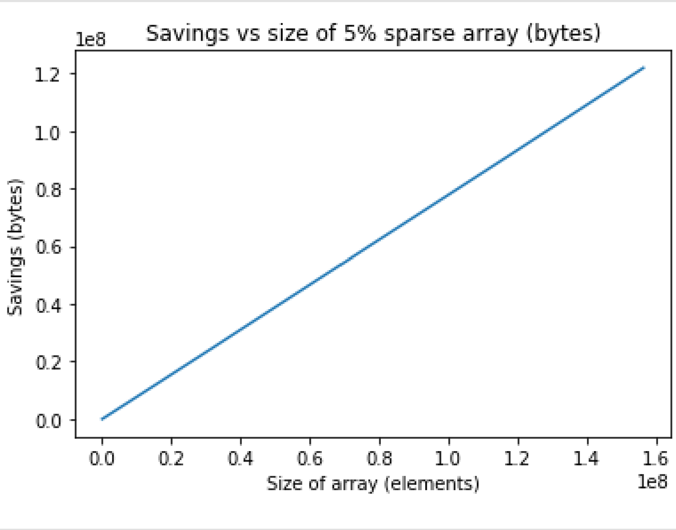
```
<ipython-input-9-30af5643cbeb>:11: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same d
imensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X and Y, or p
ass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error tw
o minor releases later.
  c = ax.pcolor(X,Y,z.transpose())
```

Out[9]:
```
<matplotlib.colorbar.Colorbar at 0x7fadc734fc70>
```



In [10]:
```python
fig, ax = plt.subplots()
#ax.set_ylabel('Sparsity of array (percent)')
ax.set_ylabel('Savings (bytes)')
ax.set_xlabel('Size of array (elements)')
ax.set_title('Savings vs size of 5% sparse array (bytes)')
x = array_five_percent['size']
z = array_five_percent['savings']

c = ax.plot(x,z)
```



In [ ]:

In [ ]: