Jimmy Patel

How to run the programs:

To run the programs, compile using gcc then execute similar to the lab.

```
$cat numbers.txt | ./<executable> <cache size>
```

Ex: `$cat numbers.txt | ./lru 500`

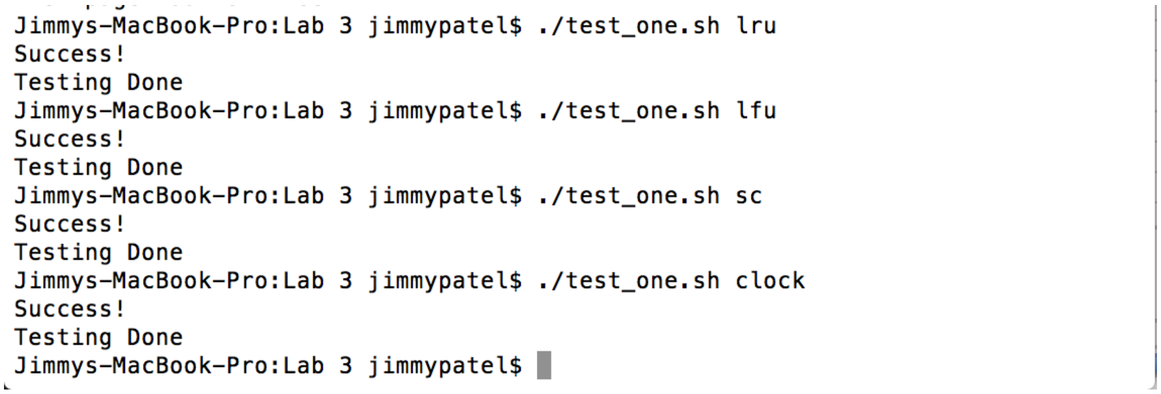
To run with your bash script you will need to run your script four times, one for each of the page replacement algorithms.

```
$ ./test_one.sh <executable>
```

Ex: `$ ./test_one.sh lru`

Demo:

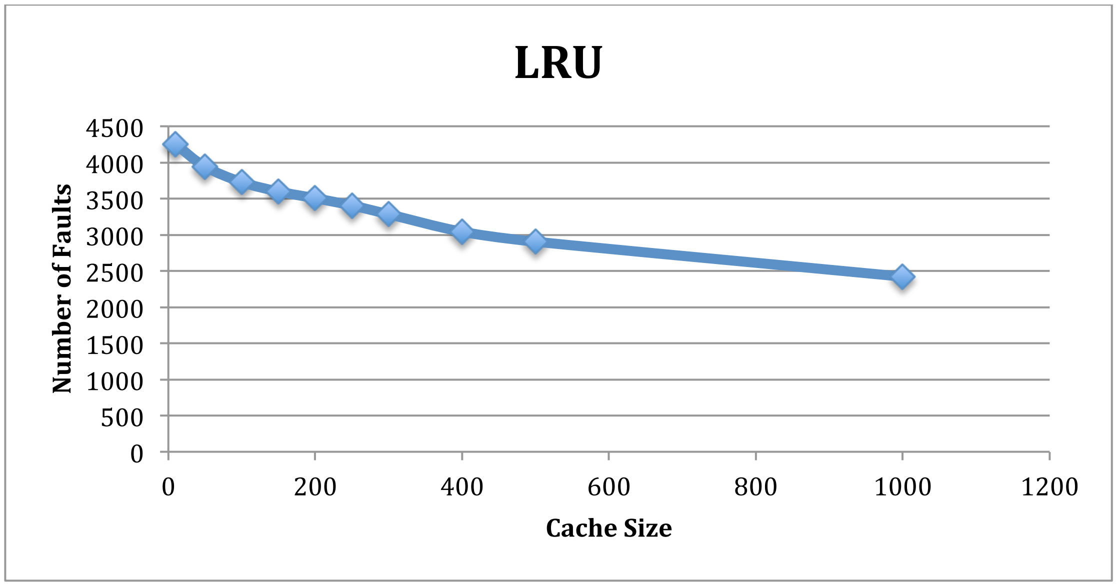
Here is a screenshot of the results for the lab. I am sorry that I was not able to come to lab this week but here is a screenshot and you can run the program yourself if you would like to for further proof.



```
Jimmys-MacBook-Pro:Lab 3 jimmypatel$ ./test_one.sh lru
Success!
Testing Done
Jimmys-MacBook-Pro:Lab 3 jimmypatel$ ./test_one.sh lfu
Success!
Testing Done
Jimmys-MacBook-Pro:Lab 3 jimmypatel$ ./test_one.sh sc
Success!
Testing Done
Jimmys-MacBook-Pro:Lab 3 jimmypatel$ ./test_one.sh clock
Success!
Testing Done
Jimmys-MacBook-Pro:Lab 3 jimmypatel$
```

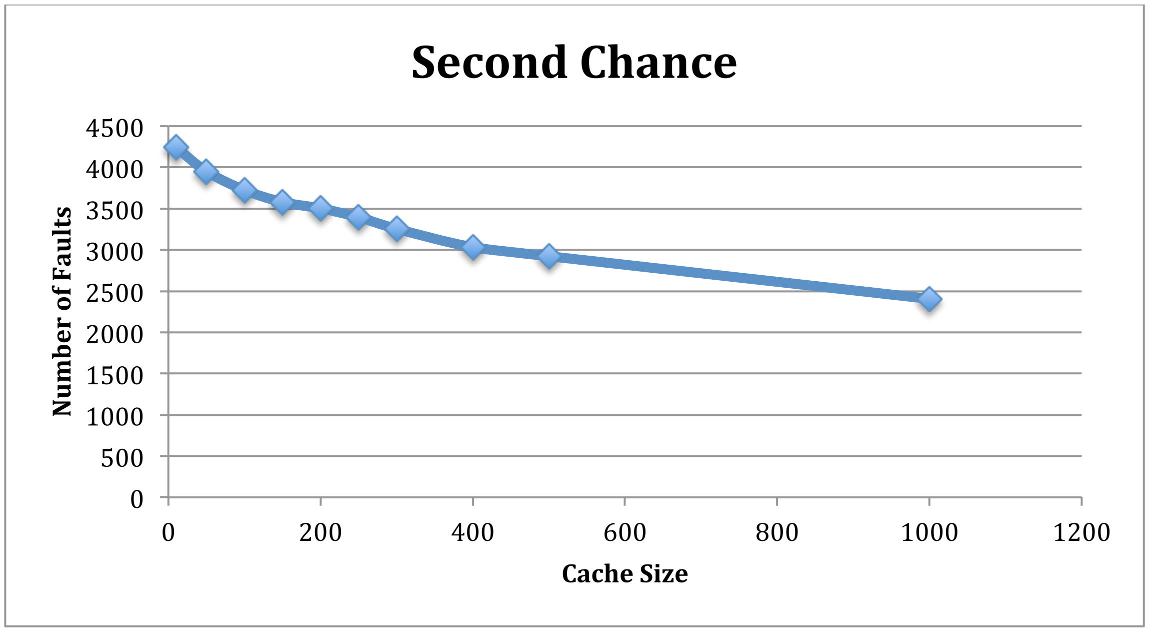
LRU:

For my LRU page replacement algorithm, I used a doubly LinkedList with a head pointer where the nodes contained one integer holding the value read from standard input. In this algorithm, I make a node for every distinct value read from standard input and add it to my LinkedList. In doing so, in every search, I can detect that a page fault occurred if the count of node found is greater than the linked list cache size or if the node was not found. In my LinkedList, the most recently used pages are at the front of the list and the older ones are propagated towards the rear of the list. If a page is not found in my LinkedList, I add it to the front. On the other hand, if it is found, I reassign the pointers of the previous and next node before making the matching node the new head of the list.



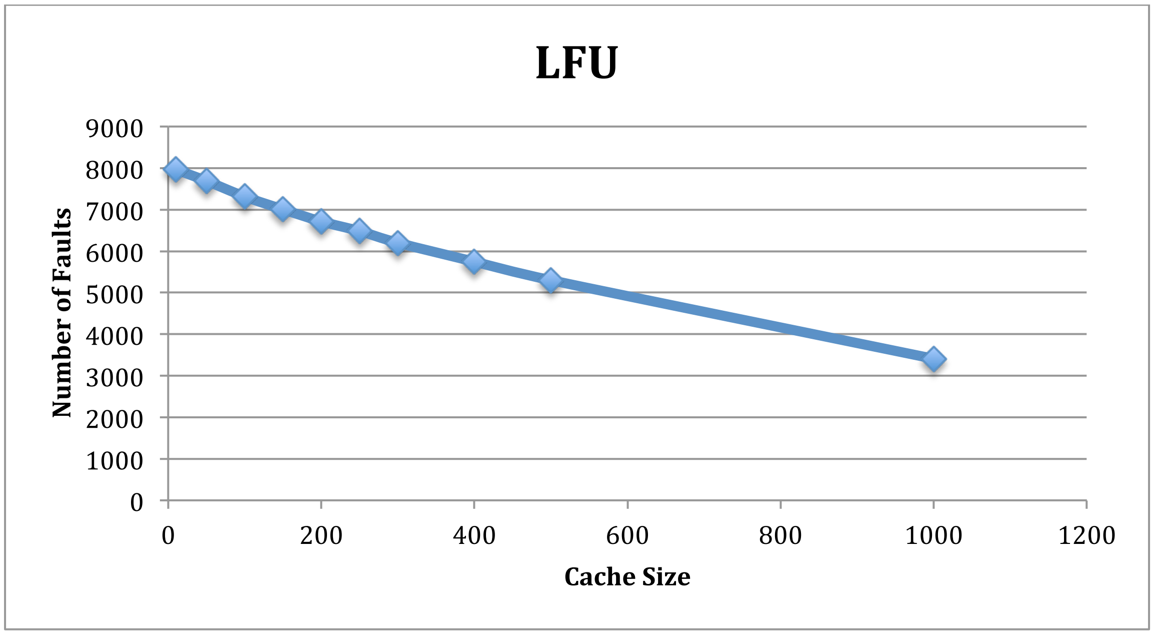
Second Chance:

For my Second Chance page replacement algorithm, I used a doubly LinkedList with a head and tail pointer. The nodes in this list contained the value of the page as well as a separate integer for keeping track of the reference bit. In this implementation, the number of nodes in my LinkedList will never exceed the max cache size. New pages get inserted towards the rear of the list and the older pages are towards the front. On each search of the LinkedList from head to tail, if the node is not in the list, all nodes with a reference bit of 1 are set to 0 and moved towards the rear of the list. Once the search sees a reference bit of 0, the new page you want to insert replaces that page. If the page was already in the LinkedList, that node's reference bit is set to 1 and all other reference bits are left unchanged. A page fault will occur whenever you search for a page number and it does not exist in the list.



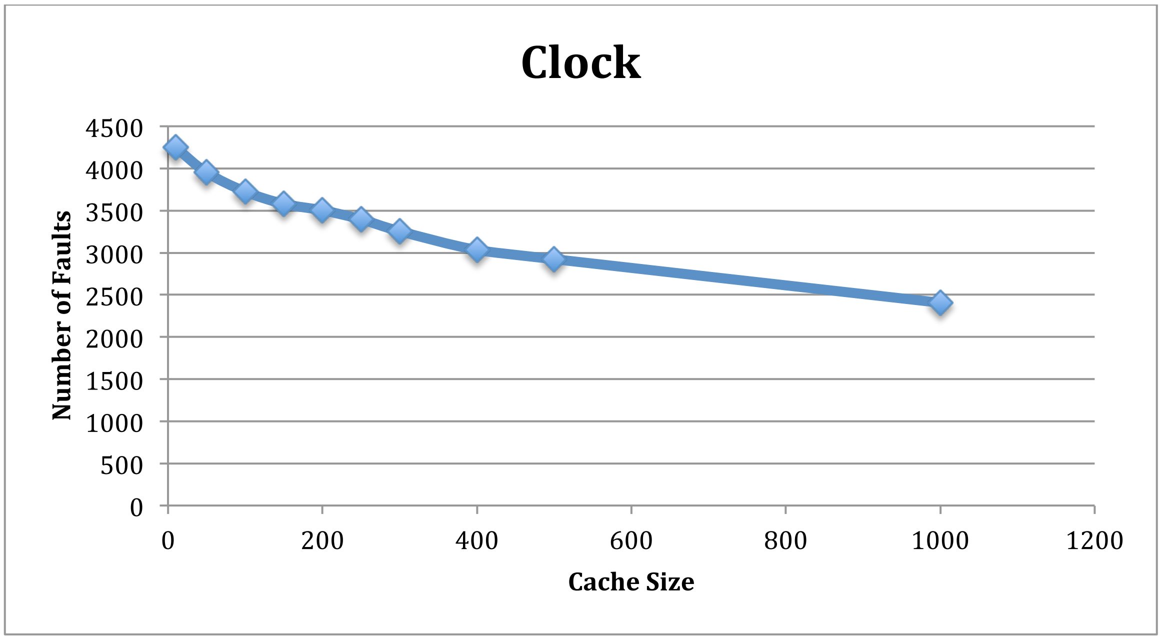
LFU:

For my LFU page replacement algorithm, I used a doubly LinkedList with a head and tail pointer. Each node in my LinkedList contained the value of the page and the frequency that the page had been accessed. The LinkedList was structured so that the latest pages requested would be towards the tail of the list and the older pages would be at the front of the list. This helps differentiate two pages in the event that you need to remove one and. The algorithm was straightforward. If the page requested was not in my linked list, a page fault would occur and the oldest page with smallest number of requests would be deleted before adding the new page you just requested at the tail. If a page you requested was in my data structure, I would remove it from the list, add it to the tail, and increment the frequency of that node being requested.



Clock:

Like all my other page replacement algorithms, I used a double LinkedList with a head and tail pointer. Because I did not want to make too many changes to the code, I did not change the LinkedList to be circular. I overcame this by resetting the hand to be the head of the list after it passed the tail. Whenever a page fault occurs, the hand of the clock cycles through to find the first node with the reference bit equal to 0, while decrementing the reference bit if it was 1. After I found this node, I reset its contents, then moved the clock hand to the next node.



Here are the Data Points for the graphs above:

LRU	LRU Faults	LFU	LFU Faults	SC	SC Faults	CLOCK	Clock Faults
10	4248	10	7963	10	4246	10	4246
50	3942	50	7690	50	3952	50	3952
100	3727	100	7313	100	3725	100	3725
150	3598	150	7005	150	3579	150	3579
200	3505	200	6714	200	3503	200	3503
250	3405	250	6483	250	3396	250	3396
300	3284	300	6201	300	3255	300	3255
400	3041	400	5746	400	3032	400	3032
500	2904	500	5300	500	2923	500	2923
1000	2421	1000	3409	1000	2406	1000	2406

Results Discussion:

Based on the plots above, we can definitively say that the page replacement algorithm with the most number of page faults was Least Frequently Used. This is because once a page gathers a lot of page requests; it is essentially stuck in the LinkedList wasting the space of a node until there is another node to get more requests. Observing the pages in memory by printing everything in the LinkedList proved this because there was a point where only the last node in the cache was being replaced because the rest of the pages were requested a lot earlier on. Second Chance and Clock were really good in terms on minimizing page faults. Because it is essentially the same algorithm with a slight modification, the graphs were identical. LRU was also a very good page replacement algorithm with slightly more page faults in the tests above (34075 vs 34017). Despite the difference in the number of page faults based on a cache size, the graphs show that each page replacement algorithm preforms better as the cache size increases.

Although, it is not a linear relationship between the cache size and number of page faults, increasing the cache did decrease the number of page faults.

To improve the performance of the page replacement algorithms, we can increase the cache size. Have a cache size of 10 lead to a large number of page faults and increasing the size decreased the number of page faults a significant amount for each of the page replacement algorithms. We can also assign priorities to the pages, for example, if the page is modified, that page will likely be more important than non-modified pages (dirty bit). So given the option, the page with the unmodified dirty bit should be thrown out so if we combine that with the Second Chance algorithm, I would expect to see the number of page faults decrease.. Not only that, but I think we to further improve the algorithms; we will need to get one step closer to the optimal page replacement algorithm.