Gabriel Borges

Yufeng Wu

CSE 3500 – Programming Assignment

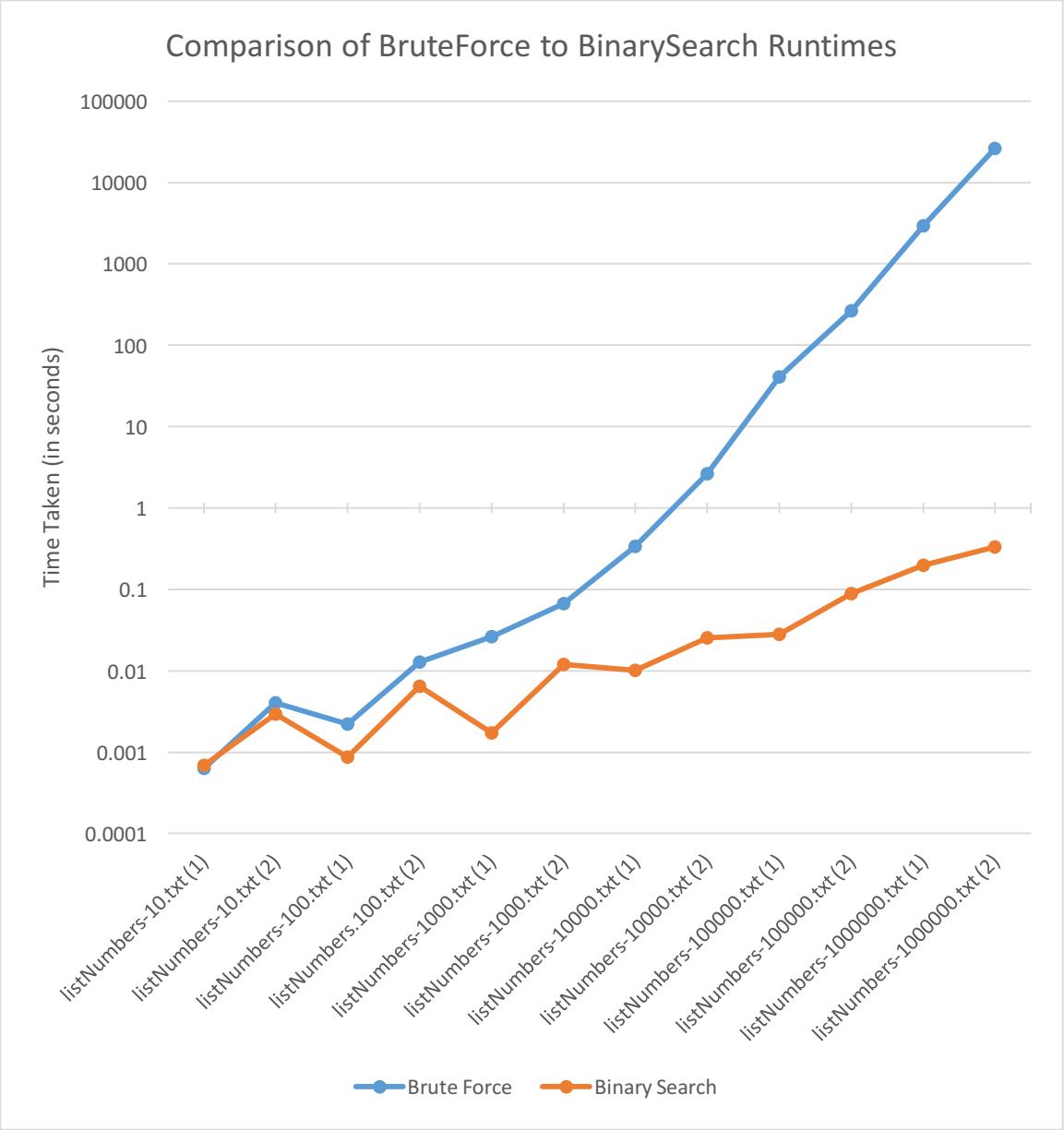
3/23/16

Programming Assignment Write-Up

Settings: For this assignment, I'm running the program on my MacBook Pro, with a 2.4Ghz i5 and 8GB 1600Mhz RAM.

Results:

FileName	*BruteForce Runtime (in seconds)*	*BinarySearch Runetime (in seconds)*
listNumbers-10.txt (1)	6.37199E-4	6.82965E-4
listNumbers-10.txt (2)	0.00401796	0.002959525
listNumbers-100.txt (1)	0.002218045	8.74264E-4
listNumbers.100.txt (2)	0.01269465	0.006410437
listNumbers-1000.txt (1)	0.026166509	0.001715684
listNumbers-1000.txt (2)	0.067301876	0.011900116
listNumbers-10000.txt (1)	0.335328108	0.010119315
listNumbers-10000.txt (2)	2.629364564	0.025217425
listNumbers-100000.txt (1)	41.075098774	0.028160041
listNumbers-100000.txt (2)	265.087019675	0.088976959
listNumbers-1000000.txt (1)	2944.801408131	0.198014289
listNumbers-1000000.txt (2)	25950.726119835	0.329783723



Conclusion:

The choice of algorithm becomes painfully clear as the sample size for a given set of data becomes larger. Although the difference in time is apparent for smaller sample sets, the perceived time elapsed is not particularly large. However, when the program is run for the sets of 100,000 and 1,000,000 sized lists, the time difference between the bruteforce algorithm and the binarysearch algorithm become more apparent, and the faster algorithm becomes the only logical option.

An interesting observation is that for small data sets, the algorithm choice does not seem to matter if the input size is also small. Looking at the difference between, as an example, the numberset 1 runtime to the numberset2 runtime for the sample size of 10, the only reason that the binarySearch algorithm runs faster is the new numberset has a significantly larger amount of inputs to check (i.e. 100 vs. 10). If the number set was still small, as it is for the first run through (listNumbers-10.txt (1)), the brute force algorithm is actually more efficient (although this may have been due to the luck of the arrangement). In both sample sets, however, as the amount of data becomes larger, the binarysearch algorithm becomes painstakingly more efficient to use. For example, the first data set for 1,000,000 numbers to check, the bruteForce algorithm took approximately 49 minutes to complete versus .2 seconds. There's not a question that binary search is more efficient.

Code: Source Code is included in submission as .java file. GitHub public repository for the project:

https://github.com/gborges0727/CSE3500-ProgrammingAssignment

Example Output (for listNumbers-1000000.txt (1)):

```
[gborges0727 (master *) programming assignment $ java NumberSearch
2780164 and 11655 add to 2791819 and exist in the list.
122138 and 856916 add to 979054 and exist in the list.
4120185 and 2485437 add to 6605622 and exist in the list.
3782766 and 8014345 add to 11797111 and exist in the list.
2202823 and 4864756 add to 7067579 and exist in the list.
549387 and 8665703 add to 9215090 and exist in the list.
1399218 and 8345199 add to 9744417 and exist in the list.
1444105 and 2321703 add to 3765808 and exist in the list.
292450 and 3673025 add to 3965475 and exist in the list.
3457205 and 457847 add to 3915052 and exist in the list.
Run time for bruteForce = 2944801408131 in nano-seconds or 2944.801408131 seconds

35 and 2791784 add to 2791819 and exist in the list.
370 and 978684 add to 979054 and exist in the list.
349 and 6605273 add to 6605622 and exist in the list.
1797188 and 9999923 add to 11797111 and exist in the list.
23 and 7067556 add to 7067579 and exist in the list.
35 and 9215055 add to 9215090 and exist in the list.
23 and 9744394 add to 9744417 and exist in the list.
358 and 3765450 add to 3765808 and exist in the list.
113 and 3965362 add to 3965475 and exist in the list.
186 and 3914866 add to 3915052 and exist in the list.
Run time for binarySearch = 198014289 in nano-seconds or 0.198014289 seconds
gborges0727 (master *) programming assignment $
```