Text

Description automatically generated with low confidence

**PROGRAM STRUCTURES & ALGORITHMS**

**INFO – 6205**

Husky Benchmarking

Final Project

[GitHub](https://github.com/sydrawat01/HuskyBenchmark) | [Excel Data](https://docs.google.com/spreadsheets/d/11-Am0QORMGr_euAYKQHqkk0UU7CHl2WWpuGxGJjo50s/)

**Siddharth Rawat – 002963295**

**Sumeet Deshpande – 002929388**

**Yash Firke – 002928176**

# 1.0 Quicksort

Do you agree that the number of swaps in “standard” quicksort is 1/6 times the number of comparisons? Is this figure correct? If not, why not. How do you explain it?

# 1.1 Approach:

While performing this task, we started by understanding that the number of swaps is

~ and the number of compares is ~ for Quick Sort, according to the lecture notes. We started with the Test-Driven Development (TDD), and created several unit tests, based on which we create the code for the Quick Sort algorithm. We do not cutoff to insertion sort in this approach because the number of swaps and compares would vary due to this. Also, the Benchmarking of the code for a data set varying from size 27 to 217 using doubling method for 1000 runs was performed. We have re-used the code from the class repository as well as the Husky sort repository and made necessary modifications to achieve the results.

# 1.2 Unit Test Cases:

Text

Description automatically generated

# 1.3 Output:

Text

Description automatically generated

The output can be found [here](https://github.com/sydrawat01/HuskyBenchmark/blob/master/logs/quickSortBasicBenchmark.log).

# 1.4 Observations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | No Of Swaps | No of Compare | Actual | Expected |
| 128 | 193 | 1125 | 0.172 | 0.167 |
| 256 | 427 | 2430 | 0.18 | 0.167 |
| 512 | 1033 | 5764 | 0.179 | 0.167 |
| 1024 | 2296 | 12911 | 0.178 | 0.167 |
| 2048 | 4972 | 29951 | 0.166 | 0.167 |
| 4096 | 11079 | 61334 | 0.181 | 0.167 |
| 8192 | 23955 | 133795 | 0.179 | 0.167 |
| 16384 | 50967 | 305105 | 0.167 | 0.167 |
| 32768 | 111437 | 630297 | 0.177 | 0.167 |
| 65536 | 235693 | 1427749 | 0.165 | 0.167 |
| 131072 | 503438 | 2915313 | 0.173 | 0.167 |
| 262144 | 1071901 | 6335320 | 0.169 | 0.167 |

Chart, line chart

Description automatically generated

The evidence can be found [here](https://docs.google.com/spreadsheets/d/11-Am0QORMGr_euAYKQHqkk0UU7CHl2WWpuGxGJjo50s/edit#gid=1640931067).

# 1.5 Conclusion:

From the graph and the observations, it can be seen that the ratio of number of swaps to number of compares for a given size of array is approximately 0.1667.

Hence it can be concluded that the number of swaps in “standard” quicksort is approximately times the number of comparisons.

# 2.0 Hibbard deletion of BST

According to the course lecture notes, after a number of (Hibbard) deletions have been made, the average height of the tree is. Do you agree with this? How does it look after modifying the deletion process to either :

(a) randomly choose which direction to look for the node to be deleted, or

(b) choose the direction according to the size of the candidate nodes.

# 2.1 Approach:

We started by creating our own implementation of Binary Search Tree referencing Sedgewick and Wayne. The BinarySearchTree implements an API called BST which accepts a Key-Value pair and has the following functions:

1. get()
2. put()
3. delete()
4. size()
5. height()
6. putAll()

Our approach here was to create a delete function which works well with different deletion strategies. Using TDD approach we created various test cases that checks the BST created for its height, size and the InOrder Traversal of the same.

For random number of deletions that have been made on the BST, we performed benchmarking where the size of the BST varies from 20 to 1600 nodes and are deleting N/2 elements at random. We performed this experiment for 100 iterations and the results are written into CSV file.

To perform the various deletion strategies on the BST we define constants viz., “Hibbard”, “Leftist” and “SizedDeletion”.

# 2.2 Unit Test Cases:

Text

Description automatically generated

# 2.3 Output:

The output can be found [here](https://github.com/sydrawat01/HuskyBenchmark/tree/master/data) for all three cases.

# 2.4 Observations:

The excel sheet with observations can be found here:

1. [BST Random Deletion](https://docs.google.com/spreadsheets/d/11-Am0QORMGr_euAYKQHqkk0UU7CHl2WWpuGxGJjo50s/edit#gid=1656416649)
2. [BST Hibbard Deletion](https://docs.google.com/spreadsheets/d/11-Am0QORMGr_euAYKQHqkk0UU7CHl2WWpuGxGJjo50s/edit#gid=512705678)
3. [BST Sized Deletion](https://docs.google.com/spreadsheets/d/11-Am0QORMGr_euAYKQHqkk0UU7CHl2WWpuGxGJjo50s/edit#gid=501116321)

# 2.4.1 Hibbard Deletion:

Chart, line chart

Description automatically generated

# 2.4.2 Random Deletion:

Random deletion of nodes from BST either by Leftist or Rightist(Hibbard) Strategy.

Chart, line chart

Description automatically generated

# 2.4.3 Sized Deletion:

Deletion of nodes from BST by comparing the size of the candidate nodes.

Chart, line chart

Description automatically generated

# 2.5 Conclusion:

We performed random deletion of nodes from BST using 3 strategies viz., Hibbard, Random(Hibbard or Leftist) and Sized Deletion.

We delete N/2 random nodes from BST using Hibbard’s deletion for a BST whose size varies from 20 to 1600. We observed that the trees with size between 300 and 700 nodes, the height of the tree tends to root n after the deletions have been made.

Similarly, when we performed deletion on BST using Random and SizedDeletion. We could observe that they follow the same trend as Hibbard’s deletion, where the height of the tree tends to root n

Hence based on our observations, we conclude that the height of BST after N/2 random deletions is approximately proportional to root n and it does not depend upon the strategy used for deletion.

# 3.0 ShellSort

A. When shell-sorting a sorted array, the number of comparisons is logarithmic. Do you agree? What is the base of the logs? Explain your observations.   
B. Can you develop an expression for the average number of comparisons for shell-sort (choose any one of the “common” gap sequences). Recall that this is “unknown.”

C. Can you develop an expression for the worst-case number of comparisons for the gap sequence you choose?

# 3.1 Approach:

Using the TDD approach, we initially created unit tests for shell sort. For this algorithm, we follow Knuth’s sequence to generate the gap(h). Knuth’s sequence is

h=3\*h+1

After creating the unit test cases for the above algorithm, we created the algorithm such that it passes all the unit tests.

To run benchmarking, we use array size varying from 5000 to 30000 and used the following loggers for logging the results:

1. Benchmark
2. TimeLogger
3. LazyLogger

# 3.2 Unit Test Cases:

Text

Description automatically generated

# 3.3 Output :

The output can be found [here](https://github.com/sydrawat01/HuskyBenchmark/blob/master/logs/ShellSortBenchMark.log).

# 3.4 Observations :

3.4.1 Best Case:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | h | | | | | | | |
| N | 1 | 4 | 13 | 40 | 121 | 364 | 1093 | 3280 |
| 5000 | 4999 | 9995 | 14982 | 19942 | 24821 | 29457 | 33364 | 35084 |
| 10000 | 9999 | 19995 | 29982 | 39942 | 49821 | 59457 | 68364 | 75243 |
| 15000 | 14999 | 29995 | 44982 | 59942 | 74821 | 89457 | 103364 | 115084 |
| 20000 | 19999 | 39995 | 59982 | 79942 | 99821 | 119457 | 138364 | 155084 |
| 25000 | 24999 | 49995 | 74982 | 99942 | 124821 | 149457 | 173364 | 195084 |
| 30000 | 29999 | 59995 | 89982 | 119942 | 149821 | 179457 | 208364 | 235084 |

3.4.2 Average Case:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | h | | | | | | | |
| N | 1 | 4 | 13 | 40 | 121 | 364 | 1093 | 3280 |
| 5000 | 6273560 | 1663385 | 584636 | 234546 | 131735 | 104868 | 105042 | 101896 |
| 10000 | 24627616 | 6548882 | 2139477 | 822933 | 397670 | 284586 | 238656 | 244578 |
| 15000 | 56017117 | 14839555 | 4670372 | 1734309 | 775092 | 485038 | 429945 | 387519 |
| 20000 | 1E+08 | 25714413 | 8420446 | 2973827 | 1278703 | 736322 | 587467 | 529751 |
| 25000 | 1.57E+08 | 40863541 | 12778699 | 4545871 | 1863261 | 1049604 | 782282 | 715562 |
| 30000 | 2.24E+08 | 57589929 | 18361852 | 6474799 | 2577899 | 1435103 | 1069223 | 901767 |

3.4.2 Worst Case:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | h | | | | | | | |
| 1 | 4 | 13 | 40 | 121 | 364 | 1093 | 3280 |
| 5000 | 12497500 | 3134996 | 976724 | 341449 | 137727 | 72502 | 53716 | 51491 |
| 10000 | 49995000 | 12519996 | 3873453 | 1307949 | 480870 | 208230 | 142610 | 121986 |
| 15000 | 1.12E+08 | 28154996 | 8699412 | 2899449 | 1032672 | 421887 | 231004 | 183374 |
| 20000 | 2E+08 | 50039838 | 15436042 | 5115809 | 1774900 | 703838 | 373467 | 278113 |
| 25000 | 3.12E+08 | 78174761 | 24133406 | 7957245 | 2731444 | 1054041 | 521758 | 350928 |
| 30000 | 4.5E+08 | 1.13E+08 | 34701677 | 11423659 | 3905920 | 1453038 | 709463 | 458992 |

# 3.5 Conclusion:

The algorithm runs for 1000 iterations, 3 times:

1. Sorted Array – We observed that the number of comparisons is logarithmic and the base for the log is “e”.

Number of compares ≈

where N is array size and h>1

For h=1 the number of compares is N-1

1. Average Case – Using Knuth’s sequence for shell sort, we could not develop an expression for average case. This is still unknown.

For h=1, Number of Compares is ¼(N^2 - N)

1. Worst Case – Following the same sequence we used above we could develop an expression for worst case where the array is sorted in reverse

Number of compares ≈

where N is array size and h>1

For h=1, Number of Compares is ½(N^2 - N)

# 4.0 General Benchmarking

Run benchmarks on at least five independent algorithms and analyze the numbers of array accesses. To what extent can you predict the execution time based solely on the number of array accesses?

# 4.1 Approach:

Using the TDD approach, we initially created unit tests for following independent algorithms:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Quick Sort
6. Shell Sort

After creating the unit test cases for the above algorithms, we created the algorithms such that they pass all respective unit tests.

To perform benchmarking on these algorithms, we created specific benchmarking files that run for at least 1000 times for varying values of N, where N is the size of the array.

For logging the benchmarking results, we use the following classes:

1. Benchmark
2. TimeLogger
3. LazyLogger

All of the above logs are automatically written into “logs/HuskyBenchmark.log”

# Unit Test Cases:

4.2.1 QuickSort

Text

Description automatically generated

4.2.2 InsertionSort

Text

Description automatically generated

4.2.3 ShellSort

Text

Description automatically generated

4.2.4 SelectionSort

Text

Description automatically generated

4.2.5 MergeSort

Text

Description automatically generated

4.2.6 BubbleSort

Text

Description automatically generated

# 4.3 Output:

The output can be found [here](https://github.com/sydrawat01/HuskyBenchmark/tree/master/logs).

# 4.4 Observations:

# 4.4.1 Bubble Sort:

|  |  |  |
| --- | --- | --- |
| **N** | **Hits** | **Time(ms)** |
| 128 | 24022 | 0.03 |
| 256 | 99052 | 0.09 |
| 512 | 395410 | 0.35 |
| 1024 | 1572996 | 1.3 |
| 2048 | 6278336 | 6.73 |
| 4096 | 25055602 | 22.47 |
| 8192 | 100523892 | 90.99 |
| 16384 | 403668890 | 496.96 |
| 32768 | 1609361800 | 2416.25 |

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

4.4.2 Merge Sort:

|  |  |  |
| --- | --- | --- |
| **N** | **Hits** | **Time(ms)** |
| 128 | 3064 | 0.1 |
| 256 | 7206 | 0.07 |
| 512 | 16440 | 0.59 |
| 1024 | 36922 | 0.48 |
| 2048 | 81708 | 1.08 |
| 4096 | 180164 | 2.52 |
| 8192 | 393084 | 6.19 |
| 16384 | 851296 | 25.09 |
| 32768 | 1833784 | 98.12 |

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

4.4.3 Insertion Sort:

|  |  |  |
| --- | --- | --- |
| **N** | **Hits** | **Time(ms)** |
| 128 | 16848 | 0.03 |
| 256 | 68910 | 0.06 |
| 512 | 260288 | 0.22 |
| 1024 | 1046364 | 0.9 |
| 2048 | 4135626 | 4.3 |
| 4096 | 16895392 | 16.38 |
| 8192 | 67388144 | 64.61 |
| 16384 | 267453436 | 254.77 |
| 32768 | 1073606938 | 1066.19 |

# Chart, line chart Description automatically generatedChart, line chart Description automatically generated

4.4.4 Selection Sort:

|  |  |  |
| --- | --- | --- |
| **N** | **Hits** | **Time(ms)** |
| 128 | 492 | 0.06 |
| 256 | 996 | 0.07 |
| 512 | 2020 | 0.27 |
| 1024 | 4068 | 1.05 |
| 2048 | 8160 | 3.96 |
| 4096 | 16356 | 11.38 |
| 8192 | 32716 | 33.5 |
| 16384 | 65484 | 133.12 |
| 32768 | 131032 | 533.77 |

# Chart, line chart Description automatically generatedChart, line chart Description automatically generated

# 4.4.5 Shell Sort

|  |  |  |
| --- | --- | --- |
| **N** | **Hits** | **Time(ms)** |
| 5000 | 479728 | 0.71 |
| 10000 | 1184372 | 1.71 |
| 15000 | 1886570 | 2.85 |
| 20000 | 2583810 | 3.94 |
| 25000 | 3541344 | 5.23 |
| 30000 | 4501166 | 6.57 |

# Chart, line chart Description automatically generatedChart, line chart Description automatically generated

# 4.4.6 Quick Sort

|  |  |  |
| --- | --- | --- |
| **N** | **Time(ms)** | **Hits** |
| 128 | 0.18 | 772 |
| 256 | 0.49 | 1748 |
| 512 | 0.35 | 4132 |
| 1024 | 0.37 | 9184 |
| 2048 | 0.43 | 19888 |
| 4096 | 0.92 | 44356 |
| 8192 | 1.98 | 95820 |
| 16384 | 3.12 | 203868 |
| 32768 | 6.28 | 445748 |
| 65536 | 13.59 | 942772 |
| 131072 | 36.64 | 2013752 |
| 262144 | 73 | 4287604 |

# Chart, line chart Description automatically generatedChart, line chart Description automatically generated

# 4.5 Conclusion:

In the quick sort algorithm, we vary the array size from 128 to 262144 and for shell sort, from 5000 to 30000. For other sorting algorithms, the size varies from 128 to 32768.

From the above graphs we can observe that the number of hits and the time taken to execute the algorithm increases linearly with respect to the varying array sizes.

# 5.0 CI/CD Pipeline-

We have implemented CircleCI for continuous integration that checks all the unit tests before merging any code into the master branch. The dashboard can be found [here](https://app.circleci.com/pipelines/github/sydrawat01/HuskyBenchmark?branch=master).