Northeastern University

Program Structures and Algorithms

INFO6205 Final Project

Summer-2022

Under the Supervision of

Prof. Robin Hillyard

Team Members

Dimpleben Kanjibhai Patel – 002965372

Pratik Hasmukh Hariya – 002929941

Shubham Rajan Patil – 002928545

Table of Content

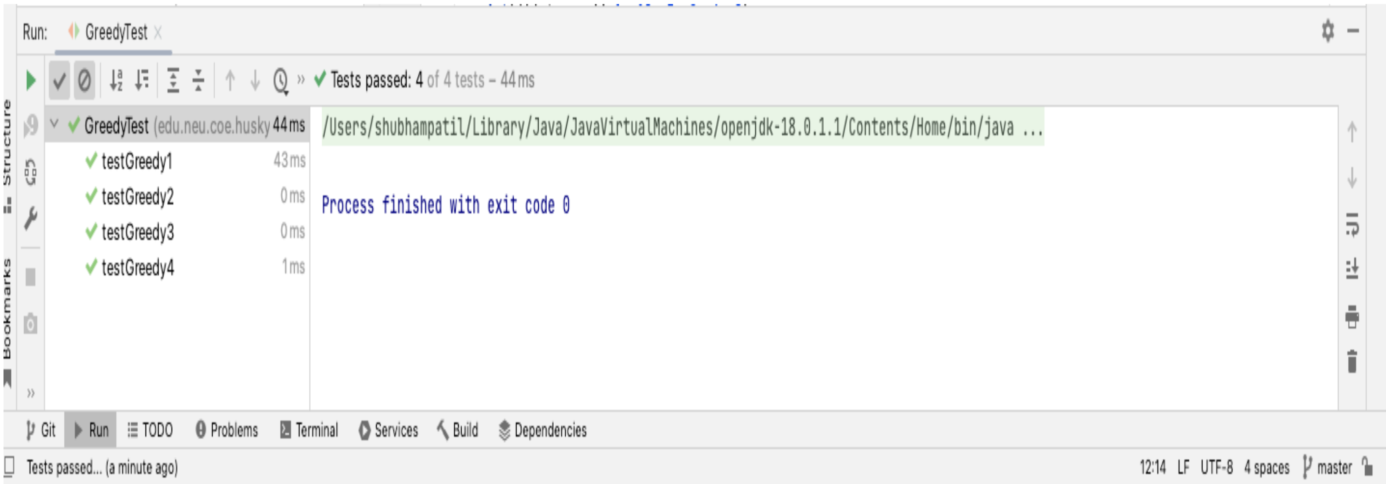
Task-1 Traveling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city. Do the benchmarking for the shortest possible route using Greedy, Genetic Algorithm and Genetic-Greedy Algorithm.

Solution Using Greedy Approach

We Started with the greedy algorithm to find the shortest path. The starting city was selected randomly. As it always makes the best decision at the time, it will choose the next city with the shortest route. Thus it will give locally optimal solutions. We referred to GeeksforGeeks for the Traveling Salesman Problem using the Greedy Algorithm and modified the algorithm to reach the starting city again after visiting all the given cities which it was choosing randomly [1].

Unit Test Output



```
✓ Tests passed: 4 of 4 tests – 44 ms

✓ GreedyTest (edu.neu.coe.husky 44 ms
    ✓ testGreedy1   43 ms
    ✓ testGreedy2    0 ms
    ✓ testGreedy3    0 ms
    ✓ testGreedy4    1 ms
```

```
/Users/shubhampatil/Library/Java/JavaVirtualMachines/openjdk-18.0.1.1/Contents/Home/bin/java ...


Process finished with exit code 0
```

```
Tests passed... (a minute ago)                    12:14  LF  UTF-8  4 spaces  master
```

Observation

Array Size	Greedy Minimum Cost
4	268
8	199
16	342
32	406
64	309
128	477

256	549
512	413
1024	474
2048	445
4096	371
8192	518
16384	446

We observed that for a small number of cities, the Greedy algorithm gives a near optimal solution. As the number of cities increases, it fails to find the globally optimal solution because the greedy algorithm only has one chance to compute the optimal solution, it never reverts and alters the decision.

Solution Using Genetic Algorithm

This algorithm takes into account the natural selection process, in which the best candidates are chosen to produce the offspring in the following generation. We have a fitness value for each of these solutions, and chromosomes for the following generations are constructed based on the fitness value [2].

Chromosome Representation

The size of the chromosome is equal to the total number of cities. Each gene of the chromosome represents the city and each chromosome represents the possible route. The Chromosome representation scheme is as shown below.

1	4	7	3	2	9	6	8	5	10

Initial Population

The initial population of the chromosome is generated by randomly selecting a distinct city. We have considered the initial population size as 5000.

Fitness Function

Fitness function calculates the cost of the path described by each chromosome.

Selection Function

We used the Tournament selection method and Roulette selection method of the Genetic Algorithm for the Traveling Salesman problem. It is observed that the

Tournament selection method outperforms roulette selection method. Hence, we chose the Tournament selection method for our algorithm.

Crossover Method

Partially Mapped Crossover technique is used for crossover. This method randomly chooses one crossover point and swaps the elements within them.

Parent1

3	4	1	7	5	8	2	6

Parent2

6	2	8	7	5	4	3	1

After performing crossover, the child will be,

6	2	8	7	5	1	4	3

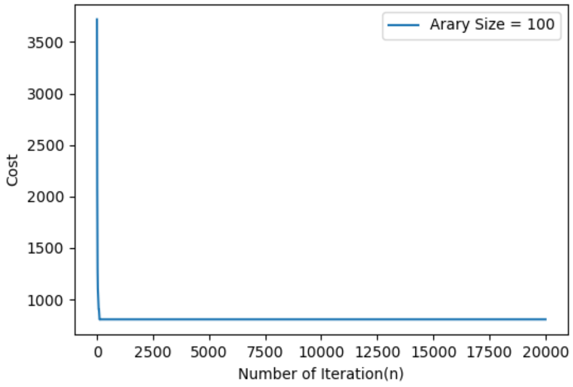
Here, we swap i^{th} element of parent1 with the equivalent element value to the i^{th} element of parent2.

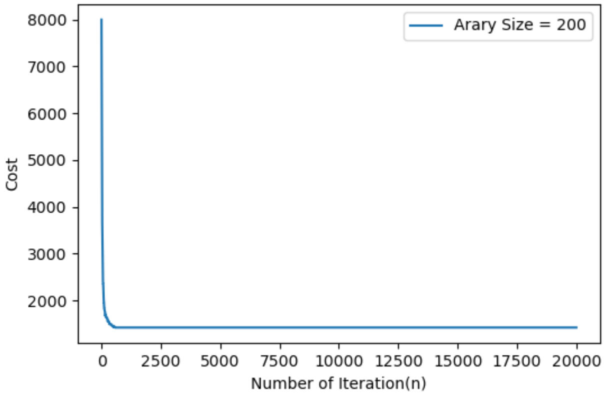
Mutation Method

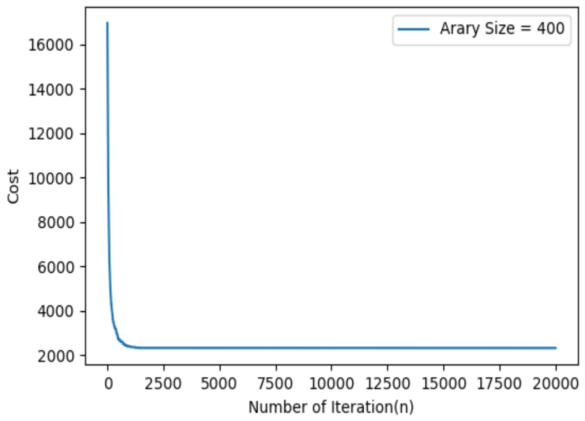
We considered the mutation rate as 0.1. If the randomly generated value is less than mutation rate then we will swap two genes in the chromosome randomly.

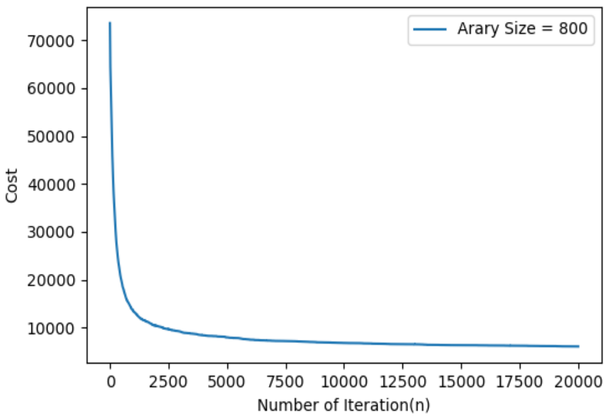
Stopping Condition

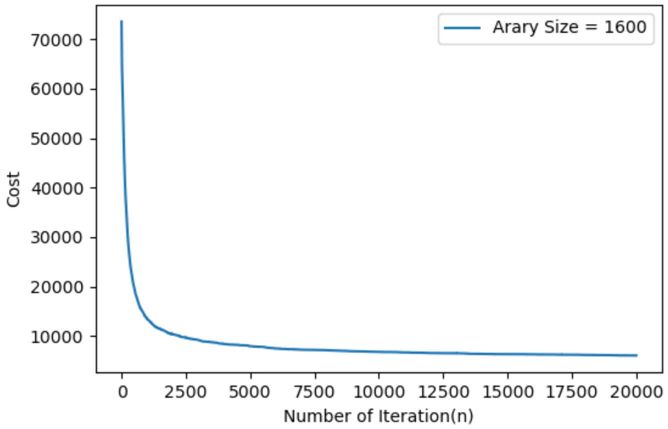
To decide the stopping condition, the experiment is performed by varying array size as 100, 200, 400, 800 and 1600 and performed 20000 iterations. The result of the experiment is as below.











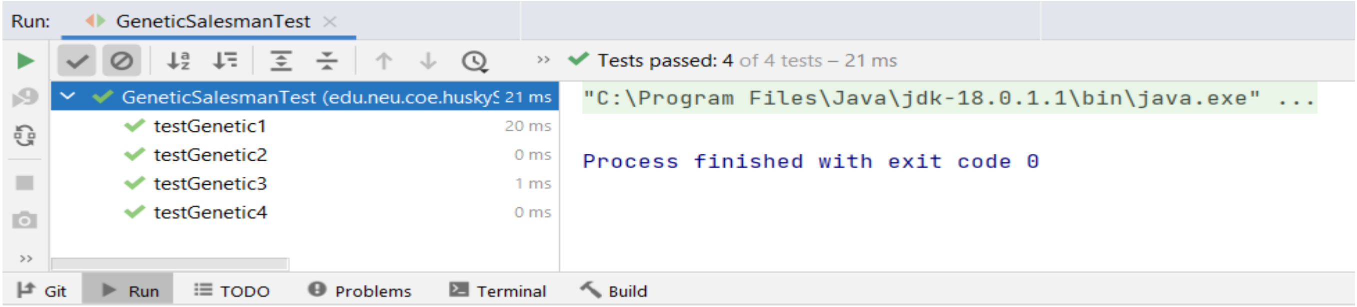
From the above simulations, it is observed that after approximately 5000 iterations, the cost of the path remains constant. Thus we fix 5000 iterations for the genetic algorithm.

Parameter Considered

Parameters Considered	**Value Considered**
Population Size	5000
Number of Iteration	5000
Selection Method	Tournament Selection Method
Size of Chromosome	Number of Cities
Crossover Method	Partially Mapped Crossover
Mutation Rate	0.1
Fitness Function	Calculate the Path Length

Unit Test

We have run unit tests for fitness function.



```
Run:   GeneticSalesmanTest ×
Tests passed: 4 of 4 tests – 21 ms
GeneticSalesmanTest (edu.neu.coe.huskyS 21 ms
    testGenetic1    20 ms
    testGenetic2     0 ms
    testGenetic3     1 ms
    testGenetic4     0 ms

"C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe" ...

Process finished with exit code 0
```

Observation

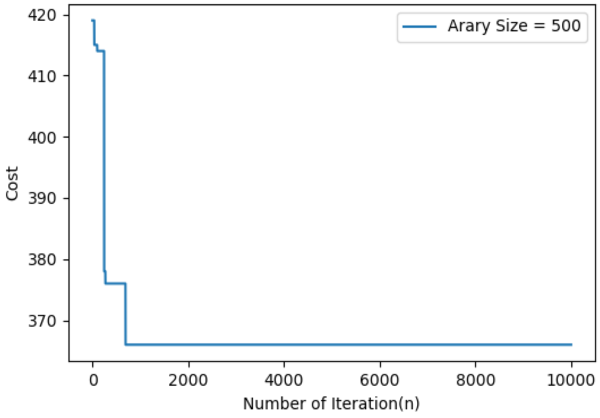
We compared the result of the Genetic Algorithm with the Greedy algorithm and found that the genetic algorithm was not performing better and it was taking a lot of time to even reach the solution of the Greedy approach. Thus we combine Genetic algorithm with Greedy approach.

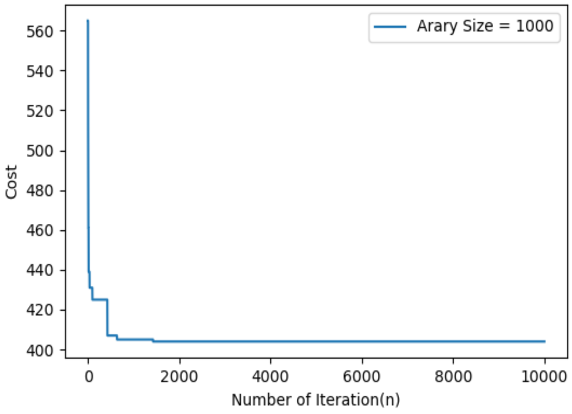
Solution Using Genetic-Greedy Algorithm

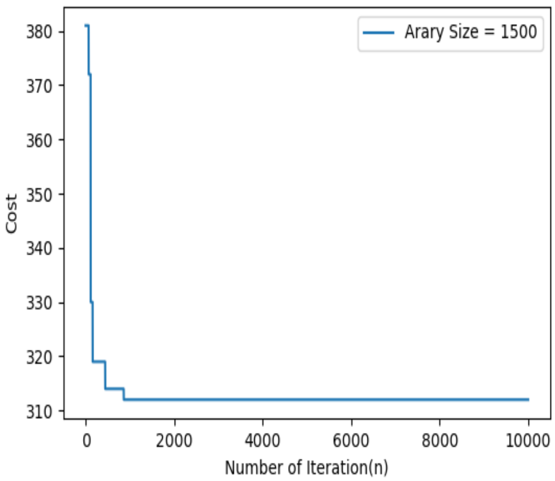
In Genetic-Greedy Algorithm, the change we made in the genetic algorithm is while generating the initial population. We took the output of the greedy algorithm as one of the chromosomes for the genetic algorithm and then performed the simulation. All the parameters considered for Genetic-Greedy Algorithm are the same as that for Genetic Algorithm. The only difference is in deciding the stopping condition.

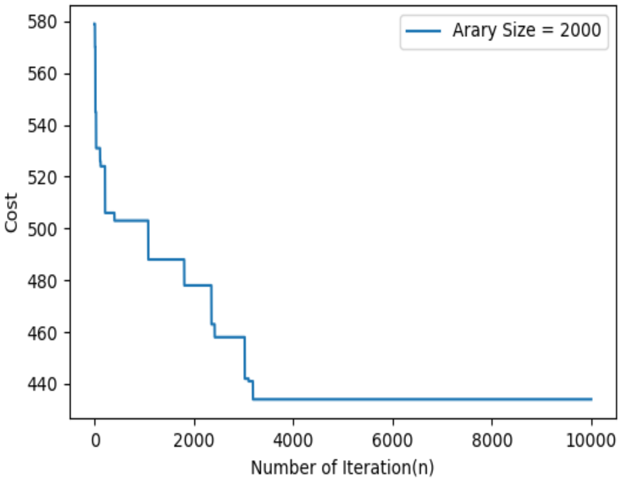
Stopping Condition

To decide the stopping condition, the experiment is performed by varying array size as 500, 1000, 1500 and 2000, and performed 10000 iterations. The result of the experiment is as below.









From the above simulations, we observe that after approximately 3000 iterations, the cost of the path remains constant. Thus we fix 3000 iterations for the Greedy-Genetic algorithm.

Parameter Considered for Greedy-Genetic Algorithm

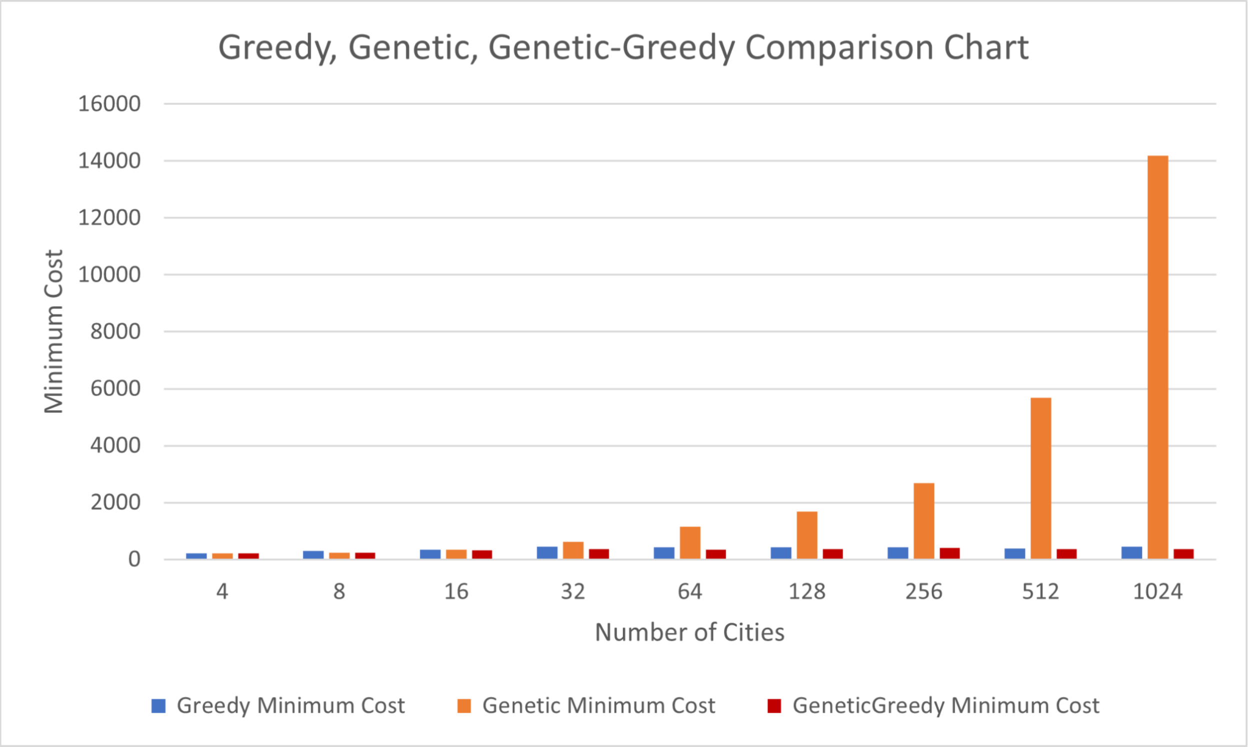
Parameters Considered	**Value Considered**
Population Size	5000
Number of Iteration	3000
Selection Method	Tournament Selection Method
Size of Chromosome	Number of Cities
Crossover Method	Partially Mapped Crossover
Mutation Rate	0.1
Fitness Function	Calculate the Path Length

Comparison Table for Greedy, Genetic and Genetic-Greedy Algorithm

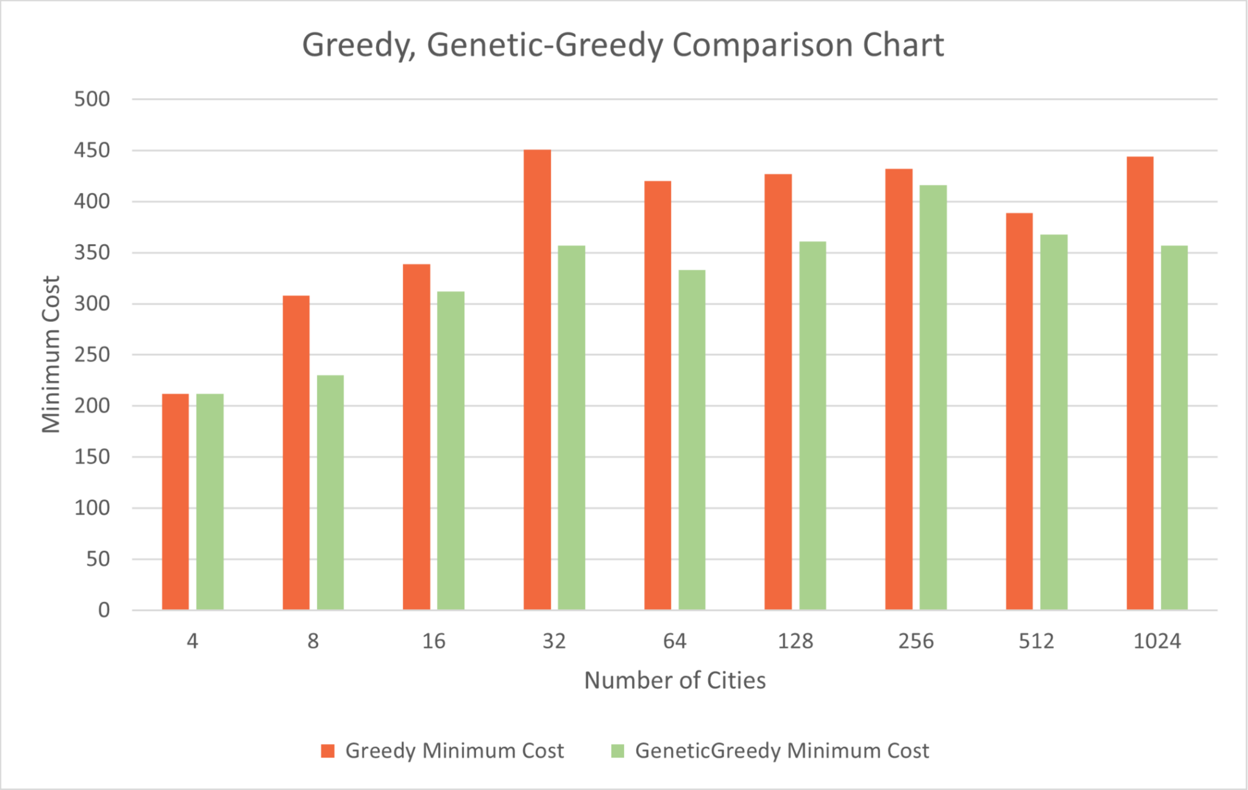
Array Size	Greedy Minimum ost	Genetic Minimum ost	GeneticGreedy Minimum Cost
4	144	144	144
8	212	212	212
16	308	230	230

32	339	352	312
64	451	621	357
128	420	1141	333
256	427	1681	361
512	432	2688	416
1024	389	5680	368
2048	444	14181	357

Comparison Graph for Greedy, Genetic and Genetic-Greedy Algorithm



Comparison Graph for Greedy and Genetic-Greedy Algorithm



Conclusion

From the above comparison table and graph, it is observed that the Genetic-Greedy Algorithm gives a near optimal solution. Genetic algorithm depends on the initial population and thus it lacks in searching for a local optimal solution. As greedy algorithm generates local optimal solution, we can take advantage of this to generate the initial population of Genetic Algorithm. Thus the Genetic-Greedy algorithm generates a good initial population and gives a near optimal solution.

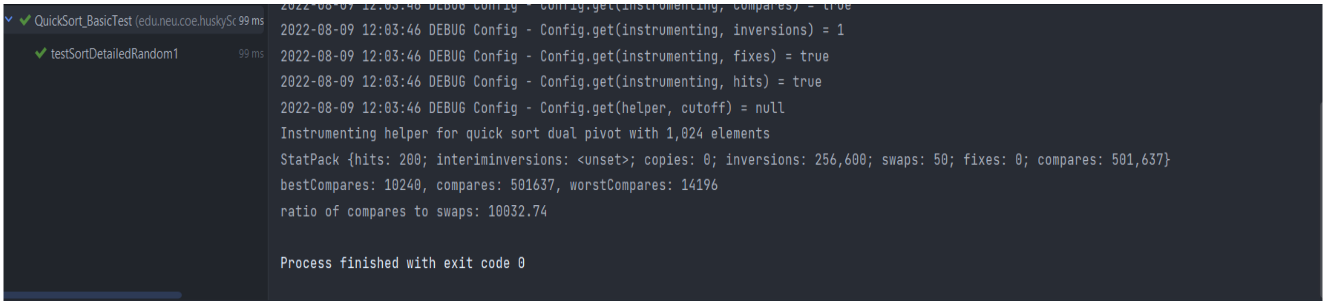
Task-2 Quick Sort

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?

Analysis

Case 1 : Quicksort on Sorted Array with Duplicate Elements

One of the problems of quicksort is that it doesn’t recognize duplicate elements and performs swaps on it. Due to this, even though the array is sorted, there are still some swaps. After multiple runs, it is observed that the probability of generating duplicate elements in a randomly generated array is very low. Hence, the swaps will not be as much as that for an unsorted array.


QuickSort_BasicTest (edu.neu.coe.huskySc 99 ms
testSortDetailedRandom1 99 ms

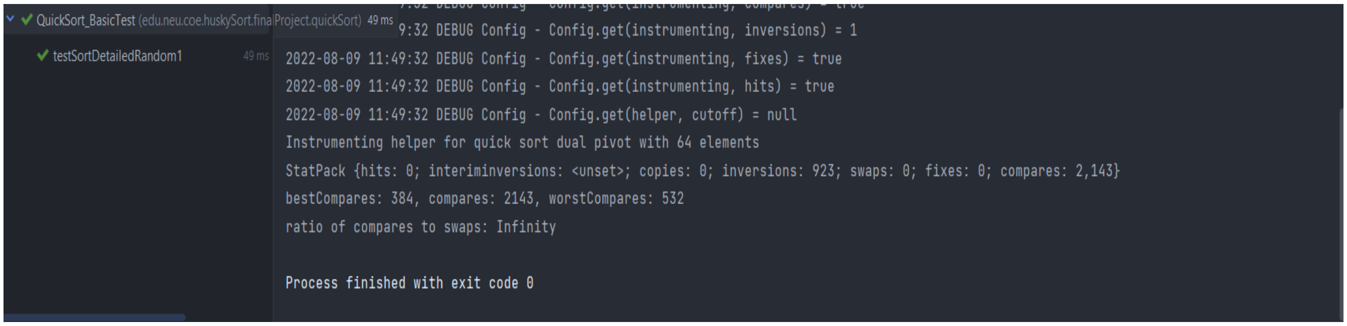
```
2022-08-09 12:03:46 DEBUG Config - Config.get(instrumenting, compares) = true
2022-08-09 12:03:46 DEBUG Config - Config.get(instrumenting, inversions) = 1
2022-08-09 12:03:46 DEBUG Config - Config.get(instrumenting, fixes) = true
2022-08-09 12:03:46 DEBUG Config - Config.get(instrumenting, hits) = true
2022-08-09 12:03:46 DEBUG Config - Config.get(helper, cutoff) = null
Instrumenting helper for quick sort dual pivot with 1,024 elements
StatPack {hits: 200; interiminversions: <unset>; copies: 0; inversions: 256,600; swaps: 50; fixes: 0; compares: 501,637}
bestCompares: 10240, compares: 501637, worstCompares: 14196
ratio of compares to swaps: 10032.74

Process finished with exit code 0
```

It is observed from the simulation that the number of swaps in this case will be equal to the number of Duplicate(d) elements. The number of compares will be more as each element will be compared with the pivot element in every partition. The number of comparisons will be approximately more than the sum of n elements.

As the number of swaps will be very low and the number of compares will be very high, hence, the ratio of swaps to compare will be much smaller than ⅙ .

Case 2 : Quicksort on Sorted Array with Duplicate Elements

It can be inferred from the below screenshot that the number of swaps is 0, as there are no duplicates. Hence, the ratio of swaps to compares will be 0 and it can never be ⅙ in such a case.



QuickSort_BasicTest (edu.neu.coe.huskySort.finalProject.quickSort) 49 ms
testSortDetailedRandom1 49 ms

```
9:32 DEBUG Config - Config.get(instrumenting, compares) = true
9:32 DEBUG Config - Config.get(instrumenting, inversions) = 1
2022-08-09 11:49:32 DEBUG Config - Config.get(instrumenting, fixes) = true
2022-08-09 11:49:32 DEBUG Config - Config.get(instrumenting, hits) = true
2022-08-09 11:49:32 DEBUG Config - Config.get(helper, cutoff) = null
Instrumenting helper for quick sort dual pivot with 64 elements
StatPack {hits: 0; interiminversions: <unset>; copies: 0; inversions: 923; swaps: 0; fixes: 0; compares: 2,143}
bestCompares: 384, compares: 2143, worstCompares: 532
ratio of compares to swaps: Infinity

Process finished with exit code 0
```

Case 3 : Quicksort on Reverse Sorted Array without Duplicate Elements

The number of swaps for this case will be $\frac{n}{2}$.

We concluded from the simulation that the number of compares for this case,

$$\frac{n(n-1)}{2} + \frac{3n}{2} - 1$$

$$= \frac{n^2 - n + 3n}{2} - 1$$

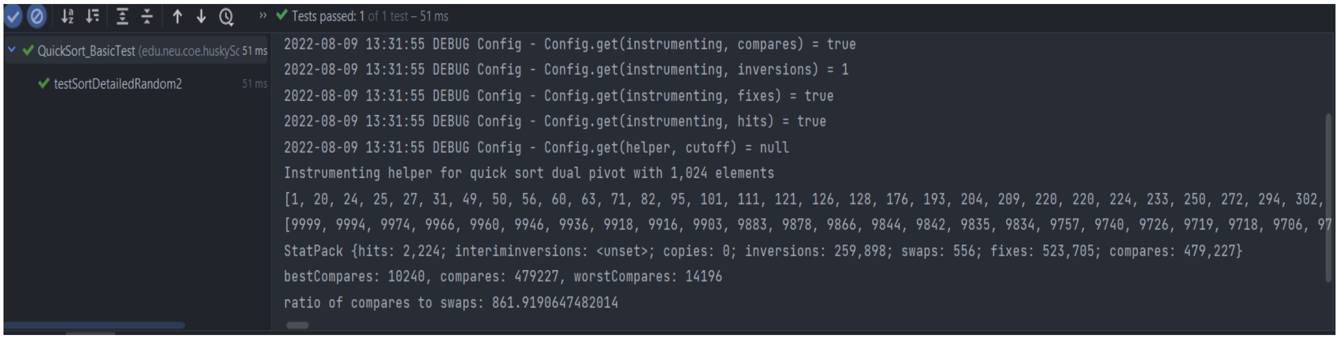
$$= \frac{n^2+2n-2}{2}$$

Therefore, the ratio of swaps to compare will be,

$$\frac{\frac{n}{2}}{\frac{n^2+2n-2}{2}} = \frac{n}{n^2+2n-2}$$

From the above equation, the ratio of swaps to compare will never be ⅙ except for n=4,5

Unit Test Case



QuickSort_BasicTest (edu.neu.coe.huskySc 51 ms
testSortDetailedRandom2 51 ms

```
2022-08-09 13:31:55 DEBUG Config - Config.get(instrumenting, compares) = true
2022-08-09 13:31:55 DEBUG Config - Config.get(instrumenting, inversions) = 1
2022-08-09 13:31:55 DEBUG Config - Config.get(instrumenting, fixes) = true
2022-08-09 13:31:55 DEBUG Config - Config.get(instrumenting, hits) = true
2022-08-09 13:31:55 DEBUG Config - Config.get(helper, cutoff) = null
Instrumenting helper for quick sort dual pivot with 1,024 elements
[1, 20, 24, 25, 27, 31, 49, 50, 56, 60, 63, 71, 82, 95, 101, 111, 121, 126, 128, 176, 193, 204, 209, 220, 220, 224, 233, 250, 272, 294, 302,
[9999, 9994, 9974, 9966, 9960, 9946, 9936, 9918, 9916, 9903, 9883, 9878, 9866, 9844, 9842, 9835, 9834, 9757, 9740, 9726, 9719, 9718, 9706, 97
StatPack {hits: 2,224; interiminversions: <unset>; copies: 0; inversions: 259,898; swaps: 556; fixes: 523,705; compares: 479,227}
bestCompares: 10240, compares: 479227, worstCompares: 14196
ratio of compares to swaps: 861.9190647482014
```

Case 4 : Quicksort on Unsorted Array

The simulation was performed for array size 8 to 524,288 using the doubling method for 1000 iterations on each array size.

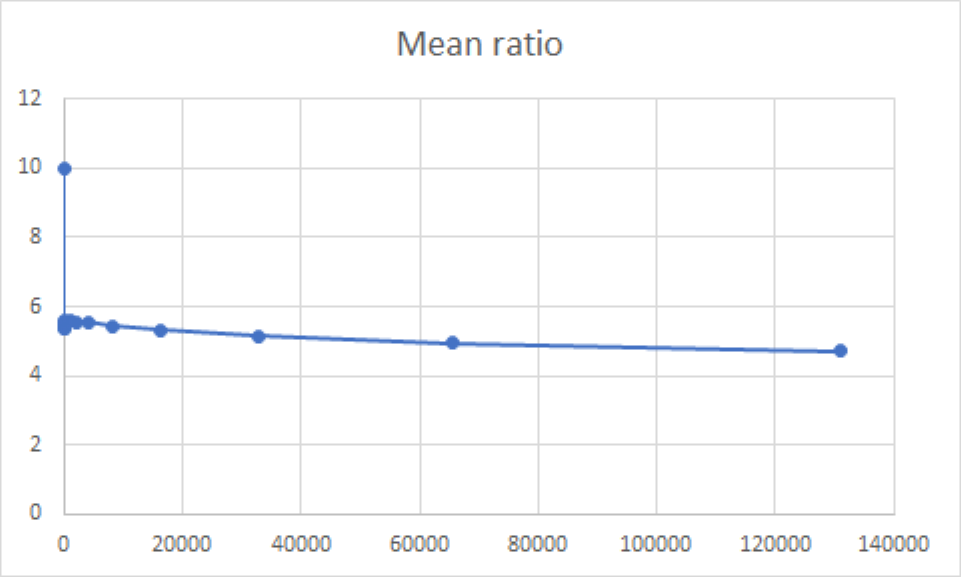
Observations

Array Size	Mean Swap	Mean compare	Mean ratio	X	Y
8	5	28	5.6	2.8	2.678571429
16	14	75	5.357142857	2.5	2.493333333
32	35	187	5.342857143	2.4	2.43315508
64	84	455	5.416666667	2.357142857	2.375824176
128	198	1081	5.45959596	2.292929293	2.317298797
256	454	2505	5.517621145	2.262114537	2.272255489
512	1027	5692	5.542356378	2.23563778	2.254919185
1024	2296	12835	5.590156794	2.220818815	2.210518115
2048	5099	28372	5.56422828	2.207099431	2.196073594
4096	11254	62307	5.536431491	2.206148925	2.162582053
8192	24828	134744	5.427098437	2.202714677	2.15341685
16384	54689	290160	5.305637331	2.204337252	2.138206507
32768	120553	620422	5.146466699	2.206664289	2.115432722
65536	266020	1312461	4.933692955	2.205082325	2.107360904
131072	586596	2765829	4.715049199	2.195613335	2.102381239
262,144	1287938	5814827	4.514834565	2.200428608	2.064333814
524,288	2846895	1.20E+07	4.216433694	-	-

$$X = \frac{Mean\ Swaps\ of\ ArraySize(A)}{Mean\ Swaps\ of\ ArraySize(A/2)}$$

$$Y = \frac{Mean\ Compares\ of\ ArraySize(A)}{Mean\ Compares\ of\ ArraySize(A/2)}$$

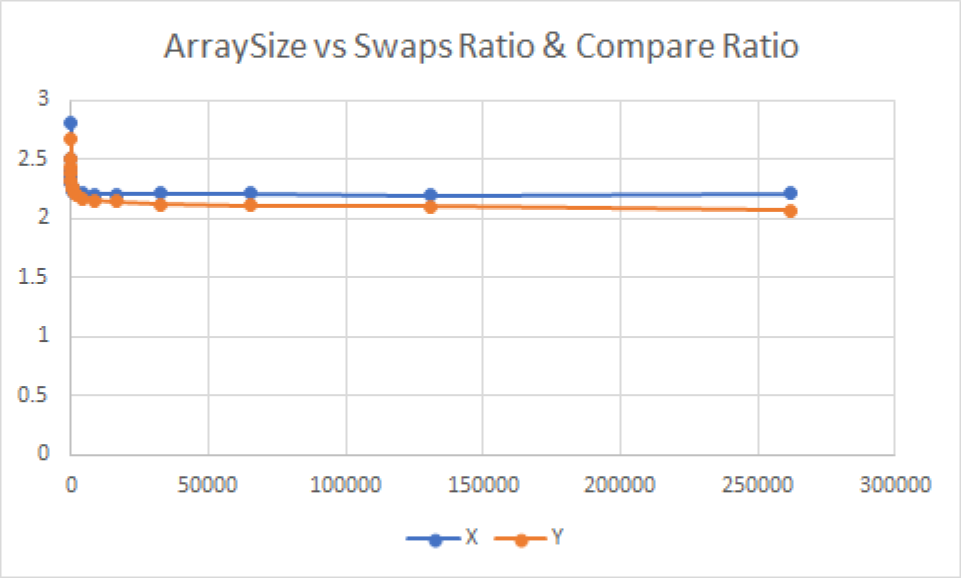
From the above table, it is observed that the ratio of compares to swaps is approximately 6 for small arrays. As the size of the array increases, the ratio of compares to swaps decreases.



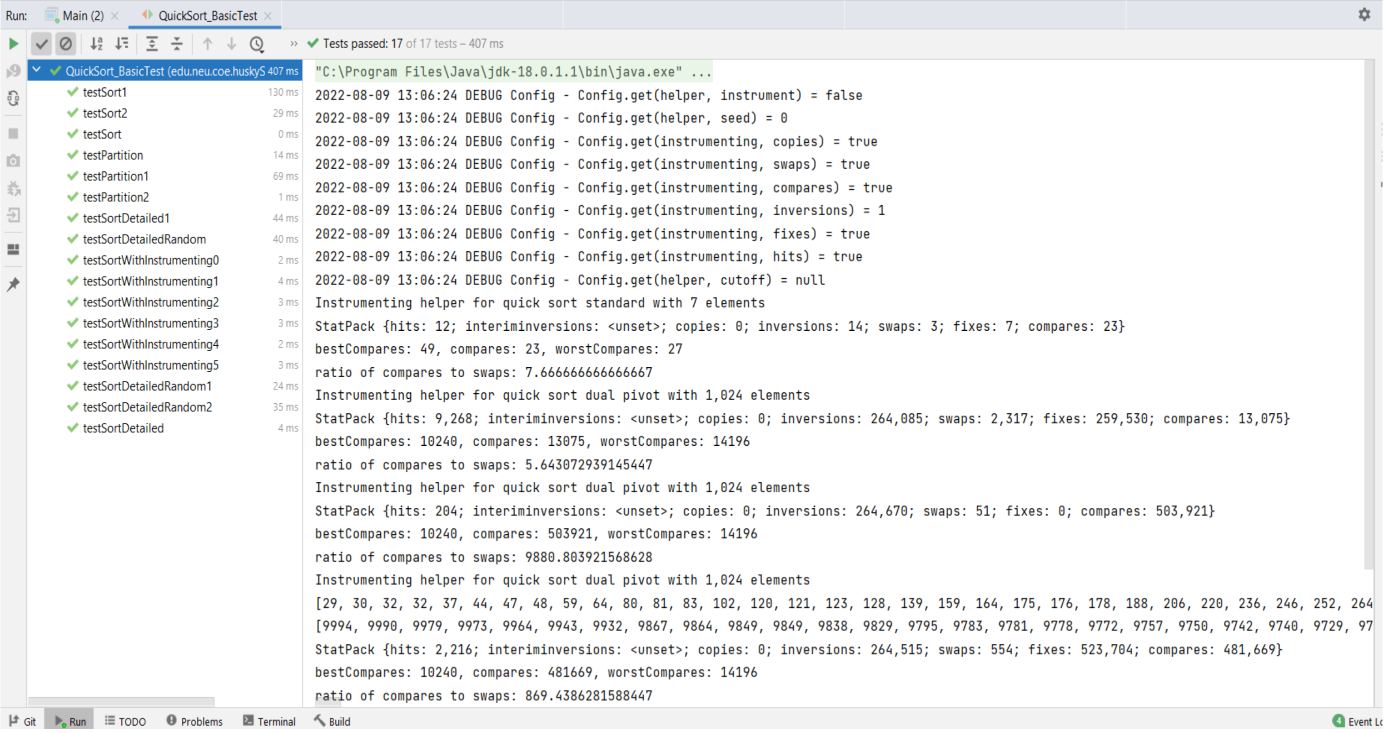
Additionally, it is observed from the above table that the value of X remains constant from Array Size 2048, while the value of Y is still in a slightly decreasing trend.

The ratio(X) of mean swaps of a particular array size(N) to the mean swaps of half the array size(N/2) is constant, while the ratio(Y) of mean compares of a particular array size(N) to the mean compares of half the array size(N/2) is decreasing.

Hence, the rate at which swaps increase with respect to array size is constant while the rate at which comparisons increase with respect to array size subside.



Unit Test Cases



```
Tests passed: 17 of 17 tests – 407 ms
```

```
QuickSort_BasicTest (edu.neu.coe.huskyS 407 ms
    testSort1                        130 ms
    testSort2                         29 ms
    testSort                           0 ms
    testPartition                     14 ms
    testPartition1                    69 ms
    testPartition2                     1 ms
    testSortDetailed1                 44 ms
    testSortDetailedRandom            40 ms
    testSortWithInstrumenting0         2 ms
    testSortWithInstrumenting1         4 ms
    testSortWithInstrumenting2         3 ms
    testSortWithInstrumenting3         3 ms
    testSortWithInstrumenting4         2 ms
    testSortWithInstrumenting5         3 ms
    testSortDetailedRandom1           24 ms
    testSortDetailedRandom2           35 ms
    testSortDetailed                   4 ms
```

```
"C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe" ...
2022-08-09 13:06:24 DEBUG Config - Config.get(helper, instrument) = false
2022-08-09 13:06:24 DEBUG Config - Config.get(helper, seed) = 0
2022-08-09 13:06:24 DEBUG Config - Config.get(instrumenting, copies) = true
2022-08-09 13:06:24 DEBUG Config - Config.get(instrumenting, swaps) = true
2022-08-09 13:06:24 DEBUG Config - Config.get(instrumenting, compares) = true
2022-08-09 13:06:24 DEBUG Config - Config.get(instrumenting, inversions) = 1
2022-08-09 13:06:24 DEBUG Config - Config.get(instrumenting, fixes) = true
2022-08-09 13:06:24 DEBUG Config - Config.get(instrumenting, hits) = true
2022-08-09 13:06:24 DEBUG Config - Config.get(helper, cutoff) = null
Instrumenting helper for quick sort standard with 7 elements
StatPack {hits: 12; interiminversions: <unset>; copies: 0; inversions: 14; swaps: 3; fixes: 7; compares: 23}
bestCompares: 49, compares: 23, worstCompares: 27
ratio of compares to swaps: 7.666666666666667
Instrumenting helper for quick sort dual pivot with 1,024 elements
StatPack {hits: 9,268; interiminversions: <unset>; copies: 0; inversions: 264,085; swaps: 2,317; fixes: 259,530; compares: 13,075}
bestCompares: 10240, compares: 13075, worstCompares: 14196
ratio of compares to swaps: 5.643072939145447
Instrumenting helper for quick sort dual pivot with 1,024 elements
StatPack {hits: 204; interiminversions: <unset>; copies: 0; inversions: 264,670; swaps: 51; fixes: 0; compares: 503,921}
bestCompares: 10240, compares: 503921, worstCompares: 14196
ratio of compares to swaps: 9880.803921568628
Instrumenting helper for quick sort dual pivot with 1,024 elements
[29, 30, 32, 32, 37, 44, 47, 48, 59, 64, 80, 81, 83, 102, 120, 121, 123, 128, 139, 159, 164, 175, 176, 178, 188, 206, 220, 236, 246, 252, 264
[9994, 9990, 9979, 9973, 9964, 9943, 9932, 9867, 9864, 9849, 9849, 9838, 9829, 9795, 9783, 9781, 9778, 9772, 9757, 9750, 9742, 9740, 9729, 97
StatPack {hits: 2,216; interiminversions: <unset>; copies: 0; inversions: 264,515; swaps: 554; fixes: 523,704; compares: 481,669}
bestCompares: 10240, compares: 481669, worstCompares: 14196
ratio of compares to swaps: 869.4386281588447
```

Conclusion

From the above observations, it can be concluded that the ratio of compares to swaps is approximately 6 for small unsorted arrays. As the size of the array increases, the ratio of compares to swaps decreases.

Task-3 Hibbard Deletion of BST

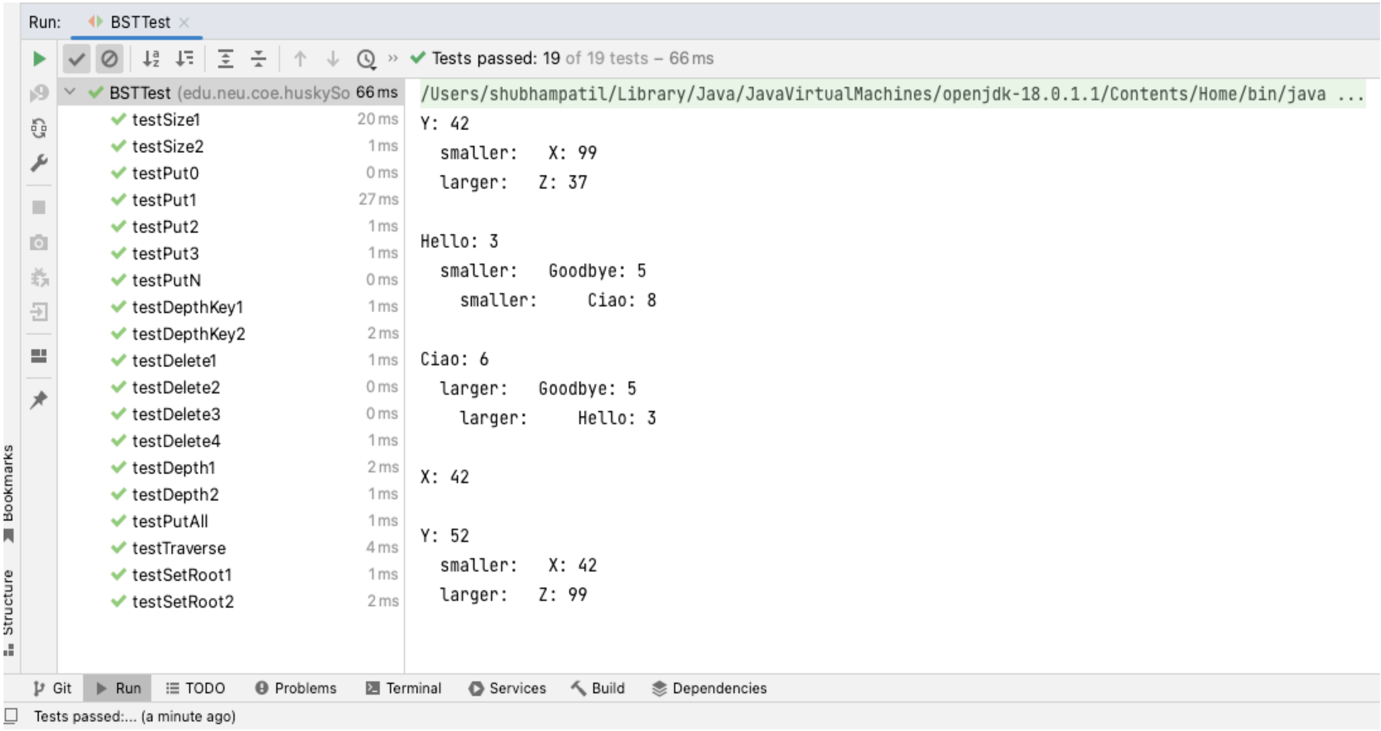
According to the course lecture notes, after a number of (Hibbard) deletions have been made, the average height of the tree is $\sqrt{n}$. Do you agree with this? How does it look after modifying the deletion process to either (a) randomly choose which direction to look for the node to be deleted or (b) choose the direction according to the size of the candidate nodes.

Analysis

We have taken the number of nodes from 2^3 to 2^{15} for 1000 iterations for all the three cases Simple, Random and Optimized deletion. Also, we have deleted n/2 nodes for each operation containing n nodes. Its analysis is shown below along with their test cases.

Test Cases

Hibbard Deletion



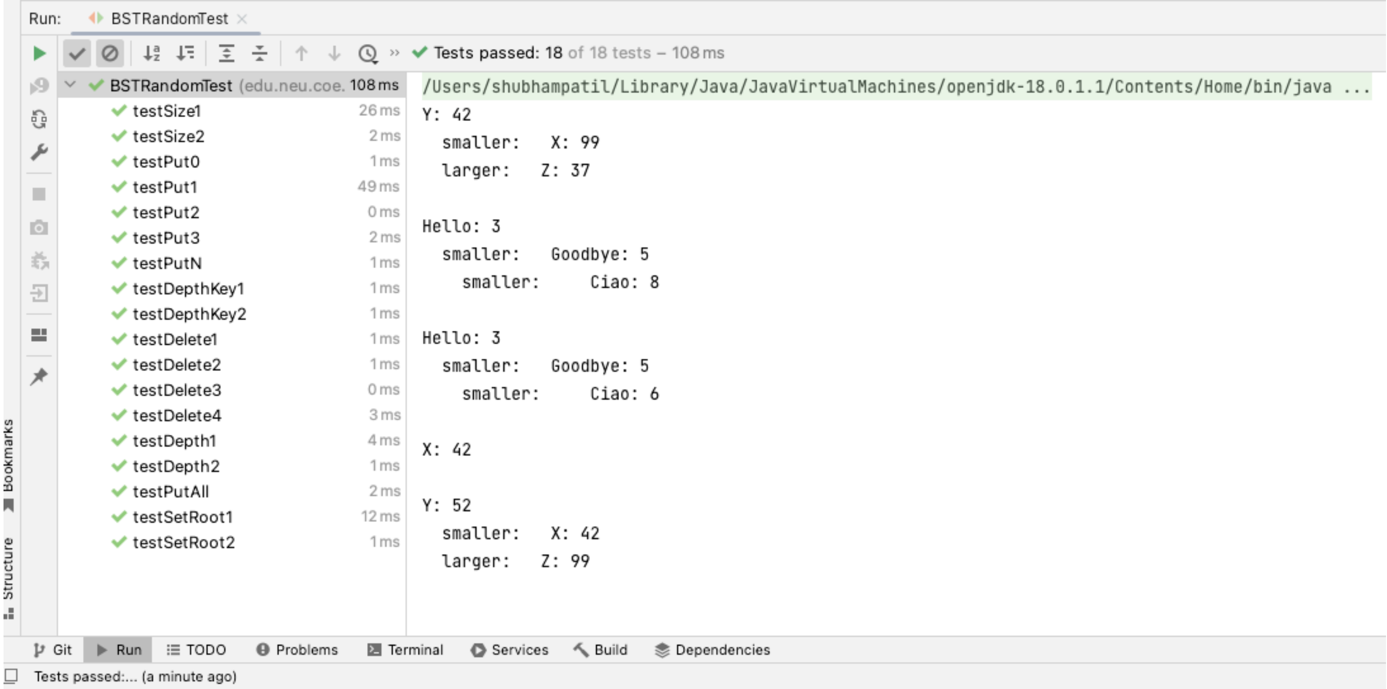
Tests passed: 19 of 19 tests – 66 ms

```
BSTTest (edu.neu.coe.huskySo 66 ms
  testSize1          20 ms
  testSize2           1 ms
  testPut0            0 ms
  testPut1           27 ms
  testPut2            1 ms
  testPut3            1 ms
  testPutN            0 ms
  testDepthKey1       1 ms
  testDepthKey2       2 ms
  testDelete1         1 ms
  testDelete2         0 ms
  testDelete3         0 ms
  testDelete4         1 ms
  testDepth1          2 ms
  testDepth2          1 ms
  testPutAll          1 ms
  testTraverse        4 ms
  testSetRoot1        1 ms
  testSetRoot2        2 ms
```

```
/Users/shubhampatil/Library/Java/JavaVirtualMachines/openjdk-18.0.1.1/Contents/Home/bin/java ...
Y: 42
  smaller:   X: 99
  larger:   Z: 37

Hello: 3
  smaller:   Goodbye: 5
    smaller:     Ciao: 8

Ciao: 6
  larger:   Goodbye: 5
    larger:     Hello: 3

X: 42

Y: 52
  smaller:   X: 42
  larger:   Z: 99
```

Tests passed:... (a minute ago)

Random Deletion



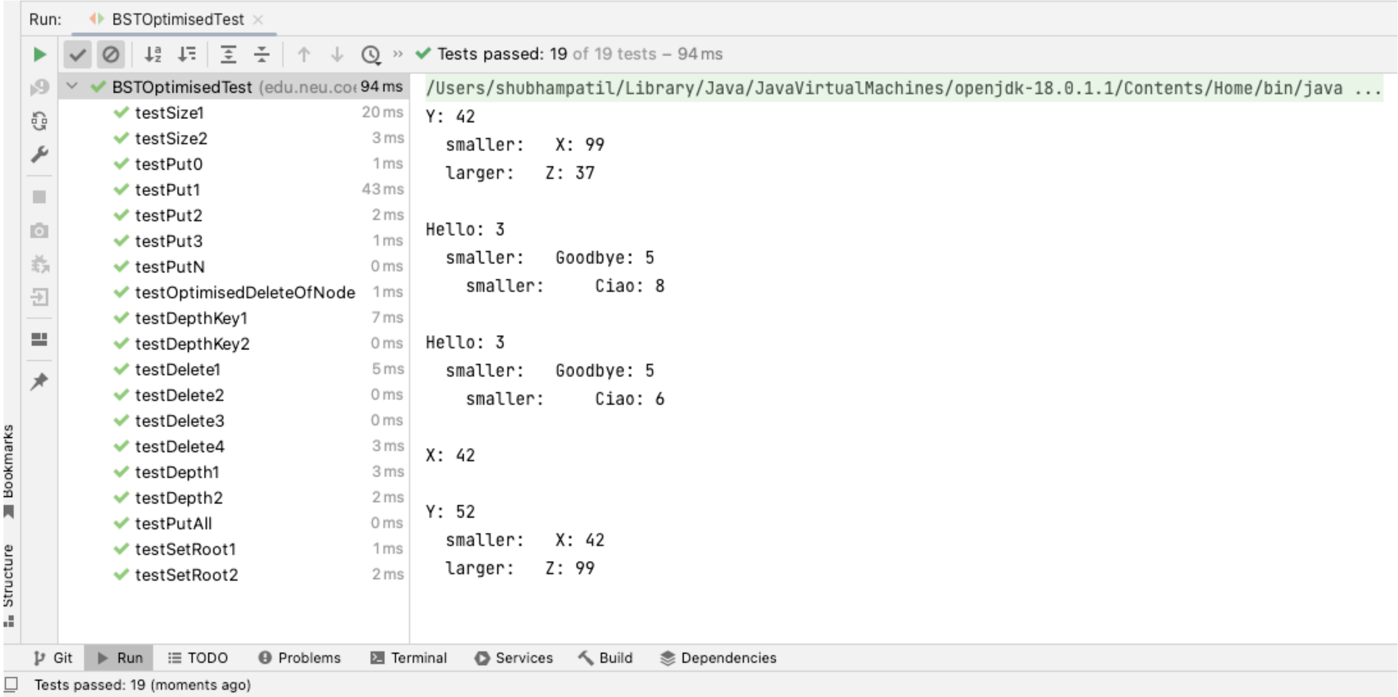
Tests passed: 18 of 18 tests – 108 ms

BSTRandomTest (edu.neu.coe.	108 ms
testSize1	26 ms
testSize2	2 ms
testPut0	1 ms
testPut1	49 ms
testPut2	0 ms
testPut3	2 ms
testPutN	1 ms
testDepthKey1	1 ms
testDepthKey2	1 ms
testDelete1	1 ms
testDelete2	1 ms
testDelete3	0 ms
testDelete4	3 ms
testDepth1	4 ms
testDepth2	1 ms
testPutAll	2 ms
testSetRoot1	12 ms
testSetRoot2	1 ms

```
/Users/shubhampatil/Library/Java/JavaVirtualMachines/openjdk-18.0.1.1/Contents/Home/bin/java ...
Y: 42
  smaller:   X: 99
  larger:   Z: 37

Hello: 3
  smaller:   Goodbye: 5
    smaller:     Ciao: 8

Hello: 3
  smaller:   Goodbye: 5
    smaller:     Ciao: 6

X: 42

Y: 52
  smaller:   X: 42
  larger:   Z: 99
```

Tests passed:... (a minute ago)

Optimized Deletion



Tests passed: 19 of 19 tests – 94 ms

BSTOptimisedTest (edu.neu.co	94 ms
testSize1	20 ms
testSize2	3 ms
testPut0	1 ms
testPut1	43 ms
testPut2	2 ms
testPut3	1 ms
testPutN	0 ms
testOptimisedDeleteOfNode	1 ms
testDepthKey1	7 ms
testDepthKey2	0 ms
testDelete1	5 ms
testDelete2	0 ms
testDelete3	0 ms
testDelete4	3 ms
testDepth1	3 ms
testDepth2	2 ms
testPutAll	0 ms
testSetRoot1	1 ms
testSetRoot2	2 ms

```
/Users/shubhampatil/Library/Java/JavaVirtualMachines/openjdk-18.0.1.1/Contents/Home/bin/java ...
Y: 42
  smaller:   X: 99
  larger:   Z: 37

Hello: 3
  smaller:   Goodbye: 5
    smaller:     Ciao: 8

Hello: 3
  smaller:   Goodbye: 5
    smaller:     Ciao: 6

X: 42

Y: 52
  smaller:   X: 42
  larger:   Z: 99
```

Tests passed: 19 (moments ago)

Notation Description

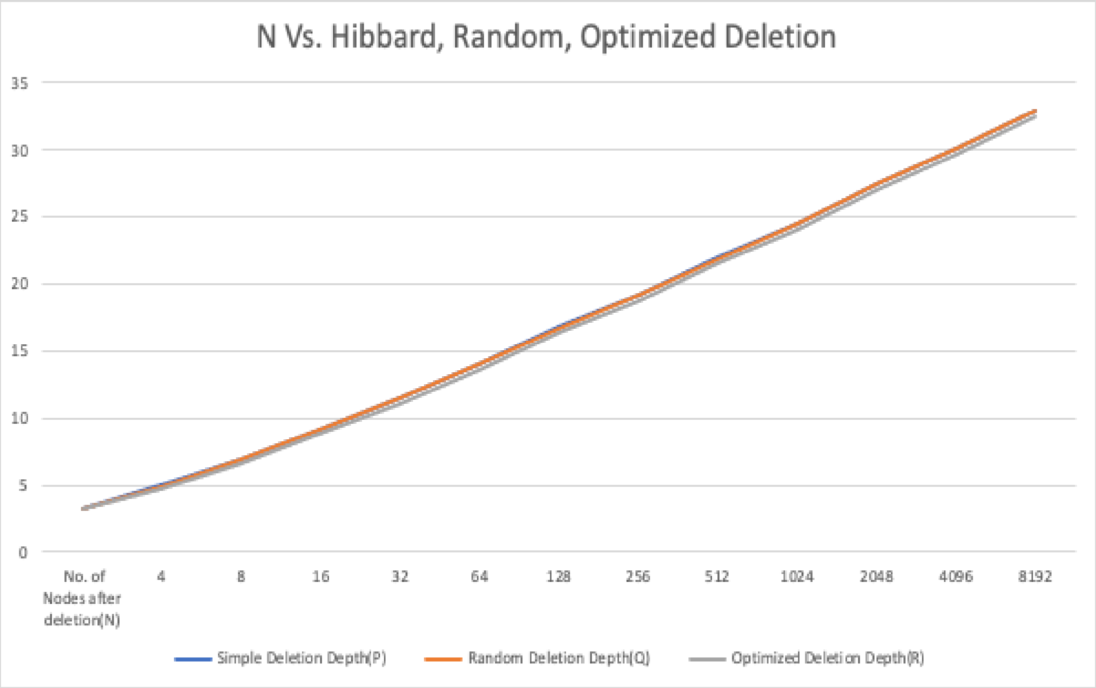
Variable Name	Description
Z	No. of nodes Inserted
N	No. of nodes after deletion
P	Mean Height after Hibbard Deletion
Q	Mean Height After Random Deletion
R	Mean Height After Optimized Deletion

Simulation Table

Z	N	P	Q	R	$\sqrt{N}$ (H)	(P/H)	(Q/H)	(R/H)	log N	2log N
8	4	3.279	3.276	3.187	2.00	1.6395	1.6380	1.5935	2	4
16	8	4.944	4.934	4.703	2.83	1.7480	1.7444	1.6628	3	6
32	16	6.9	6.901	6.599	4.00	1.7250	1.7253	1.6498	4	8
64	32	9.123	9.141	8.798	5.66	1.6127	1.6159	1.5553	5	10
128	64	11.493	11.46	11.10	8.00	1.4366	1.4325	1.3881	6	12
256	128	14.04	14.035	13.64	11.31	1.2411	1.2405	1.2062	7	14
512	256	16.775	16.735	16.33	16.00	1.0484	1.0459	1.0207	8	16
1024	512	19.256	19.237	18.82	22.63	0.8510	0.8502	0.8318	9	18
2048	1024	21.948	21.892	21.51	32.00	0.6859	0.6841	0.6721	10	20
4096	2048	24.573	24.583	24.12	45.25	0.5430	0.5432	0.5331	11	22
8192	4096	27.492	27.499	27.04	64.00	0.4296	0.4297	0.4225	12	24
16384	8192	30.112	30.1	29.65	90.51	0.3327	0.3326	0.3276	13	26
32768	16384	32.996	32.991	32.51	128.0	0.2578	0.2577	0.2540	14	28

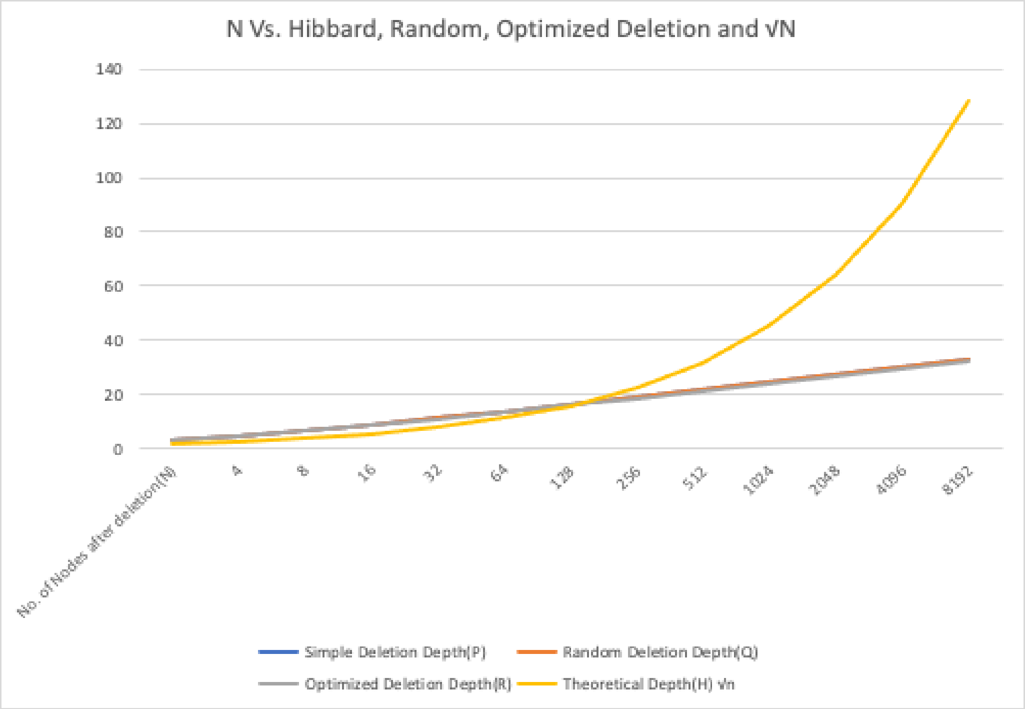
Hibbard, Random, Optimized Deletion Graph

The random and optimized deletion methods give almost similar results with respect to Hibbard Deletion in terms of height. No major difference in height can be observed as the number of elements in the lowest level of the tree can accommodate almost twice the number of nodes in the remaining tree. In some cases, a difference of height 1 can be observed in an optimized solution .



$\sqrt{N}$, Hibbard, Random, Optimized Deletion Graph

For N (No. of nodes after deletion) up to 128, the height of the tree remains close to but less than $\sqrt{N}$. For N = 256, the height is equal to $\sqrt{N}$, after which the graph of $\sqrt{N}$ increases drastically whereas the growth of height of the tree remains linear. The ratio of height after deletion(P,Q,R) to $\sqrt{N}$ (H) decreases as the size of the tree increases; it becomes 1 at N= 256 and then tends to 0 as size of the tree increases.



logN, 2logN, Hibbard, Random, Optimized Deletion Graph

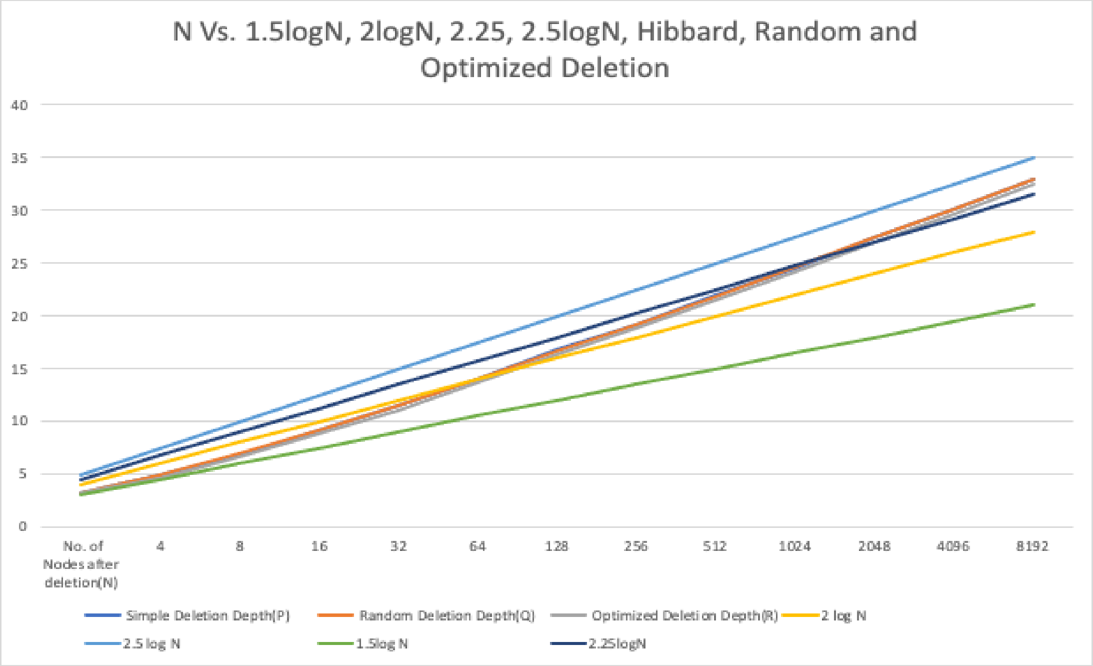
From our simulation, it can be deduced that the height of the tree (h) is,

$$h \sim c \log N$$

Where **c** starts from 1.414 and increases with the increasing number of nodes and N is the number of nodes after deletion.

Optimized Height after DeletionTh (H)	c
2	1.414
3.187	1.5935
4.703	1.56766667
6.599	1.64975
8.798	1.7596
11.10	1.85083333

13.64	1.94957143
16.33	2.041375
18.82	2.09133333
21.51	2.1508
24.12	2.19309091
27.04	2.25333333
29.65	2.28115385
32.51	2.3225



Conclusion

The height of the Binary Search Tree after performing any of the three deletion methods(Hibbard, Random and Optimized) will be $\sqrt{N}$ (N is the number of nodes after deletion) only for the N= 256 and does not stand true for other values of N.

From our observations, it can be concluded that the mean height of the Binary Search Tree after deleting half the nodes, h ~ c log N.

References

1. “Traveling Salesman Problem | Greedy Approach.” *GeeksforGeeks*, 5 January 2022, https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/. Accessed 9 August 2022.
2. “Traveling Salesman Problem with Genetic Algorithms in Java.” Stack Abuse, 8 August 2019,https://stackabuse.com/traveling-salesman-problem-with-genetic-algorithms-in-java/. Accessed 9 August 2022.
3. “rchillyard/INFO6205.” GitHub, https://github.com/rchillyard/INFO6205. Accessed 9 August 2022.
4. Lamb, Evelyn. “Travelling salesman problem.” Wikipedia, https://en.wikipedia.org/wiki/Travelling_salesman_problem. Accessed 9 August 2022.
5.