Mateusz Małecki 148265

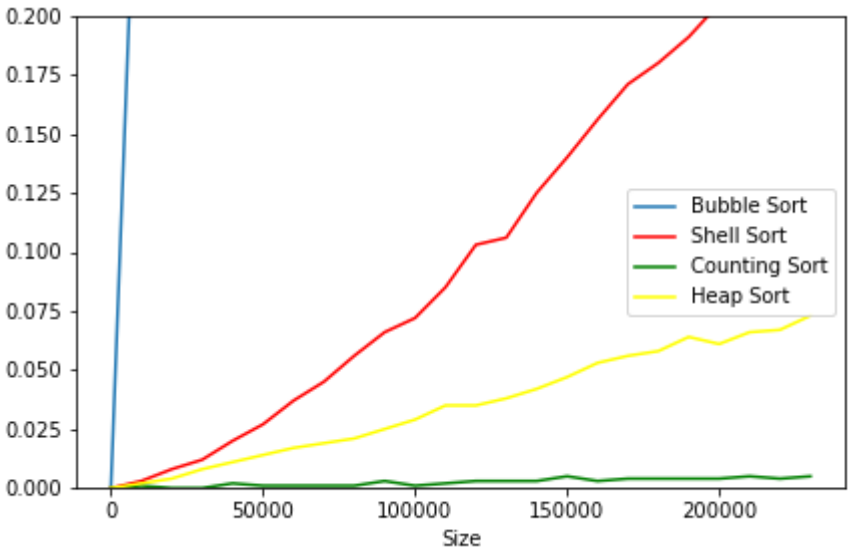
Nikodem Seidel 148269

Ex. 1

- Bubble sort:
  Complexity of Bubble sort is one of the worst in our test. It is always equal to $O(n^2)$. Algorithm is going through the whole array n^2 times, where n is a length of the array, and checking if the current element is greater than the next one. If so, then it swaps the elements. Efficiency of this algorithm is very poor. It can be clearly seen on the below chart, when the Bubble's sort line flies to the moon. Memory is used only to keep an array, and to keep one element of an array while swapping. It is not using any additional RAM to make the process of sorting any faster.
- Shell Sort:
  This algorithm is a different variant of Insertion sort mixed with Bubble sort. At the start, the algorithm is setting the so-called "gap" to the highest value, and compares elements of the array whose indices are gap size appart. As the algorithm works, the gap is decreasing, finally reaching the size of 1 when the algorithm starts to work exactly as insertion sort. Complexity of this sorting way depends on the technique of decreasing the gap. The worst case scenario is $O(n^2)$, the best is $O(n\log_2 n)$. In our case it is definitely better than Bubble sort ($O(n^2)$) but at the same time is not even close to $O(n\log_2 n)$. It is somewhere close to O(n^(4/3)). This algorithm is also purely using computer memory. Only to keep the array and to keep one element of an array while swapping.
- Counting sort:
  Sorts the elements in an array by counting the number of occurrences of each element in the array. Information about occurrences is stored in separate array and after that in the next array sorted list is done by mapping previous created array to it. In this exercises only counting algorithm isn't comparison sort and doesn't work on the primary array. On the graph below counting sort is definitely the fastest algorithm. That' s because in testing we have taken only integer numbers and relatively small range equal to 250 000. Complexity of this algorithm is the same in every case (best, average and worst) and is equal to O(n+k), where "n" is number of elements in array and "k" is a range of numbers.

- Heap sort:
  One of the most interesting sorts. The algorithm recursively creates the binary heap (by swapping elements in array in an appropriate way (according to the formula)). While it ends, the biggest element of the array is on the top of a binary tree, so it is swapped with the last element of the array. It repeats this procedure until the heap has a size of 1. Worst complexity of this algorithm is $O(n\log_2 n)$. This

algorithm uses the memory to complete all of the recursions, which allows it to be quite a decent and very consistent way to sort out arrays, which can be noticed on the below graph.



Ex. 2

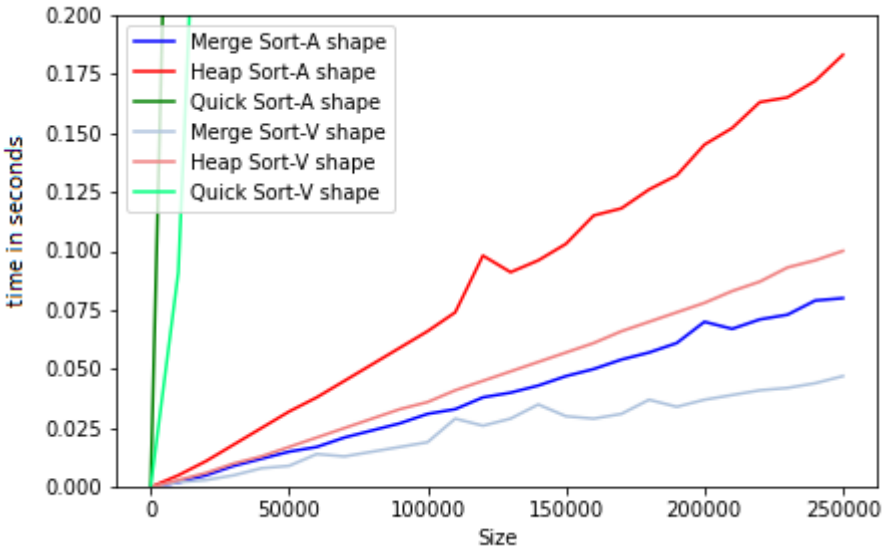
In average case quick sort has the same complexity as in the best case and it's equal to n*log n, but in the worst event (when database has shape of A, V or is constant) complexity increases to n^2. Memory usage is on the level of log n. Database with random, increasing or decreasing elements has O(n log n). Quick sort (with pivot on the middle) is used when:

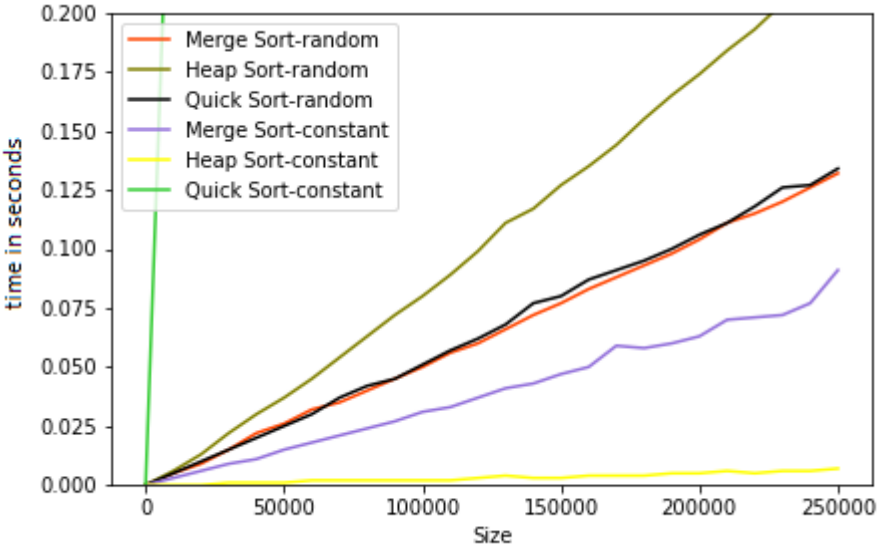
1. Database is random or sorted increasing or decreasing.
2. Stability of sorting algorithm isn't needed.
3. Cache-friendly algorithm is needed.

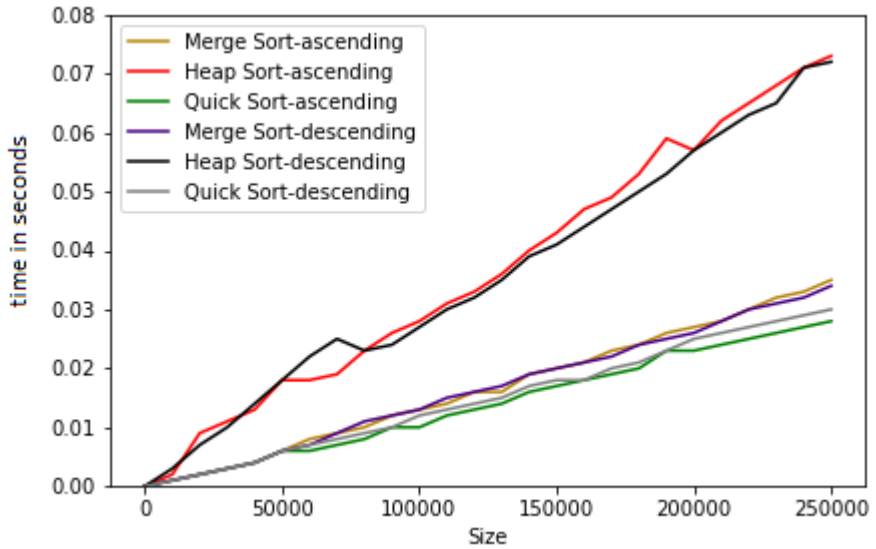
Practical usage of QS:

1. Commercial computing.
2. Search for information.
3. Numerical computation.
4. Operation research.

Comparisons of Merge, Heap, and Quick Sorts dependent of database shape.







First additional exercise.

```
time of executing quicksort according to database shape and number of elements in array


shape/size      99999        100000        100001
V-shape      9.097000      9.069000      4.538000
A-shape     26.493000     26.468000     26.483000
random       0.108000      0.108000      0.107000

Process returned 0 (0x0)   execution time : 102.763 s
Press any key to continue.
```

Second additional exercise - test (correctness and time) of every algorithm mentioned in introduction (n = 100, range = 250 000, filled randomly):

```
unsorted array:
310 34 52 284 159 19 283 13 298 308 337 237 195 33 166 108 86 310 83 50 16 118 318 42 314 256
366 177 147 63 176 127 221 129 121 292 300 299 109 167 202 91 143 283 126 350 281 324 77 291 2
86 153 50 127 367 41 123 338 363 163 84 314 42 128 296 103 396 276 4 318 397 389 286 242 140 3
79 102 240 205 25 237 330 331 154 19 359 2 288 149 376 28 265 215 286 154 243 333 10 293 178
IS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
SS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
BS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
QS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
```

```
HS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
CS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
ShS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
MS: 0.000000
2 4 10 13 16 19 19 25 28 33 34 41 42 42 50 50 52 63 77 83 84 86 91 102 103 108 109 118 121 123
 126 127 127 128 129 140 143 147 149 153 154 154 159 163 166 167 176 177 178 195 202 205 215 2
21 237 237 240 242 243 256 265 276 281 283 283 284 286 286 286 288 291 292 293 296 298 299 300
 308 310 310 314 314 318 318 324 330 331 333 337 338 350 359 363 366 367 376 379 389 396 397
```