CSCI242: Data Structures & Algorithms I

Assignment 1

Name: Zhuoyu (Benjamin) Yang

Bubble sort

**Data: “1000000almostsorted.bin” (ALMOST SORTED)**

Start time: Tue Sep 15 09:01:16.966 2020

End time: Tue Sep 15 09:35:58.415 2020

Time taken: approximately 34m 41.449s

Lower bound: 1200

Upper bound: 1300

Sample list: 1192 1193 1194 1198 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1292

**Data: “1000000numbers.bin” (UNSORTED)**

Start time: Tue Sep 15 06:45:23.143 2020

End time: Tue Sep 15 08:53:50.213 2020

Time taken: approximately 2h 8m 27.070s

Lower bound: 555000

Upper bound: 555200

Sample list: 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18198 18198 18198 18198 18198 18198 18198 18198 18198

Bubble Sort Description

The worst, average, and best time complexities of bubble sort is O(n2). Bubble sort consists of two nested loops, resulting in a quadratic time complexity. The outer loop will run *n* times and the inner loop will run approximately *n* times in most situations. The inner loop is said to run “approximately *n* times” because with bubble sort, every iteration of the outer loop results in the one more element being sorted. In our case, sorting in an ascending order, the last element becomes the largest number by the end of the first iteration; the second iteration will result in the second last element becoming the second largest number. Thus, an increasing number of indices are not iterated over with more iterations.

Typically, one will expect the runtimes of bubble sort on a randomly sorted array and an almost sorted array to be comparable. However, there is a stark difference between the runtimes of the almost sorted array (approximately 34m 41.449s) and a randomly sorted array (2h 8m 27.070s) of one million elements (*n* = 1000000). This might be because I wrote a separate swap function, swap(int arr[], int n, int m) that was invoked every time a swap was done. Operations such as handling the array that was passed in, and creating new variables n and m, might have resulted in an increase in the runtime. I ran another bubble sort with the swap instructions written inline with the 1000000 randomly sorted array and it took approximately 49m 35.657s. This hints that handling the swap instructions inline may prove to be more efficient than having the swap instructions in a separate function (potentially compromising on readability?).

There is a variant of bubble sort that features a check that breaks out of the algorithm if the last iteration has no swaps. The variation will have a best time complexity of O(n) *if* given an already sorted array. This is because the algorithm will only perform a total of (*n* – 1) comparisons before breaking out. However, this variation adds extra instructions to the algorithm with the checks, and will only be fast given a small number of scenarios; probabilistically speaking, most datasets are not perfectly sorted. Therefore, the bubble sort algorithm in this assignment does not implement the check.

Selection Sort

**Data: “1000000almostsorted.bin” (ALMOST SORTED)**

Start time: Tue Sep 15 06:12:27.642 2020

End time: Tue Sep 15 06:32:20.138 2020

Time taken: 19m 52.496s

Lower bound: 1200

Upper bound: 1300

Sample list: 1192 1193 1194 1198 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1292

**Data: “1000000numbers.bin” (UNSORTED)**

Start time: Tue Sep 15 05:41:40.94 2020

End time: Tue Sep 15 06:01:30.874 2020

Time taken: 19m 49.934s

Lower bound: 555000

Upper bound: 555200

Sample list: 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18198 18198 18198 18198 18198 18198 18198 18198 18198

Selection Sort Description

The worst, average, and best time complexities of selection sort is O(n2). Selection sort consists of two nested loops, resulting in this quadratic time complexity. The outer loop runs for (*n* – 1) iterations while the inner loop iterates over a decreasing number of elements as more elements become sorted. Essentially, selection sort starts of by assuming the first element is the smallest and holds its index in a variable, in my example, that variable is x. It then traverses the array to compare each element with that in index x. The first of these traversals takes (*n* – 1) iterations, and reduces by 1 with each successive iteration as the sorted sub list grows. This happens for (*n* – 1) times.

Evidently, the quality of the dataset, or how sorted a dataset is, does not affect the runtime of selection sort significantly. The runtime of selection sort on the 1000000 almost sorted array (19m 52.496s) is comparable to that of selection sort on the 1000000 randomly sorted array (19m 49.934s). This is because in selection sort the number of iterations remain unchanged regardless of how pre-sorted a dataset is.

Although selection sort and bubble sort have the same time complexity of O(n2), selection sort is noticeably faster (more efficient) than bubble sort because it only performs the swap operation once every time it finishes the inner loop. That is, while bubble sort performs a swap every time an adjacent element is smaller, selection sort simply searches for the smallest element and swaps it to the back of the sorted sub list. This drastically reduces the number of swap operations that needs to be done, thereby resulting in a huge runtime reduction.

Insertion Sort

**Data: “1000000almostsorted.bin” (ALMOST SORTED)**

Start time: Tue Sep 15 05:32:14.496 2020

End time: Tue Sep 15 05:32:53.158 2020

Time taken: approximately 38.662s

Lower bound: 1200

Upper bound: 1300

Sample list: 1192 1193 1194 1198 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1292

**Data: “1000000numbers.bin” (UNSORTED)**

Start time: Tue Sep 15 05:17:38.926 2020

End time: Tue Sep 15 05:27:25.396 2020

Time taken: approximately 9m 46.47s

Lower bound: 555000

Upper bound: 555200

Sample list: 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18198 18198 18198 18198 18198 18198 18198 18198 18198

Insertion Sort Description

The worst and average case time complexities of insertion sort are O(n2), owing to the two nested loops. The worst-case time complexity occurs when an array or dataset is reverse sorted. In our case, since we want to sort the array in ascending order, the worst-case scenario will be that of an array sorted in descending order. This is because a reverse sorted array will result in shifting of the entire sorted sub list with every subsequent element. The best-case time complexity of insertion sort is O(n). This is because the outer loop will only need to compare the first element of the unsorted sub list with the sorted sub list, resulting in just *n* iterations. This is apparent when comparing the runtimes of insertion sort when sorting an almost sorted array (38.662s) and a randomly sorted array (9m 46.47s) of 1000000 elements each.

Like selection sort, insertion sort has a sub list of sorted elements that grow with each iteration. However, instead of traversing the entire array looking for the smallest element, it takes the first element of the un-sorted sub list and places it in the appropriate location in the sorted sub list by means of shifting elements in the sorted sub list. Therefore, insertion sort will perform significantly less comparisons than selection sort in general.

Shell Sort

**Data: “1000000almostsorted.bin” (ALMOST SORTED)**

Start time: Tue Sep 15 05:07:56.916 2020

End time: Tue Sep 15 05:07:57.58 2020

Time taken: approximately 0.664s

Lower bound: 1200

Upper bound: 1300

Sample list: 1192 1193 1194 1198 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1292

**Data: “1000000numbers.bin” (UNSORTED)**

Start time: Tue Sep 15 05:15:03.25 2020

End time: Tue Sep 15 05:15:04.22 2020

Time taken: approximately 0.970s

Lower bound: 555000

Upper bound: 555200

Sample list: 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18192 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18193 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18194 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18195 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18196 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18197 18198 18198 18198 18198 18198 18198 18198 18198 18198

Shell Sort Description

The average-case time complexity of shell sort depends on the size of the gaps corresponding to the number of shells; one estimate puts the average time complexity at O(n3/2). The worst-case time complexity of shell sort is O(n2) because the worst-case of shell sort is basically an insertion sort. Finally, the best-case time complexity for shell sort can be said to be approximately O(*n* log *n*), and it trends towards an O(n) for a perfectly sorted dataset.

Shell sort is a modified version of an insertion sort where it tries to ameliorate the latter’s most computationally expensive process – the shifting of large numbers of elements, especially if a small element (assuming sorting in ascending order) is found at the end of an array. Shell sort does this by breaking the elements into shells of a specific gap size. Letting the gap size be *k*, shell sort treats every *k*th element as a group an performs an insertion sort on the elements within this group, thereby shifting elements over larger spans of elements. Smaller elements get progressively shifted to the front of the array with each pass. This makes subsequent passes increasingly faster. At the end of each pass, the gap size is reduced, and the same process continues. In this assignment, the algorithm reduces the gap size by dividing it by 3. Here, 3 is an arbitrary number that has been shown to result in some of the more efficient shell sorts (i.e. fastest runtime).

Compared to the other sorts in this assignment, shell sort is the most efficient for both almost sorted (approximately 0.664s) and unsorted (approximately 0.970s) data of 1000000 numbers.