**Lab 6 Sorting Algorithms**

The timings for all my different sorting algorithms look like they match up with exactly what we predicted in class. This is due to the various ways that they perform their sorting and the various operations, and ghost operations that take place within each of them. For example, in bubble sort, we have nested loops because the operation of the algorithm depends on the number of comparisons and the number of swaps which are both based on “n”. So, that means the outer nested loops runs n-1 times and the inner loop runs n times because it depends on the number of swaps it needs to do on each iteration. This means it runs at n^2 for its time complexity. Insertion Sort and Selection Sort also run on n^2 times, but the methods employed are slightly different.

I have all my code pasted below the timing graphs, and I think I’ve been better about commenting inside of it to better explain things, but I just wanted to emphasize how I defined my functions for each of the algorithms because it was something new that I learned with this lab. I’ve been using templates for class definitions, but this is where I discovered that I can write individual functions as a template as well! This was something I found recommended on cplusplus.com where they made it clear that it helps make my code a little more efficient as far as the implementation and readability. This is because I only had to write each algorithm once and that makes it usable for any data type I want to pass through it. Now, I know this sounds like our talk about using “auto” in front of our timing code because I’m not specifying the data/return type of the function, but I believe this is different. Auto just looks at whatever the given variable is and deduces what type of data is being passed through. This is local to that given variable, while templates encompass entire functions or classes.

**Screenshots of Timing Graphs**

A graph with a line and a dotted line

AI-generated content may be incorrect.A graph with a line and a dotted line

AI-generated content may be incorrect.

A graph with a line and a dotted line

AI-generated content may be incorrect.A graph with a line and a dotted line

AI-generated content may be incorrect.

A graph with a line

AI-generated content may be incorrect.A graph showing the growth of a fraction

AI-generated content may be incorrect.

A graph with a line and numbers

AI-generated content may be incorrect.A graph showing a line of fractions

AI-generated content may be incorrect.

**Code Appendix**

Just a quick explanation – this lab took me extra time simply because I became too hyper focused on getting the program to be able to do all the timings with the same vector of integers and fractions all in one go, as well as getting all the results to correctly be put into the csv file so that I could generate separate graphs.

Also, in the loops where I test the different algorithms, I’ve been using range-based loops that look like:

1. for (int size : testSize)

They talk about this in chapter 14 of our ZyBook (I don’t think you ever assigned it, but it’s super convenient to use ZyBook’s search feature to learn about all of the various syntax I can implement in C++), and it’s a much cleaner and simpler look when laying out a for loop where I don’t need to worry about the index for anything. I’m pretty sure I had used this in earlier labs, but I realized I never explained it before.

1. 1. #include "fraction.h"

2. 2. #include <algorithm>

3. 3. #include <chrono>

4. 4. #include <fstream>

5. 5. #include <iomanip>

6. 6. #include <iostream>

7. 7. #include <vector>

8. 8.

9. 9. using namespace std;

10. 10.

11. 11. /\* Ater trying, and researching a few different ways to implement these

12. 12. \* algorithms, I decided to define each one of them individually as a template.

13. 13. \* This way I don't have to write each function

14. 14. \* twice (1 for ints and 1 for fractions).\*/

15. 15. template <typename T> void bubbleSort(vector<T> &alist) {

16. 16. for (int sznum = alist.size() - 1; sznum > 0; sznum--) {

17. 17. for (int i = 0; i < sznum; i++) {

18. 18. if (alist[i] > alist[i + 1]) {

19. 19. T temp = alist[i];

20. 20. alist[i] = alist[i + 1];

21. 21. alist[i + 1] = temp;

22. 22. }

23. 23. }

24. 24. }

25. 25. }

26. 26.

27. 27. template <typename T> void insertionSort(vector<T> &alist) {

28. 28. for (int i = 1; i < alist.size(); i++) {

29. 29. T key = alist[i];

30. 30. int j = i - 1;

31. 31.

32. 32. while (j >= 0 && alist[j] > key) {

33. 33. alist[j + 1] = alist[j];

34. 34. j = j - 1;

35. 35. }

36. 36. alist[j + 1] = key;

37. 37. }

38. 38. }

39. 39.

40. 40. template <typename T> void selectionSort(vector<T> &alist) {

41. 41. for (int i = 0; i < alist.size() - 1; i++) {

42. 42. int min\_index = i;

43. 43. for (int j = i + 1; j < alist.size(); j++) {

44. 44. if (alist[j] < alist[min\_index]) {

45. 45. min\_index = j;

46. 46. }

47. 47. }

48. 48. T temp = alist[i];

49. 49. alist[i] = alist[min\_index];

50. 50. alist[min\_index] = temp;

51. 51. }

52. 52. }

53. 53.

54. 54. template <typename T> void shellSort(vector<T> &alist) {

55. 55. for (int gap = alist.size() / 2; gap > 0; gap /= 2) {

56. 56. for (int i = gap; i < alist.size(); i++) {

57. 57. T temp = alist[i];

58. 58. int j;

59. 59. for (j = i; j >= gap && alist[j - gap] > temp; j -= gap) {

60. 60. alist[j] = alist[j - gap];

61. 61. }

62. 62. alist[j] = temp;

63. 63. }

64. 64. }

65. 65. }

66. 66.

67. 67. unsigned long long getInt(int numDigits = 10) {

68. 68. unsigned long long num = 0;

69. 69.

70. 70. num = rand() % 9 + 1;

71. 71. for (int i = 0; i < numDigits - 1; i++) {

72. 72. num = num \* 10 + rand() % 10;

73. 73. }

74. 74. return num;

75. 75. }

76. 76.

77. 77. void getUniqueNums(int sz, vector<unsigned long long> &v) {

78. 78. v.clear();

79. 79. while (v.size() < sz) {

80. 80. unsigned long long num = getInt(9);

81. 81. if (find(v.begin(), v.end(), num) == v.end()) {

82. 82. v.push\_back(num);

83. 83. }

84. 84. }

85. 85. }

86. 86.

87. 87. void getUniqueFractions(int sz, vector<Fraction> &v) {

88. 88. v.clear();

89. 89. while (v.size() < sz) {

90. 90. int num = rand() % 1000 - 500;

91. 91. int den = rand() % 999 + 1;

92. 92. Fraction f(num, den);

93. 93.

94. 94. bool duplicate = false;

95. 95. for (const Fraction &existing : v) {

96. 96. if (f == existing) {

97. 97. duplicate = true;

98. 98. break;

99. 99. }

100. 100. }

101. 101.

102. 102. if (!duplicate) {

103. 103. v.push\_back(f);

104. 104. }

105. 105. }

106. 106. }

107. 107.

108. 108. int main() {

109. 109. srand(time(NULL));

110. 110. ofstream csvFile("sorting\_timing.csv");

111. 111. csvFile << "Algorithm,DataType,Size,Time(milliseconds)" << endl;

112. 112.

113. 113. cout << "Generating Timing Data..." << endl;

114. 114.

115. 115. vector<int> testSize = {10000, 20000, 30000, 40000, 50000,

116. 116. 60000, 70000, 80000, 90000, 100000};

117. 117.

118. 118. // BubbleSort algorithm

119. 119. for (int size : testSize) {

120. 120. // Generate the test data

121. 121. vector<unsigned long long> originalIntVec;

122. 122. vector<Fraction> originalFracVec;

123. 123.

124. 124. getUniqueNums(size, originalIntVec);

125. 125. getUniqueFractions(size, originalFracVec);

126. 126. cout << size << " unique elements generated!" << endl;

127. 127.

128. 128. // Test BubbleSort with integers

129. 129. vector<unsigned long long> intVecCopy = originalIntVec;

130. 130. chrono::high\_resolution\_clock::time\_point start =

131. 131. chrono::high\_resolution\_clock::now();

132. 132. bubbleSort(intVecCopy);

133. 133. chrono::high\_resolution\_clock::time\_point end =

134. 134. chrono::high\_resolution\_clock::now();

135. 135. chrono::duration<double, std::milli> time\_span = end - start;

136. 136.

137. 137. csvFile << "BubbleSort,Integer," << size << "," << fixed << setprecision(4)

138. 138. << time\_span.count() << endl;

139. 139. cout << "BubbleSort (Integer, " << size << ") completed!" << endl;

140. 140.

141. 141. // Test BubbleSort with fractions

142. 142. vector<Fraction> fracVecCopy = originalFracVec;

143. 143. start = chrono::high\_resolution\_clock::now();

144. 144. bubbleSort(fracVecCopy);

145. 145. end = chrono::high\_resolution\_clock::now();

146. 146. time\_span = end - start;

147. 147.

148. 148. csvFile << "BubbleSort,Fraction," << size << "," << fixed << setprecision(4)

149. 149. << time\_span.count() << endl;

150. 150. cout << "BubbleSort (Fraction, " << size << ") completed!" << endl;

151. 151. }

152. 152.

153. 153. // InsertionSort algorithm

154. 154. for (int size : testSize) {

155. 155. // Generate the test data

156. 156. vector<unsigned long long> originalIntVec;

157. 157. vector<Fraction> originalFracVec;

158. 158.

159. 159. getUniqueNums(size, originalIntVec);

160. 160. getUniqueFractions(size, originalFracVec);

161. 161.

162. 162. // Test InsertionSort with integers

163. 163. vector<unsigned long long> intVecCopy = originalIntVec;

164. 164. chrono::high\_resolution\_clock::time\_point start =

165. 165. chrono::high\_resolution\_clock::now();

166. 166. insertionSort(intVecCopy);

167. 167. chrono::high\_resolution\_clock::time\_point end =

168. 168. chrono::high\_resolution\_clock::now();

169. 169. chrono::duration<double, std::milli> time\_span = end - start;

170. 170.

171. 171. csvFile << "InsertionSort,Integer," << size << "," << fixed

172. 172. << setprecision(4) << time\_span.count() << endl;

173. 173. cout << "InsertionSort (Integer, " << size << ") completed!" << endl;

174. 174.

175. 175. // Test InsertionSort with fractions

176. 176. vector<Fraction> fracVecCopy = originalFracVec;

177. 177. start = chrono::high\_resolution\_clock::now();

178. 178. insertionSort(fracVecCopy);

179. 179. end = chrono::high\_resolution\_clock::now();

180. 180. time\_span = end - start;

181. 181.

182. 182. csvFile << "InsertionSort,Fraction," << size << "," << fixed

183. 183. << setprecision(4) << time\_span.count() << endl;

184. 184. cout << "InsertionSort (Fraction, " << size << ") completed!" << endl;

185. 185. }

186. 186.

187. 187. // SelectionSort algorithm

188. 188. for (int size : testSize) {

189. 189. // Generate the test data

190. 190. vector<unsigned long long> originalIntVec;

191. 191. vector<Fraction> originalFracVec;

192. 192.

193. 193. getUniqueNums(size, originalIntVec);

194. 194. getUniqueFractions(size, originalFracVec);

195. 195.

196. 196. // Test SelectionSort with integers

197. 197. vector<unsigned long long> intVecCopy = originalIntVec;

198. 198. chrono::high\_resolution\_clock::time\_point start =

199. 199. chrono::high\_resolution\_clock::now();

200. 200. selectionSort(intVecCopy);

201. 201. chrono::high\_resolution\_clock::time\_point end =

202. 202. chrono::high\_resolution\_clock::now();

203. 203. chrono::duration<double, std::milli> time\_span = end - start;

204. 204.

205. 205. csvFile << "SelectionSort,Integer," << size << "," << fixed

206. 206. << setprecision(4) << time\_span.count() << endl;

207. 207. cout << "SelectionSort (Integer, " << size << ") completed!" << endl;

208. 208.

209. 209. // Test SelectionSort with fractions

210. 210. vector<Fraction> fracVecCopy = originalFracVec;

211. 211. start = chrono::high\_resolution\_clock::now();

212. 212. selectionSort(fracVecCopy);

213. 213. end = chrono::high\_resolution\_clock::now();

214. 214. time\_span = end - start;

215. 215.

216. 216. csvFile << "SelectionSort,Fraction," << size << "," << fixed

217. 217. << setprecision(4) << time\_span.count() << endl;

218. 218. cout << "SelectionSort (Fraction, " << size << ") completed!" << endl;

219. 219. }

220. 220.

221. 221. // ShellSort algorithm

222. 222. for (int size : testSize) {

223. 223. // Generate the test data

224. 224. vector<unsigned long long> originalIntVec;

225. 225. vector<Fraction> originalFracVec;

226. 226.

227. 227. getUniqueNums(size, originalIntVec);

228. 228. getUniqueFractions(size, originalFracVec);

229. 229.

230. 230. // Test ShellSort with integers

231. 231. vector<unsigned long long> intVecCopy = originalIntVec;

232. 232. chrono::high\_resolution\_clock::time\_point start =

233. 233. chrono::high\_resolution\_clock::now();

234. 234. shellSort(intVecCopy);

235. 235. chrono::high\_resolution\_clock::time\_point end =

236. 236. chrono::high\_resolution\_clock::now();

237. 237. chrono::duration<double, std::milli> time\_span = end - start;

238. 238.

239. 239. csvFile << "ShellSort,Integer," << size << "," << fixed << setprecision(4)

240. 240. << time\_span.count() << endl;

241. 241. cout << "ShellSort (Integer, " << size << ") completed!" << endl;

242. 242.

243. 243. // Test ShellSort with fractions

244. 244. vector<Fraction> fracVecCopy = originalFracVec;

245. 245. start = chrono::high\_resolution\_clock::now();

246. 246. shellSort(fracVecCopy);

247. 247. end = chrono::high\_resolution\_clock::now();

248. 248. time\_span = end - start;

249. 249.

250. 250. csvFile << "ShellSort,Fraction," << size << "," << fixed << setprecision(4)

251. 251. << time\_span.count() << endl;

252. 252. cout << "ShellSort (Fraction, " << size << ") completed!" << endl;

253. 253. }

254. 254.

255. 255. csvFile.close();

256. 256. cout << "All tests completed! Results saved to sorting\_timing.csv" << endl;

257. 257.

258. 258. return 0;

259. 259. }