**PROGRAMA PROFESIONAL**

Ciencias de la Computación

**TÍTULO DEL TRABAJO**

Comparación de Algoritmos

**CURSO**

Análisis y Diseño de Algoritmos

**ALUMNOS**

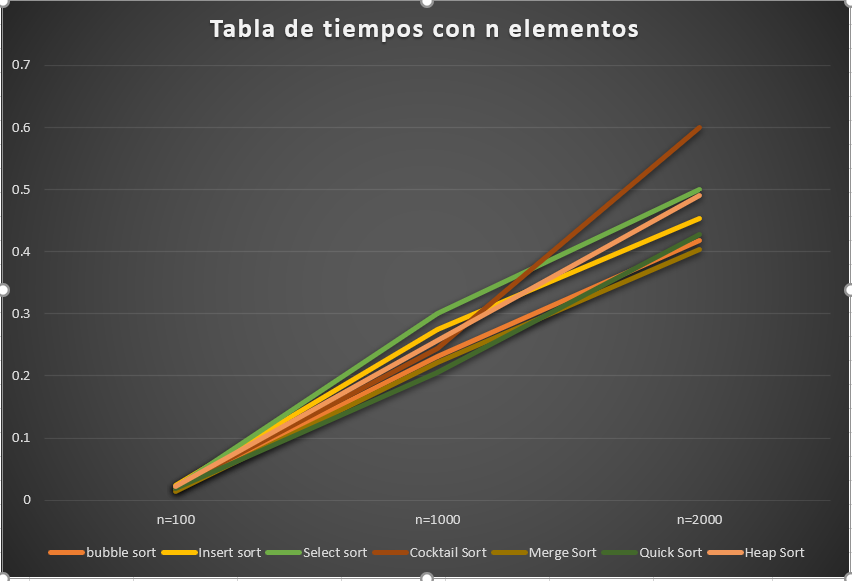
* Angel Josue Loayza Huarachi

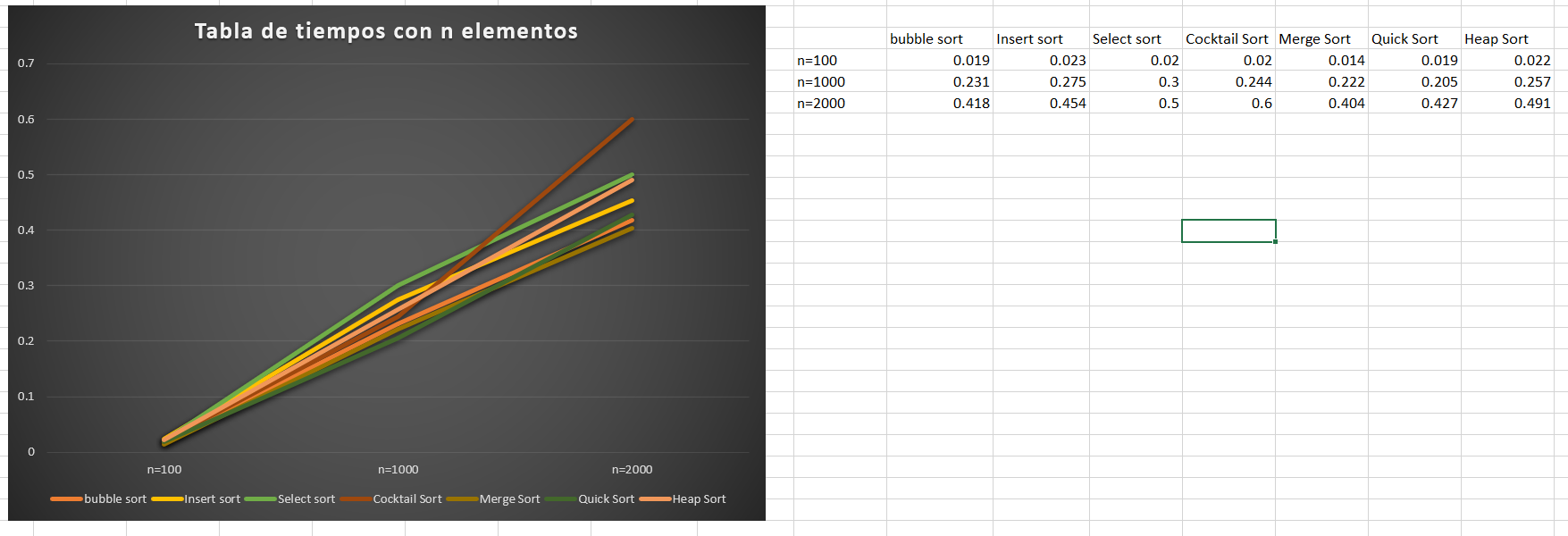
**SEMESTRE:** V

**AÑO:** 2022

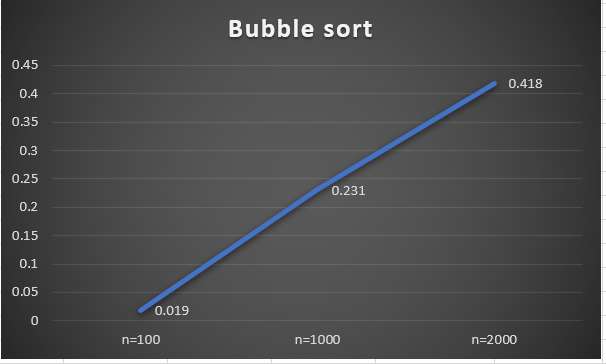
“El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo

1. Gráficos
   1. Tabla General:

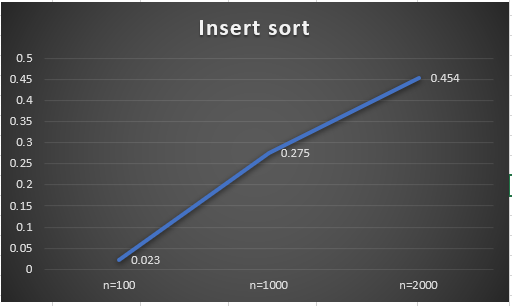




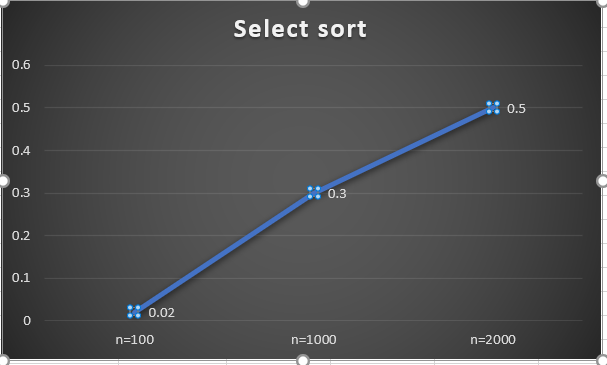
* 1. Bubble Sort



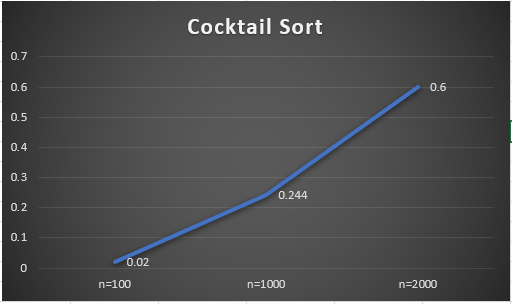
* 1. Insert Sort



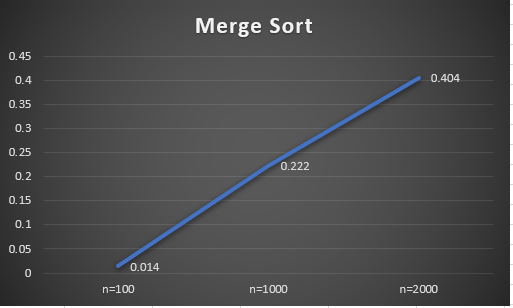
* 1. Select Sort



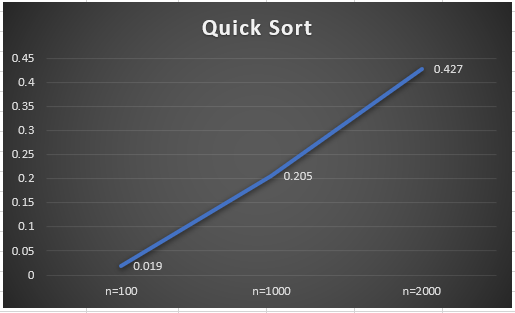
* 1. Cocktail Sort



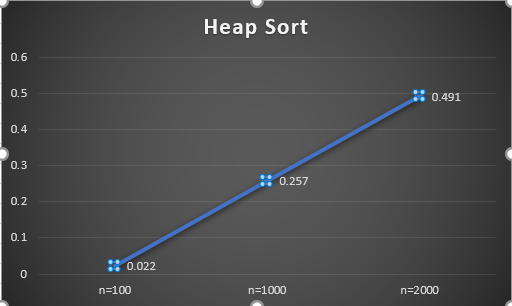
* 1. Merge Sort



* 1. Quick Sort



* 1. Heap Sort



1. Códigos
2. #include <iostream>
3. #include <ctime>
4. //#include "Header.h"
5. using namespace std;
6. void printArray(int A[], int size)
7. {
8. cout << "----------------------------------------------" << endl;
9. for (auto i = 0; i < size; i++)
10. cout << A[i] << " ";
11. }
12. //bouble random
13. void boobler(int a[], int n)
14. {
15. printArray(a, n);
16. cout << endl;
17. srand(time(NULL));
18. int temp;
19. for (int i = 0; i < n; i++) {
20. for (int j = i + 1; j < n; j++)
21. {
22. if (a[j] < a[i]) {
23. temp = a[i];
24. a[i] = a[j];
25. a[j] = temp;
26. }
27. }
28. }
29. printArray(a, n);
30. }
31. //insert random
32. void insertr(int a[], int n)
33. {
34. printArray(a, n);
35. cout << endl;
36. srand(time(NULL));
37. for (int i = 1; i < n; i++)
38. {
39. int temp = a[i];
40. int j = i - 1;
41. while (j >= 0 && temp <= a[j])
42. {
43. a[j + 1] = a[j];
44. j = j - 1;
45. }
46. a[j + 1] = temp;
47. }
48. printArray(a, n);
49. }
50. //select random
51. void selectionr(int a[], int n)
52. {
53. printArray(a, n);
54. cout << endl;
55. srand(time(NULL));
56. int i, j, loc, temp, min;
58. for (i = 0; i < n - 1; i++)
59. {
60. min = a[i];
61. loc = i;
62. for (j = i + 1; j < n; j++)
63. {
64. if (min > a[j])
65. {
66. min = a[j];
67. loc = j;
68. }
69. }
70. temp = a[i];
71. a[i] = a[loc];
72. a[loc] = temp;
73. }
74. printArray(a, n);
75. }
76. //cocktail random
77. void cocktailr(int a[], int n)
78. {
79. printArray(a, n);
80. cout << endl;
81. srand(time(NULL));
82. bool swapped = true;
83. int start = 0;
84. int end = n - 1;
85. while (swapped) {
86. swapped = false;
87. for (int i = start; i < end; ++i) {
88. if (a[i] > a[i + 1]) {
89. swap(a[i], a[i + 1]);
90. swapped = true;
91. }
92. }
93. if (!swapped)
94. break;
95. swapped = false;
96. --end;
97. for (int i = end - 1; i >= start; --i) {
98. if (a[i] > a[i + 1]) {
99. swap(a[i], a[i + 1]);
100. swapped = true;
101. }
102. }
103. ++start;
104. }
105. printArray(a, n);
106. }
107. //merge random
108. void mergeauxr(int array[], int const left, int const mid, int const right)
109. {
110. auto const subArrayOne = mid - left + 1;
111. auto const subArrayTwo = right - mid;
112. // Create temp arrays
113. auto\* leftArray = new int[subArrayOne],
114. \* rightArray = new int[subArrayTwo];
115. // Copy data to temp arrays leftArray[] and rightArray[]
116. for (auto i = 0; i < subArrayOne; i++)
117. leftArray[i] = array[left + i];
118. for (auto j = 0; j < subArrayTwo; j++)
119. rightArray[j] = array[mid + 1 + j];
120. auto indexOfSubArrayOne = 0, // Initial index of first sub-array
121. indexOfSubArrayTwo = 0; // Initial index of second sub-array
122. int indexOfMergedArray = left; // Initial index of merged array
123. // Merge the temp arrays back into array[left..right]
124. while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
125. if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
126. array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
127. indexOfSubArrayOne++;
128. }
129. else {
130. array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
131. indexOfSubArrayTwo++;
132. }
133. indexOfMergedArray++;
134. }
135. // Copy the remaining elements of
136. // left[], if there are any
137. while (indexOfSubArrayOne < subArrayOne) {
138. array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
139. indexOfSubArrayOne++;
140. indexOfMergedArray++;
141. }
142. // Copy the remaining elements of
143. // right[], if there are any
144. while (indexOfSubArrayTwo < subArrayTwo) {
145. array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
146. indexOfSubArrayTwo++;
147. indexOfMergedArray++;
148. }
149. }
150. void merger(int array[], int const begin, int const end)
151. {
152. if (begin >= end)
153. return; // Returns recursively
154. auto mid = begin + (end - begin) / 2;
155. merger(array, begin, mid);
156. merger(array, mid + 1, end);
157. mergeauxr(array, begin, mid, end);
158. }
159. //quick random
160. int partitionr(int arr[], int start, int end)
161. {
162. int pivot = arr[start];
163. int count = 0;
164. for (int i = start + 1; i <= end; i++) {
165. if (arr[i] <= pivot)
166. count++;
167. }
168. // Giving pivot element its correct position
169. int pivotIndex = start + count;
170. swap(arr[pivotIndex], arr[start]);
171. // Sorting left and right parts of the pivot element
172. int i = start, j = end;
173. while (i < pivotIndex && j > pivotIndex) {
174. while (arr[i] <= pivot) {
175. i++;
176. }
177. while (arr[j] > pivot) {
178. j--;
179. }
180. if (i < pivotIndex && j > pivotIndex) {
181. swap(arr[i++], arr[j--]);
182. }
183. }
184. return pivotIndex;
185. }
186. void quickr(int arr[], int start, int end)
187. {
188. // base case
189. if (start >= end)
190. return;
191. // partitioning the array
192. int p = partitionr(arr, start, end);
193. // Sorting the left part
194. quickr(arr, start, p - 1);
195. // Sorting the right part
196. quickr(arr, p + 1, end);
197. }
198. //heap random
199. void heapifyr(int arr[], int n, int i)
200. {
201. int largest = i; // Initialize largest as root
202. int l = 2 \* i + 1; // left = 2\*i + 1
203. int r = 2 \* i + 2; // right = 2\*i + 2
204. // If left child is larger than root
205. if (l < n && arr[l] > arr[largest])
206. largest = l;
207. // If right child is larger than largest so far
208. if (r < n && arr[r] > arr[largest])
209. largest = r;
210. // If largest is not root
211. if (largest != i) {
212. swap(arr[i], arr[largest]);
213. // Recursively heapify the affected sub-tree
214. heapifyr(arr, n, largest);
215. }
216. }
217. // main function to do heap sort
218. void heapr(int arr[], int n)
219. {
220. // Build heap (rearrange array)
221. for (int i = n / 2 - 1; i >= 0; i--)
222. heapifyr(arr, n, i);
223. // One by one extract an element from heap
224. for (int i = n - 1; i >= 0; i--) {
225. // Move current root to end
226. swap(arr[0], arr[i]);
227. // call max heapify on the reduced heap
228. heapifyr(arr, i, 0);
229. }
230. }
231. int main()
232. {
233. // -------- Creacion de numeros aleatorios ---------
234. int a[2000];
235. int n = 2000;
236. for (int i = 0; i < n; i++)
237. {
238. a[i] = rand() % (100 - 1 + 1) + 1;
239. }
240. //---------- Algoritmos ----------------------------
241. unsigned t0, t1;
242. t0 = clock();
243. //boobler(a,n);
244. //insertr(a,n);
245. //selectionr(a,n);
246. //cocktailr(a,n);
247. /\*
248. printArray(a, n);
249. cout << endl;
250. merger(a, 0, n - 1);
251. printArray(a, n);
252. \*/
253. /\*
254. printArray(a, n);
255. cout << endl;
256. quickr(a, 0, n - 1);
257. printArray(a, n);
258. \*/
259. printArray(a, n);
260. cout << endl;
261. heapr(a, n);
262. printArray(a, n);
263. t1 = clock();
264. double time = (double(t1 - t0) / CLOCKS\_PER\_SEC);
265. cout << endl << "Execution Time (s): " << time << endl;
267. return 0;
268. }