



Universidad Católica
San Pablo

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

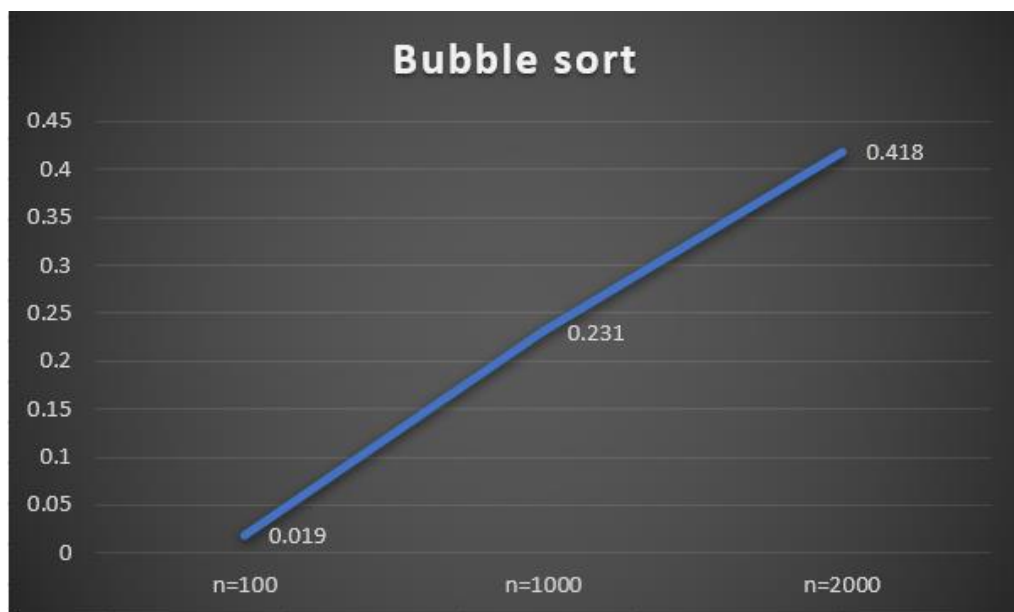
1. Gráficos

a. Tabla General:

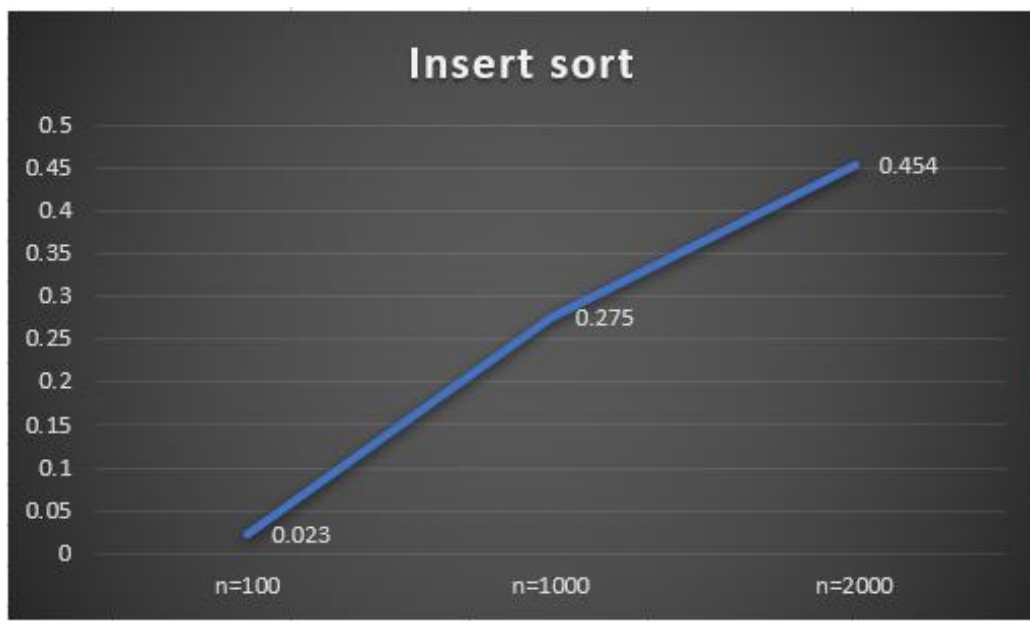


	bubble sort	Insert sort	Select sort	Cocktail Sort	Merge Sort	Quick Sort	Heap Sort
n=100	0.019	0.023	0.02	0.02	0.014	0.019	0.022
n=1000	0.231	0.275	0.3	0.244	0.222	0.205	0.257
n=2000	0.418	0.454	0.5	0.6	0.404	0.427	0.491

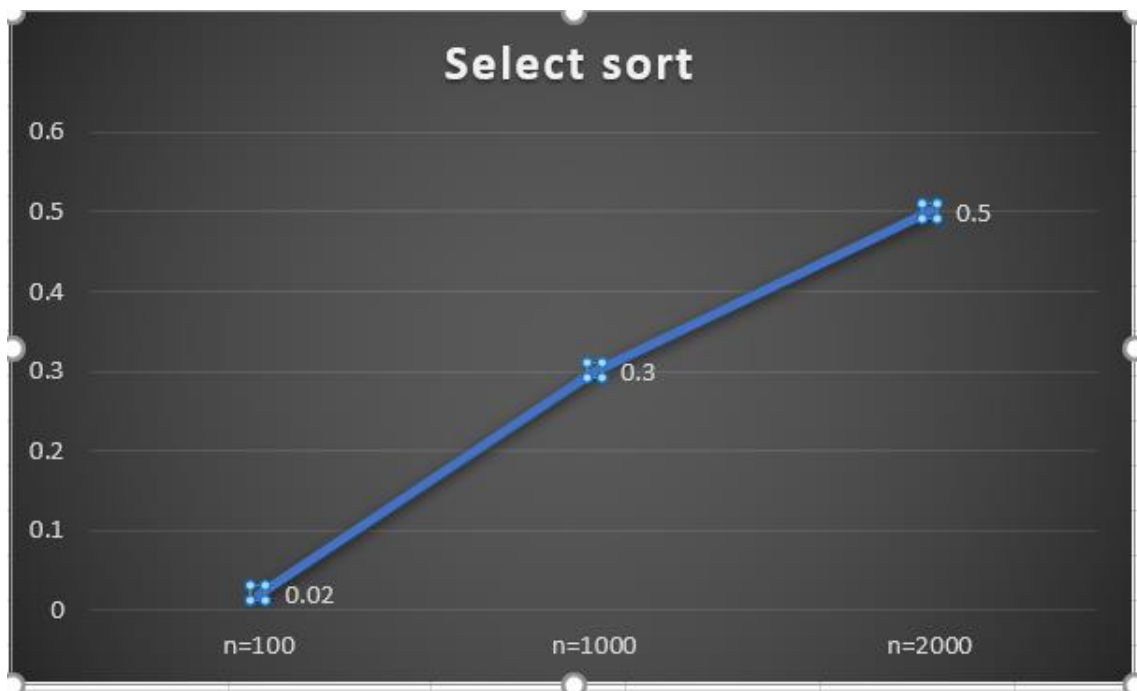
b. Bubble Sort



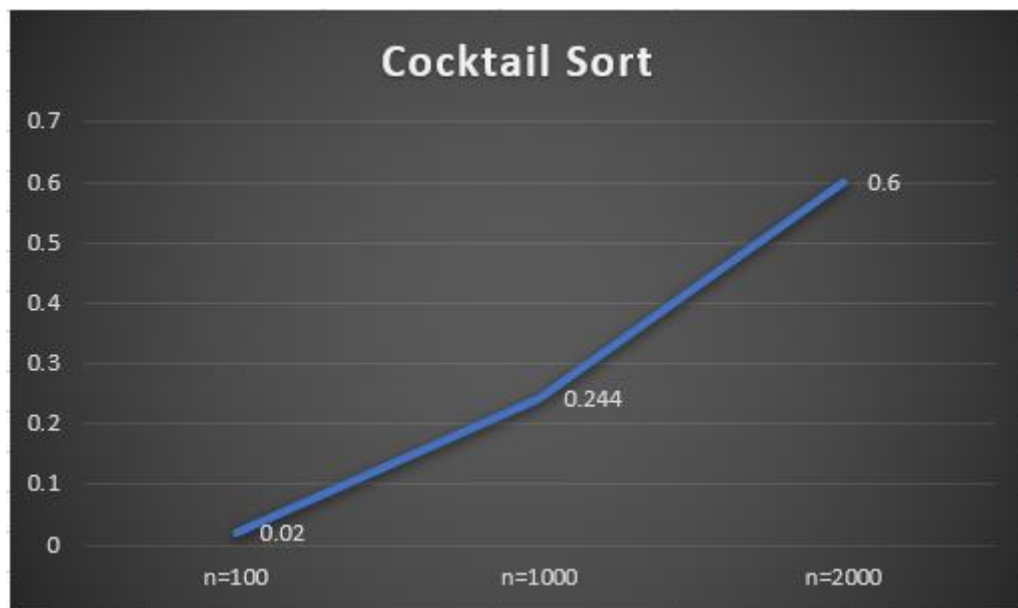
c. Insert Sort



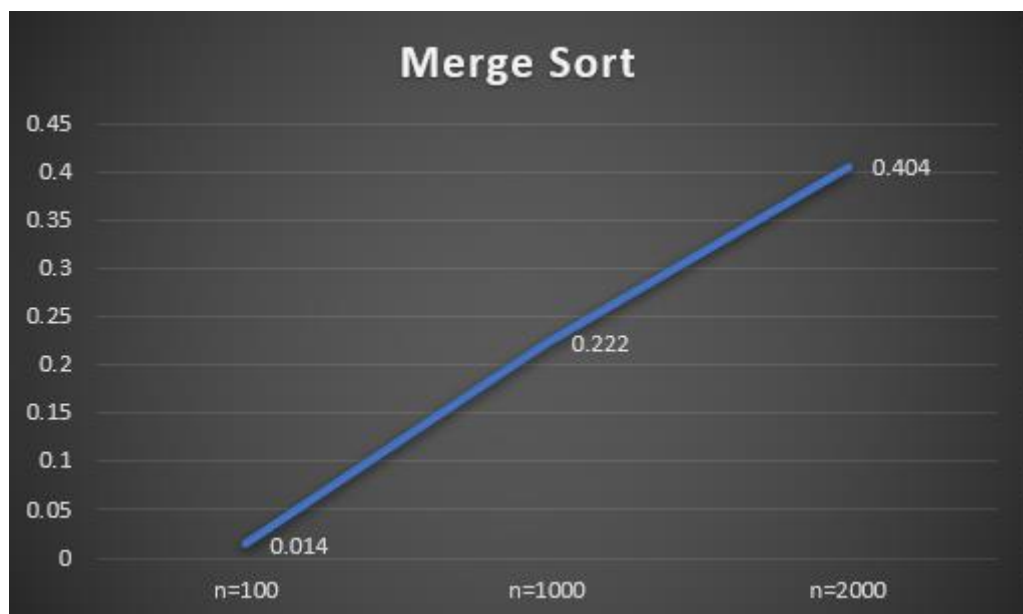
d. Select Sort



e. Cocktail Sort



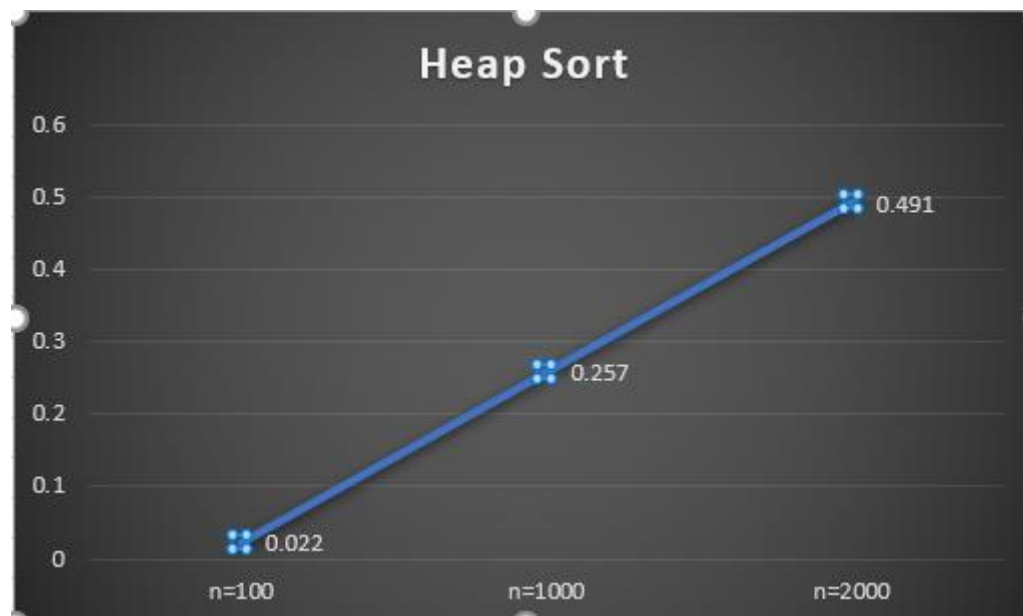
f. Merge Sort



g. Quick Sort



h. Heap Sort



2. Códigos

```
3. #include <iostream>
4. #include <ctime>
5. // #include "Header.h"
6. using namespace std;
7.
8. void printArray(int A[], int size)
9. {
10.     cout << "-----" << endl;
11.     for (auto i = 0; i < size; i++)
12.         cout << A[i] << " ";
13. }
14.
15. //bubble random
16. void boobler(int a[], int n)
17. {
18.     printArray(a, n);
19.     cout << endl;
20.
21.     srand(time(NULL));
22.     int temp;
23.
24.     for (int i = 0; i < n; i++) {
25.         for (int j = i + 1; j < n; j++)
26.         {
27.             if (a[j] < a[i]) {
28.                 temp = a[i];
29.                 a[i] = a[j];
30.                 a[j] = temp;
31.             }
32.         }
33.     }
34.     printArray(a, n);
35. }
36.
37. //insert random
38. void insertr(int a[], int n)
39. {
40.     printArray(a, n);
41.     cout << endl;
42.
43.     srand(time(NULL));
44.
45.     for (int i = 1; i < n; i++)
46.     {
47.         int temp = a[i];
48.         int j = i - 1;
49.         while (j >= 0 && temp <= a[j])
50.         {
51.             a[j + 1] = a[j];
52.             j = j - 1;
53.         }
54.         a[j + 1] = temp;
55.     }
56.
57.     printArray(a, n);
58. }
59.
60. //select random
61. void selectionr(int a[], int n)
62. {
63.     printArray(a, n);
64.     cout << endl;
```

```

65.
66.     srand(time(NULL));
67.
68.     int i, j, loc, temp, min;
69.
70.     for (i = 0; i < n - 1; i++)
71.     {
72.         min = a[i];
73.         loc = i;
74.
75.         for (j = i + 1; j < n; j++)
76.         {
77.             if (min > a[j])
78.             {
79.                 min = a[j];
80.                 loc = j;
81.             }
82.         }
83.
84.         temp = a[i];
85.         a[i] = a[loc];
86.         a[loc] = temp;
87.     }
88.
89.     printArray(a, n);
90. }
91.
92. //cocktail random
93. void cocktailr(int a[], int n)
94. {
95.     printArray(a, n);
96.     cout << endl;
97.
98.     srand(time(NULL));
99.
100.    bool swapped = true;
101.    int start = 0;
102.    int end = n - 1;
103.
104.    while (swapped) {
105.        swapped = false;
106.
107.        for (int i = start; i < end; ++i) {
108.            if (a[i] > a[i + 1]) {
109.                swap(a[i], a[i + 1]);
110.                swapped = true;
111.            }
112.        }
113.
114.        if (!swapped)
115.            break;
116.
117.        swapped = false;
118.        --end;
119.        for (int i = end - 1; i >= start; --i) {
120.            if (a[i] > a[i + 1]) {
121.                swap(a[i], a[i + 1]);
122.                swapped = true;
123.            }
124.        }
125.        ++start;
126.    }
127.

```

```

128.     printArray(a, n);
129. }
130.
131. //merge random
132. void mergeauxr(int array[], int const left, int const mid, int
const right)
133. {
134.     auto const subArrayOne = mid - left + 1;
135.     auto const subArrayTwo = right - mid;
136.
137.     // Create temp arrays
138.     auto* leftArray = new int[subArrayOne],
139.         * rightArray = new int[subArrayTwo];
140.
141.     // Copy data to temp arrays leftArray[] and rightArray[]
142.     for (auto i = 0; i < subArrayOne; i++)
143.         leftArray[i] = array[left + i];
144.     for (auto j = 0; j < subArrayTwo; j++)
145.         rightArray[j] = array[mid + 1 + j];
146.
147.     auto indexOfSubArrayOne = 0, // Initial index of first sub-
array
148.         indexOfSubArrayTwo = 0; // Initial index of second sub-
array
149.     int indexOfMergedArray = left; // Initial index of merged
array
150.     // Merge the temp arrays back into array[left..right]
151.     while (indexOfSubArrayOne < subArrayOne &&
indexOfSubArrayTwo < subArrayTwo) {
152.         if (leftArray[indexOfSubArrayOne] <=
rightArray[indexOfSubArrayTwo]) {
153.             array[indexOfMergedArray] =
leftArray[indexOfSubArrayOne];
154.             indexOfSubArrayOne++;
155.         }
156.         else {
157.             array[indexOfMergedArray] =
rightArray[indexOfSubArrayTwo];
158.             indexOfSubArrayTwo++;
159.         }
160.         indexOfMergedArray++;
161.     }
162.     // Copy the remaining elements of
163.     // left[], if there are any
164.     while (indexOfSubArrayOne < subArrayOne) {
165.         array[indexOfMergedArray] =
leftArray[indexOfSubArrayOne];
166.         indexOfSubArrayOne++;
167.         indexOfMergedArray++;
168.     }
169.     // Copy the remaining elements of
170.     // right[], if there are any
171.     while (indexOfSubArrayTwo < subArrayTwo) {
172.         array[indexOfMergedArray] =
rightArray[indexOfSubArrayTwo];
173.         indexOfSubArrayTwo++;
174.         indexOfMergedArray++;
175.     }
176. }
177.
178. void merger(int array[], int const begin, int const end)
179. {
180.     if (begin >= end)

```



```

181.         return; // Returns recursively
182.
183.         auto mid = begin + (end - begin) / 2;
184.         merger(array, begin, mid);
185.         merger(array, mid + 1, end);
186.         mergeauxr(array, begin, mid, end);
187.     }
188.
189.     //quick random
190.     int partitionr(int arr[], int start, int end)
191.     {
192.
193.         int pivot = arr[start];
194.
195.         int count = 0;
196.         for (int i = start + 1; i <= end; i++) {
197.             if (arr[i] <= pivot)
198.                 count++;
199.         }
200.
201.         // Giving pivot element its correct position
202.         int pivotIndex = start + count;
203.         swap(arr[pivotIndex], arr[start]);
204.
205.         // Sorting left and right parts of the pivot element
206.         int i = start, j = end;
207.
208.         while (i < pivotIndex && j > pivotIndex) {
209.             while (arr[i] <= pivot) {
210.                 i++;
211.             }
212.
213.             while (arr[j] > pivot) {
214.                 j--;
215.             }
216.
217.             if (i < pivotIndex && j > pivotIndex) {
218.                 swap(arr[i++], arr[j--]);
219.             }
220.         }
221.
222.         return pivotIndex;
223.     }
224.
225.     void quickr(int arr[], int start, int end)
226.     {
227.
228.         // base case
229.         if (start >= end)
230.             return;
231.
232.         // partitioning the array
233.         int p = partitionr(arr, start, end);
234.
235.         // Sorting the left part
236.         quickr(arr, start, p - 1);
237.
238.         // Sorting the right part
239.         quickr(arr, p + 1, end);
240.     }
241.
242.
243.     //heap random

```

```

244. void heapifyr(int arr[], int n, int i)
245. {
246.     int largest = i; // Initialize largest as root
247.     int l = 2 * i + 1; // left = 2*i + 1
248.     int r = 2 * i + 2; // right = 2*i + 2
249.
250.     // If left child is larger than root
251.     if (l < n && arr[l] > arr[largest])
252.         largest = l;
253.
254.     // If right child is larger than largest so far
255.     if (r < n && arr[r] > arr[largest])
256.         largest = r;
257.
258.     // If largest is not root
259.     if (largest != i) {
260.         swap(arr[i], arr[largest]);
261.
262.         // Recursively heapify the affected sub-tree
263.         heapifyr(arr, n, largest);
264.     }
265. }
266.
267. // main function to do heap sort
268. void heapr(int arr[], int n)
269. {
270.     // Build heap (rearrange array)
271.     for (int i = n / 2 - 1; i >= 0; i--)
272.         heapifyr(arr, n, i);
273.
274.     // One by one extract an element from heap
275.     for (int i = n - 1; i >= 0; i--) {
276.         // Move current root to end
277.         swap(arr[0], arr[i]);
278.
279.         // call max heapify on the reduced heap
280.         heapifyr(arr, i, 0);
281.     }
282. }
283.
284. int main()
285. {
286.     // ----- Creacion de numeros aleatorios -----
287.     int a[2000];
288.     int n = 2000;
289.
290.     for (int i = 0; i < n; i++)
291.     {
292.         a[i] = rand() % (100 - 1 + 1) + 1;
293.     }
294.     //----- Algoritmos -----
295.     unsigned t0, t1;
296.     t0 = clock();
297.
298.     //boobler(a,n);
299.     //insertr(a,n);
300.     //selectionr(a,n);
301.     //cocktailr(a,n);
302.
303.     /*
304.     printArray(a, n);
305.     cout << endl;
306.     merger(a, 0, n - 1);

```

```
307.     printArray(a, n);
308.     */
309.
310.     /*
311.     printArray(a, n);
312.     cout << endl;
313.     quickr(a, 0, n - 1);
314.     printArray(a, n);
315.     */
316.
317.     printArray(a, n);
318.     cout << endl;
319.     heapr(a, n);
320.     printArray(a, n);
321.
322.
323.     t1 = clock();
324.
325.     double time = (double(t1 - t0) / CLOCKS_PER_SEC);
326.     cout << endl << "Execution Time (s): " << time << endl;
327.
328.     return 0;
329. }
```