

Relatório Projeto 3 AED 2023/2024

Nome: Gonçalo José Carrajola Gaio

Nº Estudante: 2022224905





PL (inscrição): PL3

Email: goncalogaio12@gmail.com

IMPORTANTE:

- As conclusões devem ser manuscritas... texto que não obedeça a este requisito não é considerado.
- Texto para além das linhas reservadas, ou que não seja legível para um leitor comum, não é considerado.
- O relatório deve ser submetido num único PDF que deve incluir os anexos. A não observância deste formato é penalizada.

1. Planeamento

	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5
Insertion Sort					
Heap Sort					
Quick Sort					
Finalização Relatório					

2. Recolha de Resultados (tabelas)

N	10000	25000	50000	75000	100000	500000	1000000	5000000	10000000
Insertion Sort	0	1	1	2	2	4	5	6	7
	1	0	1	2	2	3	4	8	8
	0	1	1	2	1	3	4	6	8
	1	1	2	2	2	3	5	7	8
	0	1	1	2	2	2	4	8	8
Médias Alg1:	0.4	0.8	1.2	2	1.8	3	4.4	7	7.8
Heap Sort	2	5	7	8	10	42	73	372	745
	3	7	7	8	15	40	77	374	754
	3	7	7	9	9	41	76	375	741
	2	5	7	14	10	38	72	383	739
	2	4	7	11	10	39	76	380	763
Médias Alg2	2.4	5.6	7	10	10.8	40	74.8	376.8	748.4
Quick Sort	1	2	3	7	6	12	18	69	141
	1	3	3	5	7	13	20	70	143
	2	2	3	5	6	10	20	71	133
	1	2	3	4	7	10	20	70	160
	1	2	4	5	6	12	16	70	140
Médias Alg2	1.2	2.2	3.2	5.2	6.4	11.4	18.8	70	143.4

KEYS: A			
Valores de N:	10000	Médias Insertion Sort	0.4
	25000		0.8
	50000		1.2
	75000		2
	100000		1.8
	500000		3
	1000000		4.4
Médias Heap Sort	5000000	Médias Quick Sort	7
	10000000		7.8
	1		1.2
	3		2.2
	3		3.2
	5		5.2
	7		6.4
r-quadrado	13		11.4
	20		18.8
	70		70
	143		143.4
	0.9980371892		0.9986837299

N	10000	25000	50000	75000	100000	500000	1000000	5000000	10000000
Insertion Sort	21	77	302	639	1193	Valores demasiado elevados para serem aqui representados			
	15	105	299	684	1175				
	16	78	302	673	1164				
	16	78	301	664	1189				
	15	78	303	659	1173				
Médias Alg1:	16.6	83.2	301.4	663.8	1178.8				
Heap Sort	2	2	4	4	6	36	72	380	760
	2	3	3	4	7	35	73	380	774
	2	2	3	5	6	35	71	383	787
	3	2	3	5	6	33	70	382	769
	1	2	3	5	6	34	75	387	798
Médias Alg2	2	2.2	3.2	4.6	6.2	34.6	72.2	382.4	777.6
Quick Sort	1	2	6	5	6	17	60	187	382
	1	3	6	5	9	26	61	179	381
	1	2	7	6	6	19	45	189	376
	0	3	7	6	7	24	41	186	371
	1	2	6	6	6	19	38	187	377
Médias Alg2	0.8	2.4	6.4	5.6	6.8	21	49	185.6	377.4

KEYS: B			
Valores de N:	10000	Médias Insertion Sort	16.6
	25000		83.2
	50000		301.4
	75000		663.8
	100000		1178.8
	500000		0.9500593376
	10000000		0.9500593376
Médias Heap Sort	2	Médias Quick Sort	0.8
	2.2		2.4
	3.2		6.4
	4.6		5.6
	6.2		6.8
	34.6		21
	72.2		49
r-quadrado	382.4		185.6
	777.6		377.4
	0.9998975197		0.999215547

N	10000	25000	50000	75000	100000	500000	1000000	5000000	10000000
Insertion Sort	8	36	143	269	544	Valores demasiado elevados para serem aqui representados			
	8	52	146	278	523				
	8	38	144	284	554				
	8	41	153	281	549				
	9	40	148	277	495				
Médias Alg1:	8.2	41.4	146.8	277.8	533				
Heap Sort	1	2	4	7	11	60	141	973	2344
	2	3	5	7	11	59	157	1020	2436
	1	3	5	7	10	57	138	1008	2220
	1	2	5	7	10	57	134	1023	2601
	1	2	5	8	10	57	138	1028	2468
Médias Alg2	1.2	2.4	4.8	7.2	10.4	58	141.6	1010.4	2413.8
Quick Sort	1	3	2	4	6	31	71	395	834
	1	3	3	5	7	30	75	395	822
	1	3	3	4	5	33	72	411	792
	1	3	3	4	7	36	71	392	804
	1	3	3	4	6	29	71	420	831
Médias Alg2	1	3	2.8	4.2	6.2	31.8	72	402.6	816.6

KEYS: C			
Valores de N:	10000	Médias Insertion Sort	8.2
	25000		41.4
	50000		146.8
	75000		277.8
	100000		533
	500000		0.9428244568
	10000000		0.9428244568
Médias Heap Sort	1.2	Médias Quick Sort	1
	2.4		3
	4.8		2.8
	7.2		4.2
	10.4		6.2
	58		31.8
	141.6		72
r-quadrado	1010.4		402.6
	2413.8		816.6
	0.9937005097		0.9998006135

Comparação Keys Insertion Sort

Valores de N:	10000	Médias Keys A	0.4
	25000		0.8
	50000		1.2
	75000		2
	100000		1.8
	500000		3
	1000000		4.4
	5000000		7
Médias Keys B	10000000		7.8
	16.6	Médias Keys C	8.2
	83.2		41.4
	301.4		146.8
	663.8		277.8
	1178.8		533

Comparação Keys Heap Sort

Valores de N:	10000	Médias Keys A	1
	25000		3
	50000		3
	75000		5
	100000		7
	500000		13
	1000000		20
	5000000		70
Médias Keys B	10000000		143
	2	Médias Keys C	1.2
	2.2		2.4
	3.2		4.8
	4.6		7.2
	6.2		10.4
	34.6		58
	72.2		141.6
	382.4		1010.4
	777.6		2413.8

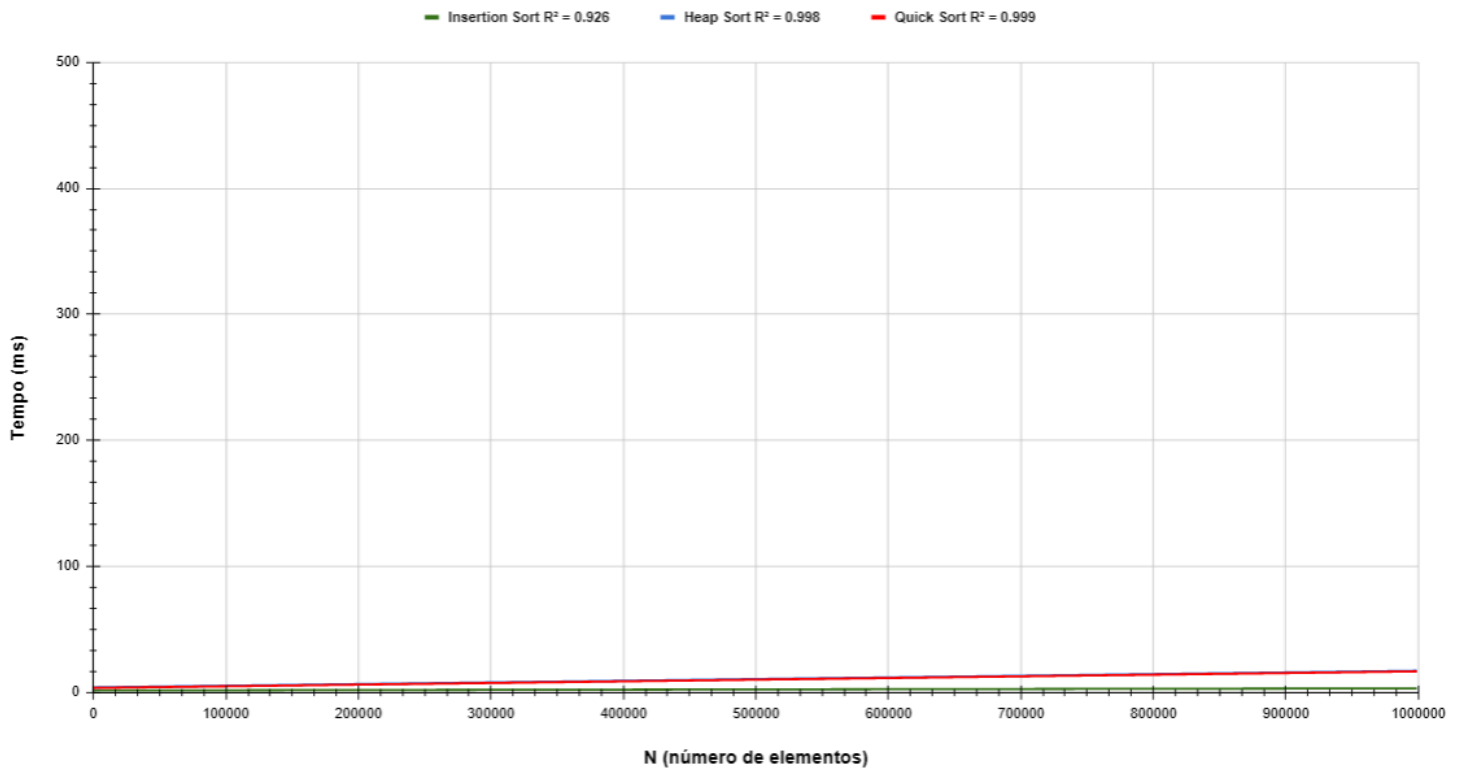
Comparação Keys Quick Sort

Valores de N:	10000	Médias Keys A	1.2
	25000		2.2
	50000		3.2
	75000		5.2
	100000		6.4
	500000		11.4
	1000000		18.8
	5000000		70
Médias Keys B	10000000		143.4
	0.8	Médias Keys C	1
	2.4		3
	6.4		2.8
	5.6		4.2
	6.8		6.2
	21		31.8
	49		72
	185.6		402.6
	377.4		816.6

3. Visualização de Resultados (gráficos)

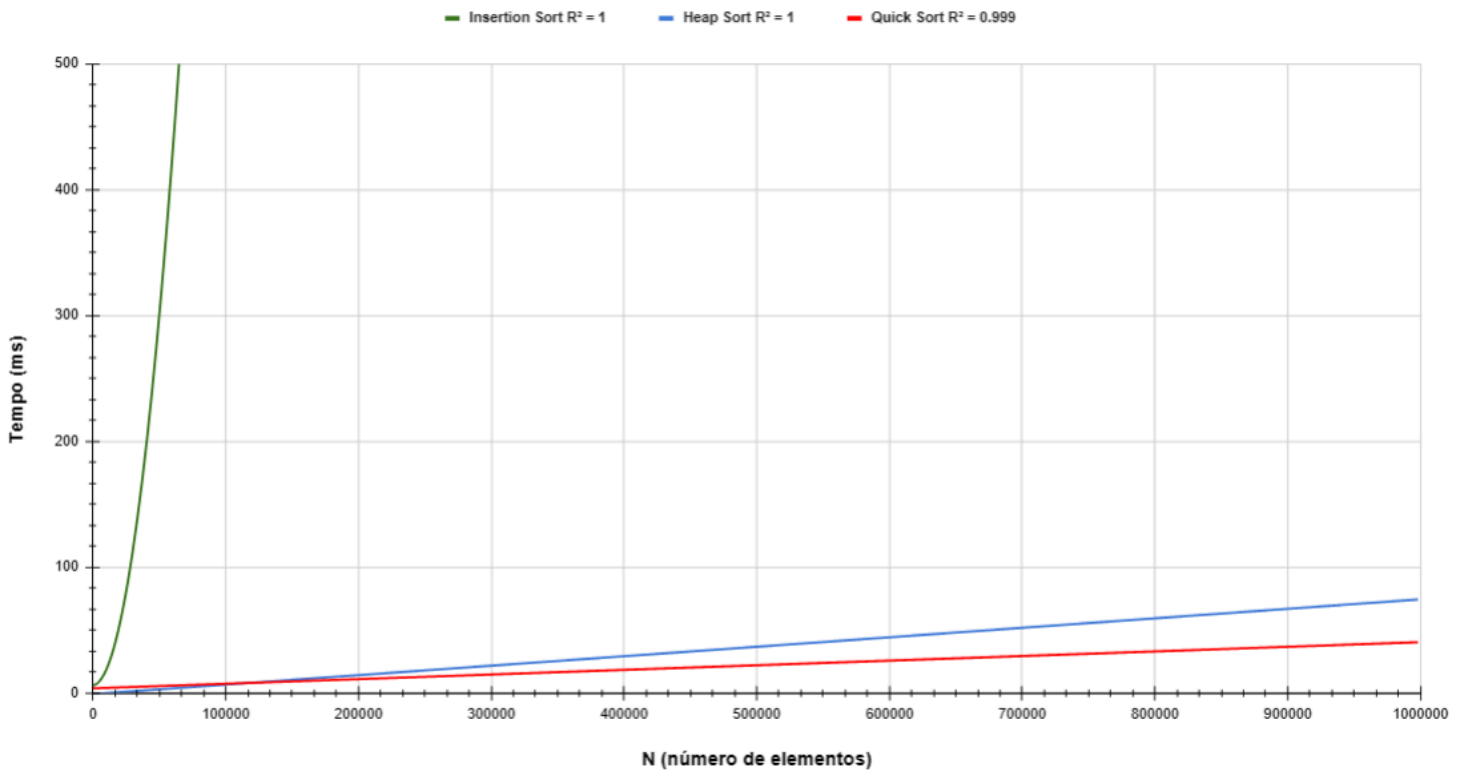
KEYS: A

Chaves por ordem crescente



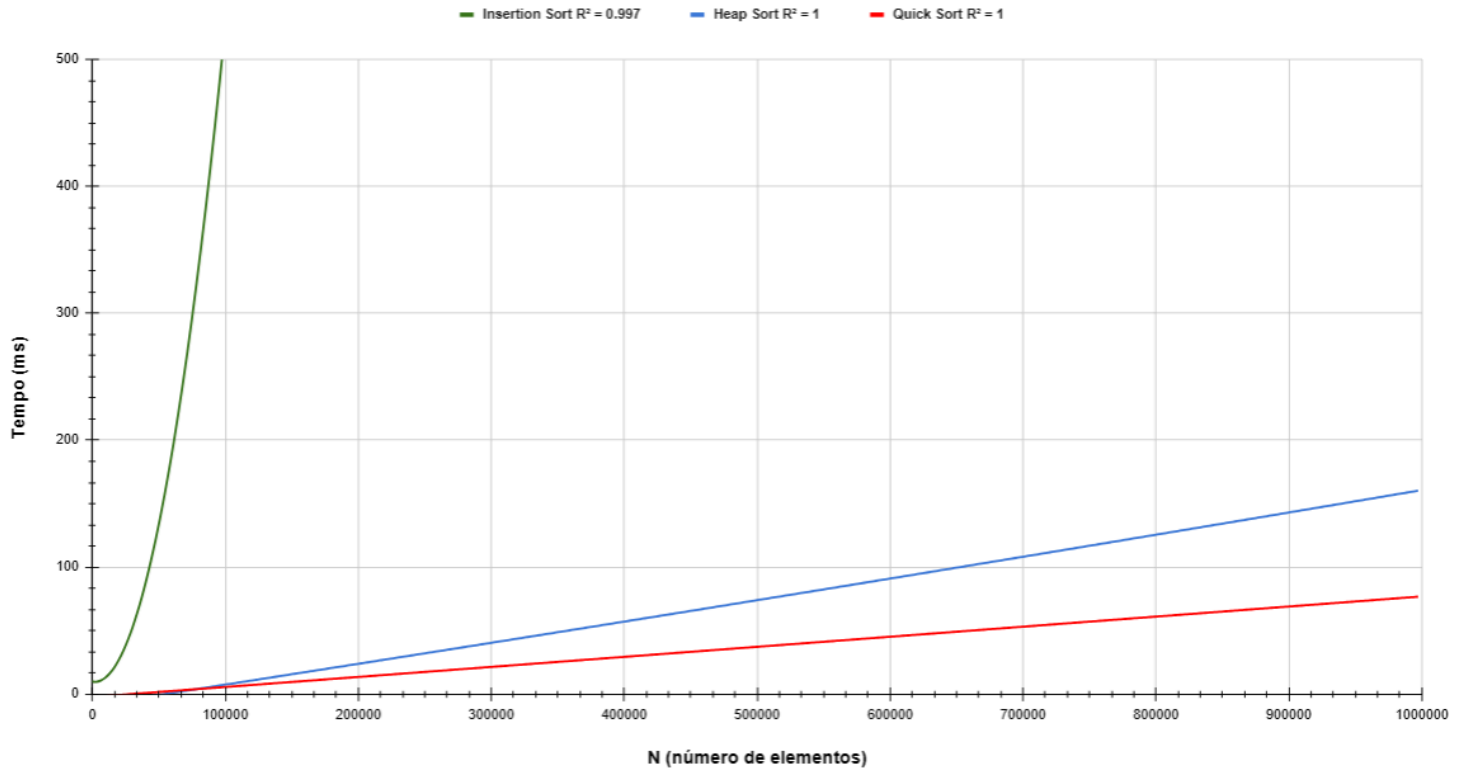
KEYS: B

Chaves por ordem decrescente



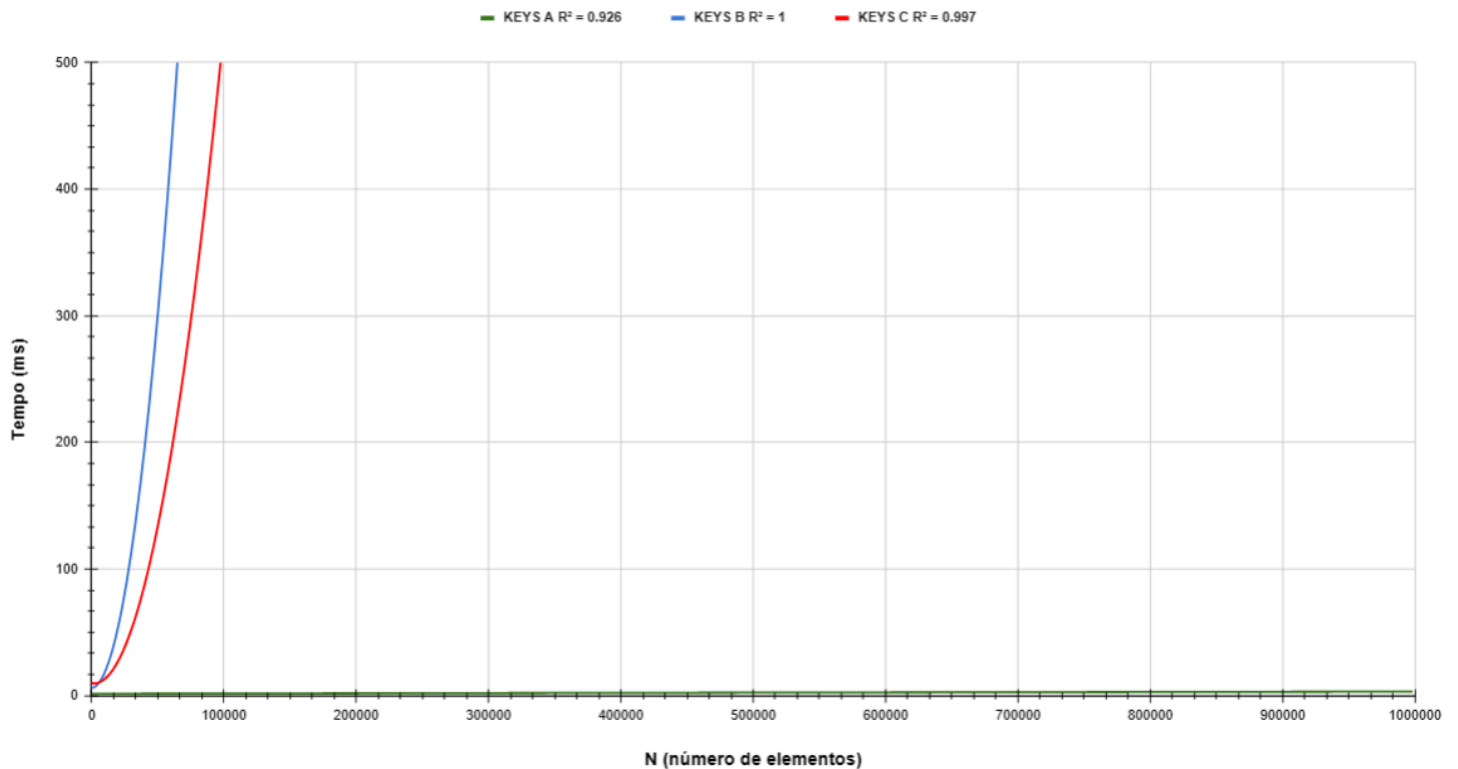
KEYS: C

Chaves por ordem aleatória



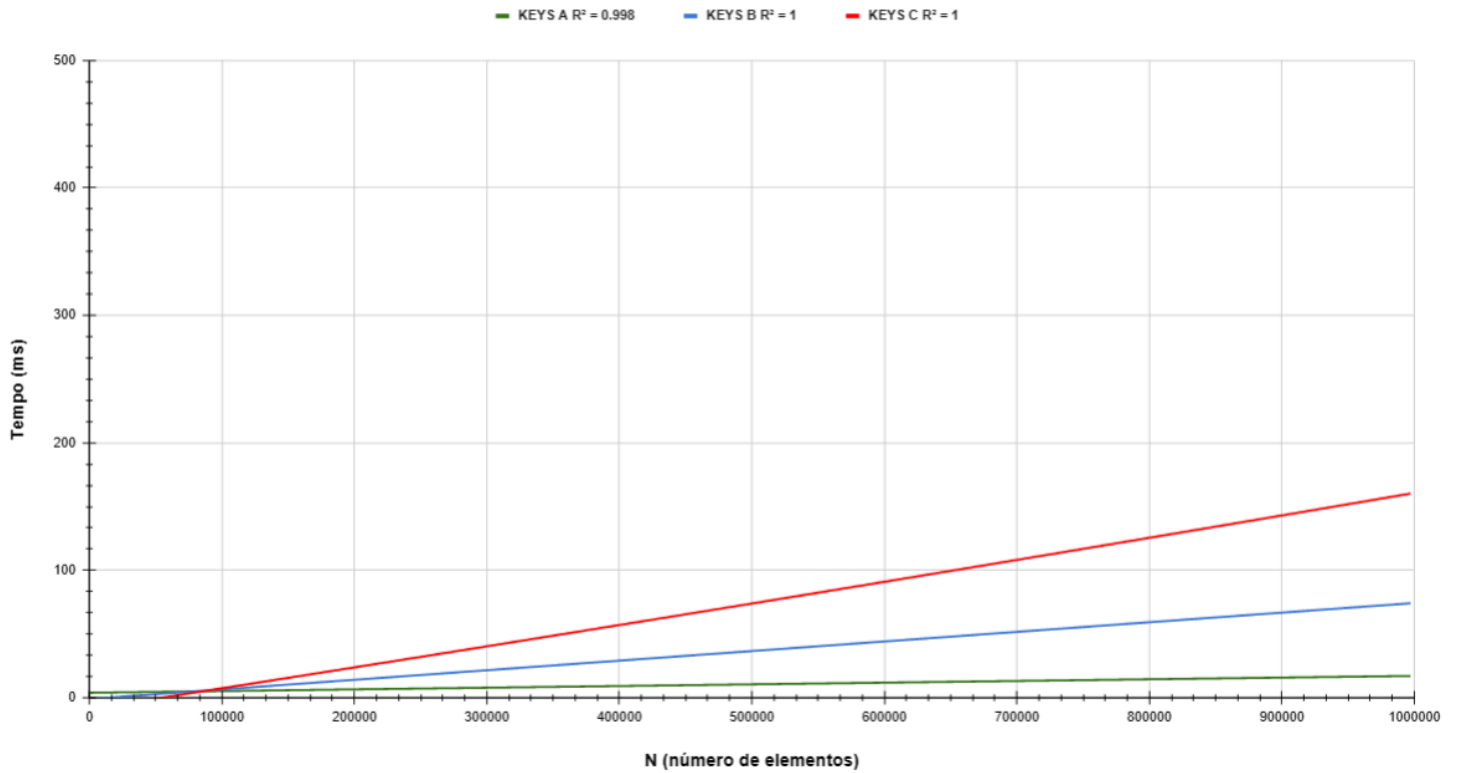
Comparação Final Keys Insertion Sort

Performance do Insertion Sort nos diversos conjuntos de chaves



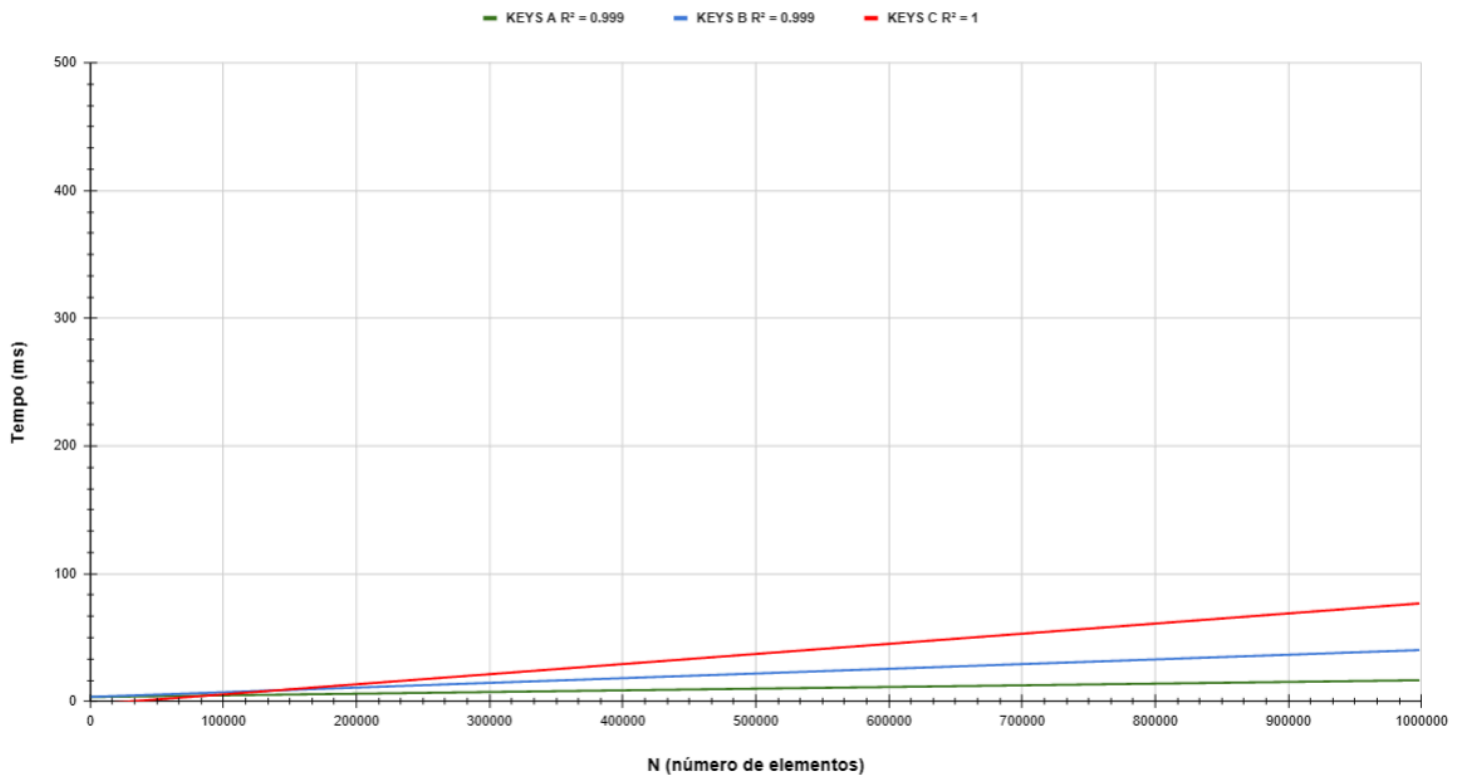
Comparação Final Keys Heap Sort

Performance do Heap Sort nos diversos conjuntos de chaves



Comparação Final Keys Quick Sort

Performance do Quick Sort nos diversos conjuntos de chaves



4. Conclusões

4.1 Tarefa 1

Nesta tarefa foi implementado o algoritmo insertion sort que consiste no percorrer do array, inserindo cada elemento no seu lugar, "empurrando" os elementos maiores para a direita deste.

Com a utilização deste algoritmo é de fácil visualização que a sua complexidade no pior caso e no caso médio, equivale a $O(N^2)$, porém como também já era previsto teoricamente no melhor caso este comporta-se como o melhor dos 3 algoritmos com complexidade $O(N)$. Este algoritmo é estável e in-place.

4.2 Tarefa 2

Nesta tarefa foi implementado o algoritmo heap sort que consiste na utilização de uma heap tree, as quais são formadas e retiradas a sua raiz como o último elemento do array (o maior) e, em seguida, são reajustadas de forma a estarem novamente em formato de max heap tree.

Com a utilização deste algoritmo é de fácil visualização que a sua complexidade em todos os cenários, ainda que com diferenças, equivale a $O(N \log(N))$ assim como previsto teoricamente. Este algoritmo não é estável mas é in-place.

4.3 Tarefa 3

Nesta tarefa foi implementado o quick sort que consiste na partição recursiva do array em 2, onde a partição da esquerda contém os elementos menores e a da direita os maiores, para isto é escolhido um dos elementos como divisor (pivot), neste caso é usada uma mediana de 3, e quando os arrays são muito pequenos é usado o insertion sort. Uma utilização deste algoritmo é de fácil visualização que a sua complexidade em todos os cenários, mesmo que com diferenças, equivale a $O(N \log(N))$ assim como previsto teoricamente. Este algoritmo não é estável mas é in-place.

É de notar que existe uma chance muito próxima de 0 da complexidade do algoritmo escalar para $O(N^2)$ no caso da escolha do pivot ser, por azar, a pior possível. Na prática, esta chance geralmente não é levada em conta devido à uma quase nula probabilidade.

Anexo A - Delimitação de Código de Autor

Nota: Todo o código para o funcionamento dos testes e implementação das tarefas foi por mim desenvolvido, desta forma, apenas serão aqui apresentadas as linhas de maior importância para que não fique demasiado extenso.

Implementação Insertion Sort

```
239  /**
240   * Implements the insertion sort algorithm.
241   * @param keys Array to sort.
242   */
243  @private static void InsertionSort(int[] keys) { 1 usage
244      /* Going through the second to last element in array between low and high indexes. */
245      for(int i = 1; i < keys.length; i++) {
246          /* Saving the current element */
247          int element = keys[i];
248          int j = i - 1;
249
250          /* Pushing the left elements greater than the current element to the right */
251          while(j >= 0 && keys[j] > element) {
252              keys[j + 1] = keys[j];
253              j--;
254          }
255          /* Replacing the current element in the correct position */
256          keys[j + 1] = element;
257      }
258  }
```


Implementação Heap Sort

```
260  /**
261   * Implements the heap sort algorithm.
262   */
263  private static class HeapSort { 2 usages
264      /**
265       * Sorts an array using the heap sort algorithm.
266       * @param keys array to sort.
267       */
268  @ public void sort(int[] keys) { 1 usage
269      /* Building max heap tree */
270      /* Note: keys.length/2 - 1 is the last node in the tree that have children */
271      for(int i = ((keys.length/2)-1); i >= 0; i--) heapify(keys, keys.length, i);
272
273      /* Sorting the array */
274      for(int i = keys.length - 1; i > 0; i--) {
275          /* Swaps the first element to the end of the unsorted part of the array */
276          swap(keys, index1: 0, i);
277
278          /* Reorganize the max heap tree */
279          heapify(keys, i, i);
280      }
281  }
282
283  /**
284   * Heapifies a heap tree recursively.
285   * @param keys Heap tree array.
286   * @param length Length of the heap tree array.
287   * @param i The index of the node to verify the heap tree properties and change if needed.
288   */
289  private void heapify(int[] keys, int length, int i) { 3 usages
290      /* Initializations */
291      int biggest = i;
292      int left = (2*i) + 1; // left child of node i
293      int right = (2*i) + 2; // right child of node i
294
295      /* Verification of the biggest node in the trio, root and children */
296      if(left < length && keys[left] > keys[biggest]) biggest = left;
297      if(right < length && keys[right] > keys[biggest]) biggest = right;
298
299      /* If the biggest isn't the root of the subtree */
300      if(biggest != i) {
301          /* Swaps the root and the biggest to let it right */
302          swap(keys, i, biggest);
303
304          /* Recursively heapifies the affected subtree */
305      @ heapify(keys, length, biggest);
306      }
307  }
308 }
```

Implementação Quick Sort

```
310      /**
311       * Implements the quick sort algorithm.
312       */
313      private static class QuickSort { 3 usages
314          /**
315           * Minimum number of elements to perform insertion sort.
316           */
317          private int MIN_ELEMENTS = 30; no usages
318
319
320          /**
321           * Default Quicksort algorithm constructor (MIN_ELEMENTS = 30).
322           */
323          public QuickSort() {} 1 usage
324
325          /**
326           * Quicksort algorithm constructor.
327           * @param MIN_ELEMENTS Minimum number of elements to perform insertion sort.
328           */
329          public QuickSort(int MIN_ELEMENTS) { this.MIN_ELEMENTS = MIN_ELEMENTS; } 1 usage
330
331
332          /**
333           * Sorts an array with quick sort algorithm.
334           * @param keys Array to sort.
335           */
336          public void sort(int[] keys) { sortRec(keys, low: 0, high: keys.length - 1); } 1 usage
337
338      /**
339       * Sorts recursively a part of an array with quick sort algorithm.
340       * @param keys Array to sort.
341       * @param low Lower index of the array to consider.
342       * @param high Higher index of the array to consider.
343       */
344      private void sortRec(int[] keys, int low, int high) { 3 usages
345          if (low < high) {
346              /* If the partition array has few elements (minus then MIN_ELEMENTS) applies insertion sort */
347              if (high - low + 1 <= MIN_ELEMENTS) insertSort(keys, low, high);
348              /* Otherwise, continue quick sorting partition method */
349              else {
350                  /* Choosing pivot index */
351                  int pivotIndex = getPivotIndex(keys, low, high);
352
353                  /* Puts the pivot at the end of the array partition */
354                  swap(keys, pivotIndex, high);
355
356                  /* Sorting array according to pivot and actualization of pivot index */
357                  pivotIndex = partition(keys, low, high);
358
359                  /* Recursive sort of left and right array */
360                  sortRec(keys, low, high: pivotIndex - 1);
361                  sortRec(keys, low: pivotIndex + 1, high);
362              }
363          }
364      }
```

```

366  /**
367   * Sorts an array putting all elements lower than pivot to the left and higher to its right
368   * and returns the new pivot index.
369   * @param keys Array to sort.
370   * @param low Lower index of the array to start considering to sort.
371   * @param high Higher index of the array to end considering to sort.
372   * @return The new pivot index.
373   */
374  @ private int partition(int[] keys, int low, int high) { 1 usage
375      int pivot = keys[high];
376
377      /* Initialization of the left index pointer */
378      int left = low - 1;
379
380      /* Sorting the array, puts all the elements lower than pivot to left and higher to right of him */
381      for(int i = low; i < high; i++) {
382          if(keys[i] < pivot) {
383              left++;
384              swap(keys, left, i);
385          }
386      }
387
388      /* Actualizing the left pointer to the first element higher than pivot */
389      left++;
390
391      /* Swap the first element higher than pivot with him */
392      swap(keys, left, high);
393
394      /* Return news pivot index */
395      return left;
396  }
397
398  /**
399   * Gets the pivot index from a median of 3 between first, middle
400   * and last element of the array, sorting them before.
401   * @param keys Array to search a pivot.
402   * @param low Lower index of the array to consider.
403   * @param high Higher index of the array to consider.
404   * @return The pivot index.
405   */
406  @ private int getPivotIndex(int[] keys, int low, int high) { 1 usage
407      /* Middle element index of the partition in the array */
408      int mid = low + (high-low)/2;
409
410      /* Sort the first, mid and last element in the partition of the array */
411      if(keys[high] < keys[mid]) swap(keys, high, mid);
412      if(keys[high] < keys[low]) swap(keys, high, low);
413      if(keys[mid] < keys[low]) swap(keys, mid, low);
414
415      /* Takes the middle element as pivot */
416      return mid;
417  }

```

```

419  /**
420   * Applies insertion sort in a part of an array.
421   * @param keys Array to sort.
422   * @param low Lower index of the array to consider.
423   * @param high Higher index of the array to consider.
424   */
425  private void insertSort(int[] keys, int low, int high) { 1 usage
426      /* Going through the second to last element in array between low and high indexes. */
427      for(int i = low + 1; i <= high; i++) {
428          /* Saving the current element */
429          int element = keys[i];
430          int j = i - 1;
431
432          /* Pushing the left elements greater than the current element to the right */
433          while(j >= low && keys[j] > element) {
434              keys[j + 1] = keys[j];
435              j--;
436          }
437          /* Replacing the current element in the correct position */
438          keys[j + 1] = element;
439      }
440  }
441  }

```

```

450  /**
451   * Swaps two elements in an array.
452   * @param keys Array of elements.
453   * @param index1 Index of one of the elements to swap with 'index2'.
454   * @param index2 Index of the other element to swap with 'index1'.
455   */
456  @ private static void swap(int[] keys, int index1, int index2) { 8 usages
457      int temp = keys[index1];
458      keys[index1] = keys[index2];
459      keys[index2] = temp;
460  }

```

Anexo B - Referências

Embora a nível de código nada tenha sido copiado ou pego como referência, todos os vídeos pelo professor recomendados para as aulas teóricas foram utilizados como referência de construção lógica dos algoritmos de forma a complementar os conhecimentos já obtidos, assim, fica a minha referência e agradecimento ao criador do canal do youtube: Michael Sambol.

URLs para o canal:

<https://www.youtube.com/@MichaelSambol>