


Algorithmics	Student information	Date	Number of session
	UO: 270534	02/24/2021	4
	Surname: Iglesias Préstamo	 Escuela de Ingeniería Informática Universidad de Oviedo	
	Name: Ángel		



Activity 1. [Time measurements for sorting algorithms]

1. Task 1 - Insertion.

• Results

N	$sorted(10^{-7} s)$	$inverse(10^{-3} s)$	$random(10^{-3} s)$
10,000	58	60	46
20,000	64	217	136
40,000	126	295	150
80,000	210	1,215	572
160,000	435	4,876	2,349
320,000	848	19,041	9,130
640,000	1,669	76,347	36,442
1,280,000	3,854	309,526	158,185
2,560,000	9,808	1,243,104	620,628
5,120,000	19,403	4,946,423	2,479,540
10,240,000	40,705	19,802,890	9,900,099
Complexity	$O(n)$	$O(n^2)$	$O(n^2)$

• Conclusion

$sorted(t)$	$inverse(t)$	$random(t)$
$t_1 = 210, n_1 = 80,000$ and $n_2 = 320,000$ $t_3 = 126, n_3 = 40,000$ and $n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \frac{320,000}{80,000} \cdot 210 = 840$ $t_4 = \frac{640,000}{40,000} \cdot 126 = 2,016$ 	$t_1 = 76,347, n_1 = 160,000$ and $n_2 = 640,000$ $t_3 = 19,041, n_3 = 320,000$ and $n_4 = 1,280,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{640,000}{160,000}\right)^2 \cdot 76,347 = 1,221,552$ $t_4 = \left(\frac{1,280,000}{320,000}\right)^2 \cdot 19,041 = 304,656$ 	$t_1 = 2,349, n_1 = 160,000$ and $n_2 = 1,280,000$ $t_3 = 9,130, n_3 = 320,000$ and $n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{1,280,000}{160,000}\right)^2 \cdot 2,349 = 150,336$ $t_4 = \left(\frac{640,000}{320,000}\right)^2 \cdot 9,130 = 36,520$

Values	$sorted(t)$	$inverse(t)$	$random(t)$
Actual ₁	848	1,243,104	158,185
Expected ₁	840	1,221,552	150,336
Actual ₂	1,669	309,526	36,442
Expected ₂	2,016	304,656	36,520

This is probably the most interesting algorithm to evaluate, maybe not as much as quicksort; as it performs as other algorithms do, but when the vector is sorted, this is a beast. I had to use tens of microseconds as the scale for measuring the times, because if I didn't I wouldn't get noticeable counts. The time complexity is linear, which is way better than any other algorithm, this suggests me that, this algorithm can be very practical for some situations. For example, when the array is almost sort, or, as it is easy to implement, for small projects.

2. Task 2 - Selection.

• Results

<i>N</i>	<i>sorted(10⁻³ s)</i>	<i>inverse(10⁻³ s)</i>	<i>random(10⁻³ s)</i>
10,000	191	176	205
20,000	527	484	518
40,000	1,897	2,127	2,141
80,000	7,558	8,557	8,466
160,000	29,829	34,115	33,340
320,000	119,689	138,498	135,676
640,000	473,457	553,992	542,704
1,280,000	1,870,023	2,671,256	2,190,816
2,560,000	7,702,096	8,863,872	8,673,784
5,120,000	30,634,978	35,467,670	34,729,056
10,240,000	122,561,536	141,812,972	139,000,344
<i>Complexity</i>	$O(n^2)$	$O(n^2)$	$O(n^2)$

• Conclusion

<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
$t_1 = 1,870,023, n_1 = 1,280,000 \text{ and } n_2 = 10,240,000$ $t_3 = 29,829, n_3 = 160,000 \text{ and } n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{10,240,000}{1,280,000}\right)^2 \cdot 1,870,023 = 119,681,472$ $t_4 = \left(\frac{640,000}{160,000}\right)^2 \cdot 29,829 = 477,264$ 	$t_1 = 34,115, n_1 = 160,000 \text{ and } n_2 = 640,000$ $t_3 = 138,498, n_3 = 320,000 \text{ and } n_4 = 1,280,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{640,000}{160,000}\right)^2 \cdot 34,115 = 545,840$ $t_4 = \left(\frac{1,280,000}{320,000}\right)^2 \cdot 138,498 = 2,215,968$ 	$t_1 = 8,466, n_1 = 80,000 \text{ and } n_2 = 320,000$ $t_1 = 33,340, n_1 = 160,000 \text{ and } n_2 = 1,280,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{320,000}{80,000}\right)^2 \cdot 8,466 = 135,456$ $t_4 = \left(\frac{1,280,000}{160,000}\right)^2 \cdot 33,340 = 2,133,760$

<i>Values</i>	<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
Actual ₁	122,561,536	553,992	135,676
Expected ₁	119,681,472	545,840	135,456

Values	<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
Actual ₂	273,457	2,671,256	2,190,816
Expected ₂	477,264	2,215,968	2,133,760

I would also use this algorithm for small projects; however, something I noticed implementing it is that – probably – it uses less memory than other algorithms, as you won't perform as many interchanges between positions as with others.

3. Task 3 - Bubble.

- Results

<i>N</i>	<i>sorted(10⁻³ s)</i>	<i>inverse(10⁻³ s)</i>	<i>random(10⁻³ s)</i>
10,000	123	152	186
20,000	460	530	788
40,000	2,993	1,984	4,960
80,000	12,057	7,998	19,820
160,000	48,404	32,048	79,341
320,000	194,581	128,929	317,974
640,000	778,324	515,716	1,271,896
1,280,000	3,113,296	2,062,864	5,034,567
2,560,000	12,453,184	8,251,456	20,350,336
5,120,000	49,812,736	33,005,824	81,401,344
10,240,000	199,250,944	132,123,224	325,605,376
Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$

- Conclusion

<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
$t_1 = 3,113,296, n_1 = 1,280,000$ and $n_2 = 10,240,000$ $t_3 = 48,404, n_3 = 160,000$ and $n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{10,240,000}{1,280,000}\right)^2 \cdot 3,113,296 = 199,250,944$ $t_4 = \left(\frac{640,000}{160,000}\right)^2 \cdot 48,404 = 774,464$ 	$t_1 = 2,062,864, n_1 = 1,280,000$ and $n_2 = 10,240,000$ $t_3 = 32,048, n_3 = 160,000$ and $n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{10,240,000}{1,280,000}\right)^2 \cdot 2,062,864 = 132,023,296$ $t_4 = \left(\frac{640,000}{160,000}\right)^2 \cdot 32,048 = 512,768$ 	$t_1 = 79,341, n_1 = 160,000$ and $n_2 = 640,000$ $t_3 = 317,974, n_3 = 320,000$ and $n_4 = 1,280,000$ <ul style="list-style-type: none"> $t_2 = \left(\frac{640,000}{160,000}\right)^2 \cdot 79,341 = 1,269,456$ $t_4 = \left(\frac{1,280,000}{320,000}\right)^2 \cdot 317,974 = 5,087,584$

Values	<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
Actual ₁	199,250,944	132,123,224	1,271,896
Expected ₁	199,250,944	132,023,296	1,269,456
Actual ₂	778,324	515,716	5,034,567

Values	<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
Expected ₂	774,464	512,768	5,087,584

4. Task 4 – Quicksort with the central element as the pivot.

- Results

<i>N</i>	<i>sorted(10⁻⁶s)</i>	<i>inverse(10⁻⁶s)</i>	<i>random(10⁻⁶s)</i>
10,000	96	95	588
20,000	139	157	1,251
40,000	382	340	2,605
80,000	740	712	5,349
160,000	1,567	1,476	11,148
320,000	2,788	2,867	23,272
640,000	5,694	5,908	49,703
1,280,000	11,970	12,209	99,942
2,560,000	24,872	25,353	204,067
5,120,000	54,417	58,101	469,408
10,240,000	118,601	131,890	1,068,294
Complexity	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$

- Conclusion

<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
$t_1 = 34,115, n_1 = 160,000$ and $n_2 = 640,000$ $t_3 = 138,498, n_3 = 320,000$ and $n_4 = 1,280,000$ <ul style="list-style-type: none"> $t_2 = \frac{\log k + \log n_2}{\log n_1} \cdot t_1 = 5,708$ $t_4 = \frac{\log k + \log n_2}{\log n_1} \cdot t_1 = 11,970$ 	$t_1 = 1,870,023, n_1 = 1,280,000$ and $n_2 = 10,240,000$ $t_3 = 29,829, n_3 = 160,000$ and $n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \frac{\log k + \log n_2}{\log n_1} \cdot t_1 = 115,888$ $t_4 = \frac{\log k + \log n_2}{\log n_1} \cdot t_1 = 6,086$ 	$t_1 = 1,870,023, n_1 = 1,280,000$ and $n_2 = 10,240,000$ $t_3 = 29,829, n_3 = 160,000$ and $n_4 = 640,000$ <ul style="list-style-type: none"> $t_2 = \frac{\log k + \log n_2}{\log n_1} \cdot t_1 = 1,023,061$ $t_4 = \frac{\log k + \log n_2}{\log n_1} \cdot t_1 = 51,634$

Values	<i>sorted(t)</i>	<i>inverse(t)</i>	<i>random(t)</i>
Actual ₁	5,694	131,890	1,068,294
Expected ₁	5,708	115,888	1,023,061
Actual ₂	11,970	5,908	49,703
Expected ₂	12,663	6,086	51,634

In almost every case we can see that the expected complexity matches to the actual ones. Well, they won't be exactly equals, but at least so close that we are noticing the expected evolution of the times – following the theoretical time complexity. As you can

see, I had to use bigger values for nTimes as the algorithm is so fast that if I didn't do such, I wouldn't get representative values. This means, results are in microseconds. It's a good algorithm but choosing the right pivot is key.

Activity 2. [QuicksortFateful]

In this case it has been chosen the left-most element of the vector – or a partition of it – as a pivot. This will cause the algorithm to slice the array into unbalanced partitions. Notice that this scenario is the worst-case for this kind of algorithm. Imagine we have a set of elements whose size is n , as you may now, the recursive calls of the quicksort algorithm have the following scheme:

- quickSort(left, i-1);
- quickSort(i+1, right);

In the best situation both calls would have the same size, this way, we are dividing by 2 the size with each iteration. But that's the best case, now, let's suppose the worst scenario, the one at which with every and each iteration, the pivot is the smallest element; for example, in the case of choosing as a pivot the left-most element and having the vector already sorted. This way, the size of the problem won't be chopped into 2 halves, what will happen is that one partition will contain no elements, and the other one with $n - 1$ elements. During its second iteration, it will happen the same thing as before, the next item, the "new" pivot is already the smallest element; this will cause the algorithm to split the array into two portions, one containing 0 elements, but the other $n - 1$ elements ($n - 2$ if we compare it to the starting size). This will occur with every iteration performing this process as many times as needed until the size is 2, when we will divide the array into two pieces one with no elements and the last with 1. This leads us to the worst possible complexity this algorithm can have: n^2 . You can calculate this by understanding that in theory the program is going to be executed: $n + (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n+1)}{2}$ times; as Gauss solved some centuries ago. Probably it won't happen exactly this when using as a pivot the left-most element – this would be the worse of the worse – but that's the key thing when talking about time complexities: the worst case. However, if you create even partitions, in order to do so, we need to get as a pivot, a value on the middle: you are constantly getting $\frac{(n-1)}{2}$ elements, which leads us to a time complexity close to: $O(n \log n)$.