Brief description of the sorting algorithms and the comparison of their performance

In this article we will discuss sorting algorithms. First off, the algorithms selected for this testing are presented with a brief description of how they work, after which testing is carried out. I decided to do 3 tests for each sorting algorithm, for an array with elements in random order, partially sorted array (the array is sorted for N/2 elements, where N is the length of the array), array with already sorted elements and for an array with elements in reversed order. The results of the tests are entered in a chart and final conclusions are drawn.

Sorting algorithms are very widely used in programming, but sometimes programmers don't even think about which algorithm works best (by the "best" I mean a combination of speed and complexity of both writing and executing the task).

To ensure the best results, all the presented algorithms will sort integer arrays of 1000, 10^6 and 10^8. The computer on which these tests will be carried out has the following specifications: Intel Core i7-7700HQ CPU @ 2.80GHz, 16GB RAM, Windows 11 Pro x64 build 19042, IDE - CLion 2021.3.3.

The following sorting algorithms were chosen for the tests:

Radix sort – This algorithm sorts the list of numbers digit-by digit from the least significant digit (LSD) to most significant digit (MSD), this determines the order of the elements in the array. Radix sort uses **counting sort** (can also use insertion sort, bubble sort or bucket sort) as a subroutine to sort individual digits. As a result, this requires additional memory space to sort digits.

Merge sort – A recursive algorithm based on Divide-and-Conquer. It divides the unsorted list in two approximately equal separate sublists. The algorithm compares and repeatedly merge these sublists to produce sorted sublists by calling the merge function.

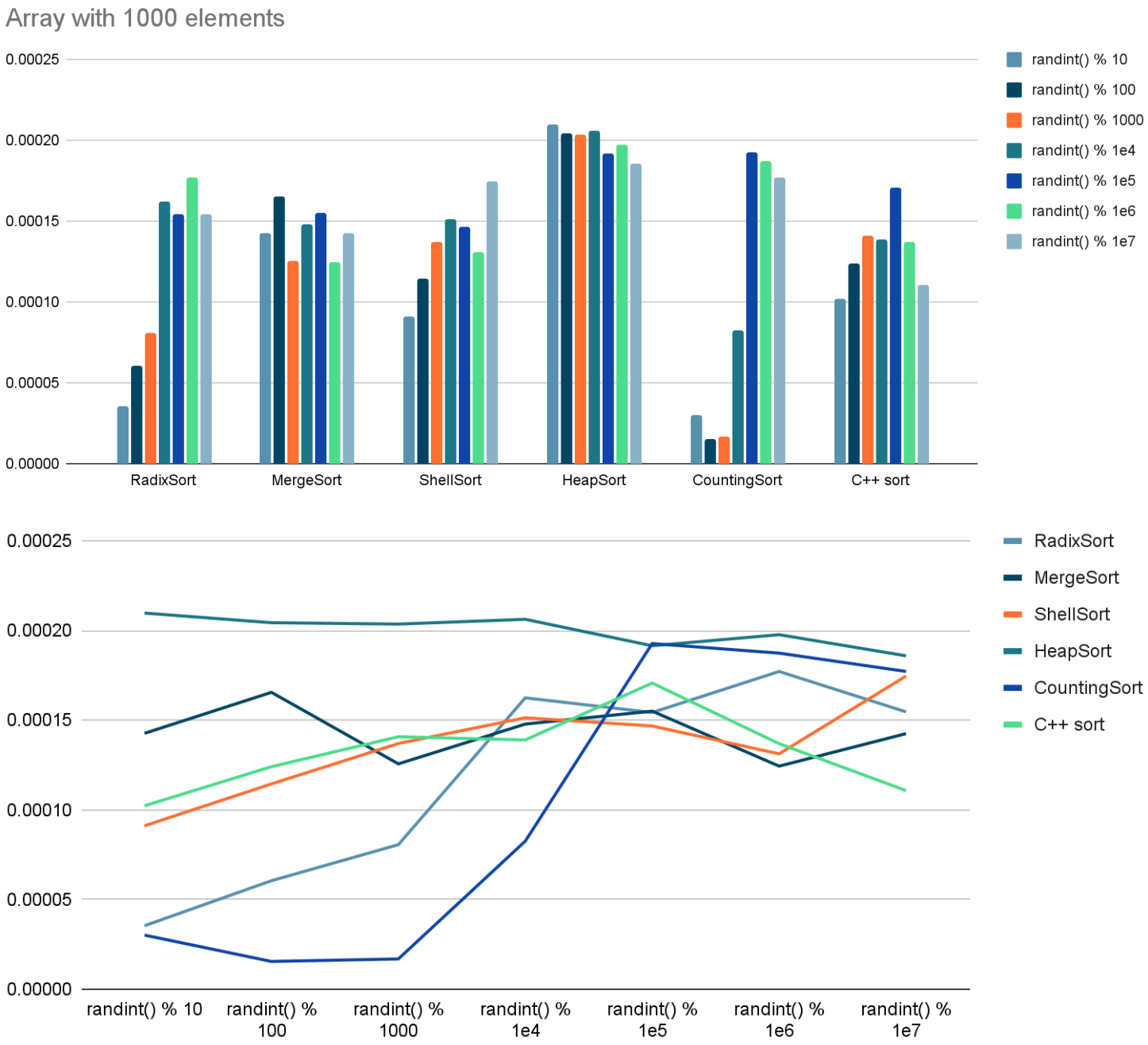
Counting sort – We create an auxiliary array of size r - l, where l – is the minimum and r – is the maximum element of the array. Then we go through the array and count the occurrences of each element. Now we can iterate through the array of values and write out each number as many times as needed.

Shell sort – It's an "add-on" to insertion sort, but instead of one pass of the array, we make several, and on the *i* pass we sort subarrays of elements that are at a interval of d_i from each other. On the last pass, d_i must be equal to 1 (that means that the last step is actually an insertion sort). Some of the optimal d_i sequences that can be used in the shell sort algorithm are: Sedgewick's increments (1, 8, 23, 77, 281, 1073, $4j+1+ 3\cdot 2j+ 1$), Hibbard's increments (1, 3, 7, 15, 31, 63, 127, 255, 511…), Knuth's increments (1, 4, 13, .., $(3^k-1)/2$), but we will use Shell's original sequence (N/2, N/4, …, 1, where N is the length of the array).

Heap sort – It is a comparison-based algorithm based on the binary heap data structure. Initially, build a max heap of elements in array. The root element of the array will be the maximum, so we swap this element with the last element of the array and then heapify the max heap excluding the last element (because it is already in the correct position), decreasing the length of heap one by one. We repeat this step until all the elements are in their correct position.

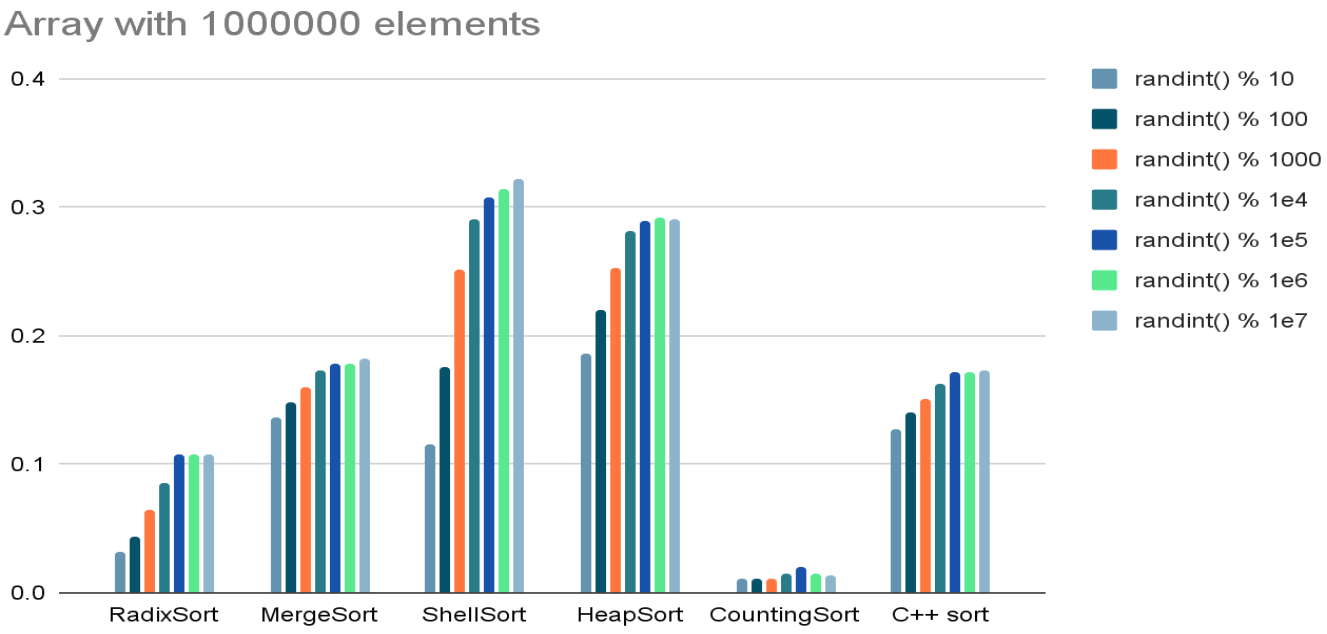
C++ sort – is a function in the C++ STL, which uses introsort, a hybrid algorithm. Standard C++ library uses a 3-part sorting algorithm: introsort is performed first (introsort itself being a hybrid of quicksort and heap sort) followed by insertion sort.

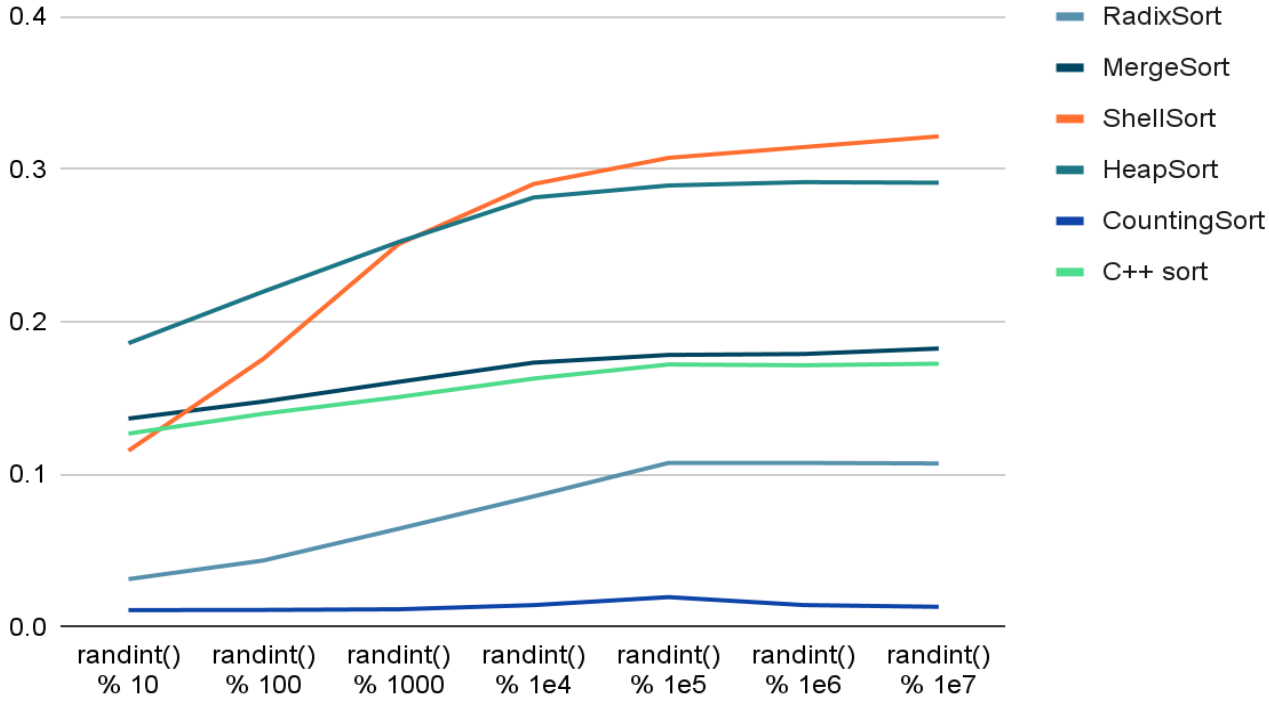
Array with randomized elements



	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.0000354	0.0000605	0.0000806	0.0001625	0.00015433	0.0001772	0.00015467	0.00011789
MergeSort	0.0001427	0.00016557	0.00012567	0.00014787	0.00015507	0.0001244	0.00014253	0.0001434
ShellSort	0.00009113	0.0001145	0.00013703	0.0001514	0.0001468	0.00013127	0.00017463	0.00013525
HeapSort	0.00020977	0.0002044	0.00020363	0.0002063	0.00019153	0.00019773	0.00018593	0.0001999
CountingSort	0.00003017	0.00001547	0.00001687	0.00008257	0.00019277	0.00018743	0.00017723	0.00010036
C++ sort	0.0001023	0.0001241	0.0001408	0.00013903	0.00017073	0.00013693	0.00011077	0.00013209

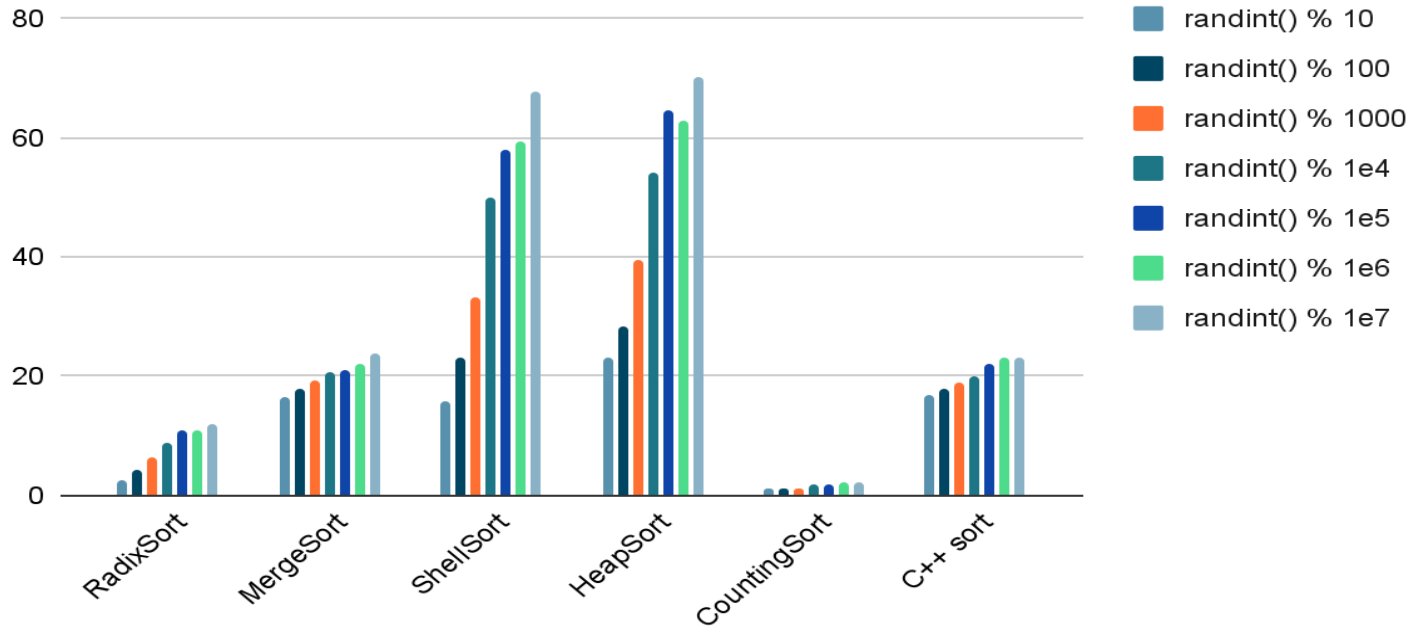
Here we can see that HeapSort has the worst time overall. Radix, Merge and C++ sort seem to have a pattern in the first half of elements, increasing the execution time with maximum elements, but then this “pattern” breaks. The graph also suggests that Counting Sort is very efficient if array contains small elements, while Merge and C++ sort have pretty constant running times.

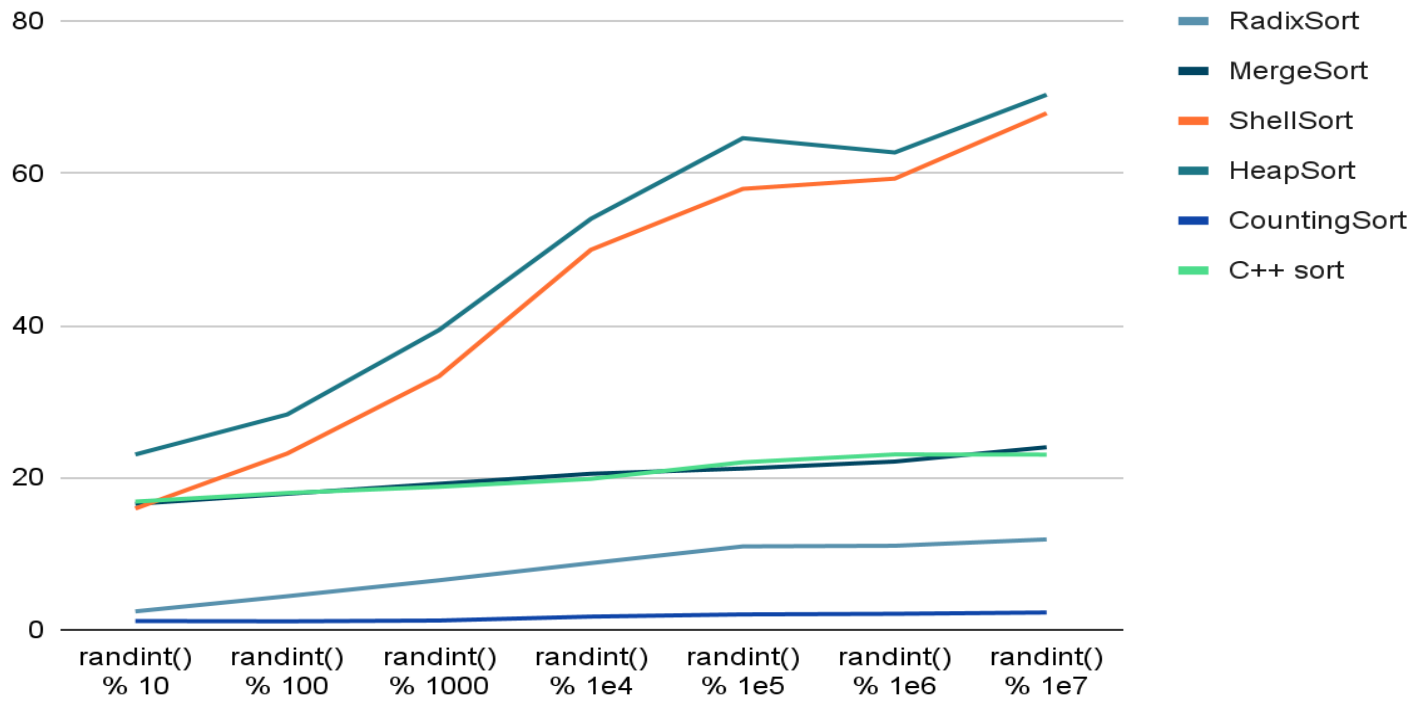




	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.031294	0.04352213	0.06431527	0.0855272	0.1074669	0.1074541	0.1071085	0.0780983
MergeSort	0.13654177	0.14770103	0.1605913	0.17321507	0.1782448	0.17886547	0.1824448	0.16537203
ShellSort	0.11546437	0.17576837	0.25087767	0.2902855	0.30744603	0.31448857	0.3215185	0.25369272
HeapSort	0.1859317	0.21980323	0.252276	0.2814726	0.28932547	0.2915045	0.29112813	0.25877738
CountingSort	0.0109659	0.0110984	0.01151817	0.0142514	0.01948387	0.01428677	0.0131422	0.01353524
C++ sort	0.12668353	0.13970847	0.15063083	0.1627488	0.17199277	0.17147533	0.17247967	0.15653134

Array with 100000000 elements



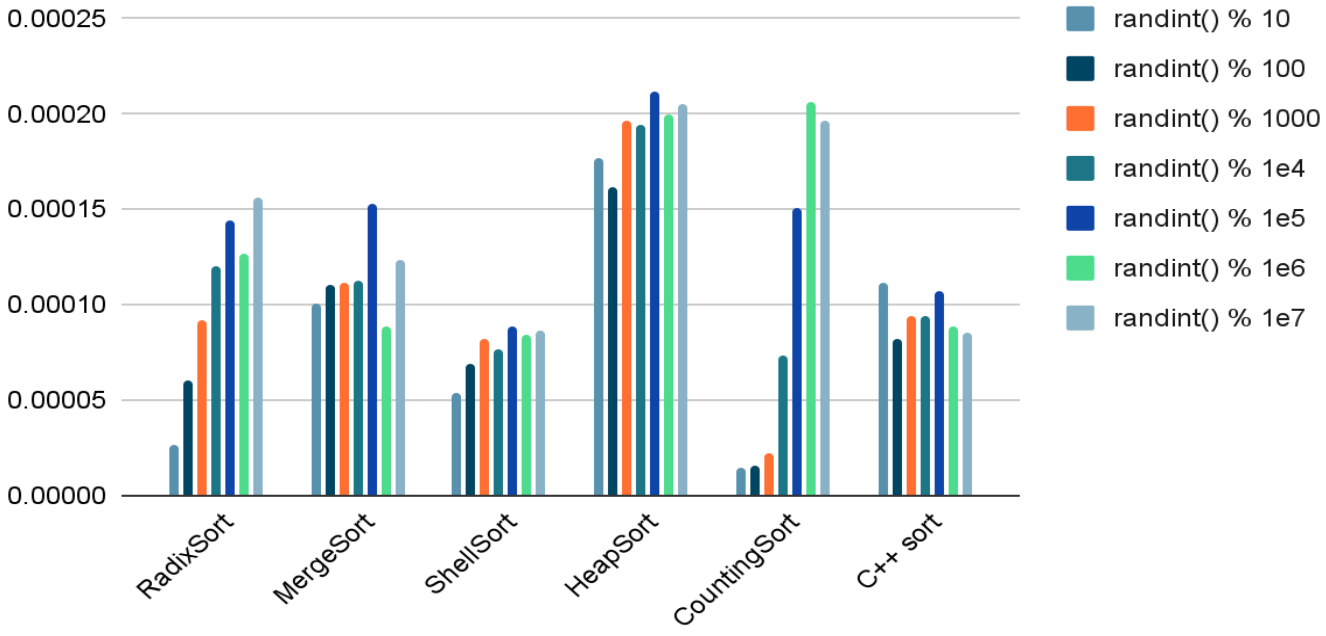


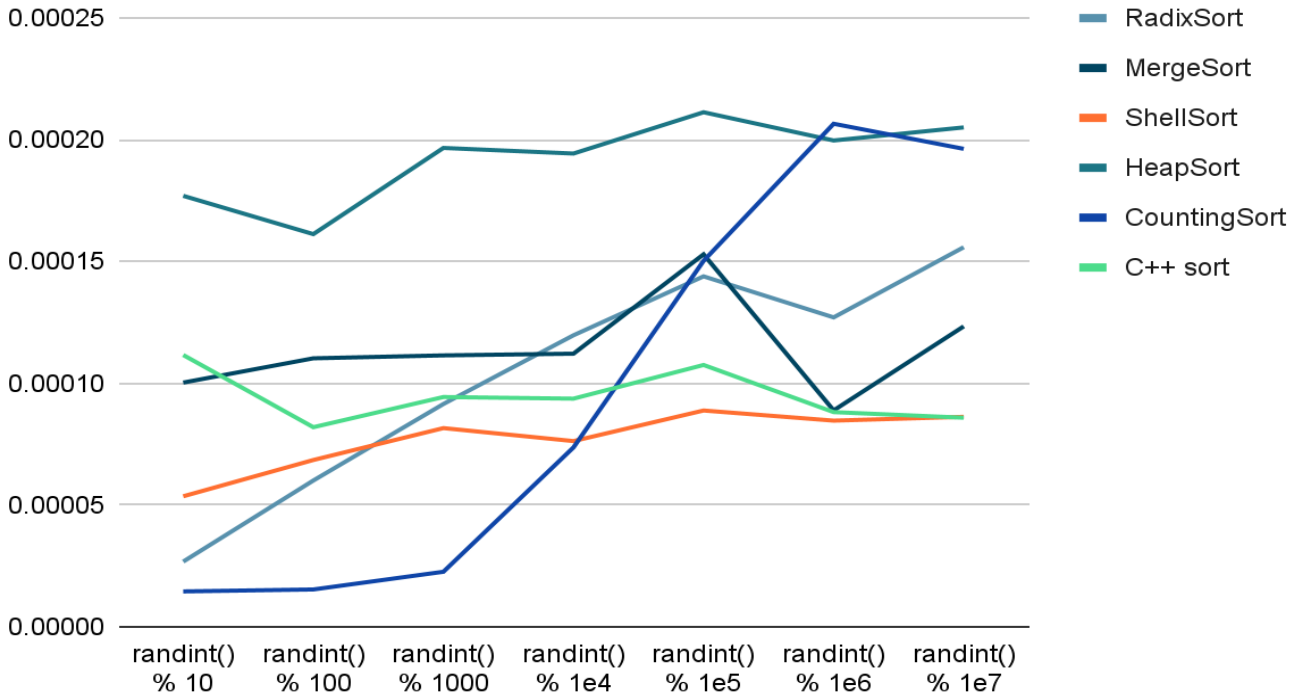
	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	2.41675537	4.4125779	6.50614827	8.76397547	10.96096247	11.04307623	11.87835363	7.99740705
MergeSort	16.55016943	17.87379103	19.18545403	20.50655383	21.19275217	22.1090369	23.99276883	20.20150375
ShellSort	15.93358813	23.164293	33.32687167	49.95040533	57.9502708	59.2956658	67.889378	43.93006753
HeapSort	23.03761733	28.30103273	39.40468833	54.01715917	64.61363083	62.73206303	70.32512437	48.9187594
CountingSort	1.15127133	1.11401567	1.2154122	1.7337168	2.02125673	2.10066367	2.27242373	1.6583943
C++ sort	16.8487944	17.98411397	18.79197847	19.8322925	22.01302663	23.04466303	23.02209847	20.21956678

My conclusions for arrays with 10^6 and 10^8 elements are exactly the same. Definitely the counting sort finishes faster than other algorithms. While radix, merge and c++ sort times are increasing consistently, Shell and Heap sort times are spiking up.

Partially sorted arrays

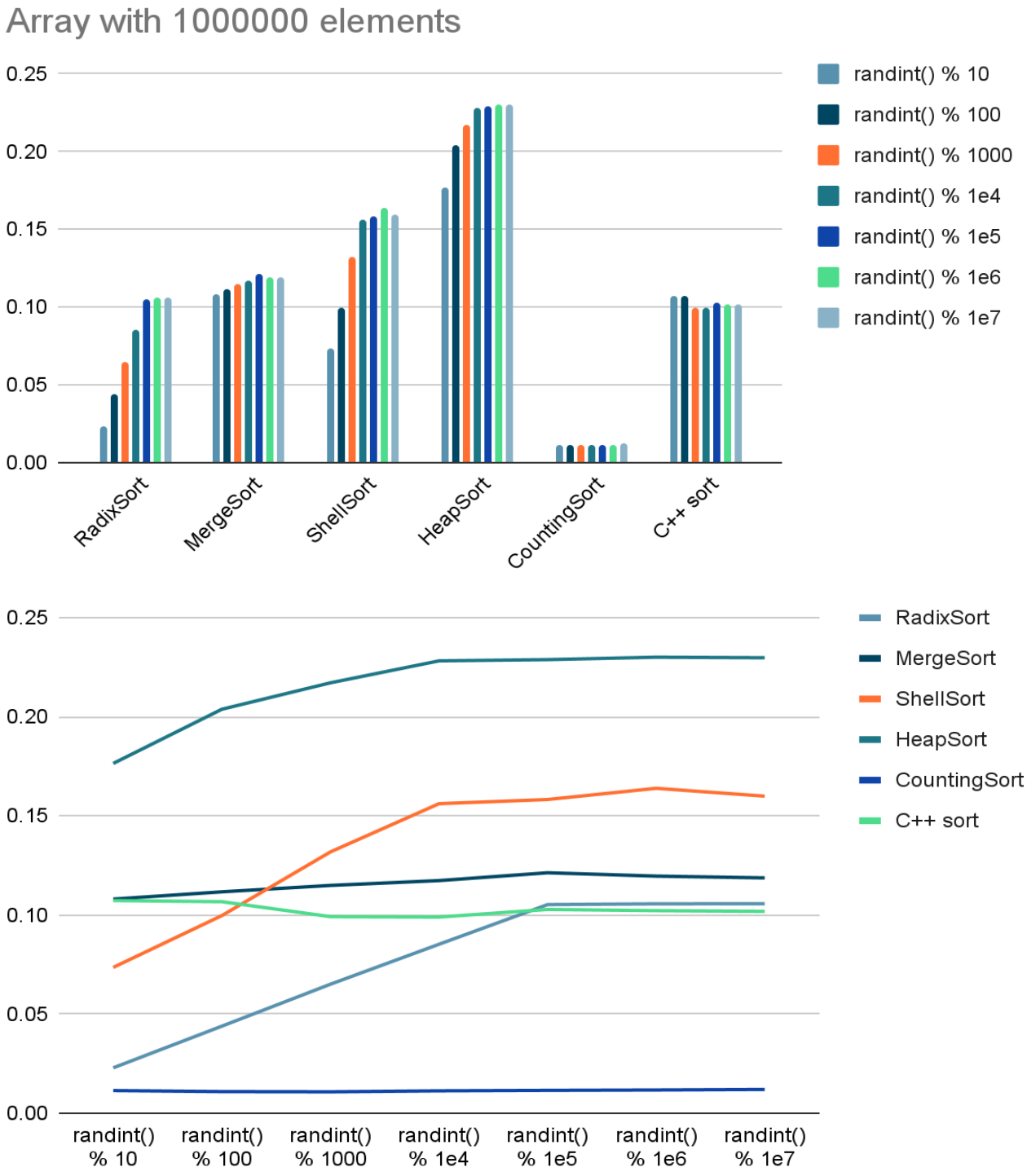
Array with 1000 elements



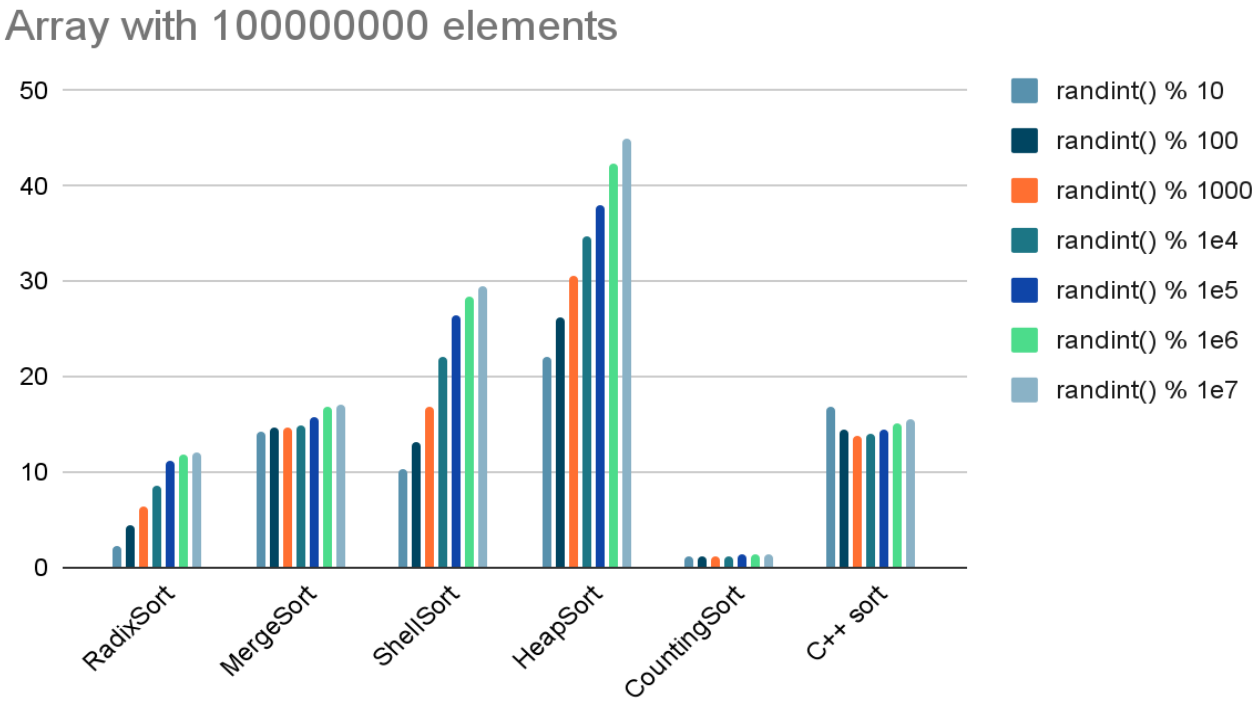


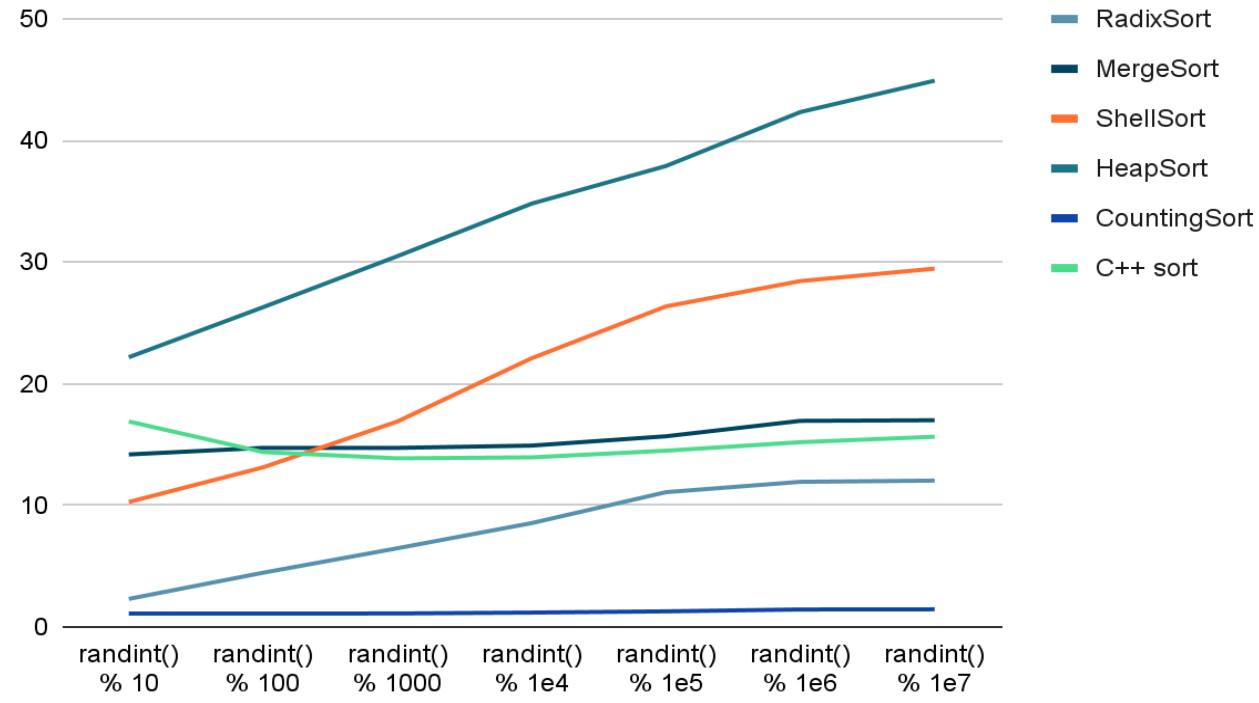
	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.00002677	0.00006007	0.0000915	0.00011973	0.00014393	0.00012707	0.0001559	0.00010357
MergeSort	0.0001003	0.00011027	0.00011147	0.00011217	0.00015297	0.00008887	0.0001234	0.00011421
ShellSort	0.0000536	0.0000685	0.0000816	0.00007627	0.00008883	0.00008467	0.00008623	0.0000771
HeapSort	0.000177	0.00016127	0.00019667	0.00019437	0.00021133	0.00019973	0.0002051	0.00019221
CountingSort	0.00001453	0.00001533	0.0000226	0.00007367	0.0001503	0.0002066	0.0001963	0.00009705
C++ sort	0.00011163	0.00008197	0.00009437	0.0000937	0.00010753	0.00008817	0.00008587	0.00009475

Same as with random elements, counting sort is efficient for numbers from 0 to 1000, but spiking up rapidly after. Shell and C++ STL sort are pretty stable. Heap sort was very weak here with very high running times.



	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.022898	0.04391233	0.06508243	0.0852449	0.10526367	0.10564137	0.10570073	0.07624906
MergeSort	0.10803347	0.11169963	0.1149378	0.1173736	0.12131613	0.1196356	0.1187274	0.11596052
ShellSort	0.0735779	0.09971347	0.13184633	0.1561798	0.15828903	0.16394147	0.15998653	0.13479065
HeapSort	0.17650393	0.20383673	0.21717597	0.22825327	0.2288816	0.23009593	0.22981283	0.21636575
CountingSort	0.0114164	0.0109007	0.0107895	0.01127803	0.0115234	0.01170727	0.011978	0.01137047
C++ sort	0.10730123	0.1067305	0.0991781	0.0989606	0.10281603	0.10220547	0.10183807	0.10271857

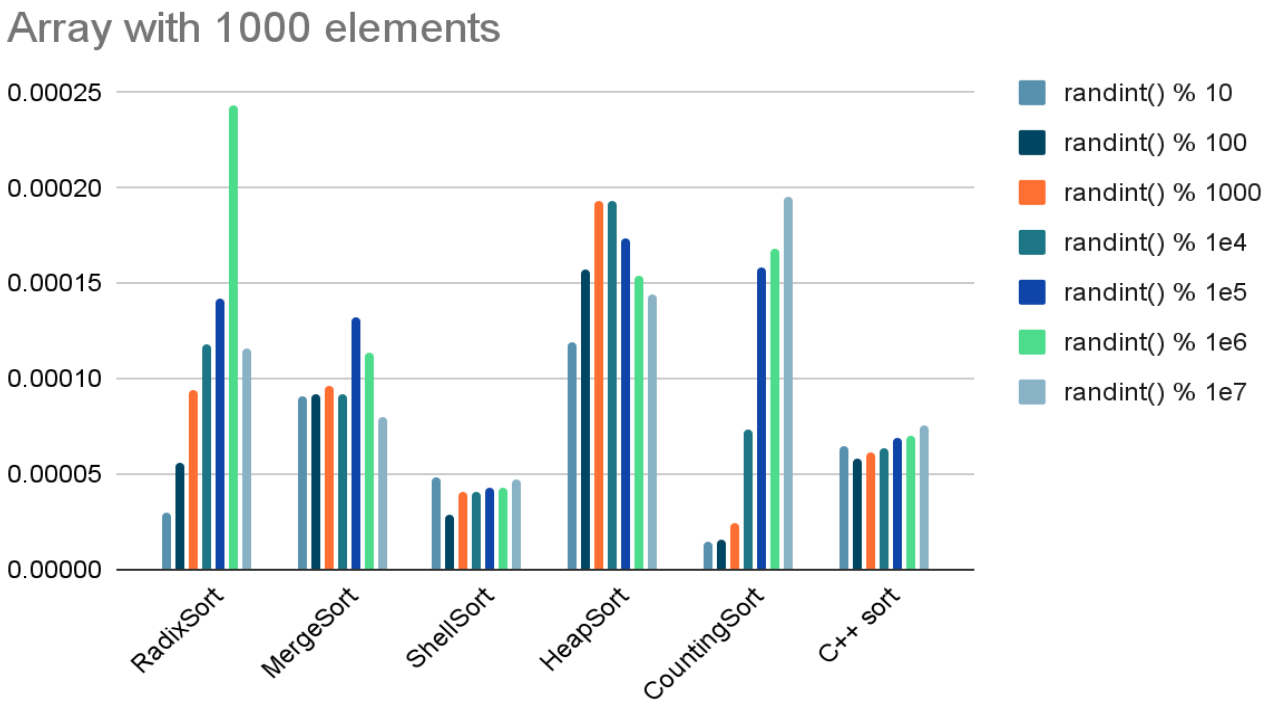


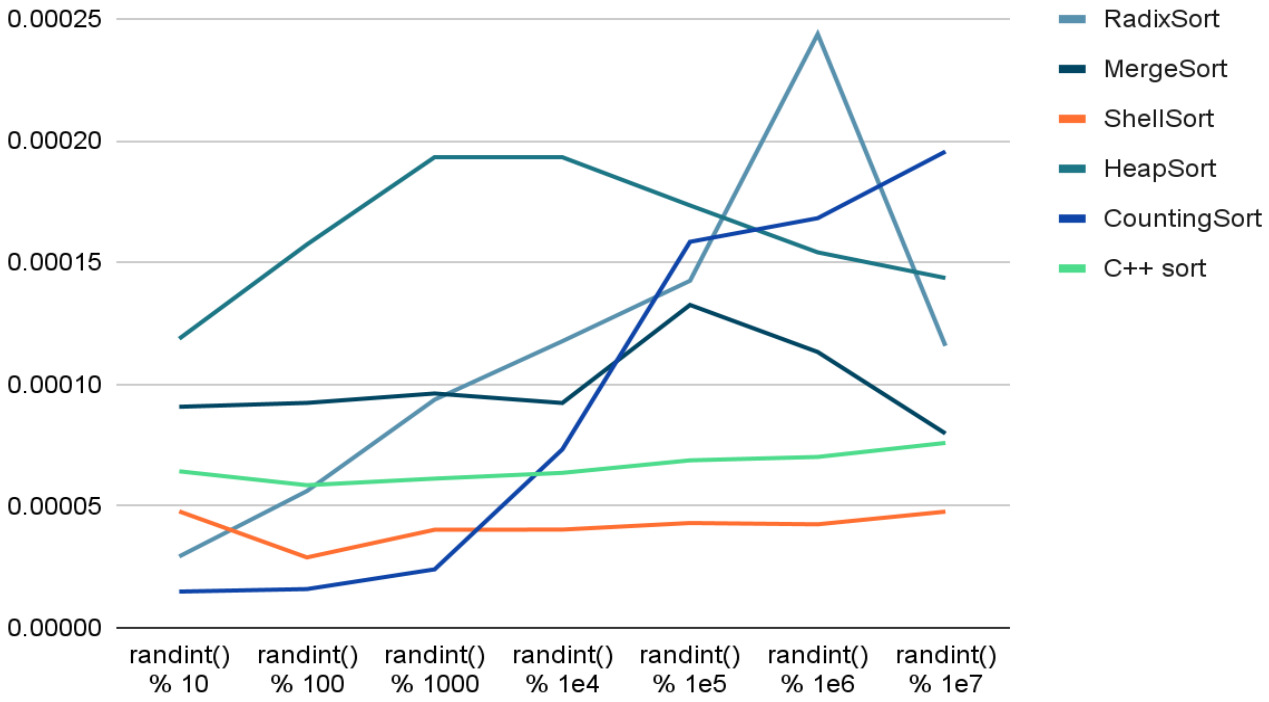


	randint() % 10	randint() % 100	randint() % 100	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	2.30507767	4.4589993	6.47073227	8.54070393	11.0841164	11.9348438	12.0353868	8.11855145
MergeSort	14.18653787	14.7295795	14.72607957	14.91431913	15.6805918	16.94886933	16.99880963	15.45496955
ShellSort	10.27509913	13.12397453	16.89666207	22.09963427	26.36322607	28.44627597	29.47109177	20.95370912
HeapSort	22.18109403	26.29663817	30.49524757	34.81799857	37.912684	42.3462925	44.93106213	34.14014528
CountingSort	1.0969527	1.1009393	1.10793453	1.18082203	1.28142933	1.4353452	1.44825573	1.23595412
C++ sort	16.89751927	14.38457127	13.8670102	13.94817317	14.49517577	15.19381433	15.64352073	14.91854068

Counting sort is still in the first place. C++ and merge sorts are stable, having pretty much equal running times. Here, heap sort is definitely in the last place, being the most inefficient one. Radix sort performs pretty good, with relative low running times, that are increasing slowly.

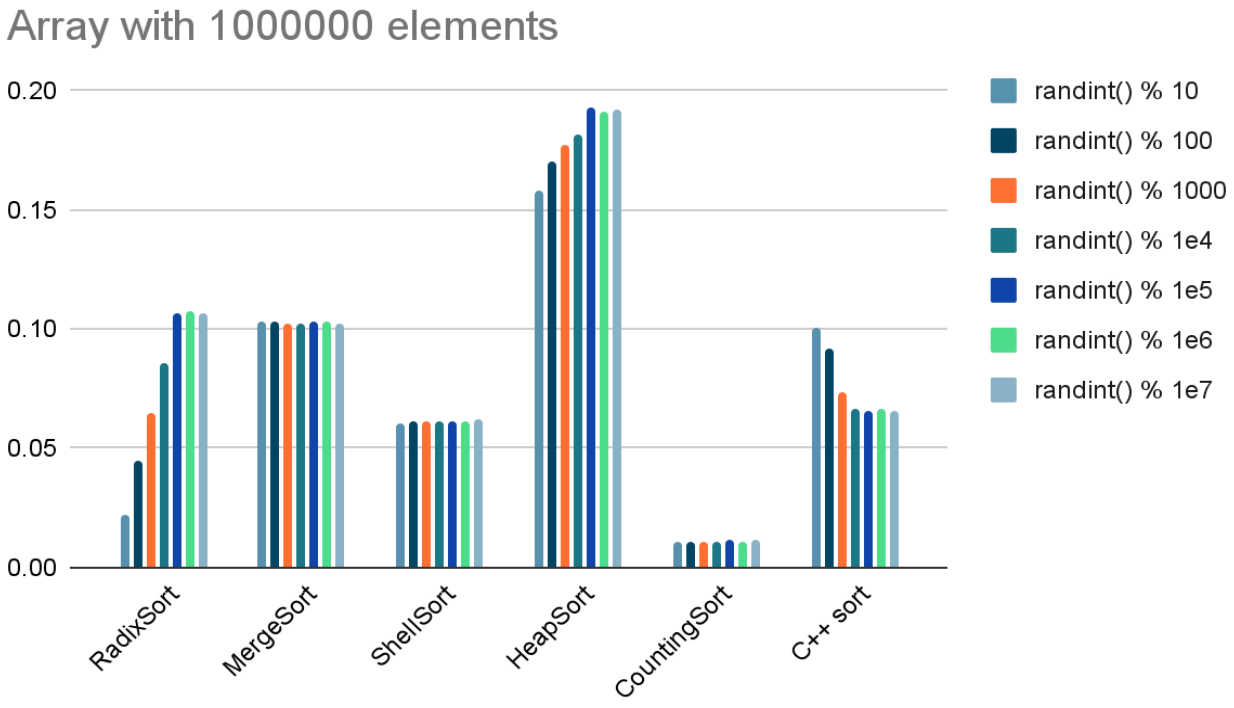
Array with already sorted elements

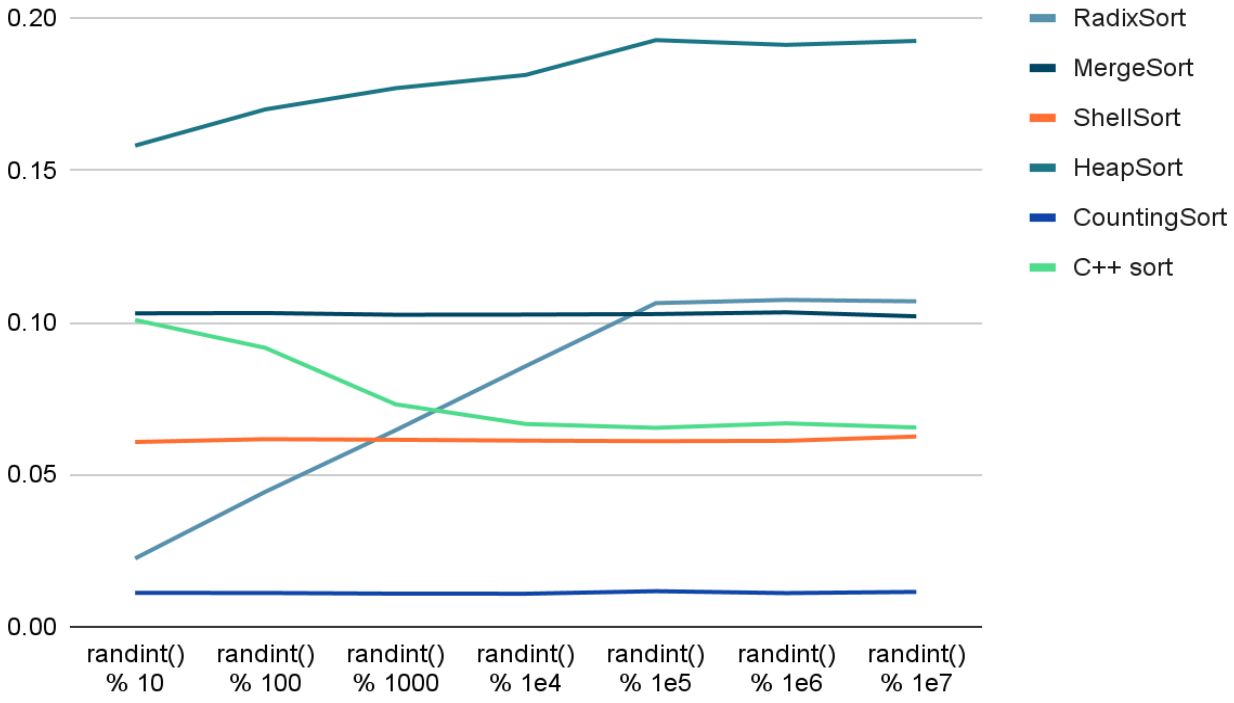




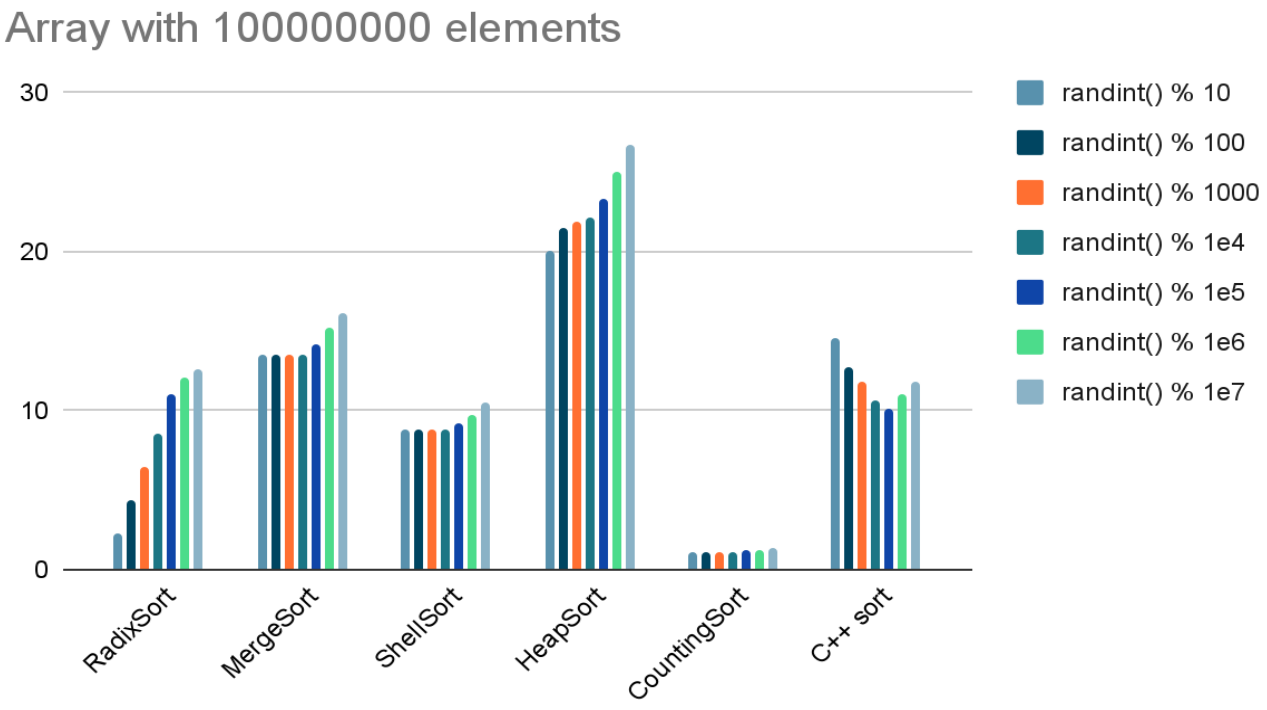
	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.0000293	0.00005617	0.0000937	0.0001177	0.00014253	0.00024383	0.0001158	0.00011415
MergeSort	0.00009077	0.00009237	0.00009623	0.00009233	0.0001326	0.00011327	0.0000798	0.00009962
ShellSort	0.0000478	0.0000289	0.00004027	0.00004033	0.00004303	0.0000425	0.0000477	0.0000415
HeapSort	0.00011873	0.00015743	0.0001933	0.00019327	0.00017347	0.0001542	0.00014367	0.00016201
CountingSort	0.00001487	0.0000159	0.000024	0.00007327	0.00015853	0.00016823	0.0001956	0.00009291
C++ sort	0.00006427	0.00005857	0.00006127	0.00006363	0.00006877	0.00007017	0.00007593	0.00006609

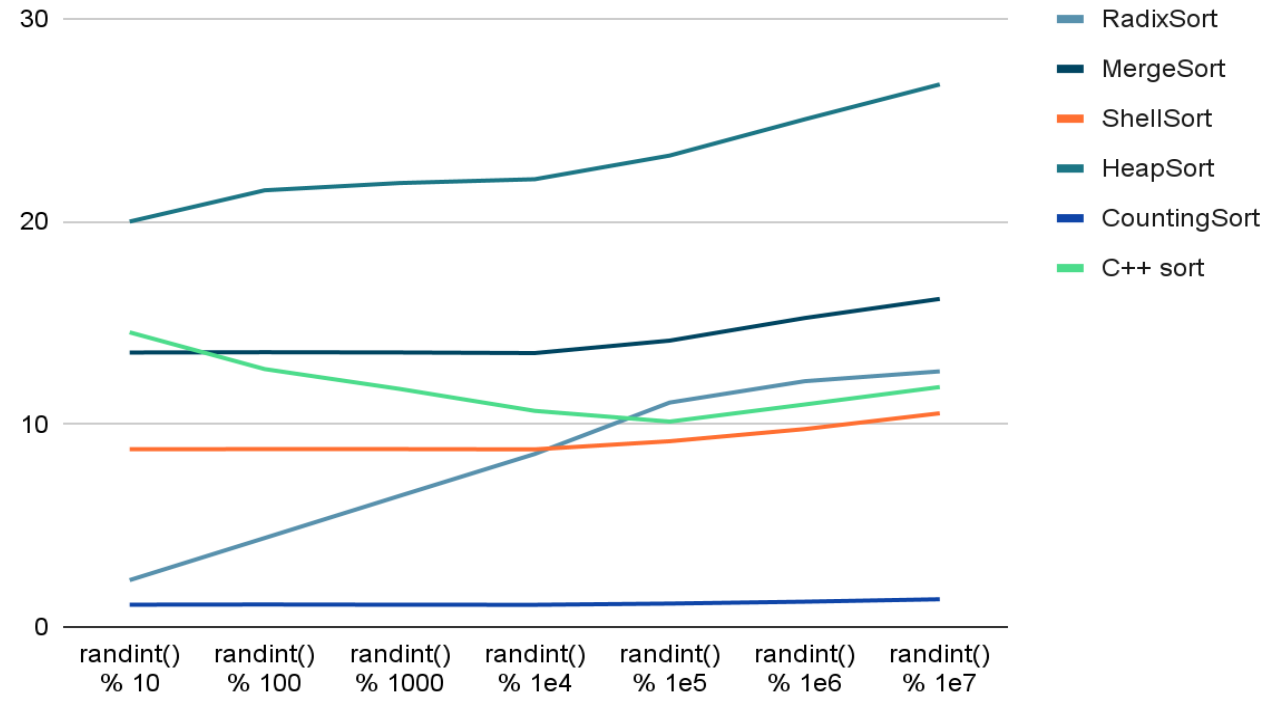
Here shell sort seems to be the efficient one if we don't know what elements we have in the array. Counting sort remains to be the winner if we know our array has small numbers (between 0 and 1000). Radix sort spikes up at the array of 10^6 elements, don’t really know what caused this.





	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.02245793	0.04435573	0.06457133	0.08565157	0.10635103	0.10740667	0.10693703	0.07681876
MergeSort	0.10299253	0.10309083	0.10252633	0.10258397	0.102755	0.1033242	0.10201173	0.10275494
ShellSort	0.0607222	0.06164727	0.06144967	0.0611639	0.0609936	0.0611091	0.06252103	0.0613724
HeapSort	0.15810227	0.17000053	0.1769634	0.181317	0.19276403	0.19119967	0.19250117	0.18040687
CountingSort	0.0111434	0.0111132	0.01088973	0.01086443	0.0116968	0.01107027	0.0114621	0.01117713
C++ sort	0.10080317	0.0916464	0.07310233	0.06662053	0.06537727	0.06685037	0.06547993	0.07569714



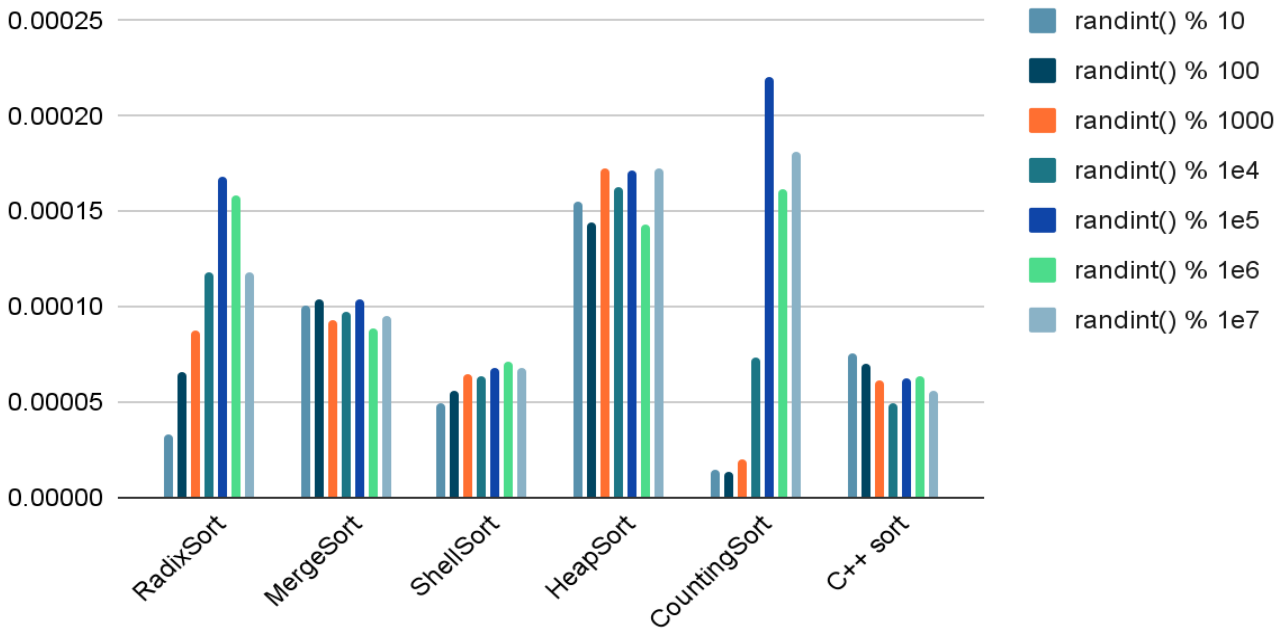


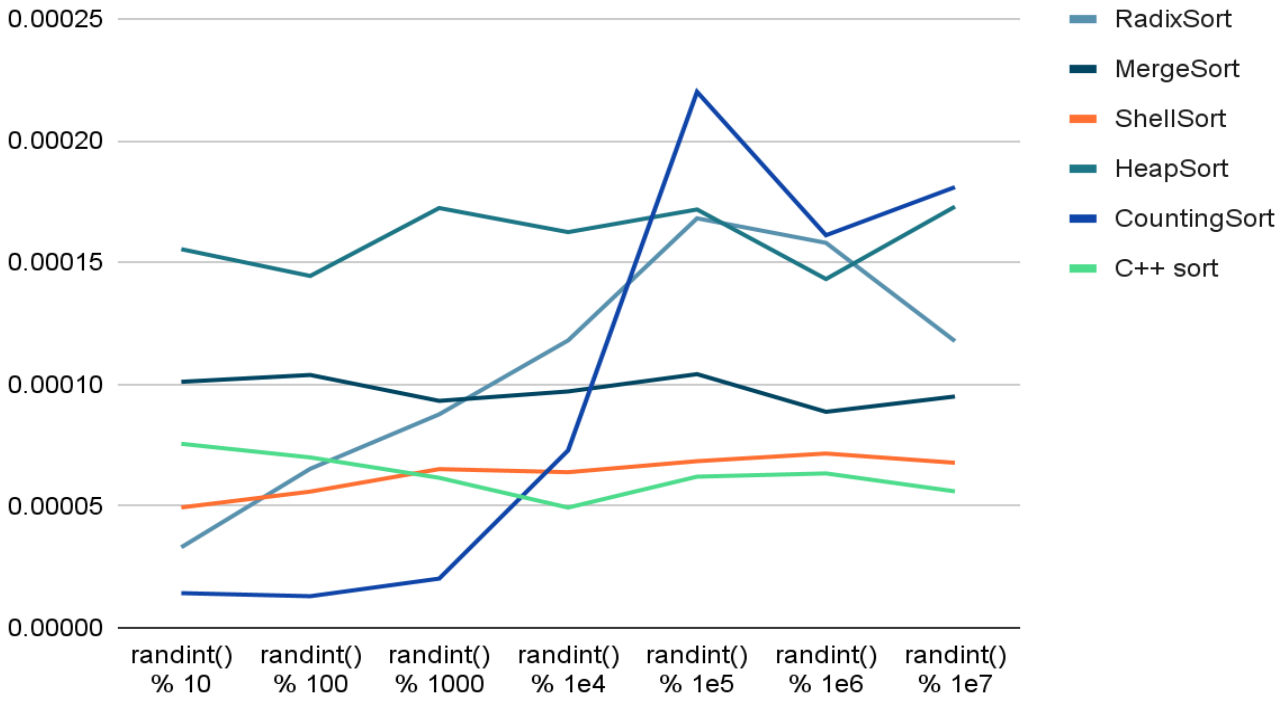
	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	2.30964977	4.38819377	6.46949373	8.53500983	11.0740355	12.12639957	12.61234773	8.21644713
MergeSort	13.54208253	13.55706813	13.5444502	13.51449933	14.1301447	15.2450333	16.18778777	14.24586657
ShellSort	8.76925927	8.7766905	8.77536703	8.7626282	9.16388803	9.7643679	10.54281927	9.22214574
HeapSort	20.0043172	21.5470551	21.9062616	22.09402923	23.26258003	25.0511366	26.77273743	22.94830246
CountingSort	1.09710487	1.10720927	1.09515613	1.09140247	1.15286633	1.24625453	1.36540233	1.16505656
C++ sort	14.53833047	12.7246877	11.74481387	10.66245997	10.1332393	10.97854983	11.83825707	11.80290546

Here we see the same results, with counting sort still in the first place. Merge and shell sorts have pretty constant running times. C++ sort time decreases as the maximum element increases.

Array with elements in reverse order

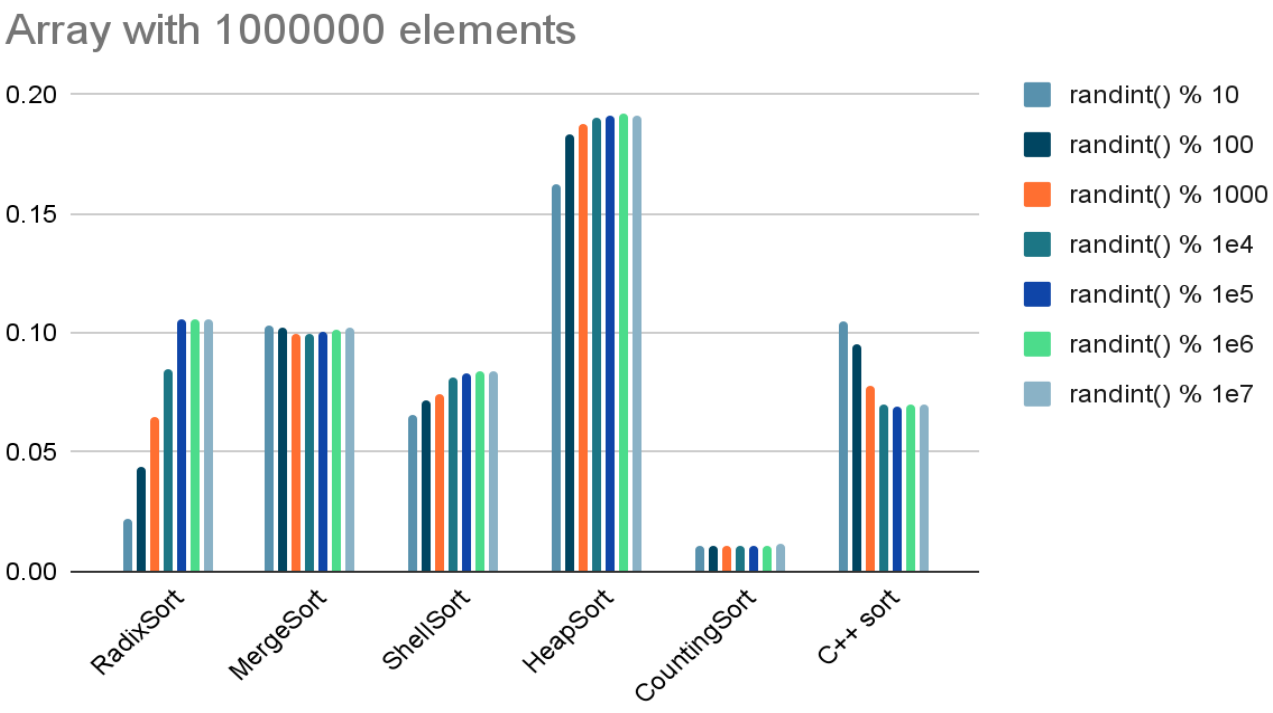
Array with 1000 elements

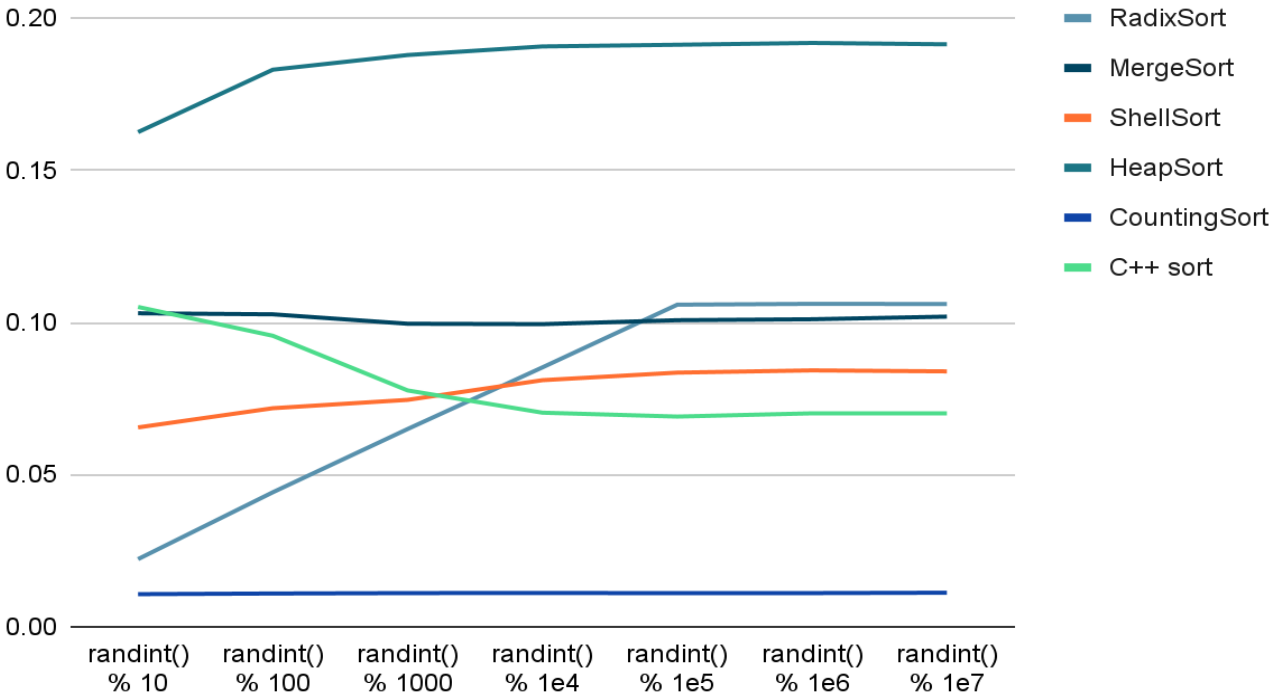




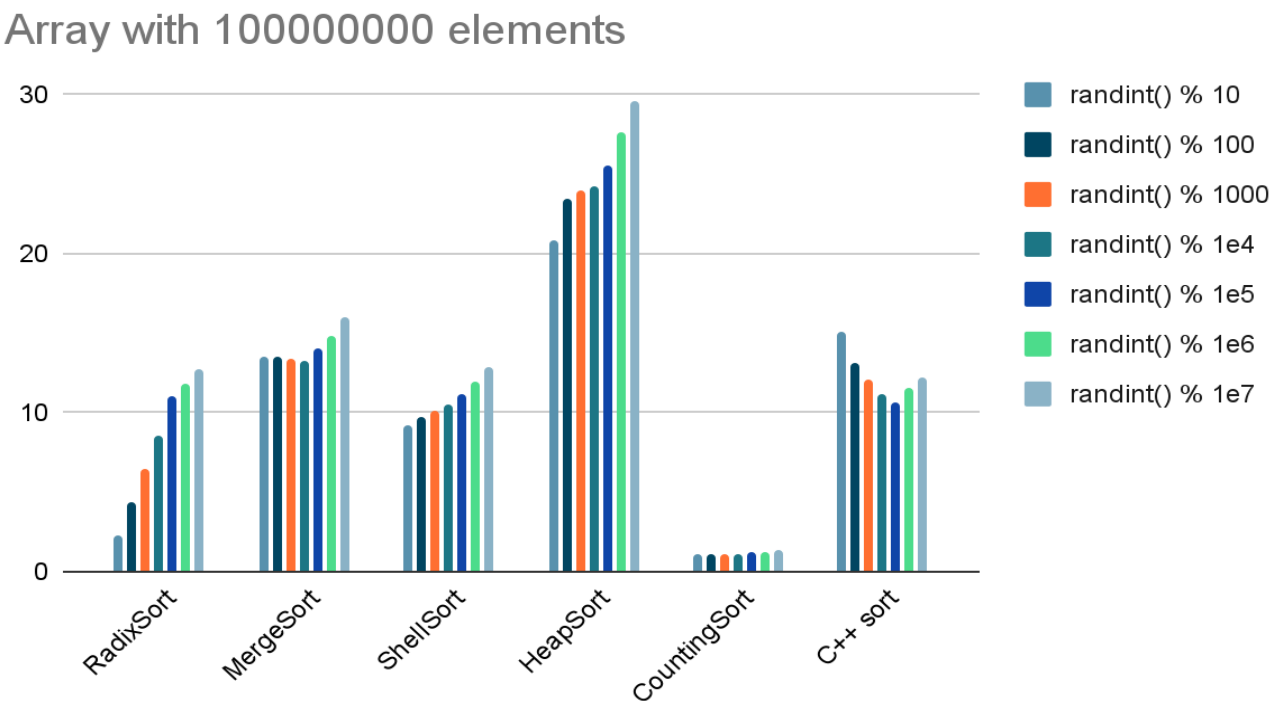
	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.00003307	0.0000653	0.00008767	0.00011803	0.00016817	0.0001581	0.00011773	0.00010687
MergeSort	0.00010103	0.00010383	0.0000932	0.00009707	0.00010417	0.0000887	0.000095	0.00009757
ShellSort	0.00004943	0.00005593	0.00006517	0.0000639	0.00006843	0.00007157	0.00006777	0.00006317
HeapSort	0.0001555	0.0001445	0.0001724	0.0001625	0.0001718	0.00014323	0.00017297	0.00016041
CountingSort	0.00001423	0.00001297	0.0000202	0.00007287	0.00022013	0.00016123	0.00018097	0.00009751
C++ sort	0.00007557	0.00006997	0.0000616	0.00004937	0.00006207	0.0000634	0.00005603	0.00006257

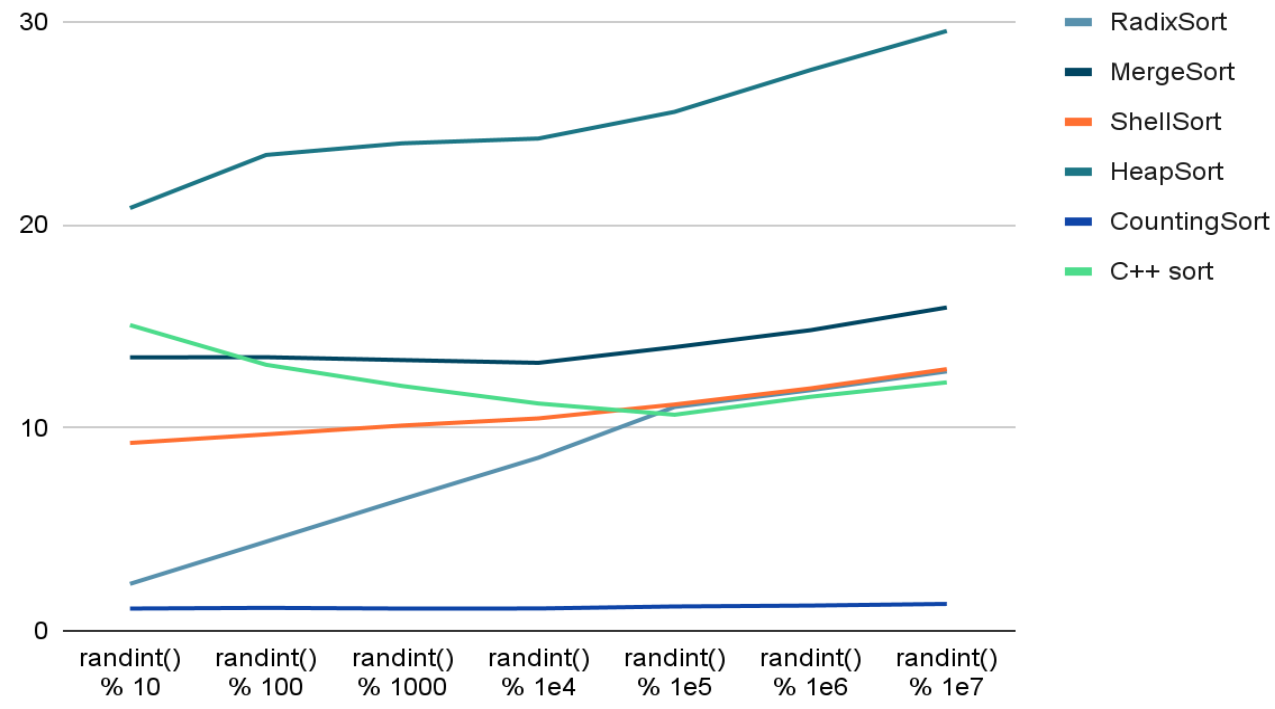
Counting sort remains to be the most efficient one for elements between 0 and 1000, but then loses to Shell and C++ sort, which once again have pretty much stable running times.





	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	0.0222307	0.04420663	0.06498747	0.08523747	0.10583623	0.10611127	0.1060558	0.0763808
MergeSort	0.10301697	0.1026702	0.09957493	0.0994391	0.10073903	0.10104867	0.1019103	0.10119989
ShellSort	0.06550847	0.071818	0.07456037	0.0810281	0.08351133	0.0842478	0.08392503	0.07779987
HeapSort	0.1625396	0.1830482	0.18786577	0.19068883	0.19124103	0.19183583	0.19139097	0.18551575
CountingSort	0.0107226	0.010934	0.01107097	0.01112267	0.01104127	0.0110729	0.0111958	0.01102289
C++ sort	0.1050904	0.09561837	0.0776587	0.07034773	0.0690747	0.07013353	0.070114	0.07971963





	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	result
RadixSort	2.3124398	4.39307087	6.48418097	8.53047187	11.03075237	11.85995123	12.782448	8.19904502
MergeSort	13.4767184	13.4835686	13.34232907	13.2069131	13.9794164	14.81644313	15.93474127	14.03430428
ShellSort	9.25829667	9.6797046	10.11659617	10.4676257	11.15153393	11.94867463	12.89616243	10.78837059
HeapSort	20.8350564	23.45320563	24.02562853	24.2605501	25.57557543	27.64176087	29.5630818	25.05069411
CountingSort	1.09616517	1.1294454	1.0955841	1.0998025	1.19915757	1.24380347	1.3237751	1.16967619
C++ sort	15.06696943	13.1114014	12.0645941	11.20222013	10.64837607	11.533492	12.24392343	12.26728237

Results are pretty much the same. Counting sort remains to be the winner here, followed by radix and shell sort. Heap sort times are spiking up again, remaining to be the most inefficient one for an array with elements in reverse order.

Final conclusions

While **counting sort** has overall the best time results (even for array with length of 1000, if elements are small it's a good choice) its weakness is in space cost. If the range of values is big, then the counting sort requires a lot of space (sometimes more than *O(n)*).

We can see that **shell sort** is not a stable sorting algorithm. Instability here comes because of the increment-based sorts that moves elements without examining the elements in between. It has pretty good results for partially sorted array, where maximum element is not that high, and for already sorted arrays, because the number of comparisons for each of the increment-based insertion is the length of the array. It also had good results for arrays with elements placed in reverse order.

Heap sort is time efficient algorithm with time complexity of *O(nlogn)* and also has a consistent performance. This means that it performs equally well in the best, average and worst cases (we can see almost the same running times for randomized, partially sorted, already sorted and arrays with elements in reverse order), even though it's not the best choice for an already sorted array, losing to merge sort.

Merge sort is very fast for larger arrays, because it doesn't go through array many times (like bubble sort for example). On the other hand, it is slow for small arrays, uses extra space to store subarrays (space complexity is *O(n)*) and also does the whole process if the array is already sorted.

Radix sort performed really great overall. It's a very fast algorithm mainly used to sort arrays with large integers, which it does pretty good.

C++ STL sorting algorithm shows pretty good times, performing well on any kind of arrays that was used for the test. It's stable no matter how many elements are there in the array. While sometimes it can be slower (for example radix is faster when the array has 10^8 partially sorted elements) it is still a safe choice.