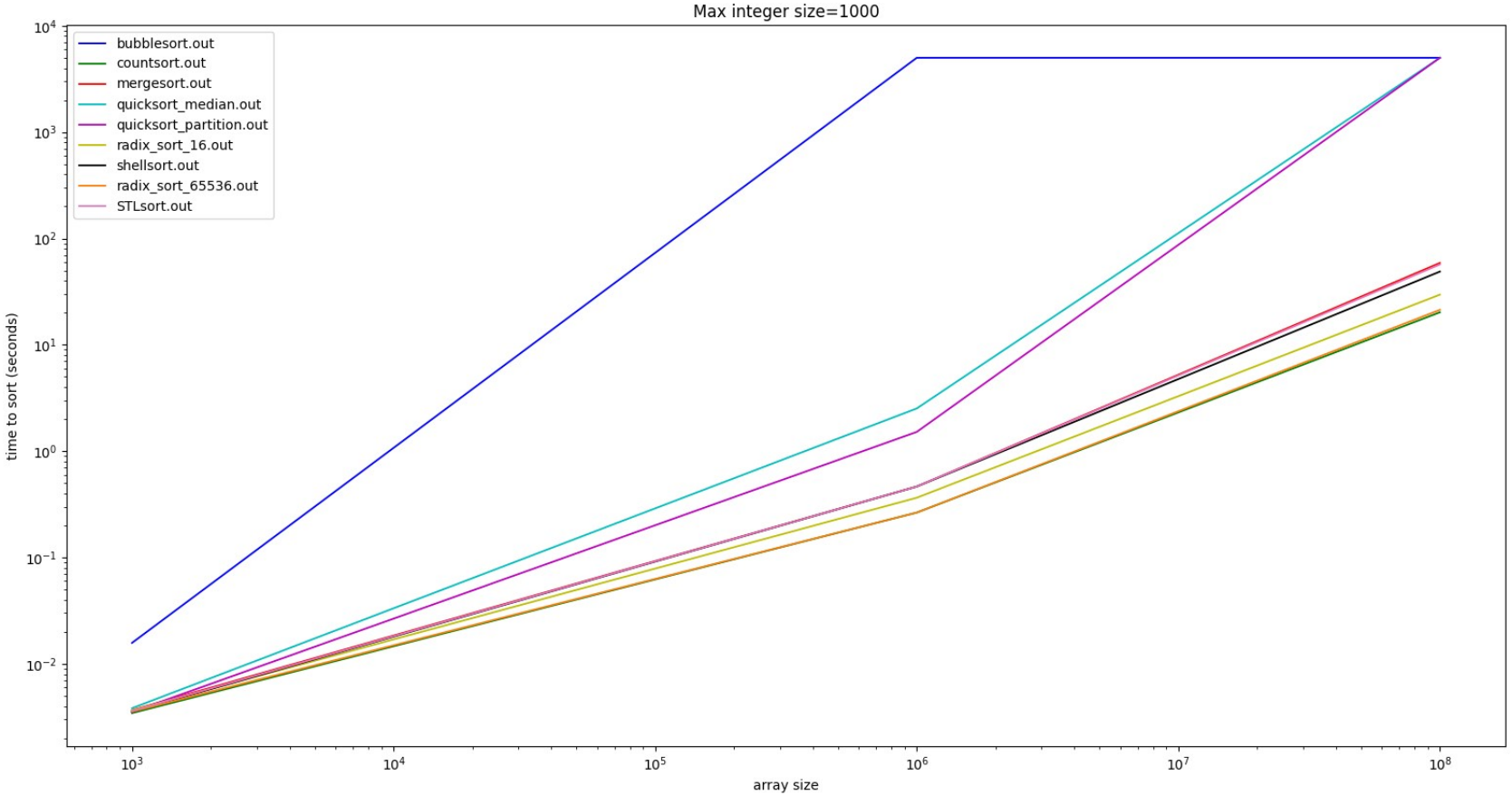
# Sorting algorithms: what to use, and when?

# Test 1




STLsort is the sorting algorithm provided by the c++ standard library It is added for comparison purposes

quicksort_partition is the quicksort algorithm with the usual partition (we pick the leftmost element)

When an algorithm reaches the top of the graph, it means it was timed out for taking too long. The algorithm is then assumed to take 500s to finish its job.

radix_sort_x=radix sort in base x

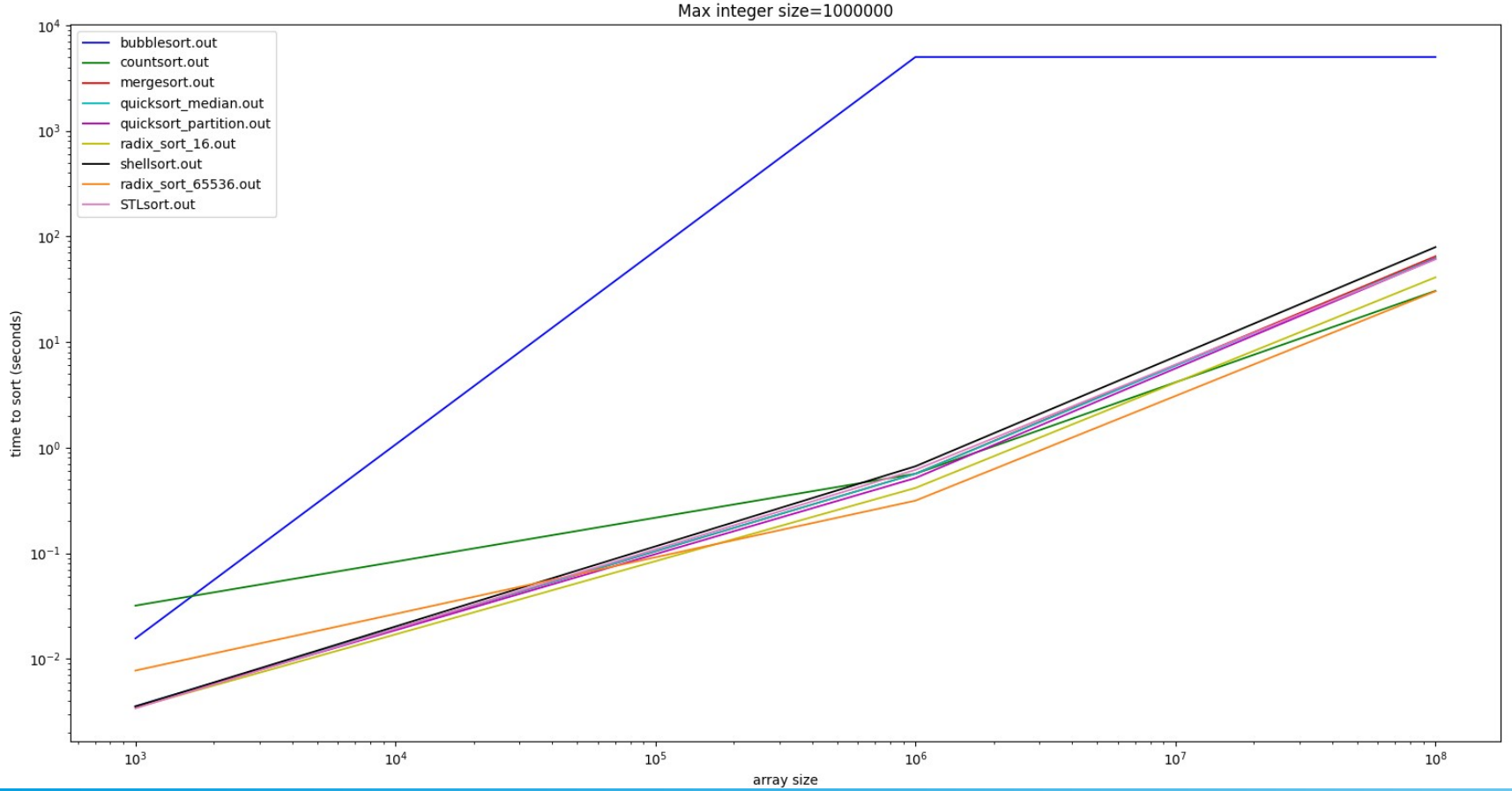
# Conclusion

- Count sort and radix sort are ultra efficient if the maximum integer size is small.
- Both quick sort algorithms timed out for the biggest array: this is because we are trying to fill an array of size 10^8 with 1000 distinct values (the maximum number inside of the array). This leads to a very clustered vector, that is relatively sorted (i.e. it looks something like [1,2,1,2,1,1,1,2,2,1,2,1]). This proves that quick sort with median of 3 is still not guaranteed to produce fast results and 3 way quick sort may “fix” the problem depending on the implementation.

# Conclusion

- Radix sort gets a lot more efficient if we increase the size of the base we are using, and it is a power of 2.
- Keep in mind that the plot has a logarithmic scale, so some algorithms may appear to be more efficient/inefficient than they actually are. Consult the results.txt file for the actual data.
- Bubble sort is the slowest of all algorithms, and gets outperformed even at sorting small random vectors
- Merge sort has a significant amount of overhead at creating auxiliary space, and for this reason it gets beaten by shell sort at sorting large arrays, even though it has a better algorithmical complexity

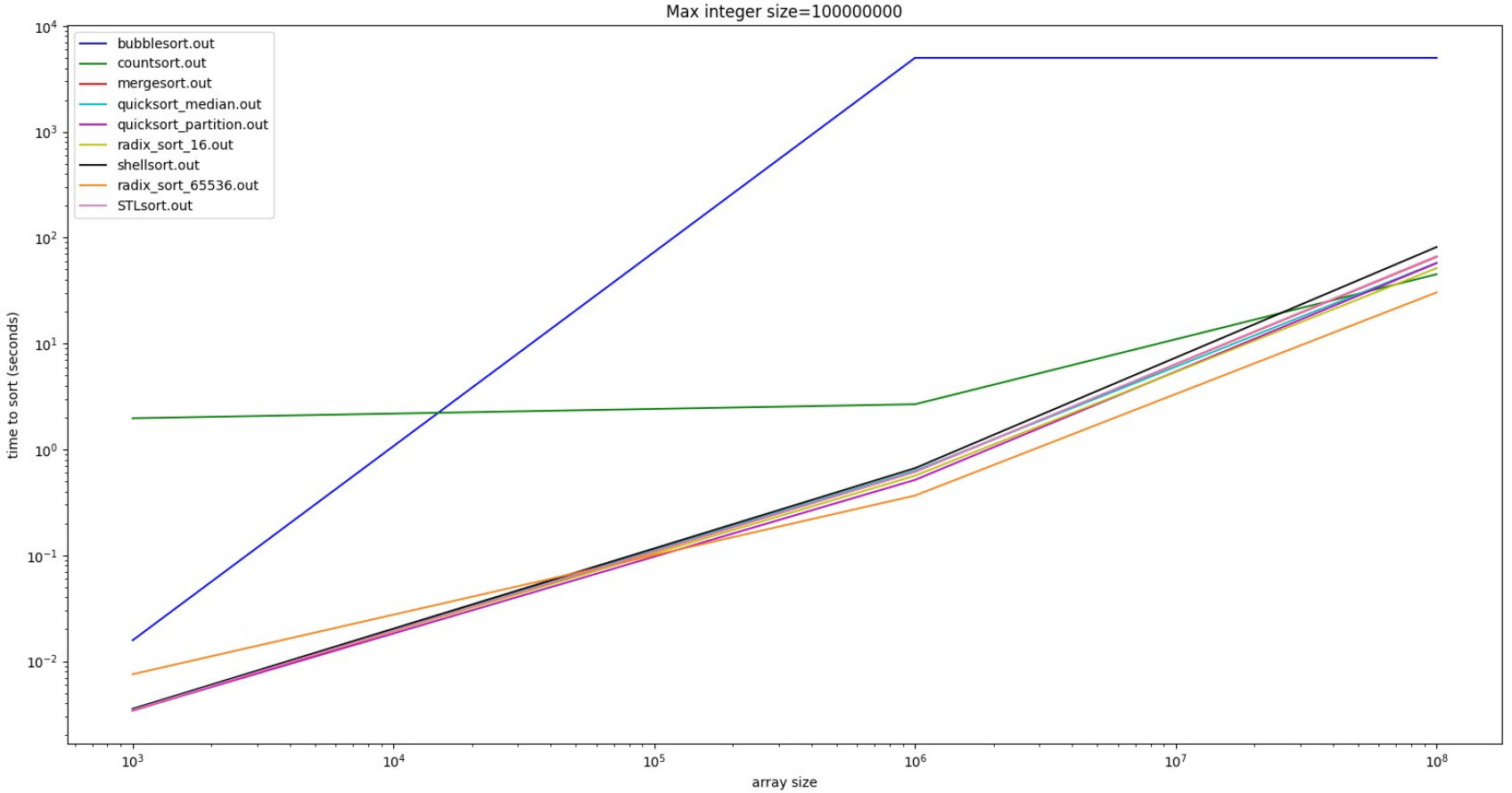
# Test 2


Max integer size=1000000
bubblesort.out
countsort.out
mergesort.out
quicksort_median.out
quicksort_partition.out
radix_sort_16.out
shellsort.out
radix_sort_65536.out
STLsort.out
time to sort (seconds)
array size
$10^{4}$
$10^{3}$
$10^{2}$
$10^{1}$
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
$10^{7}$
$10^{8}$

# Conclusion

- Count and radix sort in a big base are not suitable for sorting small vectors with a large maximum.
- They are still the most efficient algorithms at sorting large vectors (10^8) with a large maximum number (10^6)
- Bubble sort is still not a viable algorithm for sorting random arrays of integers
- Both quick sort algorithms didn’t time out: it is because the max. number is a lot bigger, and the integers are randomly distributed in the interval [0;max. number] , however if the array was sorted, they would still have timed out.

# Test 3


Max integer size=100000000
bubblesort.out
countsort.out
mergesort.out
quicksort_median.out
quicksort_partition.out
radix_sort_16.out
shellsort.out
radix_sort_65536.out
STLsort.out
time to sort (seconds)
array size
$10^{4}$
$10^{3}$
$10^{2}$
$10^{1}$
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
$10^{7}$
$10^{8}$

# Conclusion

- Count sort is massively inefficient for sorting small arrays (or arrays < 10^6) with a very large max. number. This is because it has an algorithmical complexity of O(n+d) where n is the number of integers and d is the biggest number. It is also an algorithm that is prone to failing, since it occupies ram memory at least as much as the biggest number. I recommend using radix sort instead of count sort every time, since it has the same performance for large arrays but won’t fail on you.
- Bubble sort is still the least efficient algorithm.

# Conclusion

- When the array size is small and the max integer is very big, using a radix sort in a large base is not suitable, as it has too much overhead, and radix sort in a small base is still slower than quick sort. Radix sort in a large base is still the fastest by far when the array is also very big.
- Both quick sort algorithms had the same performance overall and picking a median seems virtually useless for arrays of random integers.

# conclusion

- Again, if you want to check the precise performance of the algorithms, check out the results.txt file

- If you want to zoom in very deep on the plots, you can using matplotlib’s gui, just run the gen_plotx.py command for you plot.
- The sorting algorithm provided by the C++ STL was the most worth it to use, since it requires no effort and it is still good enough.