Measure Sort Algorithm Execution Time

Introduction

Efficiency

There are two kinds of efficiency: time efficiency and space efficiency. **Time efficiency**, also called as time complexity, indicates how fast an algorithm in question runs. **Space efficiency**, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output [Levitin, 2007].

Since the space aspect differ on storage used, such as main memory (ie, SRAM) which fastest, secondary memory (ie., RAM), and cache (ie., disk drive), the space efficiency not much concerned. So that is, time efficiency is more important indicator which used to choose better solution.

Measurement Unit of Time

There are two standard time measurement used in modern computing: Wall clock time and CPU clock time.

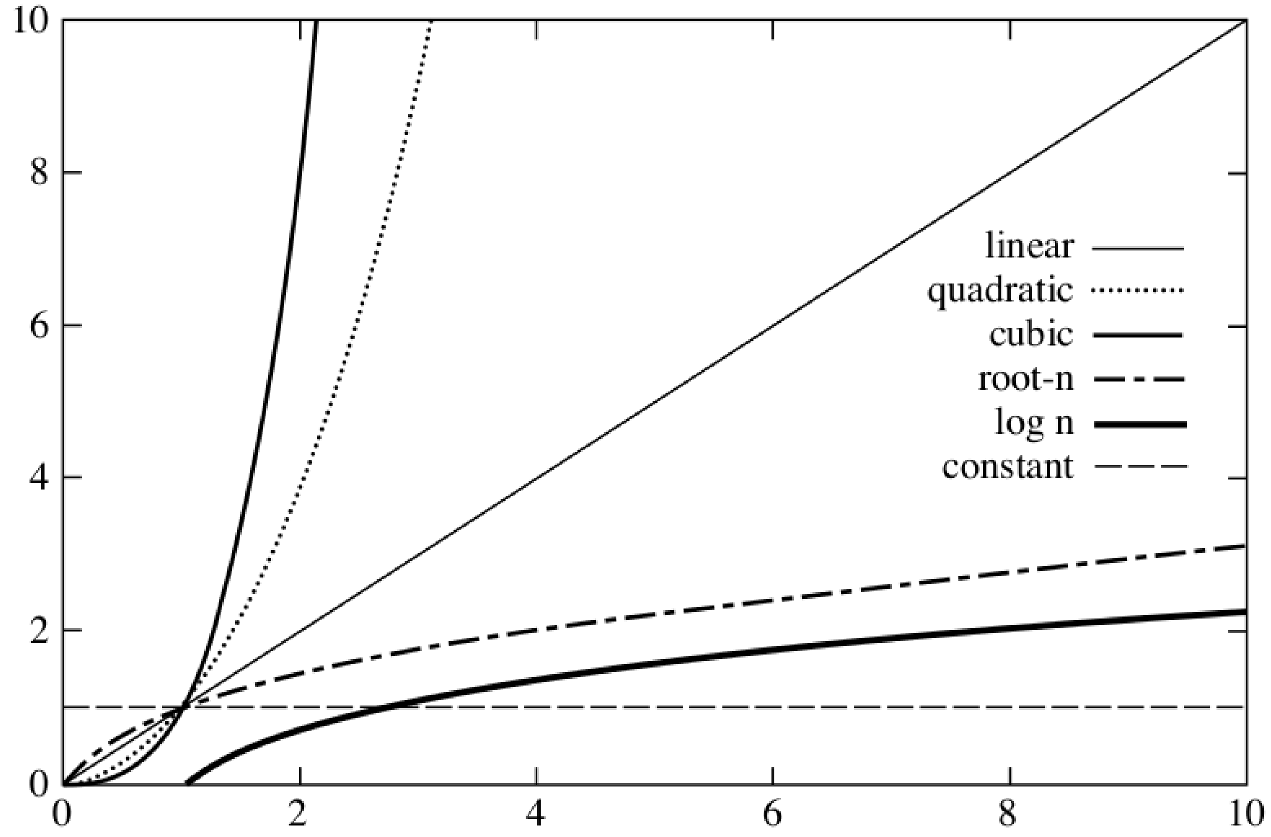
Measurement using wall clock time is gross measurement of time needed by computer to done a given process depend on the human wall clock. Its called as gross measurement because in modern computing, a CPU likely executed multiple command at a time. So, if you just give CPU a job to do while you running on modern operating system, the CPU will convert it into many small pieces of problem if possible, than solved it one by one until reached final problem. When, there is some piece of problem from another process still running, your job piece will waiting and placed in the job queue to be executed. Wall clock in C of now time is seconds since Epoch (00:00:00 GMT, 1st of January 1970).

Measurement using CPU clock time is reliable measurement of how many CPU cycle needed to executed a process. Every CPU has one or more (multi core) global clock defined in frequency (cycle per second) [Schaffer, 2017]. Since cycle is not convenient by human, so we still needed to convert CPU cycle unit into seconds. The **clock cycle time** is **1 <divided by> clock second**. So, if a CPU clock is 3 GHz, then the clock cycle time is 1/3,000,000,000 second. If we used CPU clock time to measure program execution in second, then the formula would be **clock cycle of program spent <multiply by> clock cycle time**. C provided global variable called as CLOCK_PER_SECOND which called from time.h library to convert CPU clock cycle time into clock time in seconds [GNU, 2017][Smith, 2017].

Asymptotic Analysis

Sometimes, we want to know the growth rate of execution time if *n* grows larger. To do that, we implemented derivative concept in calculus into time efficiency measurement of computer program. Rate of growth of the execution time will represented as ∂-time versus ∂-time.

Big Oh notation is a standard notation that is used to simplify the comparison between tow or more algorithms. The notation written as **O(n)** or read as “big-Oh of n”. The idea behind this notation is implementing derivative concept of calculus as measurement tool to conclude rate of growth of execution time. A task that requires the same amount of time regardless of the input size is described as O(1), or "constant time". A task that is O(n) is termed a linear time task. A task that is O(log n) is called logarithmic. A task that is O(n^2) is called quadratic. A task O(n^3) is called cubic, and so on [Todorovic, 2016]. For example: The time complexity of an algorithm is O(n) means that execution time is bounded by some constant times n or O(n) written as c*n. If input size doubles, then the execution time is c*(2n).



Experiment Specification

Machine

Hardware specification has biggest influence in CPU running time in modern computing. This fact, dramatically change programming paradigm which tend to be more convenient for general usage than concerning on performance. In this benchmark, I used a **Intel Pentium P3600** [Intel, 2017] CPU and **Corsair 2x2GB DDR3 1333 MHz** embedded on **HP Pressario CQ-42**. Here the important specification of my notebook:

• Cores	: 2
• Threads	: 2
• Processor Base Frequency	: 2.27 GHz
• Cache	: 3 MB
• Memory size	: 4 GB (Max 8 GB)
• Memory type	: DDR3 1066
• Memory channels	: 2
• Max memory bandwidth	: 17.1 GB/s
• Physical address extensions	: 36-bit

Program

The main program that we used to do benchmark is simple batch program called as **benchmark_sort**. This main program will do benchmark on six sort algorithm that have been improved as possible. Here sort algorithm program used in this benchmark:

- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- Quick short
- Merge short

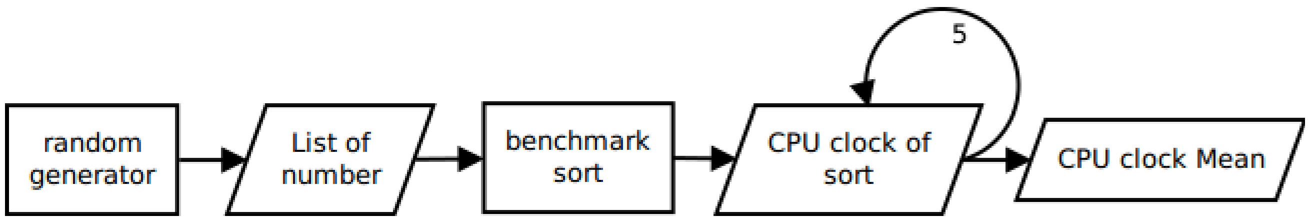
Since every implemented sort algorithm have been tested, the main program will not print out data before and after sorted. To do proper and wise benchmark, main program read list of integer from static file.

For convenient, we developed a tool to generate files contain list of integer number called as **random_generator**.

Experiment Method

First step, we generate files contain list of random number which will be used to do benchmarking process. There are ten generated files which has different length: 50,000; 100, 000; 150,000; 200,000; 250,000; 300,000; 350,000; 400,000; 450,000 and 500,000.

Second step, we run benchmark process with specific sort algorithm and a random file. This benchmark process produces elapsed wall clock of specific sort algorithm and CPU clock time spent by a specific sort algorithm. We only used CPU time in this benchmark. Somehow, the CPU clock time produced by benchmark process on same sort algorithm always different with very small miss match in micro second. So, we used mean of five CPU clock time to calculate each sort algorithm running time.



Experiment Result

The result excluded from our experiment divided into three parts: **Individual clock time** intended to explain raw result of our execution time benchmark. To ensure high quality evidence, we commit every calculate precisely as possible where looking smallest gap result. Each individual benchmark result will be sorted using only human eye or selective skill; **Average clock time** intended to explain average result of each algorithm execution time; ; **Overall execution time** intended to explain overall execution time of all algorithm, so we can see the head to head execution time between sort algorithm.

Individual clock time

In the first experiment using 5000 list of integers, shell sort faster than another comparison based sort, insertion, selection and bubble sort respectively. While, in divide and conquer based, merge sort faster that quick sort.

Data Length: 50000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.01	0.01	0.01	0.01	0.01
2	Quick	0.06	0.05	0.05	0.05	0.05
3	Shell	3.42	3.42	3.42	3.42	3.42
4	Insertion	3.64	3.65	3.64	3.69	3.64
5	Selection	6.12	6.1	6.1	6.09	6.1
6	Bubble	25.59	25.59	25.56	25.58	26.63

In the second experiment using 100000 list of integers, the ranked result looked same like before.

Data Length: 100000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.02	0.02	0.02	0.02	0.02
2	Quick	0.2	0.2	0.2	0.2	0.2
3	Shell	13.67	13.67	13.67	13.66	13.69
4	Insertion	14.5	14.57	14.49	14.49	14.5
5	Selection	24.38	24.39	24.43	25.7	26.03
6	Bubble	99.95	99.85	100.34	102.47	100.43

In the third experiment using 150000 list of integers, the ranked result looked same like before.

Data Length: 150000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.04	0.04	0.04	0.04	0.04
2	Quick	0.44	0.44	0.44	0.44	0.44
3	Shell	27.12	27.1	27.1	27.09	27.1
4	Insertion	32.76	32.94	32.74	32.76	32.76
5	Selection	58.32	58.26	58.27	58.29	58.26
6	Bubble	241.36	233.11	237.48	238.72	251.66

In the third experiment using 200000 list of integers, the ranked result looked same like before.

Data Length: 200000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.05	0.05	0.05	0.05	0.05
2	Quick	0.77	0.78	0.77	0.77	0.77
3	Shell	55.04	54.91	54.89	54.95	54.93
4	Insertion	58.05	58.04	58.03	58.03	58.04
5	Selection	97.51	97.51	97.51	97.5	97.5
6	Bubble	403.1	402.67	401.95	401.11	402.62

In the third experiment using 250000 list of integers, the ranked result looked same like before.

Data Length: 250000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.06	0.06	0.06	0.06	0.06
2	Quick	1.19	1.19	1.19	1.21	1.19
3	Shell	78.73	79.58	79.02	78.66	78.69
4	Insertion	90.69	90.58	90.64	90.65	90.57
5	Selection	152.35	152.36	152.38	152.33	152.35
6	Bubble	627.68	631.17	637.48	636.57	659.44

In the third experiment using 300000 list of integers, the ranked result looked same like before.

Data Length: 300000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.08	0.08	0.08	0.08	0.08
2	Quick	1.71	1.71	1.71	1.71	1.72
3	Shell	110.36	108.24	108.35	108.03	108.15
4	Insertion	130.85	130.93	131.31	131.26	131.05
5	Selection	230.02	228.21	230.2	234.23	228.16
6	Bubble	945.93	985.66	975.09	996.5	1014.53

In the third experiment using 350000 list of integers, the ranked result looked same like before.

Data Length: 350000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.09	0.09	0.09	0.09	0.09
2	Quick	2.32	2.32	2.32	2.32	2.35
3	Shell	142.54	142.93	142.75	142.67	142.73
4	Insertion	179.53	179.52	179.92	179.27	180.36
5	Selection	302.26	300.98	300.01	301.28	298.97
6	Bubble	1246.26	1287.91	1276.32	1376.69	1352.93

In the third experiment using 400000 list of integers, the ranked result looked same like before.

Data Length: 400000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.1	0.1	0.1	0.1	0.1
2	Quick	3.02	3.02	3.02	3.02	3.02
3	Shell	227.99	222.64	220.35	231.6	221.9
4	Insertion	236.6	232.65	232.59	232.51	232.76
5	Selection	390.43	395.63	390.31	398.63	393.49
6	Bubble	1751.91	1702.37	1609.75	1613.61	1610.09

In the third experiment using 450000 list of integers, the ranked result looked same like before.

Data Length: 450000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.12	0.12	0.12	0.12	0.12
2	Quick	3.81	3.81	3.86	3.84	3.86
3	Shell	269.83	265.79	263.38	268.21	274.62
4	Insertion	296.72	309.17	295.03	295.47	295.24
5	Selection	494.05	494.42	510.51	494.71	494.2
6	Bubble	2227.08	2038.93	2048.7	2054.6	2054.98

In the third experiment using 500000 list of integers, the ranked result looked same like before.

Data Length: 500000						
No	**Algorithm**	**Clock time (second)**				
		1	**2**	**3**	**4**	**5**
1	Merge	0.13	0.13	0.13	0.13	0.13
2	Quick	4.69	4.7	4.69	4.76	4.69
3	Shell	315.01	315.52	314.51	314.32	314.42
4	Insertion	363.25	363.85	367.41	365.33	370.12
5	Selection	611.74	613.38	611.29	613.35	611.28
6	Bubble	2752.86	2713.31	2533.99	2534.92	2536.49

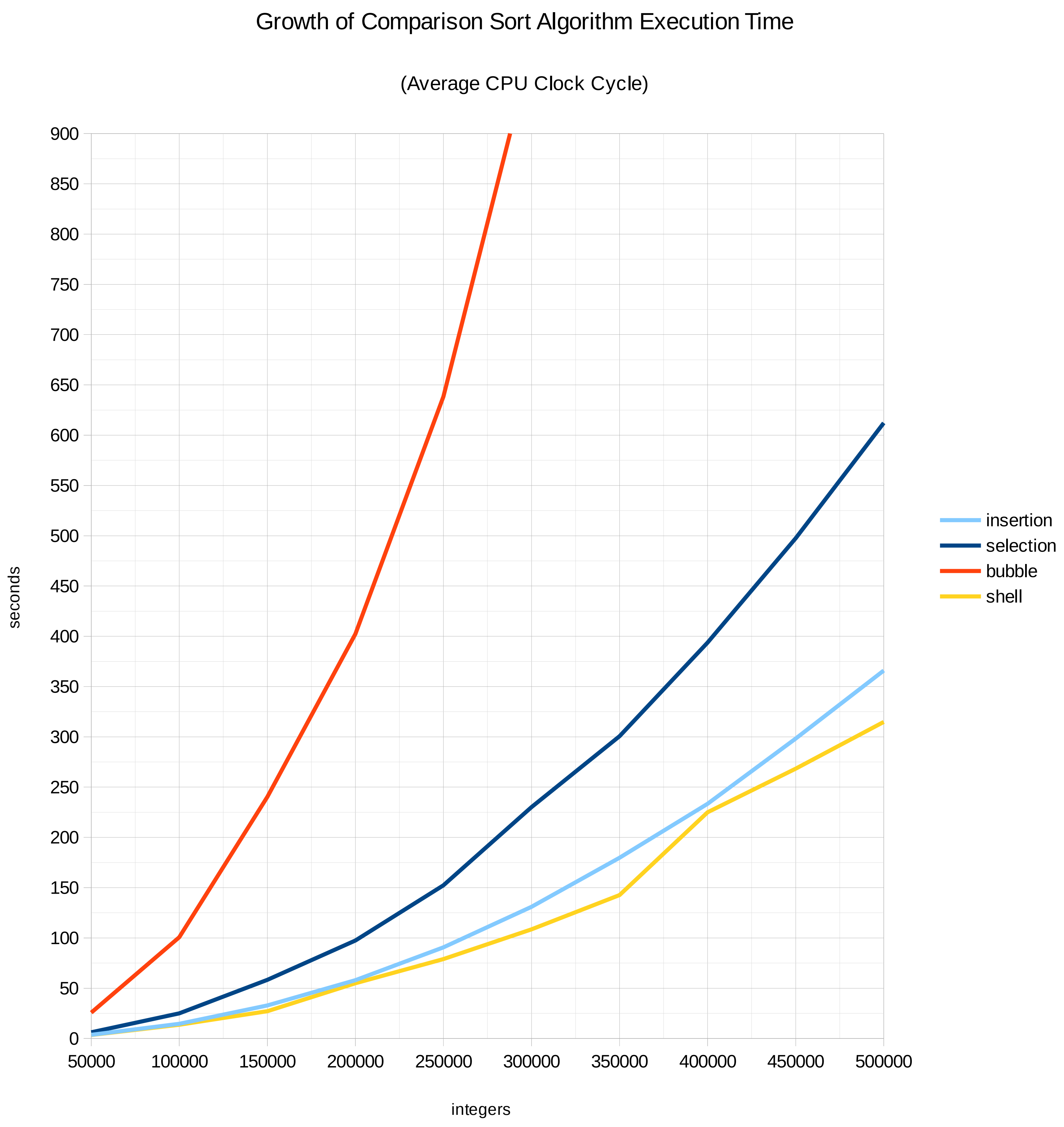
Average clock time

We calculate the average clock cycle time for each algorithm to build graph of growth of time.

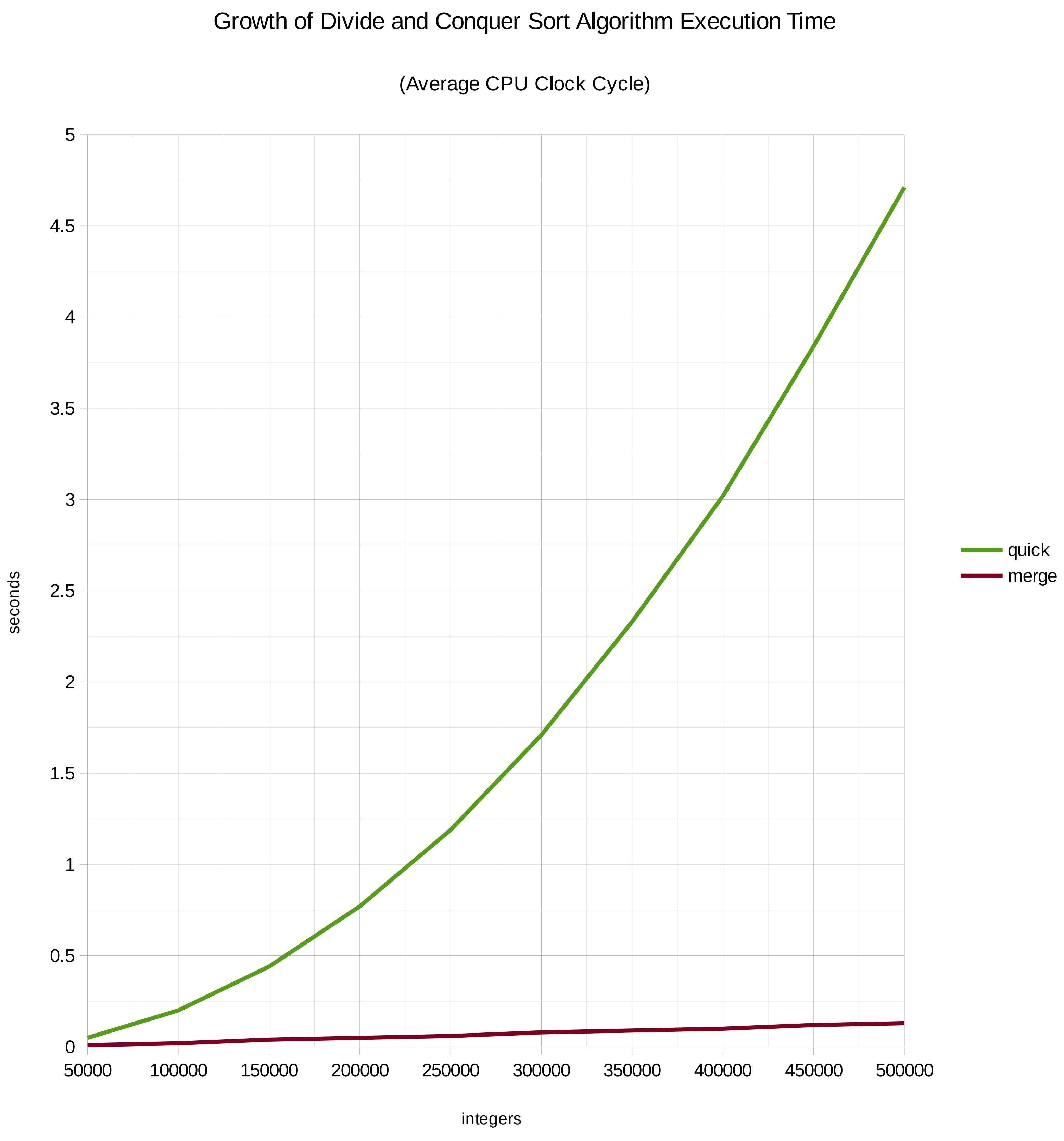
No	**Data length**	**Average Clock time (second)**					
		insertion	**selection**	**bubble**	**shell**	**quick**	**merge**
1	50000	3.65	6.1	25.79	3.42	0.05	0.01
2	100000	14.51	24.99	100.61	13.67	0.2	0.02
3	150000	32.79	58.28	240.47	27.1	0.44	0.04
4	200000	58.04	97.51	402.29	54.94	0.77	0.05
5	250000	90.63	152.35	638.47	78.94	1.19	0.06
6	300000	131.08	230.16	983.54	108.63	1.71	0.08
7	350000	179.72	300.7	1308.02	142.72	2.33	0.09
8	400000	233.42	393.7	1657.55	224.9	3.02	0.1
9	450000	298.33	497.58	2084.86	268.37	3.84	0.12
10	500000	365.99	612.21	2614.31	314.76	4.71	0.13

Overall Execution Time

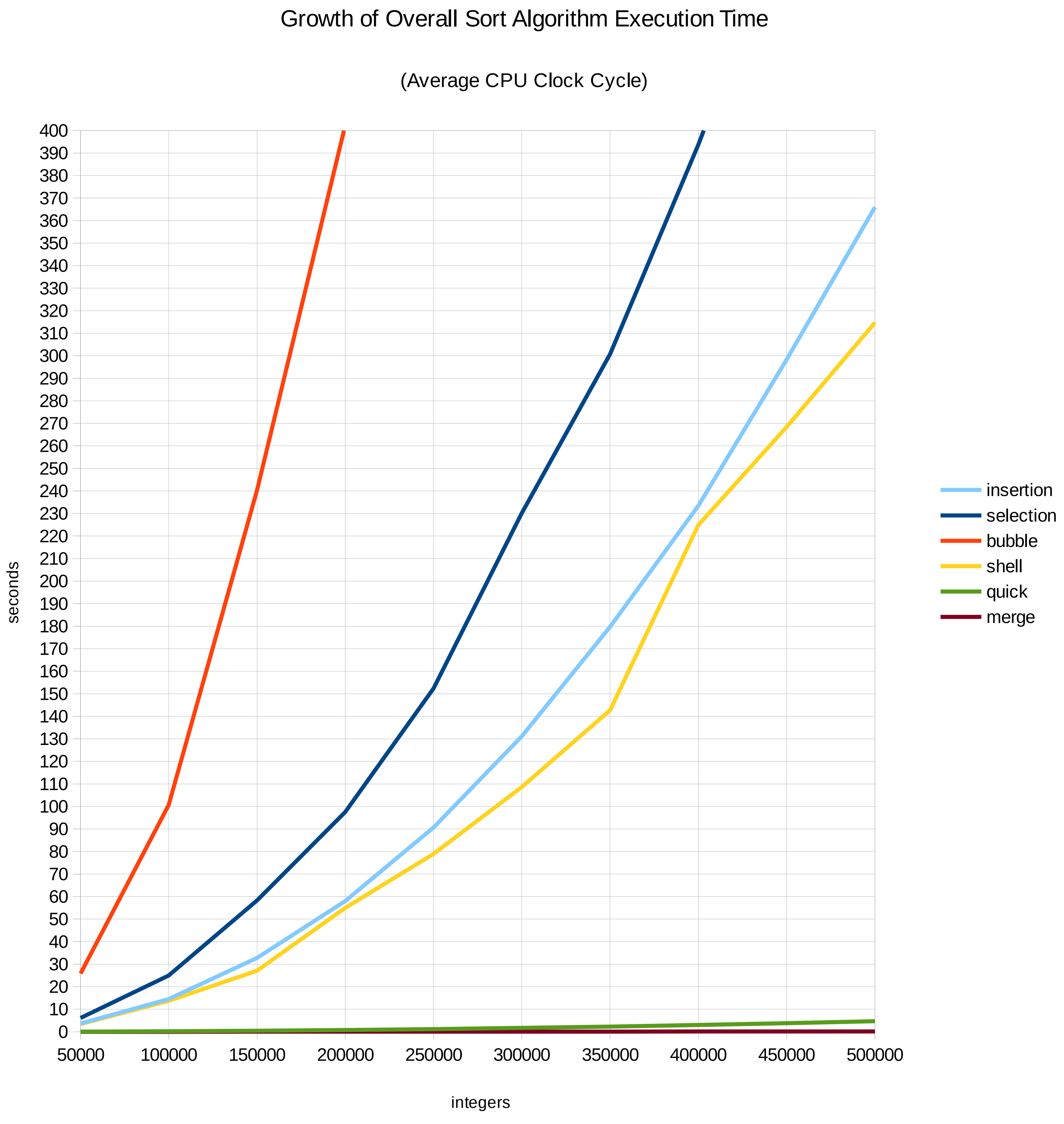
1. Comparison Based Sort Algorithm



2. Divide and Conquer Based Sort Algorithm



3. All Sort Algorithm



Algorithmic Analysis

Show only head to head growth of time for each algorithm is not quietly useful for practical purpose because we have very limited dataset. In this experiment, we only used a kind of dataset which allowed duplicated data. In the practical purpose, some sort algorithm used in specific case, for example: selection sort often used in case with sorted dataset or in small size dataset.

We used asymptotic notation to observe rate of growth of execution time for each algorithm. The asymptotic will be represented as table where left most row is growth of time and top column is growth of input size. We divided this section into two: asymptotic of comparison method and asymptotic of divide and conquer method.

Asymptotic of Comparison Method

All comparison based sort algorithms are quadratic task. Almost all comparison based sort have time complexity equal to O(n^2), except shell shot has unique quadratic pattern. The execution time of shell sort would be **double** if only if the input size is equal to **log$_2$**, so the time complexity of shell sort is O($n^{\log_2(i)}$).

Comparison based

c*n	insertion	selection	bubble	shell
1	1	1	1	1
2	3.9753424658	4.0967213115	3.9011244668	3.9970760234
3	8.9835616438	9.5540983607	9.3241566499	7.9239766082
4	15.901369863	15.985245902	15.59868166	16.064327485
5	24.830136986	24.975409836	24.756494765	23.081871345
6	35.912328767	37.731147541	38.13648701	31.763157895
7	49.238356164	49.295081967	50.718107794	41.730994152
8	63.950684932	64.540983607	64.271035285	65.760233918
9	81.734246575	81.570491803	80.839860411	78.470760234
10	100.27123288	100.36229508	101.36913532	92.035087719

Asymptotic of Divide and Conquer Method

Unluckily, we have implemented 'not good' pivoting technique in the implementation of quick short that lead the time complexity of quick sort became quadratic or almost O(n^2) with small fluctuation.

The implemented merge sort looked better that implemented quick sort. The time complexity of implemented merge sort is almost linear. The execution time of merge sort would be **increase by one unit** if only if the input size is equal to **$\mathbf{log_3}$**, so the time complexity of merge sort is O($n+\log_3(i)$).

Divide and conquer

c*n

1	1	1
2	4	2
3	8.8	4
4	15.4	5
5	23.8	6
6	34.2	8
7	46.6	9
8	60.4	10
9	76.8	12
10	94.2	13

Conclusion

An comparison based sort algorithm produce some fluctuation time between trial run, although we have anticipated by using CPU clock time as measurement unit and repeat execution until produced reasonable execution time. Sometimes, we should executed more than five run, bad proven algorithm like bubble sort have been executed until ten run because sometime our machine over heated or produced very big gap. Besides that, we realized that CPU clock timing used in this experiment is not very precise because still affected by other program activity running on CPU. We have affirmed this phenomenon by executed algorithm while executing another program, then compare the execution time with another execution time produced by algorithm execution with less user activity environment.

Beside fluctuation problem, the analysis of experiment show that:

1. Bubble sort is not suitable sort algorithm in any case that used array type dataset.
2. In comparison based method, shell sort is the best, followed by insertion sort very closely. Shell sort became the best because it has time complexity $n^{\log_2(i)}$.
3. In divide an conquer based method, merge sort overlapped quick sort execution time with ratio 1/26.
4. Implemented quick sort has 'not good' pivoting technique that lead quadratic time efficiency.

Future Experiment

We should implement another quick sort and merge variant to investigate big execution time ratio between quick and merge sort produced in this experiment. We want to know clearly how the pivoting technique influenced quick sort time efficiency and how far merge sort can be developed as best solution in various situations.

We will exclude bubble sort from benchmark list because bubble sort proven not suitable for array type dataset and has too long total execution time.

References

[Levitin, 2007] Levitin, A (2007). *Introduction to the design & analysis of algorithms* (Vol. 2). Pearson Education.

[Schaffer, 2017] Schaffer, David Black. **The Role of Performance**. https://www.it.uu.se/edu/course/homepage/dark/ht11/dark6-performance.pdf.

[GNU, 2017] GNU. **The GNU C Library: CPU Time**. http://www.gnu.org/software/libc/manual/html_node/CPU-Time.html. (Accessed 16 August 2017).

[Intel, 2017] Intel. **Intel® Pentium® Processor P6300 (3M Cache, 2.27 GHz) Product Specifications**. http://ark.intel.com/products/51680/Intel-Pentium-Processor-P6300-3M-Cache-2_27-GHz.

[Meglicki, 2001] Meglicki, Zdzislaw. **Timing a job in C**. 2001. http://beige.ucs.indiana.edu/B673/node104.html. (Accessed 16 August 2017).

[Todorovic, 2016] Todorovic, Sinisa. 2016. http://web.engr.oregonstate.edu/~sinisa/courses/OSU/CS261/CS261_Textbook/Chapter04.pdf. (Accessed 17 August 2017)

[Smit, 2017] Smith, Rick. **Timing Execution inside a GCC Program**. http://courseweb.stthomas.edu/resmith/c/cisc130/c9sp/gcc-timing.html. (Accessed 16 August 2017).