Chenlin Li

Sorting Algorithm Research Report

The purpose of the research

In this research, I implement 5 sorting algorithms, Bubble-sort, Insertion-sort, Spin-the-bottle sort, Shell sort, and Annealing sort to determine their real-world running times.

Set up of the research

First, I implemented 5 sorting algorithms with C++. For the Shell sort and Annealing sort, I implemented them with gaps and temps from two professional papers which be cited over 1000 times. According to these two papers, the gaps and temps could help me significantly improving the performance of Shell sort and Annealing sort.

After I finished the implementation, I used random shuffle to create random test cases to test the real- world time. Besides, I created almost sorted test cases to test the difference of running time between random numbers and almost sorted numbers.

The result time data will be stored in the timing.csv. For the graph part, I used anaconda and Spyder 3.3.1 to run the python code to generate the performance graph.

Generate Random number

```
mt19937 get_mersenne_twister_genreator_with_current_time_seed(){
    unsigned seed = chrono::system_clock::now().time_since_epoch().count();
    return mt19937(seed);
}
```

```
void shuffle_vector(vector<int>& nums){
    for(int i = nums.size(); i >= 0; i--){
        mt19937 mt = get_mersenne_twister_genreator_with_current_time_seed();
        int j = randint(mt,0,i);
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

This is the code of mt19937 and the Fisher Yates algorithm which will be used to create test cases.

Generate almost sorted numbers

```
vector<int> almost_random_generate(int n){
    int pair_num = 2*log2(n);
    vector<int> nums;
    for(int i = 0; i < n; i++){
        nums.push_back(i);
    }
    for(int j = 0; j < pair_num; j++) {
        mt19937 mt = get_mersenne_twister_genreator_with_current_time_seed();
        int x = randint(mt, 0, n - 1);
        mt19937 mt1 = get_mersenne_twister_genreator_with_current_time_seed();
        int y = randint(mt1, 0, n - 1);
        int temp = nums[x];
        nums[x] = nums[y];
        nums[y] = temp;
    }
    return nums;
}
```

Finding the slop of the sort algorithm running time graph.

Finding the function from the log–log plot [edit]

The above procedure now is reversed to find the form of the function $F(x)$ using its (assumed) known log–log plot. To find the function F, pick some *fixed point* (x_0, F_0), where F_0 is shorthand for $F(x_0)$, somewhere on the straight line in the above graph, and further some other *arbitrary point* (x_1, F_1) on the same graph. Then from the slope formula above:

$$m = \frac{\log(F_1/F_0)}{\log(x_1/x_0)}$$

which leads to

$$\log(F_1/F_0) = m\log(x_1/x_0) = \log[(x_1/x_0)^m].$$

Notice that $10^{\log_{10}(F_1)} = F_1$. Therefore, the logs can be inverted to find:

$$\frac{F_1}{F_0} = \left(\frac{x_1}{x_0}\right)^m$$

or

$$F_1 = \frac{F_0}{x_0^m}\, x^m,$$

which means that

$$F(x) = \text{constant} \cdot x^m.$$

In other words, F is proportional to x to the power of the slope of the straight line of its log–log graph. Specifically, a straight line on a log–log plot containing points (F_0, x_0) and (F_1, x_1) will have the function:

$$F(x) = F_0\left(\frac{x}{x_0}\right)^{\frac{\log(F_1/F_0)}{\log(x_1/x_0)}},$$

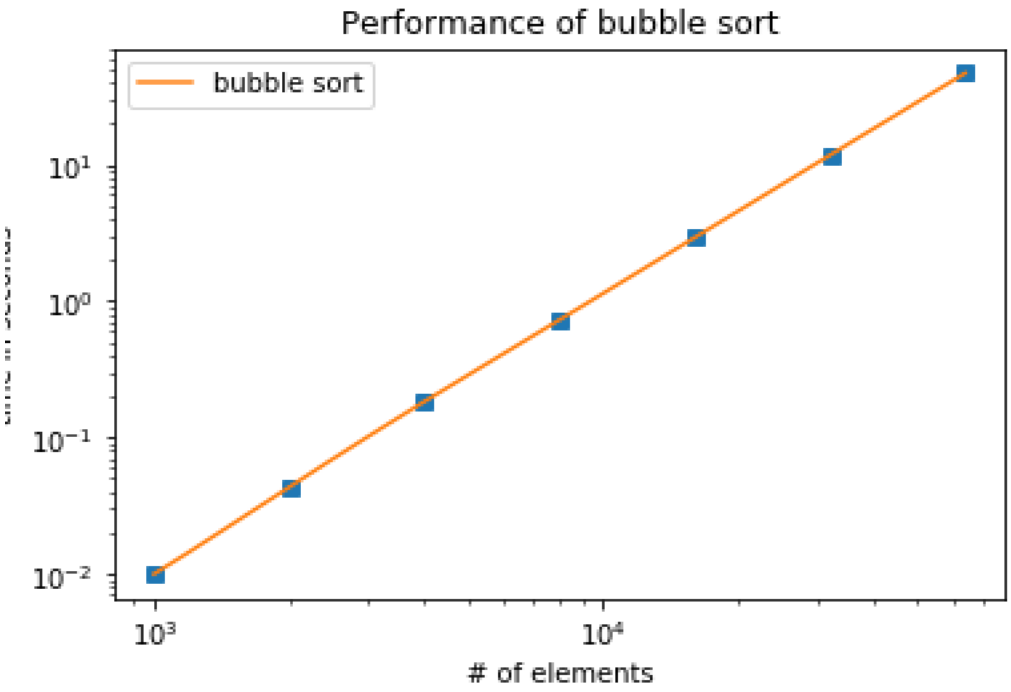
Of course, the inverse is true too: any function of the form

$$F(x) = \text{constant} \cdot x^m$$

Bubble sort algorithm

Random numbers:

funcname	n	seconds
bubble_sort	1000	0.01
bubble_sort	2000	0.0433333
bubble_sort	4000	0.183333
bubble_sort	8000	0.726667
bubble_sort	16000	2.93
bubble_sort	32000	11.73
bubble_sort	64000	46.7133



I used the power of 2, and the base of log was 2.

M = log(0.0433/0.01)/log(2000/1000) = 2;

M = log(0.1833/0.0433)/log(4000/2000)= 2;

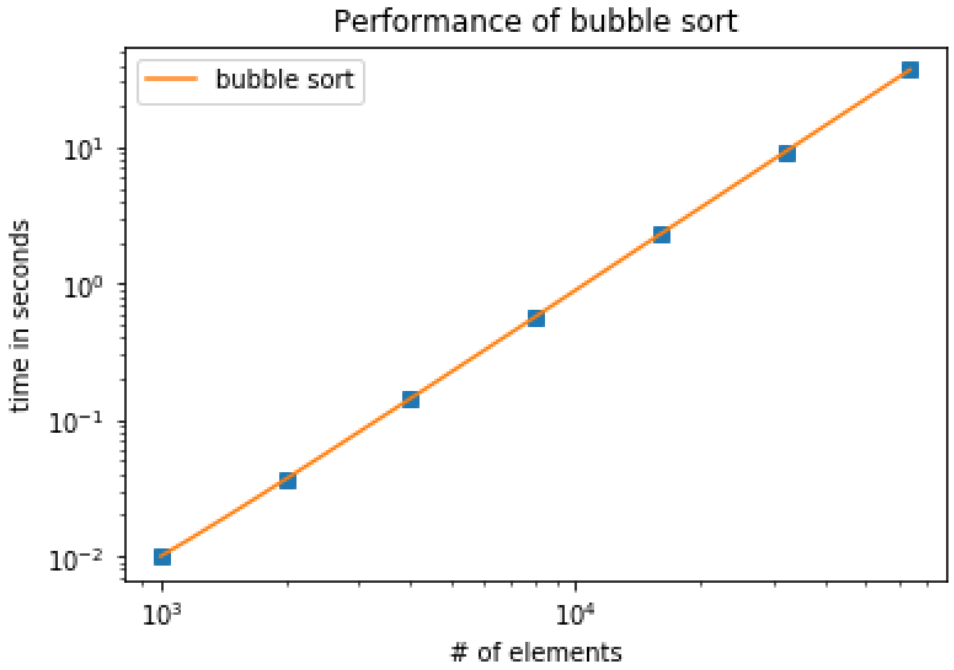
M = log(0.7267/0.1833)/log(8000/4000)=2;

Form the data that I collected, I calculated the slope m was 2.

If the m equaled 2, the function of the running time was constant*n^2 . The real time of the bubble-sort sorting random number was O(N^2).

Almost sorted numbers:

	A	B	C
1	funcname	n	seconds
2	bubble_sort	1000	0.01
3	bubble_sort	2000	0.0366667
4	bubble_sort	4000	0.143333
5	bubble_sort	8000	0.57
6	bubble_sort	16000	2.3



I used the power of 2, and the base of log was 2.

M = log(0.03667/0.01)/log(2000/1000) = 2;

M = log(0.14333/0.03667)/log(4000/2000)= 2;

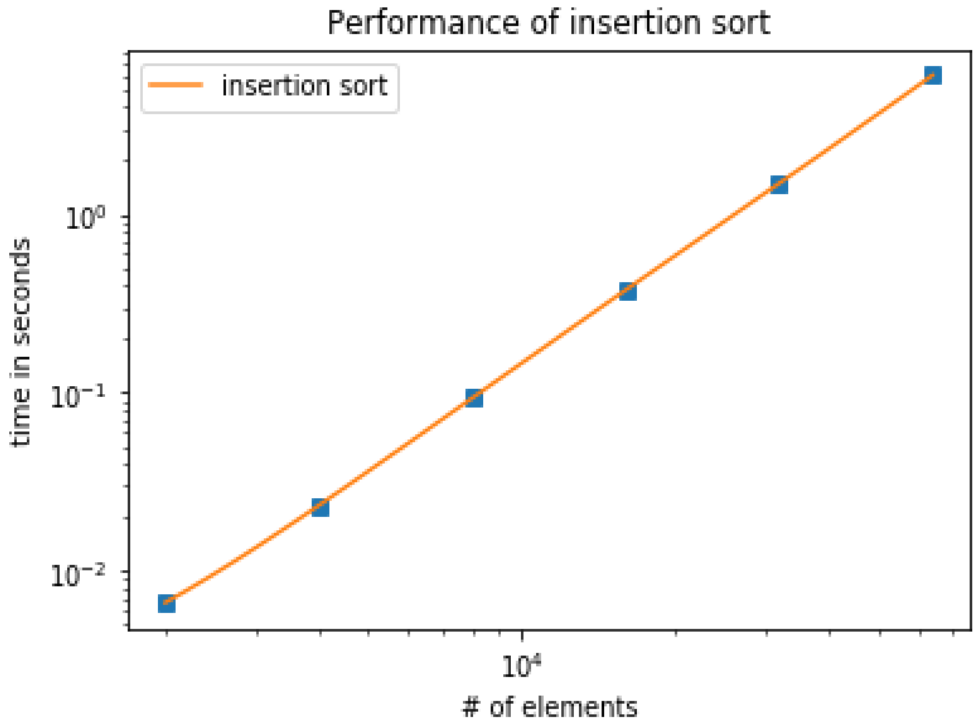
M = log(0.57/0.14333)/log(8000/4000)=2;

Form the data that I collected, I calculated the slope m was 2. The m equaled 2, and the function of the running time was constant*n^2. The real time of the bubble-sort sorting the almost sorted numbers was O(N^2).

Insertion sort algorithm

Random Numbers:

	A	B	C
1	funcname	n	seconds
2	insertion_	2000	0.006667
3	insertion_	4000	0.023333
4	insertion_	8000	0.093333
5	insertion_	16000	0.376667
6	insertion_	32000	1.49
7	insertion_	64000	6.02333



I used the power of 2, and the base of log was 2.

M = log(0.0233/0.0067)/log(4000/2000) = 2;

M = log(0.0933/0.0233)/log(8000/4000)= 2;

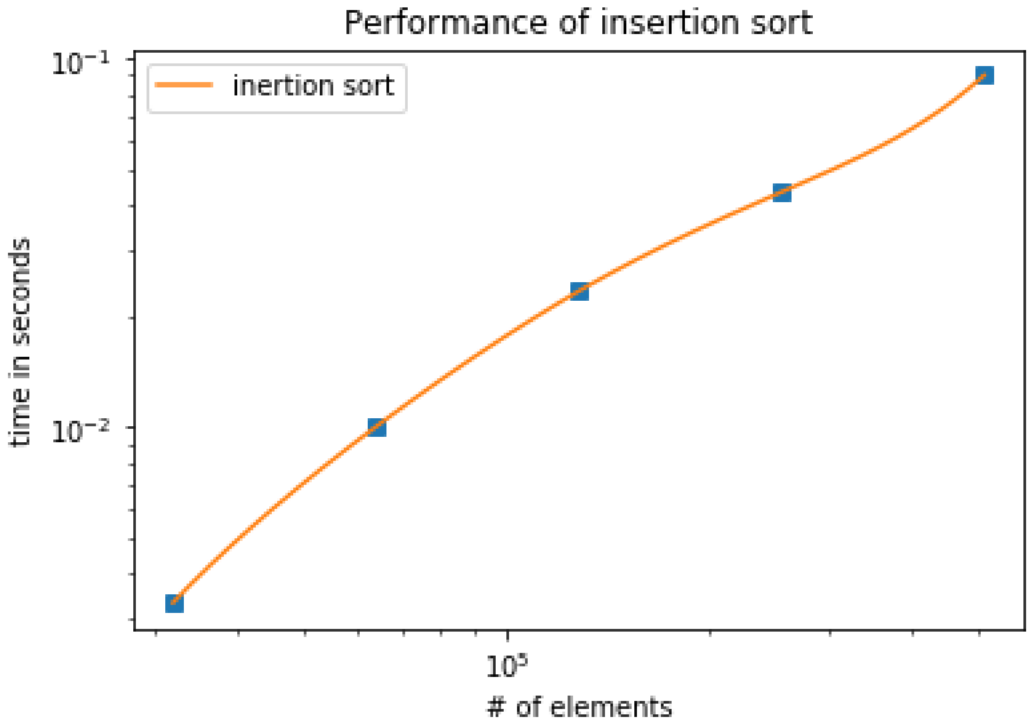
M = log(0.3766/0.0933)/log(16000/8000)=2;

Form the data that I collected, I calculated the slope m is 2.

The m = 2, and the function of the running time was constant*n^2. The real time of the insertion-sort sorting random number was O(N^2).

Almost sorted numbers:

	A	B	C
1	funcname	n	seconds
2	insertion_	32000	0.003333
3	insertion_	64000	0.01
4	insertion_	128000	0.023333
5	insertion_	256000	0.043333
6	insertion_	512000	0.09



Before the 32000, the test case was way too small to show the seconds, so I took off all the result date before 32000.

I used the power of 2, and the base of log was 2.

M = log(0.01/0.0033)/log(64000/32000) = 1;

M = log(0.023/0.01)/log(128000/64000) = 1;

M = log(0.043/0.023)/log(256000/128000) = 1;

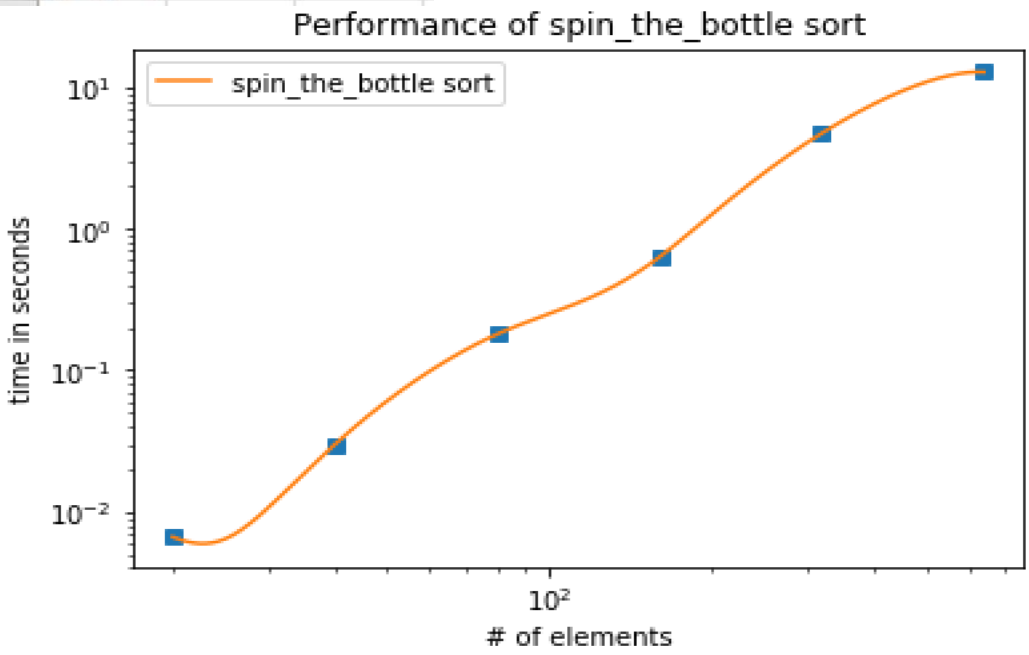
The increasing rate was 2N, and O(2N) = O(N);

In the almost sorted situation, the insertion sort was very fast.The real time of the insertion-sorting almost sorted numbers was O(N).

Spin the bottle sort algorithm

Random Numbers:

	A	B	C
1	funcname	n	seconds
2	spin_the_	20	0.006667
3	spin_the_	40	0.03
4	spin_the_	80	0.18
5	spin_the_	160	0.63
6	spin_the_	320	4.71667
7	spin_the_	640	12.7867



Spin the bottle sorting algorithm based on random sorting algorithm.

I used the power of 2, so the base of log was 2.

M = log(0.03/0.0067)/log(40/20) = 2;

M = log(0.18/0.03)/log(80/40)= 2;

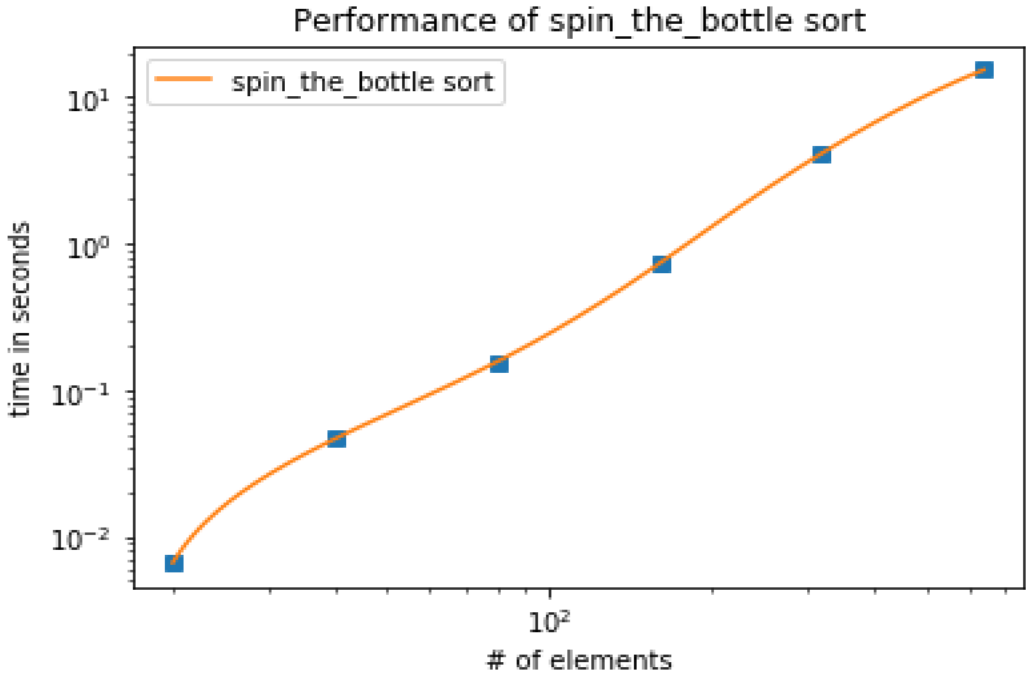
M = log(0.63/0.18)/log(160/80)=2;

Form the data that I collect, I calculated the slope m is 2.

The m = 2, and the function of the running time was constant*n^2. The real time of the spin bottle sorting random number was O(N^2).

Almost Sorted Numbers:

	A	B	C
1	funcname	n	seconds
2	spin_the_	20	0.006667
3	spin_the_	40	0.046667
4	spin_the_	80	0.156667
5	spin_the_	160	0.733333
6	spin_the_	320	4.13
7	spin_the_	640	15.3567



I used the power of 2, and the base of log was 2.

M = log(0.46/0.0067)/log(40/20) = 3;

M = log(0.157/0.047)/log(80/40) = 2;

M = log(0.733/0.15)/log(16000/8000), 3>M>2

Form the data that I collectd, I calculated the slope m was larger than 2.

If the m was larger than 2, the function of the running time could be n^2*logn.

It means the real time of the insertion-sort sorting random number may be O(N^2*longN).

Shell sort

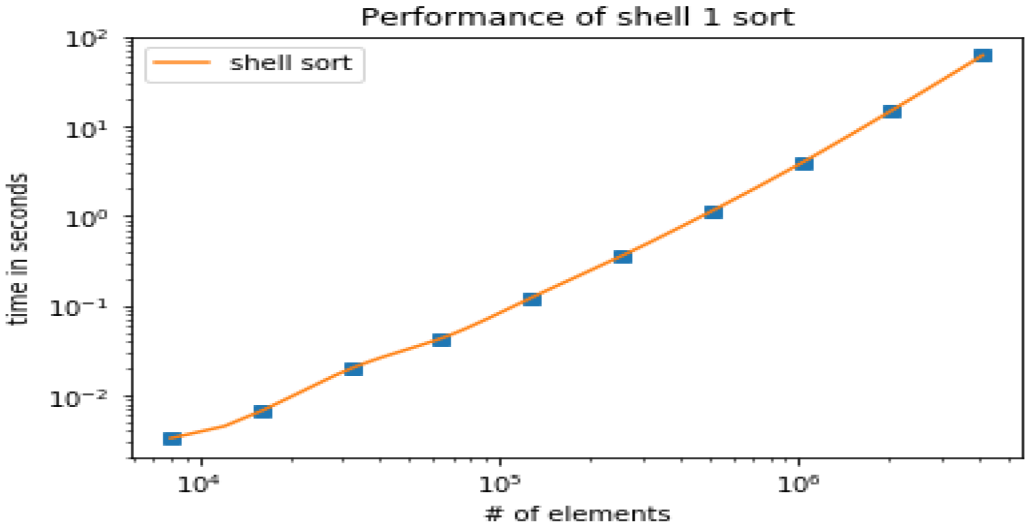
Implementation 1

Gaps: {1073,281,77,23,8,1}

This Gaps comes from Theorem (Sedgewick, 1982). The theory running time of the shellsort should be O(N^4/3)

Random Numbers:

funcname	n	seconds
shell_sort	8000	0.003333
shell_sort	16000	0.006667
shell_sort	32000	0.02
shell_sort	64000	0.043333
shell_sort	128000	0.123333
shell_sort	256000	0.36
shell_sort	512000	1.14
shell_sort	1024000	3.95
shell_sort	2048000	15.2033
shell_sort	4096000	62.0067



I used the power of 2, and the base of log was 2.

M = log(0.0067/0.0033)/log(16000/8000) , 2>M>1;

M = log(0.02/0.0067)/log(32000/16000), >2M>1 ;

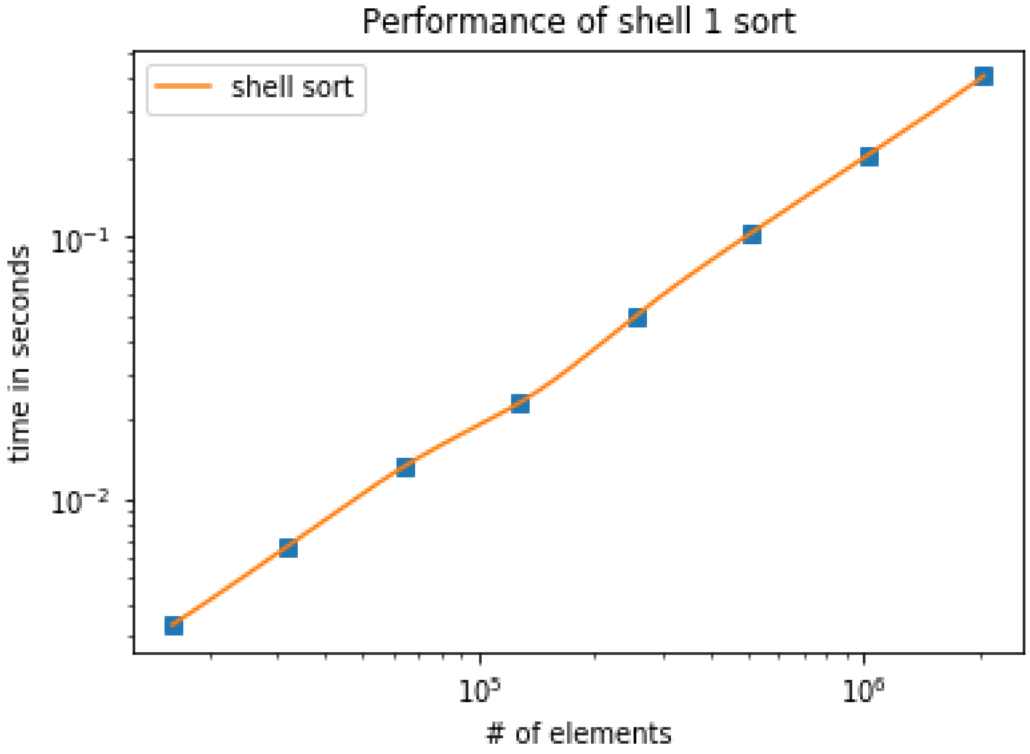
M = log(0.0433/0.02)/log(64000/32000), 2>M>1;

M = log(0.1233 /0.0433)/log(64000/32000), 2>M>1;

The collect data satisfied the theory of the running time was O(n^4/3).

Almost Sorted Numbers:

funcname	n	seconds
shell_sort	16000	0.003333
shell_sort	32000	0.006667
shell_sort	64000	0.013333
shell_sort	128000	0.023333
shell_sort	256000	0.05
shell_sort	512000	0.103333
shell_sort	1024000	0.203333
shell_sort	2048000	0.406667



The collect data and the graph were very similar as the graph and data of the random numbers. The running time of the random and almost sorted numbers was same. It was O(N^4/3).

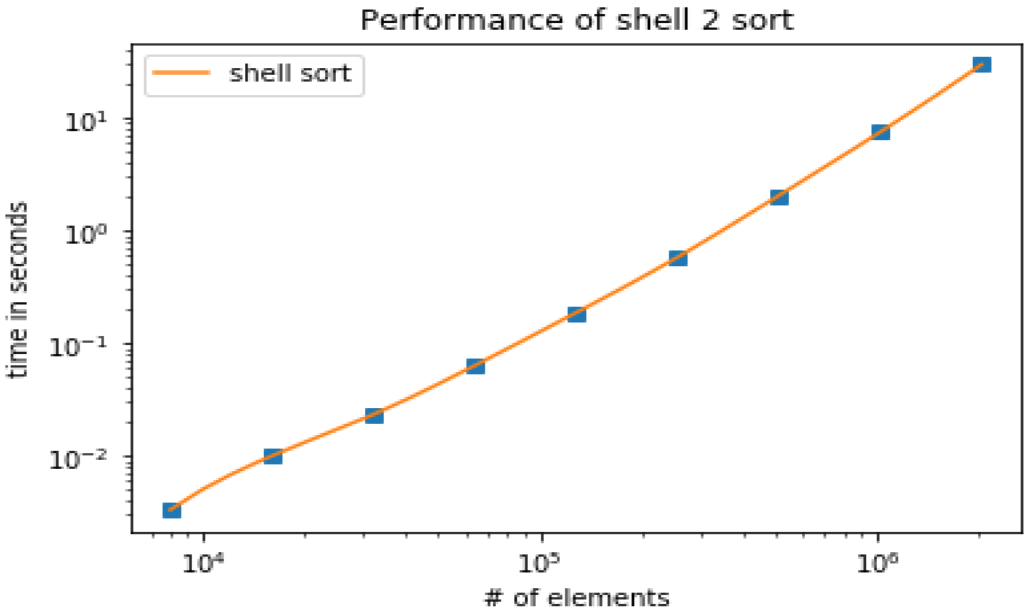
Implementation 2

Gaps: {511,255,127,63,31,15,7,3,1}

This Gaps come from the : Theorem (Papernov-Stasevich, 1965; Pratt, 1971). The theory running time of the shell sort was O(N^3/2).

Random Numbers:

funcname	n	seconds
shell_sort	8000	0.003333
shell_sort	16000	0.01
shell_sort	32000	0.023333
shell_sort	64000	0.063333
shell_sort	128000	0.186667
shell_sort	256000	0.58
shell_sort	512000	2.04667
shell_sort	1024000	7.51
shell_sort	2048000	29.86



M = log(0.01/0.0033)/log(16000/8000) , 2>M>1;

M = log(0.023/0.01)/log(32000/16000), >2M>1 ;

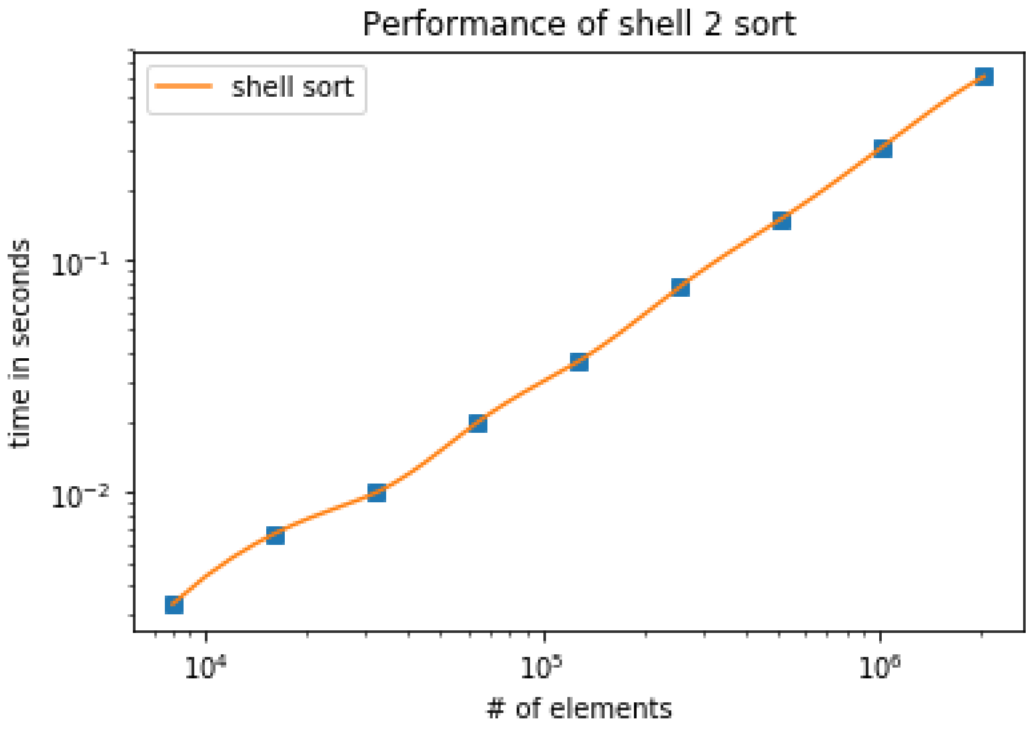
M = log(0.0633/0.023)/log(64000/32000), 2>M>1;

M = log(0.1866 /0.063)/log(64000/32000), 2>M>1;

The M was larger than the M of the implementation1, and these data can show the running time was larger than the O(N^4/3), and it could be O(N^3/2).

Almost Sorted Numbers:

	A	B	C
1	funcname	n	seconds
2	shell_sort	8000	0.003333
3	shell_sort	16000	0.006667
4	shell_sort	32000	0.01
5	shell_sort	64000	0.02
6	shell_sort	128000	0.036667
7	shell_sort	256000	0.076667
8	shell_sort	512000	0.15
9	shell_sort	1024000	0.306667
10	shell_sort	2048000	0.613333



The collect data and the graph were very similar as the graph and data of the graph from random numbers. The running time of the random and almost sorted numbers was same. It was O(N^3/2).

Annealing sort Algorithm

First implementation:

Temp: {1000,500,250,125,65,33,17,9,5,4,2,1}

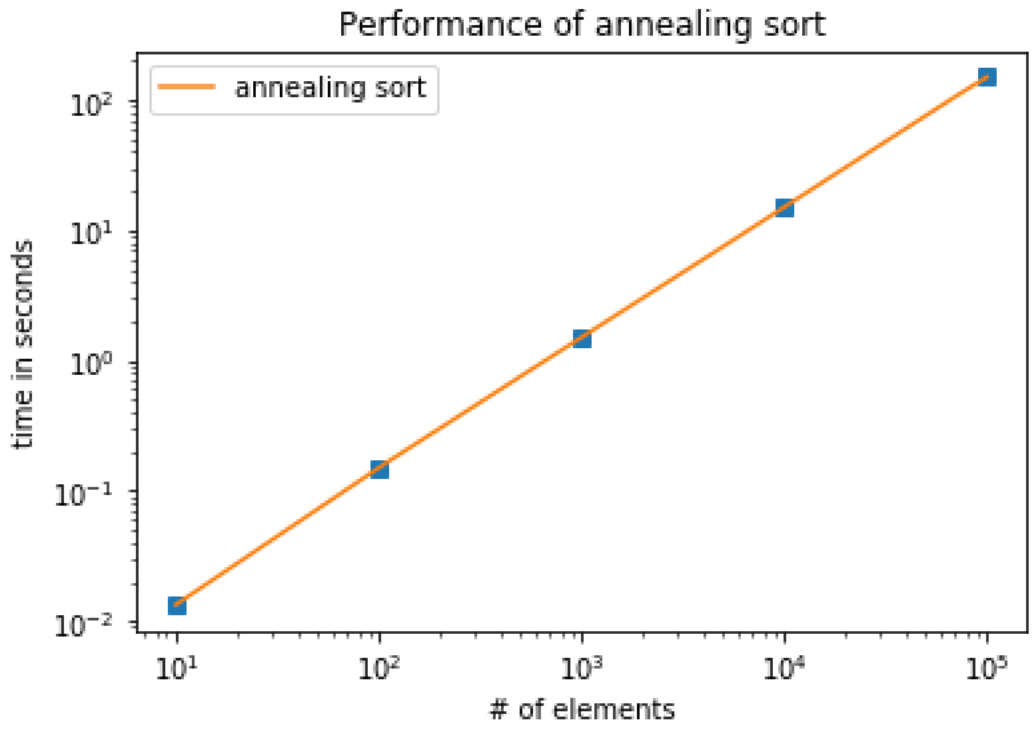
Reps: {2,2,2,2,2,2,2,2,2,2,2,2}

The temps could be varied with the length of the random number. For my testing, the largest temp of the temp needed equal to the size of the random numbers. For example, if the size of random number was 100000, the highest temp should be 100000(2n), and the rest of the temps should follow the n, n/2 ... qlog6n. The reps size should as same as the temps.

The temp sequence and reps came from the paper of Dr. Goodrich. The phase. “For this phase, let T1 = (2n,2n,n,n,n/2,n/2,n/4,n/4…, qlog6n,qlog6n)… let R = (c,c,…,c) be equal-length repetition.”

Random Numbers:

	A	B	C
1		n	seconds
2	annealing_sort	10	0.0133333
3	annealing_sort	100	0.15
4	annealing_sort	1000	1.5
5	annealing_sort	10000	15.0133
6	annealing_sort	100000	150.24



With using 10 power, my log was 10 base log.

We could find the graph looks like g(n) = nlgn graph.

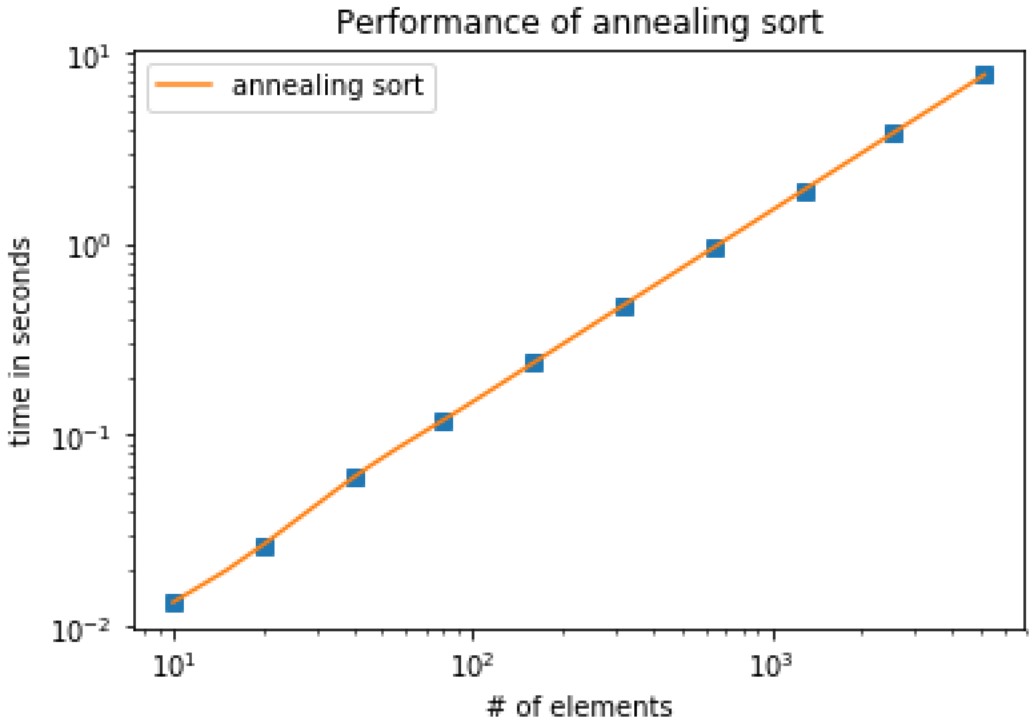
Based on my calculation, I got the m = 1.15 which is nearly 1. However, due to the logN so the m larger than 1.

From the data that I collected, we noticed the growth rate from the first plot to the second plot was 1.15 instead of 1.

The real world running time of the annealing sort algorithm could be O(NlgN)

Almost Sorted Numbers:

funcname	n	seconds
anealing_sort	10	0.013333
anealing_sort	20	0.026667
anealing_sort	40	0.06
anealing_sort	80	0.12
anealing_sort	160	0.24
anealing_sort	320	0.48
anealing_sort	640	0.963333
anealing_sort	1280	1.92667
anealing_sort	2560	3.85333
anealing_sort	5120	7.71



I used the power of 2, and the base of log was 2.

M = log(0.026 /0.0133)/log(20/10)=1;

M = log(7.71/3.85)/log(5120/2560)=1;

The M was 1, but the increase number was larger than 1. That could be a sign of the running time was logN. The running time was O(NlogN).

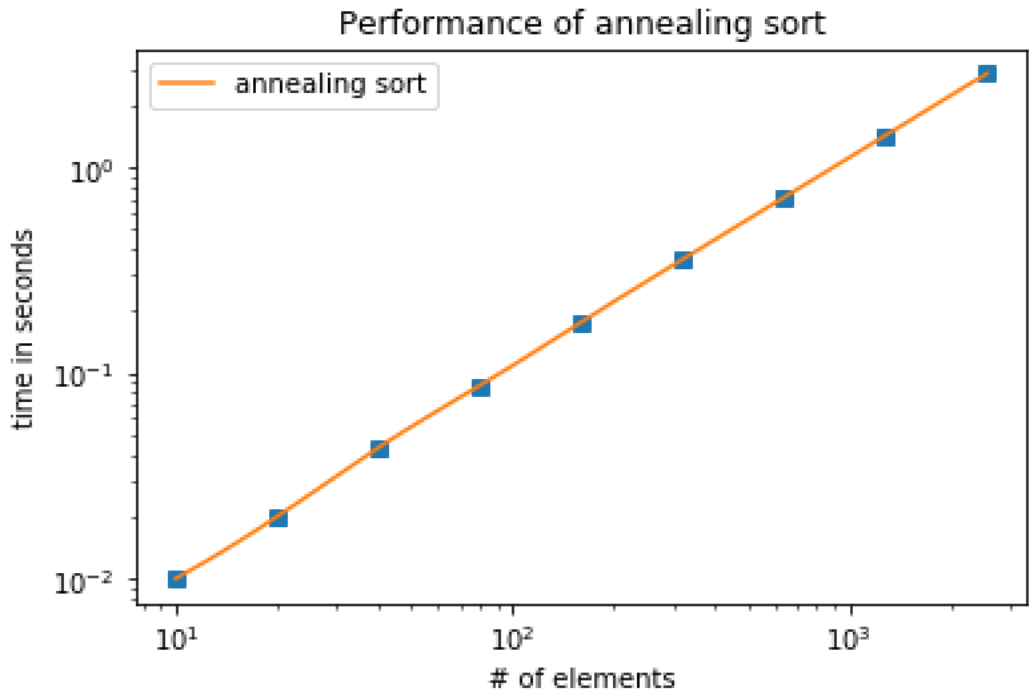
Second implementation.

Temp: {1024,512,256,128,64,32,16,8,1}

Reps: {2,2,2,2,2,2,2,2,2,2,2,2,2}

Random Numebers:

	A	B	C
1	funcname	n	seconds
2	annealing	10	0.01
3	annealing	20	0.02
4	annealing	40	0.043333
5	annealing	80	0.086667
6	annealing	160	0.176667
7	annealing	320	0.356667
8	annealing	640	0.71
9	annealing	1280	1.42667
10	annealing	2560	2.85



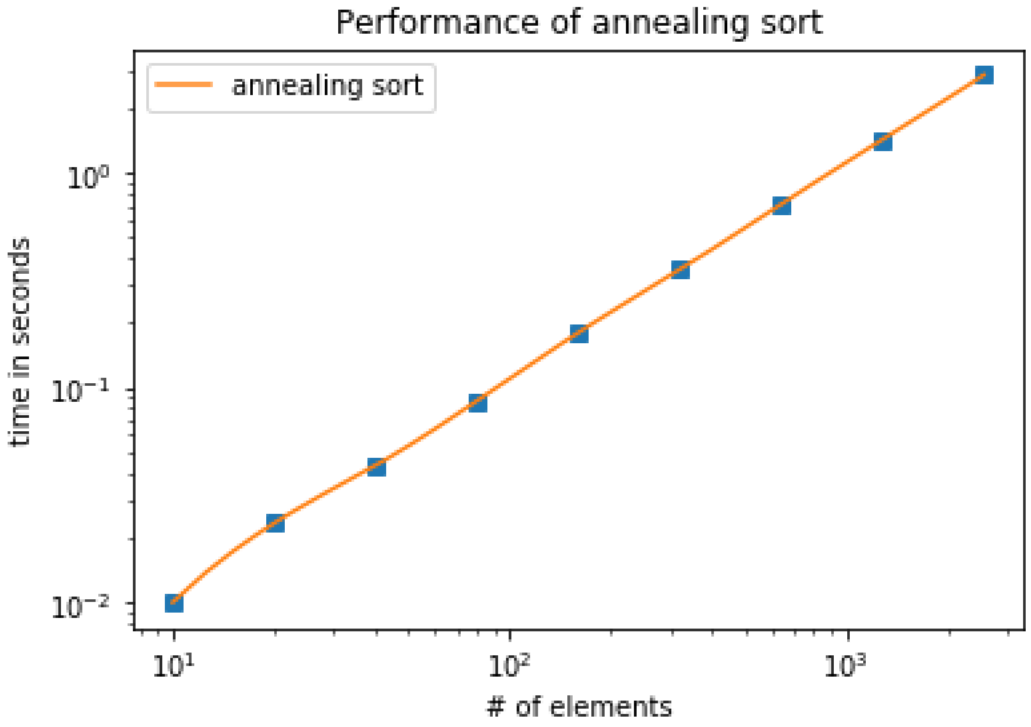
We could find the graph looks like g(n) = nlgn graph.

From the data that I collected, we noticed the growth rate from the first plot to the second plot was 1.15 instead of 1.

The real world running time of the annealing sort algorithm could be O(NlgN)

Almost Sorted Numbers:

	A	B	C
1	funcname	n	seconds
2	anealing_	10	0.01
3	anealing_	20	0.023333
4	anealing_	40	0.043333
5	anealing_	80	0.086667
6	anealing_	160	0.18
7	anealing_	320	0.353333
8	anealing_	640	0.713333
9	anealing_	1280	1.42667
10	anealing_	2560	2.85333



The graph and the data were as similar as the graph and number of the random numbers.

Conclusion:

The Bubble-sort and Spin-the-bottle sort could be very expensive to be used in sorting random and almost sorted numbers. Although the running time of Insertion-sort in sorting random numbers was O(N^2), the Insertion-sort could sort almost sorted numbers in very short times, O(N). It will be the best choice to be used sorting almost sorted numbers.

The running time of Shell sort could be varied with the gaps. Dr. Sedgewick improved the running time of Shell sort to O(N^4/3).

Using the good temps and reps, we could improve the running time of Annealing sort to O(NlogN). It will be very fast.