Xiaohu Zheng

Sorting Algorithm analysis

7/10/2021

Encountered problems:

1. Should I use different arrays for different sorting algorithms, or should I test the same array for different sorting algorithms?

In order to test the efficiency of an algorithm, we should use random numbers or arrays. However, if we want to compare the efficiency for different algorithms, the array to be tested for these algorithms should be the same. And we need to generate random numbers for this array. I realized this problem after two runs. So, I changed the code to make sure the arrays with the same size need to be the same. In other words, for different sorting algorithms, they will test the same array with the same elements. (I made this mistake and think this is a very important point to point out before we analyze the data)

Below is the data for two runs with array of different elements.

Not the same array for the same size N!!

Time chart

N	Insertion sort	Merge Sort	Heap Sort	Quick sort	Quick Sort(Rand)	Radix Sort
10(nano)	10200	26500	4700	2800	4200	30500
100(nano)	102400	312900	82800	34100	42700	88500
1000(nano)	9956600	2766100	1249700	561600	625200	734700
10000(nano)	1566555500	27358200	13465700	6662500	6247800	6044600
100000(nano)	482562196	309352100	192974000	128173500	155178900	56108200
1000000(milli)	Unknown	3341.75	2196.46	7408.68	133.968	582.105

Time chart

N	Insertion sort	Merge Sort	Heap Sort	Quick sort	Quick Sort(Rand)	Radix Sort
10(nano)	24400	30000	4400	2700	3500	26800
100(nano)	184500	271000	127600	36800	32700	112100
1000(nano)	16115400	2866400	1140100	554500	461900	623100
10000(nano)	1076259500	30549700	15219300	7152000	6931400	6292300
100000(nano)	770993676	319299300	178631300	126390600	127910900	61564300
1000000(milli)	Unknown	3579.07	2334.71	7807.48	138.004	619.74

We can see at least two problems in the shown pictures.

1. The output for insertion sort with size of N = 1000000 is "Unknown". I waited for more than 30mins for it to finish, but it is really slow and will not stop running. After several tries, it didn't give any results. I marked the running time as unknown, but it is not really "Unknown", it is just too slow. We don't have to wait for it to finish to get an exact data because we already knew the reason why it is slow. The reason is that insertion sort is an $O(n^2)$ algorithm, with the size of N increases, the running time of the insertion sort algorithm grows exponentially. 1 million is already a huge number of inputs, so it is expected that the insertion sort algorithm to be really slow with this size of input. Even though we don't get an

exact data for the last input for insertion sort algorithm, it will not affect us to analyze the data and to compare the sorting algorithms later.

2. Notice the when the input size N = 100000, the running time of insertion sort are 482562196, 770993676. They are smaller than the running time when the input size N is 10000, which is unusual. We should expect the running time to be larger when the input size of N in bigger. This happened because the running time is too huge that exceed the maximum value that it can hold, so we get some wrong numbers. If we test a couple of more times, we sometimes get negative numbers too, this is the same reason above. In the picture, we can see one way to fix the problem is to use milliseconds. Because when the input number N is 1000000, for other sorting algorithms, we will also get wrong results. I have already used milliseconds for N = 1000000.

3. Another problem we can not directly see in the data is what I have mentioned in the beginning of report. I think the data is not too accurate because we didn't test the same array for the same input size N. For example, when N = 10, we should expect the performance of the insertion sort is better than other sorting algorithms, but in the result is really slow.

We see the result of improved code, below are the 3 runs with arrays with same elements

For the same size of N, the tested array for all algorithms are the same!

Time chart

N	Insertion sort	Merge Sort	Heap Sort	Quick sort	Quick Sort(Rand)	Radix Sort
10(nano)	3900	45800	4400	3100	4100	27800
100(nano)	170800	292900	67900	40500	32900	82600
1000(nano)	14272100	2765700	1058300	592200	558200	531700
10000(nano)	1615126100	28438000	15148900	7568100	6242400	5824000
100000(nano)	156787.943300(milli)	321766000	188821900	140817100	139542800	62471100
1000000(milli)	Unknown	3541.78	2335.66	8100.12	133.94	601.899

Time chart

N	Insertion sort	Merge Sort	Heap Sort	Quick sort	Quick Sort(Rand)	Radix Sort
10(nano)	2600	30300	5200	3100	3900	28000
100(nano)	226200	290500	82800	43800	35100	85000
1000(nano)	14042400	2835900	1293900	658400	617300	642100
10000(nano)	1882707600	40356200	15333300	7438700	6784400	6250200
100000(nano)	209936.962700(milli)	356526500	186897200	134074200	153080300	60542000
1000000(milli)	Unknown	3610.36	2357.36	8357.13	144.444	641.761

Time chart

N	Insertion sort	Merge Sort	Heap Sort	Quick sort	Quick Sort(Rand)	Radix Sort
10(nano)	2400	28600	5100	2800	3300	47200
100(nano)	140600	299000	84200	39500	44100	74600
1000(nano)	13671700	2976700	1287200	511300	566300	750300
10000(nano)	1738271900	29921100	14934700	6700900	6862800	6280200
100000(nano)	125481.664200(milli)	327312500	185182800	139778800	129418300	60669700
1000000(milli)	Unknown	3596.1	2363.98	8189.15	137.034	621.065

I randomly picked 3 pictures, they look similar, so we know the results are good for analyzing!

Analyzing the data for each sorting algorithms!

The stable below means average running time and worst case running time! It means the performance of a sorting algorithm, not the relative orders!

1. Insertion sort.

Insertion sort is the fastest algorithm when N = 10, this is what we expected because we know when the size of array is small and the array is almost sorted, insertion sort can sort the array in a very short amount of time. However, it is still a $O(n^2)$ sorting algorithm. It grows exponentially, as we can see when N >= 100, it is almost slower than all the other sorting algorithms listed above.

Advantage: good when the array size is small, and when the array is already sorted.

Disadvantage: O(n^2) sorting algorithm, thus very bad performance when n is huge.

2. Merge sort.

Merge sort uses divide and conquer strategy, it is the first sorting algorithm that runs at O(nlogn) time, be more accurate, it runs $\theta(nlogn)$ time. We don't see any better when the input size N is small like 10 or 100, but when N gets to 1000 or even larger, we can see the time complexity is improved. It is better than insertion sort, which is a huge improvement. Why when N is small, the performance of merge sort doesn't look so well? It is because of the Merge operation, we need to copy the data into a buffer array first, then copy the data from the buffer array back to the original array, which takes times. So in the picture, we see it is slower than insertion sort when N is small.

Advantage: first O(nlogn) sorting algorithm, it is good to use when input number is large.

Disadvantage: need to use O(n) extra space for merging, comparing to other O(nlogn) sorting algorithm like quick sort, the space complexity is bad.

3. Heap sort

Heap sort uses special tree structure array to sort. It is also a $\theta(nlogn)$ sorting algorithm. From the data we can see, except for small size of N, its performance is really good, and it does not use any extra space.

Advantage: O(nlogn) sorting algorithm, which is faster than merge sort. And comparing to the normal quick sort algorithm, the performance is really "stable". Unlike quick sort, the worst case can hit O(n^2). The worst case for heap sort is just O(nlogn). And it does not need any extra space.

Disadvantage: Real time performance is worse than quick sort. Need to maintain its tree structure, need to rebuild after its (insert, delete, update) operation, which is unnecessary.

4. Quick sort

Quick sort also uses divide and conquer strategy by choosing pivot point and divide the array. The real time performance for quick sort is really good, as we can see in the chart, it is faster than all the other algorithms. (Random pivot point not counted since it is also quick sort)

Advantage: Like its name, really fast. In place algorithm, so it does not need any extra space.

Disadvantage: The average running time can be O(nlogn), however, it is not "stable" that sometimes its worst time running time can hit $O(n^2)$, which is really bad. The reason is that if the right value we chose is always the maximum value for the subarray, it does not improved anything.

5. Quick sort(random pivot point)

Because the disadvantage introduced above, we can use a new way of choosing our pivot point. We will use a random number in the array to be our pivot point instead of just use the right value. This small change can make a huge difference. It can most likely guarantee that the sorting algorithm will be O(nlogn). As we see in the chart when N = 1000000, the quick sort with random pivot point is nearly 70 times faster than quick sort.

Advantage: fastest! No extra space needed! "Stable"!

Disadvantage: Not stable (the relative order of the elements will be changed).

6. Radix sort

It is an O(n) time sorting algorithm. It needs to iterate the array 4 times and needs extra space. Therefore, for small number of N <= 10000. Its performance is not improved too much. However, it is still an O(n) sorting algorithm which grows linearly, so when N > 10000, we can see its running time can be very fast like quick sort.

Advantage: O(n) time sorting algorithm, fast with big input size.

Disadvantage: It needs a lot of extra space.

In conclusion, from the above data we know that to best sort an array. When the input size is big, we should use quick sort with random pivot point, and when the input size is small, we can just use insertion sort algorithm.s