Algorithm Assignment 2

20161662 허남규.

Table of Contents

Overview

In this assignment, I look analyze various implementations of integer sorting algorithms to understand performance, time-complexity. I also experiment with various methods to design a very efficient sorting algorithm.

Averaged Running Time Measurements

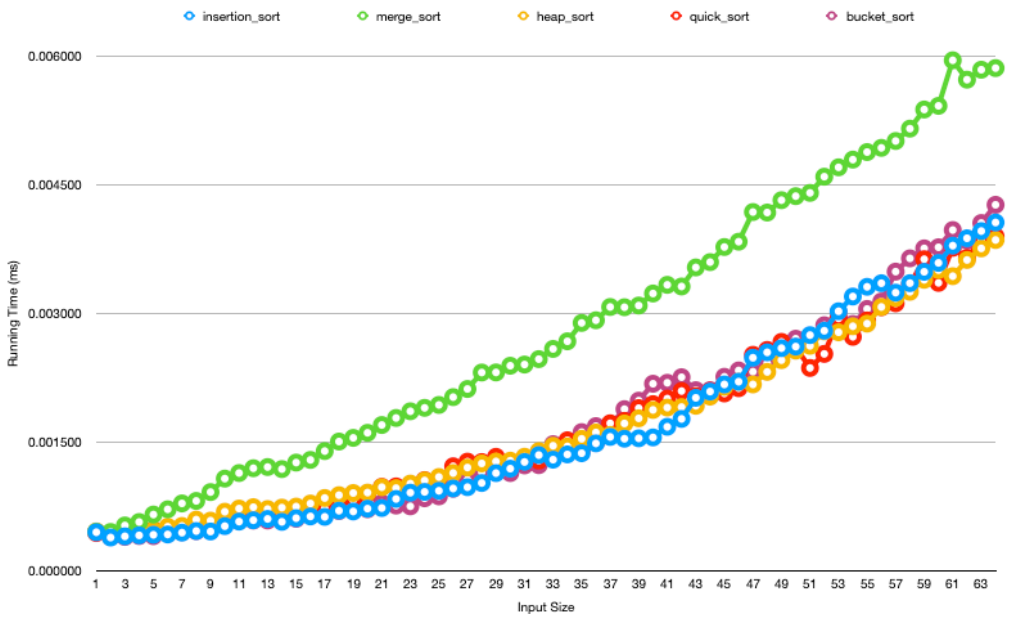
All running times are averages of multiple repeated runs. The number of iterations depends on the input size, ranging from 10,000 for small inputs with less than 32 elements, down to 1 for inputs larger than 2^15. **All running times are in milliseconds.**

Performance Comparisons

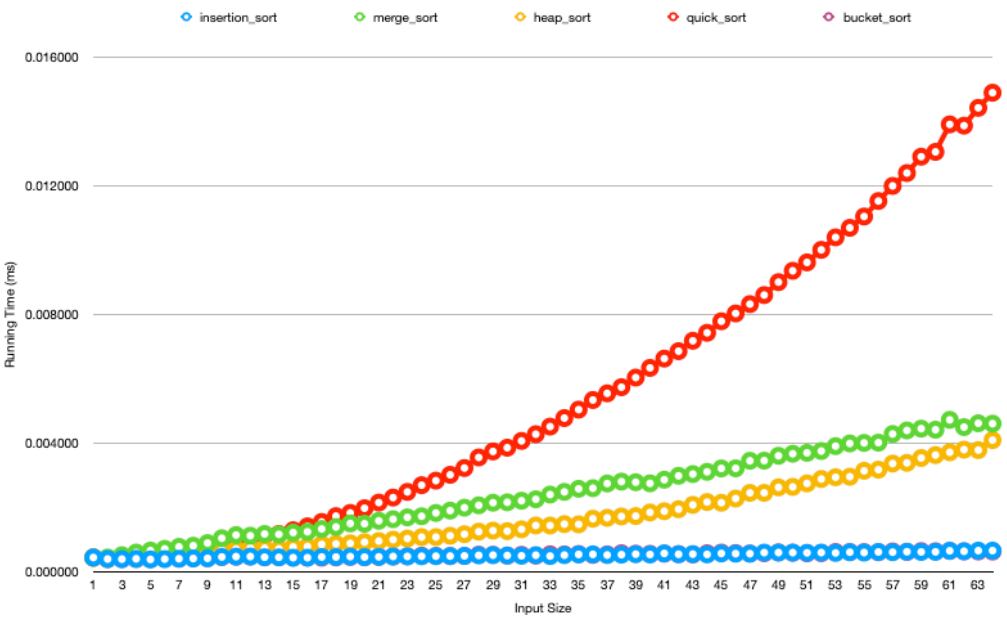
Here are the comparisons between unoptimized versions of 4 major sorts: insertion sort, merge sort, heap sort (not included in the final executable file), quick sort, and an optimized radix sort.

Linear Order

Random Order Case



Descending Order Case



Logarithmic Order

Random Order Case

N	Insertion	Merge	Heap	Quick	Radix
1024	0.740	0.137	0.109	0.110	0.744
2048	3.026	0.274	0.248	0.240	2.993
4096	12.070	0.588	0.523	0.551	1.169
8192	47.837	1.239	1.134	1.115	1.780
16384	191.436	2.647	2.481	2.453	2.849
32768	760.922	5.523	5.221	5.155	4.712
65536		11.466	11.151	10.843	8.500
131072		24.088	24.240	23.943	15.981
262144		50.480	52.991	49.043	30.536
524288		106.095	115.237	102.910	60.697
1048576		219.312	246.405	215.891	118.372

Descending Order Case

N	Insertion	Merge	Heap	Quick	Radix
1024	0.004	0.081	0.094	3.214	0.004
2048	0.008	0.159	0.200	12.709	0.008

4096	0.015	0.334	0.415	50.651	1.187
8192	0.030	0.688	0.873	203.407	1.716
16384	0.061	1.418	1.859	809.882	2.912
32768	0.138	3.067	3.812	3263.834	4.684
65536		5.993	7.849		8.716
131072		12.695	16.507		15.937
262144		25.770	34.324		31.072
524288		53.492	71.609		59.766
1048576		110.195	148.713		117.922

*Note that I omitted running times for $O(N^2)$ cases, as they would require too much time.

Observations

I have found that the algorithms' running time follow their asymptotic time complexities accordingly with minor discrepancies in the descending order case. However, insertion sort shows a very fast linear time complexity in the descending order case.

Optimized Algorithm (#4)

I take 2 approaches to this task. First, I employ common methods to optimize quick-sort as best I can, such as early stopping and median pivot selection. I also introduce a new heuristic method where I use a moving-average-like method for pivot selection. Secondly, I use a highly optimized implementation of radix-sort, as these usually show high performance for limited domain data such as integers.

After extensive optimization, I was not able improve the quick-sort method to perform better than the radix-sort approach. I share the details below.

Experiment Environment

CPU	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (10 cores) (shared)
Cache	25MB *(Intel Smart Cache)*
RAM	64GB (shared)
OS	Ubuntu 16.04.2 LTS, GNU/Linux 4.4.0-130-generic

Experiment Details

Input Range	-2,147,483,648 ~ 2,147,483,647 (randomly distributed)
Input Size	2^20 = 1,048,576
Input Ordering	Random, Inverse Order
Metric	Runtime

Approach 1. Optimized Quick Sort

To optimize the performance of my quick-sort based method, I introduced 5 techniques as follows:

A. Median of 3 pivot selection: selects the median of beginning, middle and ending values to use as the pivot.
B. Random pivot selection: uses a random indexed element for the pivot.
C. Pre-local sorting: pre-sort the list by inverting all local decreasing sequences.
D. Shifted mean pivot selection

   This technique takes advantage of the fact that most data is distributed evenly. Initially, I take the **mean of the max/min values of the entire list to find a quasi-median value (QMED)**. This takes $O(N)$ time. At the next recursive level, instead of repeating this min/max search, I simply reuse the min/QMED/max

values from the previous level. The QMED values of the next two lists can be expressed as the following equations.

$$QMED_{i+1,l} = \frac{MIN_i + QMED_i}{2}$$
$$QMED_{i+1,r} = \frac{QMED_i + MAX_i}{2}$$

Note that $QMED_i$ does not exactly correspond to the max/min values of the left/right lists that follow. Because of this, the QMED will continue to deviate from the real median element. To mitigate this issue, **I continuously adjust the QMED by shifting it closer to the median of three (MED_3)** mentioned in method A. This can be expressed in the following equation.

$$QMED_{adjusted} = MED\_3 * \alpha + QMED * (1 - \alpha)$$

where $\alpha \in [0,1]$ indicates how strongly I adjust the QMED using MED_3.

When I detect that the QMED has deviated too far, I re-initialize it with the true min-max-mean value.

This is similar to using the **moving average of MED_3, with a semi-accurate starting value.**

E. Early stopping: uses insertion sort to sub-lists smaller than some threshold number N_{stop}.

Logarithmic Time Comparison of Quick Sorts (Random Order)

N	Baseline	Optimizations					C++ Library
		A	B	BC	D	DE	
1024	0.1096	0.1204	0.1184	0.1280	0.1228	0.0932	0.1374
2048	0.2402	0.2682	0.2678	0.2700	0.2686	0.2220	0.2990
4096	0.5510	0.5732	0.5748	0.6116	0.5756	0.4744	0.6218
8192	1.1150	1.2658	1.2766	1.2790	1.2104	1.0148	1.3454
16384	2.4532	2.7306	2.6882	2.8102	2.7332	2.1386	2.8316
32768	5.1550	5.5820	5.8070	5.7420	5.4240	4.5950	5.9850
65536	10.8430	11.7190	11.6630	12.1890	11.5960	10.0590	12.6740
131072	23.9430	24.5960	25.7560	26.9660	24.3790	21.2280	27.0100
262144	49.0430	51.7670	52.5380	54.4290	50.7600	46.5520	58.0360
524288	102.9100	108.2250	109.9380	115.9520	106.9430	93.6560	121.3560
1048576	215.8910	226.9450	234.0730	245.6820	219.5030	198.2350	253.2800

Logarithmic Time Comparison of Quick Sorts (Descending Order)

N	Baseline	Optimizations					C++ Library
		A	B	BC	D	DE	
1024	3.2142	0.1174	0.0850	0.0940	0.0866	0.0452	0.0514
2048	12.7094	0.1970	0.1924	0.1950	0.1896	0.1048	0.1168
4096	50.6506	0.4498	0.3948	0.3992	0.3910	0.2344	0.2468
8192	203.4070	1.0830	0.8346	0.8774	0.8252	0.4826	0.5214
16384	809.8820	2.2640	1.8148	1.8392	1.7172	1.0792	1.0860
32768	3263.8340	4.5580	3.7300	3.9770	3.4520	2.2040	2.3600
65536	0.0000	9.7810	8.5970	7.9850	7.1710	4.8070	4.8470
131072	0.0000	22.1270	16.6130	16.9880	14.8620	10.2290	10.4270
262144	0.0000	40.5210	35.8650	36.6620	30.0840	21.7780	21.9950
524288	0.0000	85.8450	70.3660	74.7610	61.0680	46.0930	46.2040
1048576	0.0000	202.8950	150.3670	155.6530	130.3860	104.7290	96.7260

*Note that I omitted running times for baseline quick-sort on larger descending data, as they would require too much time.

*I used an empirically tested optimal early stopping size of $N_{stop} = 23$

*For the shifted mean optimization, I used an α value of 0.25.

Observations

- Median of 3 introduces additional overhead, which degrades performance in the random case, but practically bounds the algorithm to $O(nlogn)$ in the worst case.
- Pre-local sorting has the same effect, but does not improve performance further, as pivot selection in quick-sort does not take advantages of locally sorted chunks.
- The new technique that I introduced—the shifting quasi-median technique—increases performance significantly. It provides a 20% boost in performance compared to the sorting function provided by the standard library and nears its performance in the worst case.

Approach 2. Radix Sort

Despite extensive optimizations in quick-sort mentioned above, I have found that an optimized implementation of radix-sort trumps all other methods.

As the optimizations only pertain to language specific tweaks instead of algorithmic changes, I will not explain further.

One variable I must consider is the size of the base. To utilize faster bitwise operations, I only considered base sizes in the form 2^{base}. From empirical analysis, I have found that a value of $base = 8$ is optimal. Here are the running times compared to other methods.

Logarithmic Time Comparisons (Random Order)

N	C++ Library	Merge Sort	Heap Sort	Radix Sort	Quick Sort
1024	0.137	0.137	0.109	0.744	0.093
2048	0.299	0.274	0.248	2.993	0.222
4096	0.622	0.588	0.523	1.169	0.474
8192	1.345	1.239	1.134	1.780	1.015
16384	2.832	2.647	2.481	2.849	2.139
32768	5.985	5.523	5.221	4.712	4.595
65536	12.674	11.466	11.151	8.500	10.059

131072	27.010	24.088	24.240	15.981	21.228
262144	58.036	50.480	52.991	30.536	46.552
524288	121.356	106.095	115.237	60.697	93.656
1048576	253.280	219.312	246.405	118.372	198.235

Logarithmic Time Comparisons (Descending Order)

N	C++ Library	Merge Sort	Heap Sort	Radix Sort	Quick Sort
1024	0.051	0.081	0.094	0.004	0.045
2048	0.117	0.159	0.200	0.008	0.105
4096	0.247	0.334	0.415	1.187	0.234
8192	0.521	0.688	0.873	1.716	0.483
16384	1.086	1.418	1.859	2.912	1.079
32768	2.360	3.067	3.812	4.684	2.204
65536	4.847	5.993	7.849	8.716	4.807
131072	10.427	12.695	16.507	15.937	10.229
262144	21.995	25.770	34.324	31.072	21.778
524288	46.204	53.492	71.609	59.766	46.093
1048576	96.726	110.195	148.713	117.922	104.729

Observations

Though radix-sort is sub-optimal for reverse-ordered data, it absolutely trumps other methods in the random case. It provides a 214% speed increase compared to the standard library sort.

Code Design & Documentation

Through a modularized design I were able to maintain a very clean codebase. All sorting functions follow the same signature in the following form to allow for code reuse.

```
typedef sort_func(int *array, int begin, int end);
```

I also utilize C++'s OOP features to create a Data class that encapsulates core tasks such as data generation, saving, loading, copying, comparing etc.

Documentation

To maximize readability, I focused on using appropriate variable names and dividing up tasks into functions. Consider these names as a substitute for comments. **Actual comments were used sparingly, only in situations where it helped readability.**