Subject4: Implement and Analysis of Algorithms

Group: 47 **Members:** Shuting Guo(3170104871),Fan Ye(3170102410)

Instructors: Yin Zhang, Shenjing Tian, Haihui Xiong

Project1: Implement and Analysis of Common Sorting Algorithms

In this project, we tried 8 different algorithms, which are insertion-sort, bubble-sort, selection-sort, counting-sort, quick-sort, merge-sort, heap-sort and intro-sort. Here give a table to compare these good algorithms.

	Best Time Complexity	Worst Time Complexity	Average Time Complexity	Stability	Advantage	Disadvantage
Insertion sort	O(n)	O(n^2)	Θ(n^2)	T	Quick in small test cases	O(n^2) is not acceptable when n is big
Bubble sort	O(n^2)	O(n^2)	Θ(n^2)	T	Easy to write	O(n^2) is not acceptable when n is big
Selection sort	O(n^2)	O(n^2)	Θ(n^2)	F	Easy to understand	O(n^2) is not acceptable when n is big
Counting sort	O(n)	O(n)	O(n)	T	The fastest algorithm when sorting bounded integers	Needs too much space, is not suit for abstract elements
Quick sort	O(nlogn)	O(n^2)	O(nlogn)	F	Quick	Time complexity can worsen to O(n^2) when data is special
Merge sort	O(nlogn)	O(nlogn)	Θ(nlogn)	T	Stable and quick	Needs O(n) extra space
Heap sort	O(nlogn)	O(nlogn)	Θ(nlogn)	F	Time complexity is stable and quick	Coding complexity is higher, not quick as quick-sort
Intro sort	O(nlogn)	O(nlogn)	Θ(nlogn)	F	The complexity will not worsen as quick-sort and even faster in big test cases	Coding complexity is very high
Radix sort	O(nlogk)	O(nlogk)	Θ(nlogk)	T	When integers are small, it can sort quickly	Cost a large sum of memory

After writing codes of these algorithms, we made a test for them.

Here goes the result (***<u>seconds, running 100 times</u>***):

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Algorithm / testcases	**[3,1,10]**	**[7,1,100]**	**[8,1,100]**	**[10,1,100]**	**[50,1,100]**	**[100,1,100]**	**[1e3,1,1e4]**	**[1e4,1,1e4]**	**[1e5,1,1e5]**	**[1e6,1,1e6]**	**[1e4,1,1]**	**[1e4,1,2]**	**[1e6,1,100]**

Bubble sort	3.4e-05	3.8e-05	3.9e-05	6e-05	0.000152	0.000741	0.081903	12.9937	N/A	N/A	6.00655	12.2812	N/A
Counting sort	0.000346	0.000283	0.000358	0.000266	0.000297	0.00035	0.003551	0.009406	0.106647	1.71637	0.002507	0.002031	0.132891
Heap sort	5.2e-05	3.8e-05	4.4e-05	3.9e-05	9.1e-05	0.000136	0.005266	0.076338	1.07909	15.251	0.003955	0.031763	9.12897
Insertion sort	6.8e-05	4.9e-05	3.4e-05	3.5e-05	9.4e-05	0.00022	0.016319	1.7313	N/A	N/A	0.001183	0.85815	N/A
Merge sort	4.5e-05	9.2e-05	6.2e-05	9.8e-05	0.000155	0.000382	0.005251	0.087687	1.00966	11.6311	0.020728	0.029245	7.54492
Quick sort	9.2e-05	4.1e-05	3.7e-05	4.2e-05	7.3e-05	0.000185	0.003582	0.066211	0.827287	9.33422	3.41382	1.69959	345.842
Selection sort	4.7e-05	3.8e-05	3.9e-05	4e-05	0.000127	0.000881	0.038085	3.85988	N/A	N/A	3.42656	3.38535	N/A
Radix sort	0.000123	0.000133	0.000199	0.000172	0.00068	0.003483	0.01631	0.151731	1.32204	12.8391	0.127762	0.13206	14.3869
STL sort	3.4e-05	5.5e-05	3.6e-05	3.9e-05	6.1e-05	9.1e-05	0.002389	0.057353	0.643091	7.82551	0.007178	0.014152	3.8654
Intro sort	3.3e-05	3.7e-05	6e-05	4.2e-05	5.8e-05	0.000165	0.002325	0.06138	0.708481	8.77974	0.008467	0.014123	4.29275

Two specially organized test cases:

Test14:
[100000,1000000,1100000]+[100000,900000,1000000]+[100000,800000,900000]+[100000,700000,800000]+[100000,600000,700000]+[100000,500000,600000]+[100000,400000,500000]+[100000,300000,400000]+[100000,200000,300000]+[100000,100000,200000]

Bubble sort	Counting sort	Heap sort	Insertion sort	Merge sort	Quick sort	Selection sort	Radix sort	STL sort	Intro sort
N/A	Error	14.7471	N/A	11.2109	9.46302	N/A	13.0001	8.04907	8.65674

Test15:
[100000,100000,200000]+[100000,200000,300000]+[100000,300000,400000]+[100000,400000,500000]+[100000,500000,600000]+[100000,600000,700000]+[100000,700000,800000]+[100000,800000,900000]+[100000,900000,1000000]+ [100000,1000000,1100000]

Bubble sort	Counting sort	Heap sort	Insertion sort	Merge sort	Quick sort	Selection sort	Radix sort	STL sort	Intro sort
N/A	Error	14.5477	N/A	10.9559	8.80519	N/A	12.9844	7.67523	8.45927

Note: 1. [n,l,r] means that test case contains n random integers between l and r.

2. Optimize option -O2 was active.

We can see that, in some special testcases such as testcase 11,12,13, the unoptimized quick-sort is as slow as the O(n^2) sorting algorithms.

After analyzing the data above, we found that heap-sort and merge-sort are usually 1.2-2 times slower than quick-sort but the time cost still grows by O(nlogn).

The ***Intro-sort*** is an algorithm which combines quick-sort, heap-sort and insertion-sort together. At the beginning, this algorithm uses quick-sort to sort the array, when the recursion depth grows too much, the algorithm detects it and change to use heap-sort to sort the rest elements. When the length of the range becomes short, it turns to use insertion-sort when O(n^2) is faster than O(nlogn). And the optimized quick-sort can run faster.

We have tried many ways to optimize the speed of the code. And spent a lot of time reading STL files. However, our code is still a little bit slower than std::sort function, but has made a great improvement compared with the other algorithms.

And we found an interesting skill from the STL: when we try to calculate middle value of two integers, we write as:

```
Mid=(l+r)>>1;
```

But this has a problem, that is, if l and r are both very big may be 2e9, then 2e9+2e9 will cause error of integer exceeding. Therefore, we should write this way:

```
Mid=l+((r-l)>>1);
```

This will make you avoid exceeding easyly.

Project2: Friend Filter

First, we read info into an array. And get tree copy of the info. Sort three arrays, one by hash value of name, one by height and the other one by weight. Find info by dichotomy.

Following operations are provided:

f name : find a person by name

qh l r : query l-th to r-th tall person

qw l r : query l-th to r-th heavy person

gh x y : find all persons whose height is between x and y

gw x y : find all persons whose weight is between x and y

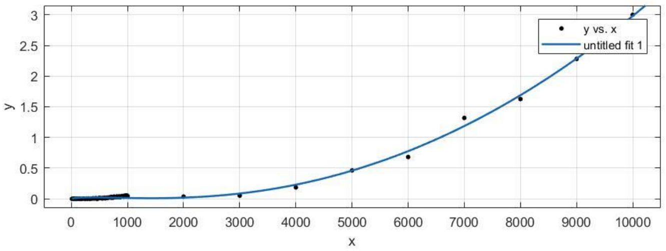
Note: For operation qh and qw, for example, in series [1,1,2,4,4,4,5,5,6], 1s are considered as the first, 4s are considered as the third etc..

To deal with the rank, we prepared a suffix-array, to store the ranks of each elements.

Because the array was sorted, all query operations are O(logn). So, the time complexity of initializing is O(nlogn+len), where len is the total length of names(Hash algorithm is O(len)), and the time complexity of each query if O(logn).

However, though the query operation is quick enough, when n becomes large, printing the result can cost a large sum of time. Printing all of the result cost O(m*n) time. Therefore, the total time complexity is O(mn+nlogn+n)=O(n^2).

We made a test and analyzed the time-cost with Matlab. As we can see, the time grows by O(n^2) with a small coefficient.



Fitting curve

```
Linear model:
    f(x) = a*x*x + b*x*log(x)/log(2) + c*x
Coefficients (with 95% confidence bounds):
     a = 5.274e-08 (5.031e-08, 5.516e-08)
     b = -5.766e-05(-6.406e-05, -5.126e-05)
     c = 0.0005368 (0.0004723, 0.0006013)
Goodness of fit:
  SSE: 0.06523  R-square: 0.9964
  Adjusted R-square: 0.9963  RMSE: 0.02469
```