Comparative Analysis of Numerical Sorting Algorithms in Java Language

Lingling Zhao [1, a], Xuemei Liu [1,b] and Xiufeng Shao [1,c]

[1] Beijing city University, Beijing, 100083, China

[a]zhaolingling@bcu.edu.cn,[b]xuemliu@163.com,[c]shaoxiufeng@bcu.edu.cn

Keywords: Internal sorting; Time complexity; Stability

Abstract. Sorting is one of the important algorithms in computer programming. The ordinary 6 sorting algorithms are analyzed and compare from the algorithmic time complexity and stability. The executive efficiency of 6 sorting algorithms is verified by Java program. The costing time and stability of 6 sorting algorithms are compared to provide certain reference for sorting algorithm.

Introduction

Sorting is a critical algorithm in the data processing and widely used in the real world. How to efficiently sort has become one of important issues in the PC application. In the sorting process, sorting is divided into internal and external sorting. For internal sorting, the files are processed in memories and no exchange across external and internal memories is involved in sorting. The external sorting refers to the sorting process where the data must be exchanged across the internal and external memories. The internal sorting is suitable to record few small tables, and external sorting is suitable to record a few big tables where the total records cannot be placed in the memories.

According to the different algorithms, the internal sorting is divided into 5 types: exchange sort, insertion sort, selection sort, merge sort and radix sort. 6 ordinary sorting algorithms are herein selected: bubble sort, quick sort, straight insertion sort, Shell sort, heap sort and merge sort to comparatively analyze the sorting algorithms through running results of Java program.

Sorting algorithm description and time complexity

Given that the record sequence to be sorted uses the sequential storage structure, i.e. a 1-dimensional array is used for storage, then n records to be sorted are placed in the array R[1]~R[n],and the sequence list structure is as below:

```
Typedef struct{
   int key;
}RcdType;
```

Bubble sort. Basic idea: the bubble sort belongs to an exchange sort. Firstly, to compare the first and second records, if at reverse sequence, that is, R[1].key>R[2].key, then both records are exchanged, thus the record with lesser keyword value "floats"(moves to left), the record with larger keyword value "drops"（moves to the right） as a stone; The second and third records are then compared, and so forth till the end of comparing the n-1 and n records. A round of bubble sort is finished like this. After the first round of sorting, the maximal record is exchanged to the n^{th} position; the second round of bubble sort is performed afterwards as the previous procedure of above n-1 record to exchange the record with larger keyword value to the n-1 position. The above procedure is repeated till there is no operation of record exchange during a round of sorting.

Time complexity: Provided that the 1^{st} traversal comparison times is n-1, the 2^{nd} traversal comparison times is n-2, and so forth, the i^{th} traversal comparison times is n-i, then the time complexity is expressed as:

$f(n)=(n-1)+(n-2)+...(n-i)+\ldots 0=(n-1)*n/2$

The time complexity of bubble sort is O (n^2) and the average time complexity is O (n^2).

Quick sort. The quick sort belongs to an exchange sort. For the quick sort, with a method of scanning from both ends to the middle, to randomly take a record (usually the 1st record) from n records to be sorted, place the record in an appropriate position; the data sequence is divided into 2 parts by the record. All data with keyword value is less than the record is placed in the front part and all data with keyword value is larger than the record is placed in the rear part, and the record is arranged in the middle of both parts. This process is called as a round of quick sort. Provided that the sequence to be sorted is R[s], R[s+1],..., R[t], firstly, to select the R[s] as a pivot, then rearrange the rest records according to the above principle, and finally, separate the sequence R[s], R[s+1],..., R[t] into two sub-sequences R[s], R[s+1],..., R[i-1] and R[i+1],..., R[t].

Time complexity: the average time complexity of quick sort is O ($n\log_2 n$). When the keyword of record set is sequential, the time complexity of quick sort is O (n^2). In such case, the quick sort is slow.

Straight insertion sort. The straight insertion sort belongs to an insertion sort. N records to be sorted are stored in the array R [1] ˜R[n] to divide the array into a sorted list and an unordered list. At the beginning, there is only one record R [1] in the ordered list and there are n-1 records, R [2] ˜R[n], in the unordered list contains. During the sorting, to take a record from the unordered list each time and insert it in an appropriate positions in the ordered list so that it becomes a new ordered list. After n-1 inserting times, the unordered list becomes a null list and the ordered list contains all n records. The sorting is ended.

The time complexity: the straight insertion sort is a stable sorting algorithm. In the best case of straight insertion sort (positive sequence), the "moving" time is 0. In the worse case (reverse sequence), the "moving" time is $\sum_{i=2}^{n}(i+1)=\frac{(n+4)(n-1)}{2}$, therefore, the time complexity of straight insertion sort is O (n^2).

Shell sort. Shell sort belongs to an insertion sort. The record sequence to be sorted is divided into several groups. Within each group, the records are separately processed with straight insertion sort so that the overall record sequence are partial sequential; the procedure is repeated till all the records are in one group. Finally, all records are processed with a straight insertion sort.

Time complexity: the time performance of Shell sort is between O (n^2) and O ($n\log_2 n$). When n falls within a certain range, the necessary comparison times and record moving times are approximately O ($n^{1.3}$) in Shell sort.

Heap sort. The heap sort belongs to a selection sort. Firstly, to structure the record sequence to be sorted into a heap, then select the 1st maximal records in the heap, remove it from the heap, and readjust the rest records into a heap so that the 2nd maximal record is found and so forth till there is merely one record in the heap. This process is known as a heap sort.

Time complexity: the main time spent in the heap sort is in repeated "screening" during the initial heap construction and new heap adjustment. For the heap sort, the time complexity at the worse case is O ($n\log_2 n$). This is the biggest advantage of the heap sort.

Merge sort. Provided that an initial sequence contains n records, which could be viewed as n ordered subsequences, each subsequence is 1 in length, then by merging pairwise, to give n/2 ordered sequences with length 2 or 1, then merge them pairwise and repeatedly till an ordered sequence with length n is given. This sorting method is known as a 2-way merge sort.

Time complexity: the time complexity of merge sort is O ($n\log_2 n$) in any case.

Example of sorting algorithm in Java language

Given that the original data 49,38,65,97,76,13,27,49,78,34,12,64, n = 12, the original data is sorted respectively using algorithms of bubble sort, quick sort, straight insertion sort, Shell sort, heap sort and merge sort. Hereinafter, the example of core code parts in 3 typical algorithms is taken.

For exchange sort. To take the bubble sort as an example; the Java core code is as follows:

```
long startTime=System.nanoTime();
    for (int i = 0; i <n; i++) {
```

```
            for(int j = 0; j<n-i-1; j++){
              if(a[j]>a[j+1]){
               int temp = a[j];a[j] = a[j+1];a[j+1] = temp; } } }
        long endTime=System.nanoTime();
      System.out.println("Program running time:"+(endTime-startTime)/1000+" microsecond");
```

For insertion sort. To take the straight insertion sort as an example; the Java core code is as follows:

```
        long startTime=System.nanoTime();
          for (int i = 1; i < n; i++) {
               int temp = a[i];
              int j;
              for (j = i-1; j>=0; j--)
                        { if(a[j]>temp) { a[j+1] = a[j];}
                          else  break;            }
              a[j+1] = temp;
           }
        long endTime=System.nanoTime();
      System.out.println("Program running time:"+(endTime-startTime)/1000+" microsecond");
```

For the selection sort. To take the heap sort as an example; the Java core code is as below:

```
        long startTime=System.nanoTime();
        int arrayLength=n;
          for(int i=0;i<arrayLength-1;i++){
            buildMaxHeap(a,arrayLength-1-i);
            swap(a,0,arrayLength-1-i);
            System.out.println(Arrays.toString(a));
          }
        long endTime=System.nanoTime();
      System.out.println("Program running time:"+(endTime-startTime)/1000+" microsecond");
        public static void buildMaxHeap(int[] data, int lastIndex){
          for(int i=(lastIndex-1)/2;i>=0;i--){
            int k=i;
            while(k*2+1<=lastIndex){
              int biggerIndex=2*k+1;
              if(biggerIndex<lastIndex){
                if(data[biggerIndex]<data[biggerIndex+1]){
                  biggerIndex++;
                }
              }
             if(data[k]<data[biggerIndex]){ swap(data,k,biggerIndex); k=biggerIndex; }
            else  break;
            }
        private static void swap(int[] data, int i, int j) {
          int tmp=data[i];
          data[i]=data[j];
          data[j]=tmp;
        }
      }
```

Analysis on sorting performance

The time consuming of sorting method is shown in Table 1 and the performance of each sorting method is shown in Table 2.

Table 1 The time consuming of sorting method

Sorting method	Time consuming （Microsecond）
Bubble sort	105
Quick sort	172
Straight insertion sort	96
Shell sort	92
Heap sort	1365
Merge sort	116

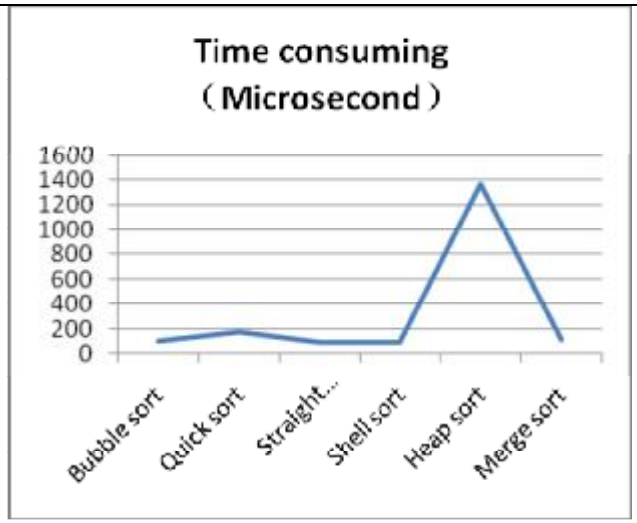

Table 2 The performance of each sorting method

Sorting method	Time complexity			stability	complexity
	Average	Worst	Best		
Bubble sort	O（n^2）	O（n^2）	O（n）	Stable	simple
Quick sort	O（$n\log_2 n$）	O（n^2）	O（$n\log_2 n$）	Instable	complex
Straight insertion sort	O（n^2）	O（n^2）	O（n）	Stable	simple
Shell sort	O（$n^{1.3}$）			Instable	complex
Heap sort	O（$n\log_2 n$）	O（$n\log_2 n$）	O（$n\log_2 n$）	Instable	complex
Merge sort	O（$n\log_2 n$）	O（$n\log_2 n$）	O（$n\log_2 n$）	Stable	complex

Conclusions

Small data size. When the data to be sorted is ordered essentially, the straight insertion sort may be selected. Where the stability is not required, it is preferable to use the simple selection sort; where the stability s required, it is preferable to use the straight insertion sort or bubble sort.

Medium data size. The quick sort is supposed to be used if the memory space can be used, the sequence is disordered and the stability is not required. The extra space for log (N) must be exhausted. The merge sort is supposed to be used if the sequence is itself ordered possibly, the stability is required and the space permits.

Large data size. The straight insertion sort and bubble sort are supposed to be used.

Acknowledgements

This work was financially supported by the Municipal professional comprehensive reform and construction project in Beijing City University.

References

[1] Zhao Yaqing, Xu Yan. Comparative analysis of numerical sorting algorithm. Computer Programming Skills & Maintenance，2015.12：5-7

[2] Jiang Yan，Zhou Jun. Table analysis of internal sorting algorithm[J]. Computer Programming Skills & Maintenance, 2014.2(1):22-23

[3] Dongarra J. The top 10 algorithms[J].IEEE Computing in Science&Engineering, 2000, 2(1):22-23

[4] Wu Weina. Comparative analysis of common sorting algorithms[J]. Computer knowledge and technology,2013.03:2014-2147

[5] Zhang Jing. Comparative analysis of common sorting algorithms[J]. Editorial Department of Hexi University,2010.2:26

[6] Gan Yan. Performance analysis of five sorting algorithms[J].Journal of ChongQing University of Arts and Sciences,2016.06:45-60