International Journal of Modern Engineering Research (IJMER)
www.ijmer.com Vol.2, Issue.4, July-Aug 2012 pp-2908-2912 ISSN: 2249-6645

Experimental Based Selection of Best Sorting Algorithm

[1] D.T.V Dharmajee Rao, [2]B.Ramesh
[1] Professor & HOD, Dept. of Computer Science and Engineering, AITAM College of Engineering, JNTUK
[2] Dept. of Computer Science and Engineering, AITAM College of Engineering, JNTUK

Abstract: *Sorting algorithm is one of the most basic research fields in computer science. Sorting refers to the operation of arranging data in some given order such as increasing or decreasing, with numerical data, or alphabetically, with character data.*

There are many sorting algorithms. All sorting algorithms are problem specific. The particular Algorithm one chooses depends on the properties of the data and operations one may perform on data. Accordingly, we will want to know the complexity of each algorithm; that is, to know the running time f(n) of each algorithm as a function of the number n of input elements and to analyses the space and time requirements of our algorithms. Selection of best sorting algorithm for a particular problem depends upon problem definition. Comparisons of sorting algorithms are based on different scenario. This work experimentally analyzes the performance of various sorting algorithms by calculating the execution time.

Keywords*: Sort-Algorithm, Sorting, Quick sort, Merge sort, Radix sort, Bubble sort, Gnome sort, Cocktail sort, Counting sort.*

I. Introduction

Sort is an important operation in computer programming. For any sequence of records or data, sort is an ordering procedure by a type of keyword. The sorted sequence is benefit for record searching, insertion and deletion. Thus enhance the efficiency of these operations.

Two categories of sort algorithms were classified according to the records whether stored in the main memory or auxiliary memory. One category is the internal sort which stores the records in the main memory. Another is the external sort which stores the records in the hard disk because of the records' large space occupation. In fact, by utilizing the splitting and merging, the external sort could be converted to internal sort. Therefore, only internal sort algorithms such as Bubble, Select, Insertion, Merge, Shell, Gnome, Cocktail, Counting, Radix and Quick Sort were discussed bellow.

For the convenience, we make two assumptions bellow. One is the sequence order, ascending is default. Another is all the records of the sequence were stored in the continuous address memory cells. In this situation, the order of records was determined by the position which stored in the memory. The sort is the move operation of records.

The two classes of sorting algorithms are $O(n^2)$ (which includes the bubble, insertion, selection, gnome, cocktail and shell sorts) and O(n log n) (which includes the heap, merge, and quick sort)

1.1 Theoretical Time Complexity of Various Sorting Algorithms

Time Complexity of Various Sorting Algorithms

Name	Best	Average	Worst
Insertion sort	n	n*n	n*n
Selection Sort	n*n	n*n	n*n
Bubble Sort	n	n*n	n*n
Shell Sort	n	$n (\log n)^2$	$n (\log n)^2$
Gnome Sort	n	n*n	n*n
Quick Sort	n log n	n log n	n*n
Merge Sort	n log n	n log n	n log n
Cocktail Sort	n	n*n	n*n
Counting Sort	-	n+ r	n+ r
Radix Sort	-	n(k/d)	n(k/d)
Heap Sort	n log n	n log n	n log n

II. Fundamental Sorting Algorithms

A. Insertion Sort

Insertion sort algorithm used in the experiments below was described by C language as:

```
Algorithm
1. For I=2 to N
2. A[I]=item ,J=I-1
3. WHILE j>0 and item<A[J]
4. A[J+1]=A[J]
5. J=J-1
6. A[J+1]=item
```

Pseudo Code

```
void insertion_sort(int b[],int N11)
{
int i,j,Temp,A[20000];
double starttime,endtime,difftime;
starttime=omp_get_wtime();
for(i=0;i<N11;i++)
A[i]=b[i];
for(i=1; i<N11; i++)
{Temp = A[i];
j = i-1;
while(Temp<A[j] && j>=0)
{A[j+1] = A[j];
j = j-1;}
A[j+1] = Temp;
}
```

B. Bubble Sort

Bubble sort algorithm used in the experiments below was described by C language as:

Algorithm

```
1.  for I=1 to N-1 (for pass)
2.  for k=1 to N-I (for comparison)
3.  if A[K]>A[K+1]
4.  swap [A(k) , A(k+1)]
```

Pseudo Code

```
void bubble(int a[],int n)
{
int i,j,t;
for(i=n-2;i>=0;i--)
 {  for(j=0;j<=i;j++)
  { if(a[j]>a[j+1])
   {t=a[j];a[j]=a[j+1];a[j+1]=t;}}}
}
```

C. Selection Sort

Selection sort algorithm used in the experiments below was described by C language as:

Algorithm

```
1. for I=1 to N-1
2. min=A [I]
3. for K=I+1 to N
4. if (min>A [I])
5. min=A [K], Loc=K
6. Swap (A [Loc],A[I])
7. Exit
```

Pseudo Code

```
void selesort(int b[],int n)
{
int i, j,a[20000],min,index;
double starttime,endtime,difftime;
for(i=0;i<n;i++)
a[i]=b[i];
for(i=0;i<n-1;i++)
    {min=a[i]; index=i;
    for(j=i+1;j<n;j++)
    if(a[j]<min)
      {min=a[j];index=j;
      }
    a[index]=a[i];
    a[i]=min;
     }
}
```

D. Quick Sort

Quick sort algorithm used in the experiments below was described by C language as:

Algorithm

```
Quick sort (A, p, r)
1.  If p < r
2.  Then q←partition (A, p ,r)
3.  Quick sort ( A, p, q-1)
4.  Quick sort (A ,q+1 ,r)
```

To sort an entire array A, the initial call is Quick sort (A, 1, length [A]).

Partition the array

The key to the algorithm is the PARTITION procedure, which rearranges the sub array A [p...r] in place.

Partition (A, p, r)

```
1.  x←A[r]
2.  i←p-1
3.  For  j←p to r-1
4.  Do if A[j]<=x
5.  Then i←i+1
6.  Exchange A[i]↔A[j]
7.  Exchange A[i+1]↔A[r]
8.  Return i+1
```

Pseudo Code

```
void quick_sort(int arr[], int low, int high) {
 int i = low;
 int j = high;
 int y = 0;
 int z = arr[(low + high) / 2];
 do {
  while(arr[i] < z) i++;
  while(arr[j] > z) j--;
  if(i <= j)
{ y = arr[i];arr[i] = arr[j]; arr[j] = y;
   i++;j--;}
 } while(i <= j);
if(low < j)
  quick_sort(arr, low, j);
 if(i < high)
  quick_sort(arr, i, high);
}
```

E. Counting Sort

Counting sort algorithm used in the experiments below was described by C language as:

Algorithm

```
1.   for  I=0  to  K
2.   C[I]=0
3.   For j=1 to length [A]
4.   C [A(J)]=C[A(J)]+1
5.   For I=1 to K
6.  C[I]=C[I]+C[I-1]
7.   For J=length(A) down to 1
8.   B[C(A(J)]=A[J]
9.  C[A(J)]=C[A(J)]-1
```

Pseudo Code

```
void countingsort(int *a, int n)
{
  int i, min, max,array[20000];
  for(i=0;i<n;i++)
  array[i]=a[i];
  min = max = array[0];
  for(i=0; i < n; i++)
  {if ( array[i] < min )
    {min = array[i];}
else if( array[i] > max )
    { max = array[i];}
  }
```

F. Shell Sort

Shell sort algorithm used in the experiments below was described by C language as:

Algorithm

```
1. for I=n to i/2 (for pass)
2. for i=h to n
3. for j=1 to j=j-h upto j>=h && k< a[j-h]
4. assign a[j-h] to a[j]
```

Pseudo Code

```
void shell_sort(int a[],int n)
{
int j,i,m,mid;
for(m = n/2;m>0;m/=2)
{for(j = m;j< n;j++)
{for(i=j-m;i>=0;i-=m)
if(a[i+m]>=a[i])
break;
else
{mid = a[i];a[i] = a[i+m];a[i+m] = mid;}}}}
}
```

G. Gnome Sort

Gnome sort algorithm used in the experiments below was described by C language as:

```
Algorithm
1. for I=1 to n (for pass)
2. if i=0 and a[i-1] < a[i] then i++
3. else swap(a[i-1],a[i])
4. return a
```

Pseudo Code

```
void gnomeSort(int a[], int n)
{
   int i = 1, j = 2;
   int temp;
   while(i < n)
   {if(a[i-1] <= a[i])
      {i = j;j++;}
      else
      {temp = a[i];a[i] = a[i-1];a[i-1] = temp;
         i--;
         if(i == 0)
         {
            i = j;
            j++;
         }}}
}
```

H. Cocktail Sort

Cocktail sort algorithm used in the experiments below was described by C language as:

```
Algorithm
1. for i=1 to n (for pass)
2. for i=n-1 to 1
3. while !i
4. swap(a[i-1],a[i])
```

Pseudo Code

```
void cocktasort(int a[], int l)
{
  int swapped = 0;
  int i,a[20000];
  do {
    for(i=0; i < (l-1); i++) {
      if ( a[i] > a[i+1] ) {
     temp = a[i];
     a[i] = a[i+1];
     a[i+1] = temp;
     swapped = 1;
}
    }
    if ( ! swapped ) break;
    swapped = 0;
    for(i= l - 2; i >= 0; i--) {
      if ( a[i] > a[i+1] ) {
  int temp = a[i];
  a[i] = a[i+1];
  a[i+1] = temp;
  swapped = 1;
    }
  }
 } while(swapped);
```

III. Problem Statement

The problem of sorting is a problem that arises frequently in computer programming. Many different sorting algorithms have been developed and improved to make sorting fast. As a measure of performance mainly the average number of operations or the average execution times of these algorithms have been investigated and compared.

3.1 Problem statement

All sorting algorithms are nearly problem specific. How one can predict a suitable sorting algorithm for a particular problem? What makes good sorting algorithms? Speed is probably the top consideration, but other factors of interest include versatility in handling various data types, consistency of performance, memory requirements, length and complexity of code, and stability factor (preserving the original order of records that have equal keys).

For example, sorting a database which is so big that cannot fit into memory all at once is quite different from sorting an array of 100 integers. Not only will the implementation of the algorithm be quite different, naturally, but it may even be that the same algorithm which is fast in one case is slow in the other. Also sorting an array may be different from sorting a linked list.

3.2 Justification

In order to judge suitability of a sorting algorithm to a particular problem we need to see, are the data that application needs to sort tending to have some pre existing order?

- What are properties of data being sorted?
- Do we need stable sort?

Generally the more we know about the properties of data to be sorted, the faster we can sort them. As we already mentioned the size of key space is one of the most important factors (sort algorithms that use the size of key space can sort any sequence for time O (n log k).

3.3 Explanation

Many different sorting algorithms have been invented so far. Why are there so many sorting methods? For computer science, this is a special case of question, “why there are so many x methods?”, where x ranges over the set of problem; and the answer is that each method has its own advantages and disadvantages, so that it outperforms the others on the same configurations of data and hardware. Unfortunately, there is no known “best” way to sort; there are many best methods, depending on what is to be sorted on what machine and for what purpose. There are many fundamental and advance sorting algorithms. All sorting algorithm are problem specific means they work well on some specific problem, not all the problems. All sorting algorithm apply to specific kind of problems. Some sorting algorithm apply to small number of elements, some sorting algorithm suitable for floating point numbers, some are fit for specific range like (0 1].some sorting algorithm are used for large number of data, some are used for data with duplicate values.

It is not always possible to say that one algorithm is better than another, as relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted; In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order. Different algorithms will perform differently according to the data being sorted.

IV. Experimental Study

In order to compare the performance of the various Sorting algorithms above, we use a desktop computer (Intel Dual Core Processor @ 2.4GHz, 2GB RAM, Windows 7 operating system) to do a serial experiments. Under VS2008, using C language, the programs test the performances of various algorithms from input scale size by utilizing random function call and time function call.

4.1 Experiments and Results

When the input sequence is produced by a random function, input sequence is positive, and the input scale varied from 1024 (1K) to 101376(99K), various sort algorithms time costs were demonstrated by table 4.1.1 and figure 4.1.1

4.1.1 Sort Algorithms Time Cost Under Positive Input Sequence

Data Size	Insert sort	Selec Sort	Bubble Sort	Shell Sort	Gnome Sort	Quick Sort	Merge Sort	Cockta Sort:	Countin Sort:	Radix Sort:
1K	0.0022	0.0037	0.0112	0.0009	0.0054	0.0003	0.0294	0.0036	0.0001	0.0002
10K	0.0452	0.0392	0.1276	0.0023	0.0614	0.0009	0.1171	0.0908	0.0002	0.0011
20K	0.4545	0.6694	2.0738	0.0167	1.0073	0.0039	0.5394	1.4707	0.0006	0.0047
30K	1.0330	1.5091	4.7020	0.0222	2.2820	0.0057	1.2182	3.3793	0.0008	0.0064
40K	1.8413	2.6861	8.3657	0.0475	3.9973	0.0077	1.1318	5.7960	0.0010	0.0085
50K	2.8438	4.0942	12.8560	0.0379	6.1155	0.0097	1.1465	8.9410	0.0012	0.0114
60K	4.0802	5.9034	18.6282	0.0561	8.9355	0.0127	1.7485	13.1773	0.0014	0.0127
70K	5.5719	8.0406	25.6148	0.0541	12.7165	0.0139	2.9771	18.6538	0.0016	0.0150
80K	7.5296	11.3767	34.5813	0.1189	16.3143	0.0156	3.2230	24.1683	0.0027	0.0225
90K	9.1565	13.1304	42.1701	0.0713	21.0288	0.0177	3.6278	31.0652	0.0028	0.0245
99K	11.4793	16.7561	52.3442	0.0675	24.7972	0.0194	2.7245	36.7319	0.0023	0.0214

From the table and the figure above, we got when the scale of input sequence was small, the difference of time cost between these algorithms was small. But with the scale of input sequence becoming larger and larger, the difference became larger and larger. Among these algorithms, the radix sort, counting sort, quick sort, shell and merge was the best, then all other traditional sorts. Whatever, the time cost curve of radix sort, counting sort, quick sort, shell sort was almost a line. It's the slowest changing with the input scale increasing.

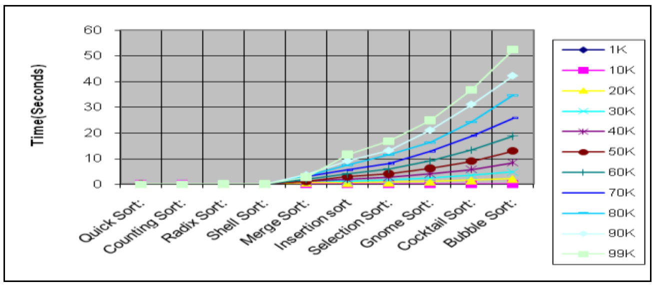

Figure 4.1.1 Time cost Comparison of various sorting algorithms with positive input sequence

When the input sequence is produced by a random function, input sequence is negative and the input scale varied from 1024 (1K) to 101376(99K), various sort algorithms time costs were demonstrated by table 4.1.2 and figure 4.1.2

4.1.2 Sort Algorithms Time Cost Under Negative Input Sequence

Data Size	Quick Sort	Count	Radix Sort	Shell Sort	Merge Sort	Inserti Sort	Select Sort	Gnome Sort	Cocktail Sort	Bubble Sort
1K	0.0003	0.0001	0.0000	0.0006	0.0263	0.0025	0.0035	0.0056	0.0037	0.01171
10K	0.0018	0.0003	0.0001	0.0071	0.2379	0.1116	0.1599	0.2425	0.3521	0.51088
20K	0.0037	0.0006	0.0001	0.0190	0.4627	0.4585	0.6459	0.9664	1.4116	2.0286
30K	0.0058	0.0009	0.0002	0.0219	0.6949	1.0562	1.4730	2.2009	3.1993	4.6123
40K	0.0077	0.0011	0.0003	0.0451	0.9435	1.8196	2.5972	3.8986	5.6744	8.1489
50K	0.0095	0.0012	0.0004	0.0357	1.1701	2.8247	4.0640	6.0874	8.8682	12.7087
60K	0.0125	0.0018	0.0006	0.0595	2.4613	4.3416	6.1430	9.2829	13.3169	18.9648
70K	0.0136	0.0016	0.0005	0.0562	1.6481	5.7460	8.4443	12.0589	17.3594	26.1405
80K	0.0156	0.0018	0.0006	0.1183	1.8738	7.2068	10.3441	15.6466	22.6671	32.6930
90K	0.0177	0.0020	0.0007	0.0669	2.1502	9.3995	13.0794	19.8742	28.7780	41.5611
99K	0.0193	0.0022	0.0007	0.0683	2.3128	11.0168	15.7181	23.9150	34.8231	50.8242

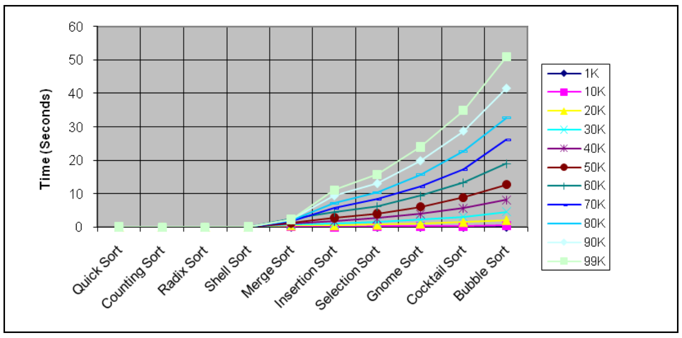

Figure 4.1.2 Time cost Comparison of various Sorting algorithms with Negative input sequence

4.2 Performances Evaluations

Two criteria to evaluate the sort algorithms: time and space. The time related to the comparison operations and move operations of records in the algorithms. The space may dependent or independent to the input sequence scale. If the additional space needed in the algorithm is independent to the input, its space complexity is O(1) . Otherwise, its space complexity is O(n) .

Let N denote the number of input records, in which there are n elements were ordered. Then we could define K, called ordered factor

$$K= (n/N)$$

K $\in [0, 1]$ reflects the sort degree of random sequence. K is bigger, more ordered exists in the sequence. Otherwise, K is smaller, more random exists in the sequence. Let KCN represent the number of comparison operation, and RCN represent the number of remove operation, T(n) and S(n) represent the algorithm time complexity and space complexity respectively. When K $\rightarrow 1$, then RCN $\rightarrow 0$ and T(n) become lesser. When K $\rightarrow 0$, then RCN and T(n) become bigger. According to S (n) whether independent or dependent to the input scale, its value is O (1) or O (n).

V. Conclusion

Every sorting algorithm has some advantages and disadvantages. In the following table we are tried to show the strengths and weakness of some sorting algorithms according to their order, memory used, stability, data type and complexity. To determine the good sorting algorithm, speed is the top consideration but other factor include handling various data type, consistency of performance, length and complexity of code, and the prosperity of stability.

Sort	Order	Worst Case	Memory	stable	Data Type	Complexity
Quick	n log n	n^2	nk+np+stack	no	all	High
Merge	n log n	n log n	nk+np+stack	yes	all	Medium
Shell	$n(\log n)^2$	n	nk+np	no	all	Low
Insertion	n^2	n^2	nk+np	yes	all	very low
Selection	n^2	n^2	nk+np	yes	all	very low
Bubble	n^2	n^2	nk+np	yes	all	very low
Counting	n	n	nk+np	yes	all	very low
Gnome	n	n^2	nk+np	yes	all	very low
Cokatail	n	n^2	nk+np	yes	all	very low

Table 5.1: Strength and Weakness of various sorting algorithm

From the average time algorithms cost, the radix sort, counting sort, quick sort, shell and merge sort are superior to other algorithms. But in the worst time situation, the quick sort cost too much time than the merge sort.

When the input scale isn't big, time cost of algorithms has not an obvious difference. But with the input scale increasing, the Radix sort has certainly on advantage over other algorithms.

For the space occupation, the quick sort and merge sort cost too much than others, their space complexity is O(log n) and O(n), the space occupation of the radix and counting sort is O(nk) and O(n+k), dependent to the input scale. Other algorithms cost little, their space complexity is O(1), independent to the input scale.

For the application, appropriate sort algorithm is selected according to the attributes of input sequence. If the input scale is small, any traditional algorithm is a good choice. But when the input scale is large Radix Sort, Counting sort, Shell Sort, Quick sort and Merge sort is the necessary choice essentially.

References

[1] MacLaren, M.D. “Internal Sorting By Radix Plus Sifting”. J. ACM 13, 3 (July1966), 404-- 411.

[2] Williams, J.W.J. “Algorithm 232: Heap sort”. Comm. ACM 7, 6 (June 1964),347-348.

[3] J. Larriba-Pey, D. Jim´enez-Gonz´alez,and J. Navarro.“An Analysis ofSuperscalar Sorting Algorithms on an R8000 processor”. In InternationalConference of the Chilean Computer Science Society pages 125-134, November1997.

[4] A. Aho, J. Hopcroft, J. Ullman, “Data Structures and Algorithms”, Pearson India reprint, 2000

[5] R. Motwani and P. Raghavan, “Randomized Algorithms”,Cambridge University Press, 2000

[6] Aditya Dev Mishra, Deepak Garg“Selection of Best Sorting Algorithm” International Journal of Intelligent Information Processing” Vol II Issue II 2008 ISSN0.973-3892, p. 233-238.

[7] Kumari A., Chakraborty S. Software Complexity: A Statistical Case Study through Insertion Sort. Applied Mathmatics and Computation. 2007, 190(1): 40-50.

[8] Jafarlou M. Z., Fard P. Y. Heuristic and Pattern Based Merge Sort. Procedia Computer Science. 2011, 3: 322-324.

[9] Nardelli E., Proietti G. Efficient Unbalanced Merge-Sort. Information Science. 2006, 176(10):1321-1337.

[10] Feng H. Analysis of the Complexity of Quick Sort for two Dimension Table. Jisuanji Xuebao. In Chinese. 2007, 30(6):963-968.