TRƯỜNG ĐẠI HỌC SƯ PHẠM TP HỒ CHÍ MINH
TẠP CHÍ KHOA HỌC
KHOA HỌC TỰ NHIÊN VÀ CÔNG NGHỆ
Tập 16, Số 6 (2019): 50-61

HO CHI MINH CITY UNIVERSITY OF EDUCATION
JOURNAL OF SCIENCE
NATURAL SCIENCES AND TECHNOLOGY
Vol. 16, No. 6 (2019): 50-61

ISSN: 1859-3100

Email: tapchikhoahoc@hcmue.edu.vn; Website: http://tckh.hcmue.edu.vn

AN INTERESTING DISCUSSION OF RUNNING TIME FOR SOME SORTING TECHNIQUES WITHOUT COMPARISON SORT

Phan Tan Quoc, Nguyen Quoc Huy

Information Technology Faculty – Saigon University, Việt Nam

[*] Corresponding author: Phan Tan Quoc – Email: quocpt@sgu.edu.vn

Received: 25/3/2019; Revised: 16/4/2019; Accepted: 17/6/2019

ABSTRACT

Sorting is one of important techniques for computer science as well as other technology areas; sorting is used mostly in searching, database management systems, scheduling, and computing algorithms. This paper aims to analyze the timing cost for some sorting techniques without comparison sorting such as Pigeonhole sort, Counting sort, Radix sort, and Bucket sort; these are sorting techniques with linear running time. Each technique is considered in running time, in-place, stable, and extra space if possible. The main contribution of the paper is experiments of sorting techniques in 90 large size test data. This is also a useful reference for working with sorting techniques.

Keywords: sorting algorithm, Pigeonhole sort, Counting sort, radix sort, Bucket sort.

1. Introduction

1.1. Sorting problems

Sorting is a process of data ordering in which data have many types such as integer, double, string, or structured one. Key of data determines the data ordering in a data collection, this is mentioned in (Nguyen, 2013). Requirement of a sorting problem is described as follows.

Input: Array of n number $a_0,a_1,\ldots,a_{n-1}$.

Output: Array of n number $a_{i0},a_{i1},\ldots,a_{in-1}$ in which $(a_{i0},a_{i1},\ldots,a_{in-1})$ is a swap of $(a_0,a_1,\ldots,a_{n-1})$ that satisfies condition $a_{i0} \le a_{i1} \le \ldots \le a_{in-1}$.

Sorting is widely used in many areas such as database management, or search engines. Sorting is also an important phase of computing; some algorithms such as binary search, greedy search, scheduling, and data classification need sorting phase before doing the next phases.

This paper aims for internal sort; it means that data must be stored in RAM at all.

Selection sort, Insertion sort, Bubble sort, Interchange sort, Shell sort, Merge sort, Quick sort, and Heap sort are in comparison sort family because the element ordering is based on comparison; these algorithms work in data type of integer, double, character, string, etc. The best running time of comparison sort algorithms is $O(n \log n)$; there is no any optimization at all.

Pigeonhole sort, Counting sort, Radix sort, Bucket sort, and Spread sort are called un-compared sorting because element ordering is not based on comparison. Running time of these algorithms is linear complexity; and sorting data has some constraints.

1.2. Features of sorting problems

The main features of sorting problems are running time, extra space (including RAM for sorting), stability (it means that elements with same value are kept their ordering), and in-place (it means that extra space is limited by a constant, and not depend on size of array) (Nguyen, 2013; Robert, 2011).

Sorting algorithms with the same big-*O* may have different average running time in different data; when an algorithm is chosen, features mentioned above need to be considered; especially if algorithms have same running time, remaining features should be considered. The same data sizes are, the same performance of algorithms are.

Comparison sorting algorithms are introduced carefully in some data structure materials (Nguyen, 2013; Robert, 2011; Neelam, 2016; Michael, 2011); so the authors summarize some main features of these algorithms to be a foundation for analysis in next sections.

Table 1. *Performance of comparison sort algorithms*

Algorithm	**Running time**	**Extra space**	**Stable**	**In place**	**Method**
Selection sort	$O(n^2)$	1	No	Yes	Selection
Insertion sort	$O(n^2)$	1	Yes	Yes	Insertion
Bubble sort	$O(n^2)$	1	Yes	Yes	Exchanging
Interchange sort	$O(n^2)$	1	No	Yes	Exchanging
Shell sort	$O(n \log^2 n)$	1	No	Yes	Insertion
Merge sort	$O(n \log n)$	n	Yes	No	Merging
Quick sort	$O(n \log n)$	lg n	No	Yes	Partitioning
Heap sort	$O(n \log n)$	1	No	Yes	Selection

2. Running time analysis for un-compared sorting algorithms

Sorting algorithms which are not based on comparison request data satisfying some constraints (this is reason why these algorithms are called special sorts). Their complexity is linear and it is also a limitation of running time.

This part discusses four algorithms: Pigeonhole sort¸ Counting sort, Radix sort, Bucket sort; However, Pigeonhole sort¸ Counting sort, and Radix sort request that sorting data must be positive integer number in range of 0.. *m*, where *m* is the maximum value of sorting elements, Bucket sort could work in real sorting data. These algorithms do not use comparison as well as replacement activities, they only use the assignment of integer indexes, so their running time is much faster than that of Quick sort (Jyoti, 2016; Hinrichs, 2015; Shama, 2015; Waqas, 2016).

2.1. Pigeonhole sort

Below is algorithm description: Let *n* pigeonholes be indexed from 0 to *n*-1, the pigeonhole *i* has weight a_i. Identify the order of pigeonholes such that their weights are in increased order.

Step 1: For *m*+1 wages indexed by the order 0..*m*; the wage *i* only contains the pigeonhole with weight of *i*; all wages contain no any pigeonhole at all.

Step 2: Pass over *n* pigeonholes, which ones have weight of *i* will be contained in the wages *i*; after this step the number of pigeonholes per wage is identified (some wages have no any pigeonhole).

Step 3: Pass over all wages from the index 0 to *m*; get whole pigeonholes from these wages; from that the authors have array of pigeonholes with increased order weights.

```
void pigeonhole(int a[], int n){
for (int i=0;i<=m;i++) b[i]=0;
for (int i=0;i<n;i++) b[a[i]]++;
int d=0;
for (int i=0;i<=m;i++)
while (b[i]>0) {
        a[d++]=i;
        b[i]--;}
}
```

The Pigeonhole sort needs an extra array *b* that its size is the max value of sorting elements. In worse case and average case, the Pigeonhole sort has running time of *O*(*n*+*m*). The Pigeonhole sort is stable, not in-place, and extra space of *O*(*m*) mentioned in (Ashok, 2014; Nguyen, 2013).

2.2. Counting sort

Step 1: Count the number of appearances a_i in original array.

Step 2: Identify the rank for each a_i (rank of a_i is the number of elements in which their values is smaller a_i).

Step 3: Number a_i with rank *r* will be put on the position $r - 1$ of resulted array *c*. If many numbers with the same values appear, they are arranged by the order of appearance in original array to make sure the stable of arrangement.

```
void countingsort(int a[], int n){
for (int i=0;i<=m;i++) b[i]=0;
for (int i=0;i<n;i++) b[a[i]]=b[a[i]]+1;
for (int i=1;i<=m;i++)
        b[i]=b[i]+b[i-1];
for(int i=n-1;i>=0;i--) {
        c[b[a[i]]-1]=a[i];
        b[a[i]]=b[a[i]]-1;}
}
```

The Counting sort needs two extra arrays *b* and *c*; the size of array *c* is the same array *a*, the size of array *b* is equal to the max value of sorting elements. In worse case or average case, the Counting sort running time has complexity $O(n+m)$. The Counting sort is stable, not in-place, and extra space must be $O(n+m)$ (Ashok, 2014; Nguyen, 2013).

In special case, sorting array has couples of different integers. The Counting sort can be adjusted by using one extra array *b*, it was mentioned in (Robert, 2011) (the same size of the max value of sorting elements) as follows:

```
void countingsort_unique(int a[], int n){
b[0]=-1;
for (int i=0;i<n;i++)
        b[a[i]]=a[i];
int d=0;
if (b[0]==0) {
        a[0]=0;
        d++;}
for (int i=1;i<=m;i++)
if (b[i]!=0)
        a[d++]=b[i];
}
```

2.3. *Radix sort*

Suppose that each sorting element has *d* digits.

Step 1: k=0; k is the index of digits.

Step 2: Set 10 blocks $b_0,b_1,\ldots,b_9$ by empty.

Step 3: for i=1..n do

Put a_i into block b_t where t is the k^{th} digit of a_i.

Step 4: Link blocks b_i together (by that process) to create array *a*.

Step 5: k=k+1; and if k<d then go to step 2; other else the algorithm is stopped.

```
void radixsort(int a[],int n){
int exp=1;
while(m/exp>0){
int radix[10]={0};
for(int i=0;i<n;i++)
        radix[a[i]/exp%10]++;
for(int i=1;i<10;i++)
        radix[i]+=radix[i-1];
for(int i=n-1;i>=0;i--)
b[--radix[a[i]/exp%10]]=a[i];
for(int i=0;i<n;i++)
a[i]=b[i];
exp*=10;
}
}
```

Suppose that the sorting elements are in a base *k* number. At that time, each index has maximum *k* values, so the running time each step of the Counting sort has complexity $O(n+k)$. Running time complexity in worse case and in average case is $O(n+k)$. The Radix sort is stable, not in-place. For the arrangement of each iteration, there is a need of using sorting algorithm which is stable, other else the result is not right (Ashok, 2014).

2.4. Bucket sort

Unlike three algorithms mentioned above, the Bucket sort can be implemented in case of sorting real numbers; the real numbers are distributed in range (0..1) in common cases (the appeared probability of real numbers is the same).

Step 1: Put sorting element into each of *k* group.

Step 2: Sort each group; comparison sorting algorithms can be used; such as selection sort, insertion sort as well as un-compared sorting algorithms.

Step 3: Combine groups by ordering to create ordered array.

In worse case, $O(n)$ numbers are put into one group, the Bucket sort has running time $O(k.n^2)$ at that time; in average case, some elements of sorting array is in each group, the Bucket sort has running time $O(k.n)$. The Bucket sort is stable, not in-place, and extra space is $O(n.k)$, it was mentioned in (Ashok, 2014; Nguyen, 2013).

When sorting elements are real numbers, the authors can put sorting elements into each group as following function Bucket_Selectionsort:

```
void Bucket_Selectionsort(float a[maxn],int n,float bucket[maxk][maxm], int n_bucket){
for (int i=0;i<n;i++)
bucket[index_bucket(n_bucket,a[i])][d[index_bucket(n_bucket,a[i])]++]=a[i];
t=0;
for (int i=0;i<n_bucket;i++){
for (int j=0;j<d[i]-1;j++){
int min = j;
for (int h = j+1; h <d[i]; h++)
if (bucket[i][h] < bucket[i][min]) min = h;
exch(bucket[i][min],bucket[i][j]);
a[t++]=bucket[i][j];
}
a[t++]=bucket[i][d[i]-1];
}
}
```

Function index bucket(int k, float x) returns the value x/(1.0/k). Similarly, it is easy to build the function Bucket_Insertionsort; the algorithm Insertion sort is applied to sort elements in each group. The function Bucket sort can be applied to non-negative integer like doing for real numbers; particular number a_i can be put into group which has index a_i/l and number a_i is put at index $d[a_i/l]$ where $l=m/k+1$; m is the maximum value of sorting array. In special case, number k of groups is equal to m; for instance, sorting array has 100 million numbers and m is 1 million, then each group has 100 numbers with the same value; the Bucket sort is the same as Pigeonhole sort in this case and it is described as follows:

```
void bucketsort(int a[], int n){
for(int i=0;i<=m;i++)
        bucket[i]=0;
for(int i=0;i<n;i++)
        bucket[a[i]]++;
for(int i=0,j=0;j<=m;j++)
for(int k=bucket[j];k>0;k--)
        a[i++]=j;
}

```

Table 2. *Performance of uncomparison sorting algorithms (nguyen, 2013)*

Algorithm	**Running time**	**Extra space**	**Stable**	**In place**
Pigeonhole sort	$O(n+m)$	$O(m)$	Yes	No
Counting sort	$O(n+m)$	$O(n+m)$	Yes	No
Radix sort	O($n.d$)	$O(n+m)$	Yes	No
Bucket sort	$O(n.k)$	$O(n.k)$	Yes	No

2.5. Validation of sorting

For comparison sorting algorithms, validation of sorting is simply to check whether input array is not decreased order (Robert, 2011). However, the method mentioned above could not be applied for algorithms with uncomparison sorting because the numbers created in result array are not based on interchange space activities.

The validation of sorting for Pigeonhole, Counting, Radix, and Bucket is implemented as follows: Using result of Quick sort as a standard; the result is stored in array a_i where i=0..n-1. Results of validated algorithms are stored in array c_i where i=0..p-1. The validated algorithm is right if n equal to p and a_i equal to c_i for every i=0..n-1.

```
int Testingsort(int a[], int n, int c[], int p){
if (n!=p) return 0;
for (int i=0;i<n;i++)
if (a[i]!=c[i])return 0;
return 1;
}
```

3. Experiences And Evaluation

This section describes in detail the experiences of sorting algorithms mentioned above and proposes some discussion about them.

3.1. Working environment

The sorting algorithms are implemented by C++ language in the programming editor DEV C++ 5.9.2; they are run in a virtual server with operation system Windows server 2008 R2 Enterprise, 64bit, Intel(R) Xeon (R) CPU E5-2660 0 @ 2.20 GHz, RAM 4GB.

3.2. Testing data

For sorting experiences, 90 random test suites were created including three groups: Group 1 includes 30 test suites which are non-negative integer data, they are randomly generated by function rand(), group 2 includes 30 test suites like group 1 but they have a constraint in which data are different from each other by couple, group 3 includes 30 test suites which are generated by the instruction 1.0*(rand()+1)/(RAND_MAX+2.

Group 1 and group 2 contain 10 test suites which have one million numbers, 10 test suites which have 10 million numbers, and 10 test suites which have 100 million numbers; group 3 has 10 test suites which have 100 thousands numbers, 10 test suites which have 500 thousands numbers, and 10 test suites which have 1 million numbers (refer to Table 3).

Table 3. *Description of experient test suites*

Group 1		**Group 2**		**Group 3**	
n	*Range*	*n*	*Range*	*n*	*Range*
1000000	[0..99999)	1000000	[0..2000000)	100000	(0..1)
10000000	[0..999999)	10000000	[0..20000000)	500000	(0..1)
100000000	[0..999999)	100000000	[0..200000000)	1000000	(0..1)

3.3. Experimental results and evaluation

Experiment results of algorithms with comparison sorting implemented in 30 test suites of group 1 are in Tables 4, 5; where running time (measured by second) of each algorithm in each group with the same size (*n*=1 million, 10 millions, 100 millions) is average sum of running time of test suites by that size.

Table 4. *The averaged running time of sorting complexity O(n2) with comparison in 10 test suites of group 1*

n	Selection	Insertion	Bubble	Interchange
1000000	407.910	198.398	1977.435	1268.338

The Selection sort, Insertion sort, Bubble sort and Interchange sort: The running time of Selection sort is linear complexity with large records but small keys (Nguyen, 2013), the running time of Insertion sort is linear complexity with ordered files (Nguyen, 2013). The experiments show that the running time of Selection sort and Insertion sort is shorter than that of Bubble sort and Interchange sort; where running time of Insertion sort is 48.6% that of Selection sort; the running time of Insertion sort is 10.0% of that of Bubble sort; the running time of Insertion sort is 15.6% of that of Interchange sort. The running time of Interchange sort is 64.1% of that of Bubble sort. Of all algorithms with complexity of $O(n^2)$, running time of Insertion sort is the shortest one.

Table 5. *Average running time of comparison sorting complexity of o(n log n) in 30 test suites of group 1*

n	Shell	Merge	Quick	Heap
1000000	0.251	0.139	0.119	0.183
10000000	3.880	1.610	1.382	3.322
100000000	61.587	17.457	14.158	49.035

The Shell sort, Merge sort, Quick sort and Heap sort: Complexity of running time in worse case Quick sort is $O(n^2)$, and in the average case is $O(n \log n)$; this is the fastest sorting in case of algorithms with complexity $O(n \log n)$; and this algorithm is also used the most in practical. The authors use Quick sort to compare with other sorting algorithms. Consider in whole test data, the running time of Quick sort is 35.4% of that of Shell sort; running time of Quick sort is 84.3% of that of Merge sort; and 45.3% of that of Heap sort. The larger the size of data is, the more efficient running time of Shell sort, Merge sort and Heap sort is. In case of Shell sort, this one needs very little bit code of program to running, number of comparison is smaller than $n^{6/5}$(Robert, 2011); and experimental results in test suites with 1 million numbers showed in Tables 4, 5 mean that the running time of Shell sort is 0.13% of that of Insertion sort.

Experimental results of uncomparison sorting algorithms in 30 test suites of group 1 are showed in Table 4.

Table 6. *The average running time of uncomparison sorting algorithms n 30 test suites of group 1*

n	Pigeonhole	Counting	Radix	Bucket
1000000	0.005	0.029	0.091	0.004
10000000	0.077	0.990	1.128	0.076
100000000	0.666	10.860	11.330	0.654

The Pigeonhole sort, Counting sort, Radix sort, and Bucket sort: Running time of Pigeonhole sort and Bucket sort (in case of the *k* number of groups is equal *m* as mentioned at section II) is shorter than that of two remained algorithms. Consider in the whole of test data with size 1 million, 10 million, and 100 million numbers, the running time of Bucket sort is respectively 96.8%, 9.5%, and 5.7% of that of Pigeonhole sort, Counting sort, and Radix sort. The running time of Bucket sort is 4.6% of that of Quick sort. Figure 1 shows this comparison. In practical, the sorting algorithms with non-negative integer play an important role and are popular in many areas; so they are very necessary in applications of comparison sorting.

Experimental results of comparison sorting with complexity $O(n \log n)$ and the *Counting sort_unique* in 30 test suite of group 2 are showed in Table 7; where running time of each algorithm in data group with the same size (n=1 million, 10 millions, 100 millions) is average sum of running time in all data with the same size.

Table 7. *Average running time of Counting sort_unique and others sorting in 30 test suites of group 2*

n	Shell	Merge	Quick	Heap	Counting_ unique
1000000	0.276	0.148	0.134	0.192	0.019
10000000	4.285	1.676	1.558	3.495	0.314
100000000	64.717	19.236	17.155	58.521	4.316

The Shell sort, Merge sort, Quick sort, Heap sort, and Counting sort_unique: The running time of Counting sort_unique is 19.8% that of Quick sort. The running time of Shell sort, Merge sort, Quick sort, Heap sort in data with distict key is lower at least 6.8% that of normal standard as showed in Table 5 (It is really to highlight that the running time of algorithms in Table 5 is for test suites of group 1, whereas in Table VII is for test suites of group 2). Figure 2 shows this comparison. In practical, data with distict key plays a crucial role in many areas so Counting sort_unique is very necessary in practical.

Experimental results of three algorithms including Quick sort, Bucket sort combined with Selection sort, and Bucket sort combined with Insertion sort in 30 test suites are real numbers of group 3 showed in Table 8; comparison between Bucket sort and Quick is showed in column CompQS; where column *n*_bucket shows bucket number used in two real versions of Bucket sort.

Table 8. *Average running time of Bucket sort and Quick sort in 30 test suites of group 3*

n	**Quick**	**Bucket_ Selection**	**Comp QS**	**Bucket_ Insertion**	**Comp QS**	***n*_bucket**
100000	0.0130	0.0072	55.4%	0.0040	31.8%	4000
500000	0.0651	0.0328	50.4%	0.0205	31.5%	20000
1000000	0.1292	0.0662	51.2%	0.0268	20.7%	40000

Consider in all 30 test suites, the running time of Bucket sort combined with Selection sort is 52.3% of that of Quick sort; the running time of Bucket sort combined with Insertion sort is 27.7% of that of Quick sort. The Figure 2 shows this comparison.

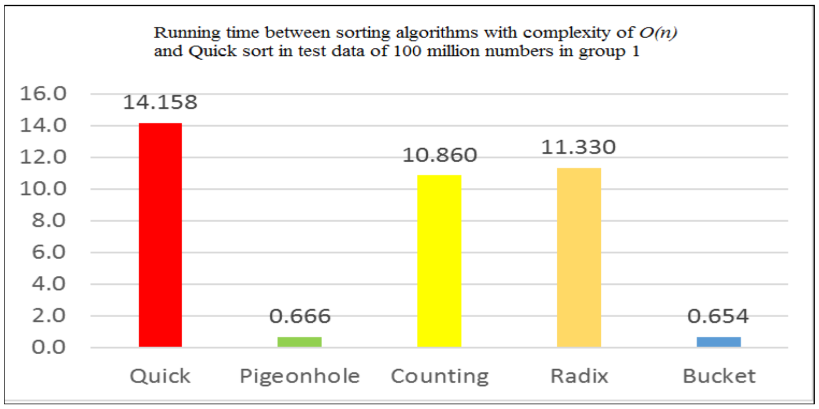

Figure 1. *Running time between Quick sort and others O(n)*

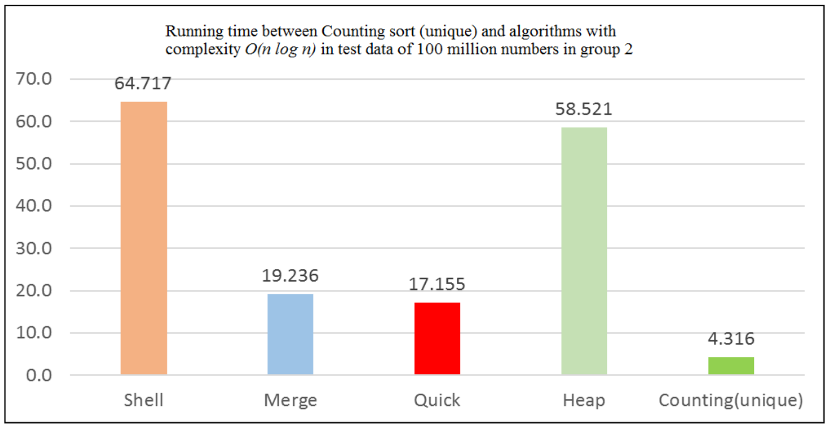

Figure 2. *Running time between Counting_unique and others O(nlog n)*

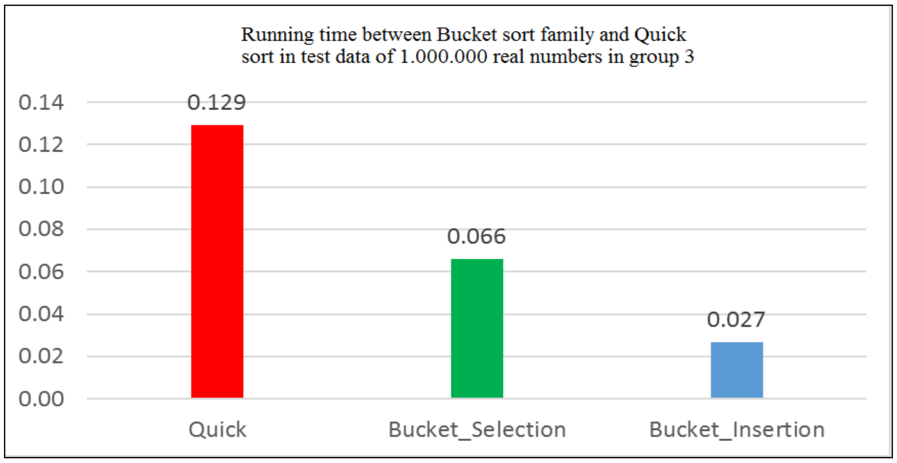

Figure 3. *Running time of Bucket versions and Quick sort*

4. Conclusion and discussion

This paper analyses some common sorting techniques with linear complexity in non-negative integer numbers (Pigeonhole sort, Counting sort, Radix sort, and Bucket sort) and in real numbers (Bucket sort). It is easy to apply them to integer numbers by shifting in negative numbers. Similarly, all sorting algorithms can be used (not only for Bucket sort) to sort real numbers by multiplicating each real number with constant of 10^k to convert a real number to an integer number.

To summary, it is focused on running time analysis, stable, in-place, and extra space for uncomparison sorting algorithms such as Pigeonhole sort, Counting sort, Radix sort, and Bucket sort; these are kinds of sorting which needs data constraint and linear complexity. The authors implemented experiences Pigeonhole sort, Counting sort, Radix sort, and Bucket sort and validate in 90 test suites which are generated randomly with size up to 100 million numbers. With the integer test suites, the running time of Pigeonhole sort, Counting sort, Radix sort, and Bucket sort (for *m* buckets) is respectively 4.7%, 57.4%, 79.1%, 4.6% of that of Quick sort. With the non-negative integer test suites in which couple data are different from each other, the running time of Counting sort_unique is 19.8% of that of Quick sort. With the real number test suites, the running time of Bucket combined with Selection sort's is 52.3% of that of Quick sort, and running time of Bucket sort combined with Insertion sort's is 27.7% of that of Quick sort. Experimental results and the discussion are really useful for users who are working with basic sorting algorithms.

❖ ***Conflict of Interest:*** *Authors have no conflict of interest to declare.*

REFERENCES

Ashok Kumar Karunanithi. (2014). *A survey, discussion and comparison of sorting algorithms*. Department of Computing Science, Umea University.

Jyoti Totla. (2016). *Review on execution time of sorting algorithms - a comparative study*. International Journal of Computer Science and Mobile Computing, *5*(11), 158-166.

L. Hinrichs (2015). *Sorting Algorithms & Run-Time Complexity*. Nebraska Wesleyan Univ.

Michael T. Goodrich, Roberto Tamassia, David M. Mount. (2011). Data *structures and algorithms in C++*. John Wiley & Sons. Inc., 500-551.

Nguyễn Đức Nghĩa. (2013). *Cấu trúc dữ liệu và giải thuật*. NXB Bách khoa.

Neelam Yadav& Sangeeta Kumari. (2016). Sorting algorithms. *International Research Journal of Engineering and Technology*, *3*(2), 528-531.

Robert Sedgewick & Kevin Wayne. (2011). *Algorithms*. Fourth edition, Addsion-Wesley.

Robert L. Kruse & Alexander J. Ryba (2000). *Data structures and program design in C++*. Prentice Hall, 317-378.

Shama Raheja, Vinay kukreja (2015). *Enhancements in sorting algorithms: a review*. *3*(1), 73-82.

Waqas Ali, Tahir Islam, Habib Ur Rehman, Izaz Ahmad, Muneeb Khan, Amna Mahmood (2016). Comparison of different sorting algorithms. *International Journal of Advanced Research in Computer Science and Electronics Engineering*, *5*(7), 63-71.

MỘT PHÂN TÍCH THÚ VỊ VỀ THỜI GIAN CHẠY ĐỐI VỚI CÁC KĨ THUẬT SẮP XẾP KHÔNG SO SÁNH

Phan Tấn Quốc, Nguyễn Quốc Huy

Trường Đại học Sài Gòn

[*] *Corresponding author: Phan Tấn Quốc – Email: quocpt@sgu.edu.vn*

Ngày nhận bài: 25-3-2019; ngày nhận bài sửa: 16-4-2019; ngày duyệt đăng: 17-6-2019

TÓM TẮT

Sắp xếp là một trong những kĩ thuật quan trọng trong ngành khoa học máy tính cũng như trong nhiều lĩnh vực khác; sắp xếp dùng nhiều trong tìm kiếm, các hệ quản trị cơ sở dữ liệu, lập lịch và các thuật toán máy tính. Bài báo này tập trung vào việc phân tích chi phí thời gian của một số kĩ thuật sắp xếp không so sánh như Pigeonhole, Counting, Radix, Bucket; đây là những kĩ thuật sắp xếp với thời gian tuyến tính. Trong mỗi thuật toán chúng tôi xét đến các tiêu chí như thời gian chạy, tính tại chỗ, tính chắc chắn, và không gian bộ nhớ phụ. Đóng góp chính của bài báo là những thực nghiệm trên dữ liệu lớn. Đây chắc chắn là phần tham khảo cần thiết cho những độc giả làm việc với các kĩ thuật sắp xếp.

Từ khóa: thuật toán sắp xếp, sắp xếp Pigeonhole, sắp xếp Counting, sắp xếp cơ số, sắp xếp Bucket.