Bahig *Journal of the Egyptian Mathematical Society* (2019) 27:3
https://doi.org/10.1186/s42787-019-0004-2

Journal of the Egyptian Mathematical Society

ORIGINAL RESEARCH Open Access

Complexity analysis and performance of double hashing sort algorithm

Hazem M. Bahig[1,2]

Correspondence: Hazem.m.bahig@gmail.com; hbahig@sci.asu.edu.eg
[1]Computer Science Division, Department of Mathematics, Faculty of Science, Ain Shams University, Cairo 11566, Egypt
[2]College of Computer Science and Engineering, Hail University, Hail, Kingdom of Saudi Arabia

Abstract

Sorting an array of n elements represents one of the leading problems in different fields of computer science such as databases, graphs, computational geometry, and bioinformatics. A large number of sorting algorithms have been proposed based on different strategies. Recently, a sequential algorithm, called double hashing sort (DHS) algorithm, has been shown to exceed the quick sort algorithm in performance by 10–25%. In this paper, we study this technique from the standpoints of complexity analysis and the algorithm's practical performance. We propose a new complexity analysis for the DHS algorithm based on the relation between the size of the input and the domain of the input elements. Our results reveal that the previous complexity analysis was not accurate. We also show experimentally that the counting sort algorithm performs significantly better than the DHS algorithm. Our experimental studies are based on six benchmarks; the percentage of improvement was roughly 46% on the average for all cases studied.

Keywords: Sorting, Quick sort, Counting sort, Performance of algorithm, Complexity analysis

Mathematics Subject Classfication: 68P10, 68Q25, 11Y16, 68W40

Introduction

Sorting is a fundamental and serious problem in different fields of computer science such as databases [1], computational geometry [2, 3], graphs [4], and bioinformatics [5]. For example, to determine a minimum spanning tree for a weighted connected undirect graph, Kruskal designed a greedy algorithm and started the solution by sorting the edges according to weight in increasing order [6].

Additionally, there are several reasons why the sorting problem in the algorithm aspect is important. The first pertains to finding a solution for a problem in which data are sorted according to certain criteria; this routine should be more efficient, in running time, than when the data are unsorted. For example, searching for an element in an unsorted array requires $O(n)$; the searching requires $O(log\ n)$ time when the array is sorted [6]. The second reason is that the sorting problem has been solved by a lot of algorithms using different strategies such as brute-force, divide-and-conquer, randomized, distribution, and advanced data structures [6, 7]. Insertion sort, bubble sort, and selection sort are examples of sorting algorithms using brute-force; merge sort and quick sort algorithms are examples of the divide-and-conquer strategy. Radix sort and

Springer Open

© The Author(s). 2019, corrected publication 2019. **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

flash sort algorithms are examples of sorting using the distribution technique; heap sort is an example of using an advanced data structure method. Additionally, the randomization techniques have been used for many previous sorting algorithms such as randomized shell sort algorithm [8]. The third reason is the lower bound for the sorting problem which is equal to $\Omega(n \log n)$ was determined based on a comparison model. Based on a determined lower bound, the sorting algorithms are classified into two groups: (1) optimal algorithms such as merge sort and heap sort algorithms and (2) non-optimal algorithms such as insertion sort and quick sort algorithms. The fourth reason is when the input data of sorting problem are taken from the domain of integers $[1,m]$, different strategies are suggested to reduce the time of sorting from $O(n \log n)$ to linear, $O(n)$. These strategies are not based on a comparison model. Examples for this kind of sorting are counting sort and bucket sort [6].

The sorting problem has been studied thoroughly, and many research papers have focused on designing fast and optimal algorithms [9–16]. Also, some studies have focused on implementing these algorithms to obtain an efficient sorting algorithm on different platforms [15–17]. Additionally, several measurements have been suggested to compare and evaluate these sorting algorithms according to the following criteria [6, 7, 18]: (1) Running time, which is equal to the total number of operations done by the algorithm and is computed for three cases: (i) best case, (ii) worst case, and (iii) average case; (2) the number of comparisons performed by the algorithm; (3) data movements, which are equal to the total number of swaps or shifts of elements in the array; (4) in place, which means that the extra memory required by the algorithm is constant; and (5) stable, which means that the order of equal data elements in the output array is similar in which they appear in the input data.

Recently, a new sorting algorithm has been designed and called the double hashing sort (DHS) algorithm [19]. This algorithm is based on using the hashing strategy in two steps; the hash method used in the first step is different than in the second step. Based on these functions, the elements of the input array are divided into two groups. The first group is already sorted, and the second group will be sorted using a quick sort algorithm. The authors in [19] studied the complexity of the algorithm and calculated three cases of running time and storage of the algorithm. In addition, the algorithm was implemented and compared with a quick sort algorithm experimentally. The results reveal that the DHS algorithm is faster than the quick sort algorithm.

In this paper, we study the DHS algorithm from three viewpoints. The first aspect involves reevaluating the complexity analysis of the DHS algorithm based on the relation between the size of the input array and the range of the input elements. Then, we prove that the time complexity is different than that is calculated in [19] for most cases. The second aspect involves proving that a previous algorithm, counting sort algorithm, exhibits a time complexity less than or equal to that of the DHS algorithm. The third aspect refers to proving that the DHS algorithm exhibits a lower level of performance than another certain algorithm from a practical point of view.

The results of these studies are as follows: (1) the previous complexity analysis of the DHS algorithm was not accurate; (2) we calculated the corrected analysis of the DHS algorithm; (3) we proved that the counting sort algorithm is faster than the DHS algorithm from theoretical and practical points of view. Additionally, the percentage of improvement was roughly 46% on the average for all cases studied.

The remainder of this work is organized into four sections. In the "Comments on DHS algorithm" section, we discuss briefly the DHS algorithm, its analysis, and then, we provide some commentary about the analysis of DHS algorithm that was introduced by [19]. In the "Complexity analysis of DHS algorithm" section, we analyze the DHS algorithm using different methods. Also, we show that a previous algorithm exhibits a time complexity less than that of the DHS algorithm in most cases. We prove experimentally that the DHS algorithm is not as fast as the previous algorithm in the "Performance evaluation" section. Finally, our conclusions are presented in the "Conclusions" section.

Comments on DHS algorithm

The aim of this section is to give some comments about DHS algorithm. So, we mention in this section shortly the main stages and complexity analysis of DHS algorithm. Then, we give some comments about the analysis of the algorithm.

DHS algorithm

The DHS algorithm is based on using two hashing functions to classify input elements into two main groups. The first group contains all elements that have repetitions greater than one; the second group contains all elements in the input array that do not have repetition. The first hashing function is used to compute the number of elements in each block and to determine the boundaries of each block. The second hashing function is used to give a virtual index to each element. Based on the values of the indices, the algorithm divides the input into two groups as described previously. The algorithm sorts the second group using a quick sort algorithm only. The algorithm consists of three main stages [19]. The first stage involves determining the number of elements belonging to each block assuming that the number of blocks is nb. The block number of each element, a_i, can be determined using $\lceil a_i/sb \rceil$, where sb is the size of the block and equal $\lceil(\mathrm{Max}(A) - \mathrm{Min}(A) + 1)/nb\rceil$. The second stage refers to determining a virtual index for each element that belongs to the block $b_i, \forall\, 1 \le i \le nb$. The values of the indices are integers and float numbers according to the equations in [19]. The final stage involves classifying the virtual indices into two separate arrays, *EqAr* and *GrAr*. The *EqAr* array is used to represent all elements that have repetitions greater than 1; the *GrAr* array is used to represent all input elements that do not have repetition. The *EqAr* array stores all virtual integer indices and its repetitions; the *GrAr* array stores all virtual float indices. The algorithm sorts only the *GrAr* array using a quick sort algorithm.

The running times of the first and the second stages are always $O(n)$, because we scan an array of size n. The running time of the third stage varied from one case to another; the running time of the DHS algorithm is based mainly on the third stage. Based on the concept of complexity analysis for the running time of the algorithm, we have three cases: best, worst, and average. The running time of the DHS algorithm is based on the size of the array, n, and the maximum element, m, of the input array. The authors [19] analyzed the running time of the DHS algorithm as follows:

1. Best case: this case occurs if the elements of the input array are well distributed and either of n or m is small. In this case, the running time of the third phase is $O(n)$.

2. Worst case: this case occurs if the value of n is large and m is small. Therefore, the number of elements that belong to the array *GrAr* is large. So, the running time of DHS algorithm is $O(n + x \log m)$, where $x \ll n$.
3. Average case: the authors in [19] do not specify the value of n and m in this case. The running time of the DHS algorithm is $O(n + x \log m)$, where $x \ll n$.

Comments on DHS algorithm

In this part, we provide two main comments about the DHS algorithm. The first category of comments is related to the theoretical analysis of the DHS algorithm; the second category of comments is related to the data generated in the practical study.

For the first category, we found that the running times for the DHS algorithm have the following three notes.

The first note is that the running time calculated for the best case is correct when m is small; the running time calculation is not correct when n is small. When n is small and m is large, the number of repetitions in the input array is very small in general. Therefore, most of the elements belong to the *GrAr* array. This situation implies that the DHS algorithm uses the quick sort algorithm on the *GrAr* array. Therefore, the running time of the third phase is $O(n \log n)$, not $O(n)$.

Example 1 Let $n = 10$, $m = n^2 = 100$, and the elements of A is well distributed as follows:

1	2	3	4	5	6	7	8	9	10
77	18	35	63	4	21	29	89	46	35

It is clear that the value of n is small compared with m. Therefore, in general, the number of elements that belong to the *EqAr* array is very small compared with the *GrAr* array that contains most of the input elements.

The second note is that the calculated running time for the worst case is not correct if the value of n is large and m is small. This situation implies that the number of repetitions in the input array is large because the n elements of the input array belong to a small range. Therefore, the maximum number of elements belonging to the *GrAr* array is less than m, say α. On the other hand, the array *EqAr* contains $n - \alpha$ elements. Therefore, the statement "the number of elements belong to the *GrAr* array is large" [19] is not accurate. It should be small since m is small. Therefore, the calculated running time for the worst case of the DHS algorithm, $O(n + x \log m)$, is not accurate in the general case.

Example 2 Let $m = 4$, $n = m^2 = 16$ and A is given as follows

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	3	4	1	1	1	1	1	1	1	1	2	1	1

It is clear that the number of non-repeated elements is 3 and the *GrAr* array contains only 3 different elements, 2, 3, and 4; the *EqAr* array contains 13 elements from 16.

The third note is that no determination when the average case occurs, which is why the running time is $O(n + x \log m)$.

For the second category, the data results in [19] for the three cases reveal how that the percentages of elements in the *EqAr* array, repeated elements, is at least 65%, which is too much and do not represents the general or average cases. This situation means that the input data used to measure the performance of DHS algorithm do not represent various types of input. For example, in the average case, $n = 100{,}000$ and the range of elements equals 100,000, the number of elements in the *EqAr* array is 70,030 [19].

Complexity analysis of DHS algorithm

In this section, we study the complexity of the DHS algorithm using another method of analysis. The DHS algorithm is based on dividing the elements of an input array into many slots; each slot contains elements in a specific range. Therefore, we mainly analyze the DHS algorithm based on the relation between the size of the array, n, and the domain of the elements in the array, m. There are three cases for the relation between n and m.

Case 1: $O(m) < O(n)$. In this case, the range of values for the elements of the input array is small compared with the number of elements in A. This case can be formed as $A = (a_1, a_2, \ldots, a_n)$, where $a_i < m$ and $m < n$. We use big Oh notation to illustrate that the difference between n and m is significant. For example, let $m = \sqrt{n}$ and $m = \log n$ and if $n = 10{,}000$, then $m = 100$ and 4, respectively.

Case 2: $O(m) = O(n)$. In this case, the range of the values for the elements of the input array is equal to the number of elements. This case can be formed as $A = (a_1, a_2, \ldots, a_n)$, where $a_i \leq m$, $n \approx m$, and $m = \alpha\ n \pm \beta$ such that α and β are constant. For example, let $m = 2n$ and $m = n + 25$; if $n = 1000$, then $m = 2000$ and 1025, respectively.

Case 3: $O(n) < O(m)$. In this case, the range of the values for the elements of the input array is greater than the number of elements. This case can be formed as $A = (a_1, a_2, \ldots, a_n)$, where $a_i < m$, $m > n$. For example, let $m = n^k$, where $k > 1$. If $n = 100$ and $k = 3$, then $m = 1{,}000{,}000$.

Now, we study the complexity of the DHS algorithm in terms of three cases.

Case 1: $O(m) < O(n)$. The value of m is small compared with the input size n; the array contains many repeated elements. In this case, the maximum number of slots is m, and there is no need to map the elements of the input array to n slots such as mapping sort algorithm [20], where the index of the element a_i is calculated using the equation: $\lfloor((a_i - \mathrm{Min}(A)) \times n)/(\mathrm{Max}(A) - \mathrm{Min}(A))\rfloor$.

The solution to this case can be found using an efficient previous sorting algorithm called counting sort (CS) algorithm [6]. Therefore, there is no need to use the insertion sort, quick sort, and merge sort algorithms as in [19, 20] to sort un-repeated elements. The main idea of the CS algorithm is to calculate the number of elements less than the integer $i \in [1, m]$. Then, we use this value to allocate the element a_j in a correct location in the array $A, \forall\, 1 \leq j \leq n$. The CS algorithm consists of three steps. The first step of the CS algorithm starts with scanning the input array A and computing the number of repetitions each element occurs within the input array A. The second step of the CS algorithm is to calculate, for each $i \in [1, m]$, the starting location in the output array by updating the array C using the prefix-sum algorithm. The prefix-sum of the array C is to compute $C[i] = \sum_{j=1}^{i} C[j]$. The final step of the CS algorithm allocates each $i \in [1, m]$ and its repetition in the output array using the array C.

Additionally, the running time of the CS algorithm is $O(n + m) = O(n)$, because $O(m) < O(n)$. The running time of the CS algorithm does not depend on the distribution of the elements, uniform and non-uniform, over the range m. Also, the CS algorithm is independent of how many repeated and unrepeated elements are found in the input array.

The following example illustrates how to use the CS algorithm in this case; there is no need to distribute the input into two arrays, *EqAr* and *GrAr*, as in the DHS Algorithm.

Example 3 Let $m = 5$, $n = m^2 = 25$, and the elements of the input array A as in Fig. 1a. As a first step, we calculate the repetition array C, where $C[i]$ represents the number of repetitions of the integer $i \in [1, m]$ in the input array A as in Fig. 1b. It is clear that the number of repetition for the integer "1" is 6; the integer "4" has zero repetition. In the second step, we calculate the prefix-sum of C as in Fig. 1c, where the prefix-sum for $C[i]$ is equal to $\sum_{j=1}^{i} C[j]$. In the last step, the integer 1 is located from positions 1–6; the integer 2 is located from positions 7–14 and so on. Therefore, the output array is shown as in Fig. 1d.

Remark Sometimes the value of m cannot fit in memory because the storage of the machine is limited. Then, we can divide the input array into k ($<m$) buckets, where the bucket number i contains the elements in the range $[(i - 1)m/k + 1, i\ m/k]$, $1 \leq i \leq k$. For a uniform distribution, each bucket contains n/k elements approximately. Therefore, the running time to sort each bucket is $O(n/k + k)$. Hence, the overall running time is $O(k(n/k + k)) = O(n + k^2) = O(n)$. For non-uniform distributions, the number of elements in each bucket i is n_i such that $\sum_{i=1}^{k} n_i = n$. Therefore, the overall running time is $O(n + k) = O(n)$.

Case 2: $O(m) = O(n)$. The value of m is approximately equal to the input size n. If the elements of the array are distributed uniformly, then the number of repetitions for the elements of the array is constant. In this case, we have two comments about the DHS algorithm. The first comment is that there is no need to construct two different arrays, *GrAr* and *EqAr*. The second comment is that there is no need to use the quick sort algorithm in the sorting because we can sort the array using the CS algorithm.

If the distribution of the elements for the input array is non-uniform, then the number of repetitions for the elements of the array is varied. Let the total number of

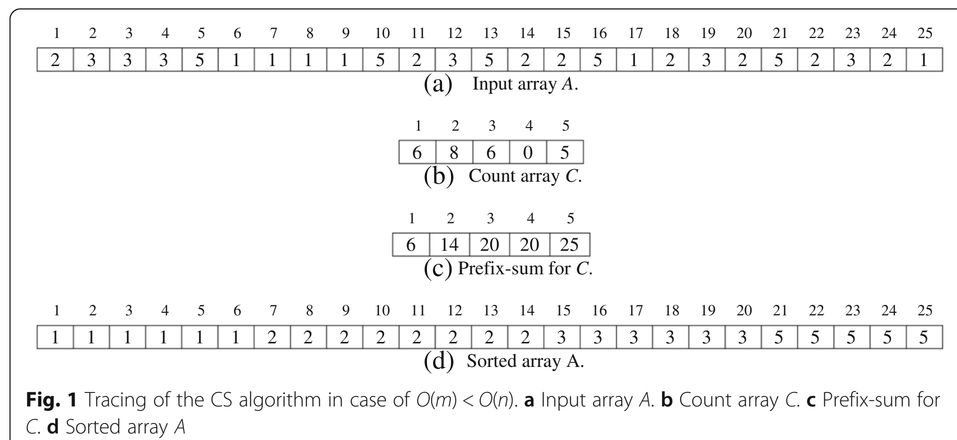


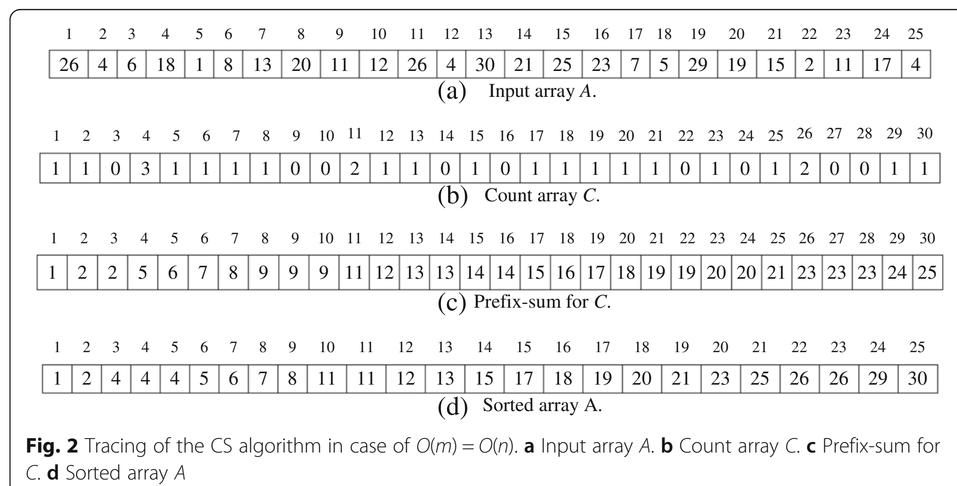
Fig. 1 Tracing of the CS algorithm in case of $O(m) < O(n)$. **a** Input array A. **b** Count array C. **c** Prefix-sum for C. **d** Sorted array A

repetitions for all the elements of the input array be $\phi(n)$. Therefore, the array *EqAr* contains $\phi(n)$ elements; the array *GrAr* contains $n - \phi(n)$ elements. The running time for executing the DHS algorithm is $O(n + (n - \phi(n)) \log (n - \phi(n)))$, where the first term represents the running time for the first two stages and the second term represents applying the quick sort algorithm on the *GrAr* array. In the average case, we have $n/2$ repeated elements, so the running time of the DHS algorithm is $O((n/2) \log (n/2)) = O(n \log n)$. In this case, the CS algorithm is better than the DHS algorithm. On the other side, if $\phi(n) \approx n$, then the running time of the DHS algorithm is $O(n)$.

Example 4 Let $m = 30$, $n = 25$. Fig. 2 shows how the CS algorithm can be used instead of the DHS algorithm in the case of a uniform distribution.

Case 3: $O(n) < O(m)$. The value of m is large compared with the input size n, so the elements of the input array are distinct or the number of repetitions in the input array is constant in general. The DHS and CS algorithms are not suitable for this case. Reasons for not considering these strategies include the following:

1. All of these algorithms require a large amount of storage to map the elements according to the number of slots. For example, if $m = n^2$ and $n = 10^6$ (this value is small for many applications), then $m = 10^{12}$ which is large.
2. If the machine being used contains a large amount of memory, then the running times of the DHS algorithm are $O(n \log n)$. But the main drawbacks of the DHS algorithm are (1) the output of the second hashing function is not unique; (2) the equations used to differentiate between repeated elements and non-repeated elements are not accurate which means that there is an element with certain repetitions and another element without repetition have the same visual indices generated by the suggested equations. Therefore, merge sort and quick sort are better than the DHS algorithm.
3. In the case of CS, the algorithm will scan an auxiliary array of size m to allocate the elements at the correct position in the output. Therefore, the running time is $O(m)$, where $O(m) > O(n)$. If $m = n^2$, then the running time is $O(n^2)$ which is greater than merge sort algorithm, $O(n \log n)$.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	4	6	18	1	8	13	20	11	12	26	4	30	21	25	23	7	5	29	19	15	2	11	17	4

(a) Input array *A*.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	1	0	3	1	1	1	1	0	0	2	1	1	0	1	0	1	1	1	1	1	0	1	0	1	2	0	0	1	1

(b) Count array *C*.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	2	2	5	6	7	8	9	9	9	11	12	13	13	14	14	15	16	17	18	19	19	20	20	21	23	23	23	24	25

(c) Prefix-sum for *C*.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	2	4	4	4	5	6	7	8	11	11	12	13	15	17	18	19	20	21	23	25	26	26	29	30

(d) Sorted array A.

Fig. 2 Tracing of the CS algorithm in case of $O(m) = O(n)$. **a** Input array *A*. **b** Count array *C*. **c** Prefix-sum for *C*. **d** Sorted array *A*

From the analysis of the DHS algorithm for the three cases based on the relation between m and n, there is a previous sorting algorithm that is associated with less time complexity than the DHS algorithm.

Performance evaluation

In this section, we studied the performance of the DHS and CS algorithms from a practical point of view based on the relation between m and n for the two cases, $O(m) < O(n)$ and $O(m) = O(n)$. Note that both algorithms are not suitable in the case of $O(n) < O(m)$.

Platforms and benchmarks setting

The algorithms were implemented using C language and executed on a computer consisting of a processor with a speed of 2.4 GHz and a memory of 16 GB. The computer ran the Windows operating system.

The comparison between the algorithms is based on a set of varied benchmarks to assess the behavior of the algorithms for different cases. We build six functions as follows.

1. Uniform distribution [U]: the elements of the input are generated as a uniform random distribution. The elements were generated by calling the subroutine random() in the C library to generate a random number.
2. Duplicates [D]: the elements in the input are generated as a uniform random distribution. The method then selects *log n* elements from the beginning of the array and assigns them to the last *log n* elements of the array.
3. Sorted [S]: similar to method [U] such that the elements are sorted in increasing order.
4. Reverse sorted [RS]: similar to method [U] such that the elements are sorted in decreasing order.
5. Nearly sorted [NS]: similar to [S]; we then select 5% random pairs of element swaps.
6. Gaussian [G]: the elements of the input are generated by taking the integer value for the average of four calling for the subroutine random().

In the experiment, we have three parameters affecting the running time for both algorithms. The first two parameters are the size of the array n and the domain of the input m; the third parameter is the data distribution (six benchmarks). Based on the relation between n and m, say $O(m) < O(n)$, we fixed the size of the array n and adopted different values of m, m_i, such that $m_i \in O(m) < O(n)$. For example, let $n = 10^8$, and the values of m are $m_1 = 10^6$, $m_2 = 10^5$, $m_3 = 10^4$, $m_4 = 10^3$, and $m_5 = 10^2$. For each fixed value of n and m_i, we generated six different input data values based on the six benchmarks (U, D, S, RS, NS, G). For each benchmark, the running time for an algorithm was the average time of 50 instances, and the time was measured in milliseconds. Therefore, the running time for the algorithm, *Alg*, using the parameters n, m and a certain type of data distribution, is given by the following equation.

$$\frac{1}{n_m}\sum_{i=1}^{n_m}\left(\frac{1}{50}\sum_{j=1}^{50} t_i(n, m_i, dd, Alg)\right)$$

where

- m_i is one of the values for m such that m_i satisfies either $O(m) < O(n)$ or $O(m) = O(n)$. In the experiment, if $n = 10^x$, then $10^2 \le m_i \le 10^{x-2}$.
- n_m is the number of different values for m_i. In the experiment, if $n = 10^x$, then $n_m = x$-3, because $10^2 \le m_i \le 10^{x-2}$.
- dd is the type of data distribution used in the experiment, and the value of dd is one of six benchmarks (U, D, S, RS, NS, G).
- Alg is either the CS or DHS algorithm.
- t_i is the running time for the Alg algorithm using the parameters n, m_i, and the data distribution dd.

In our experiments for both cases, we choose the value of n equal to 10^8, 10^7, 10^6, and 10^5, because the running times for both algorithms are very small when n is less than 10^5.

Experimental results

The results of implementing the methodology to measure the running time of the CS and DHS algorithms considering all parameters that affect the execution times are shown in Figs. 3 and 4. Each figure consists of four subfigures (a), (b), (c), and (d) for $n = 10^5$, 10^6, 10^7, and 10^8, respectively. Also, each subfigure consists of six pairs of bars. Each pair of bar represents the running times for the CS and DHS algorithms using a

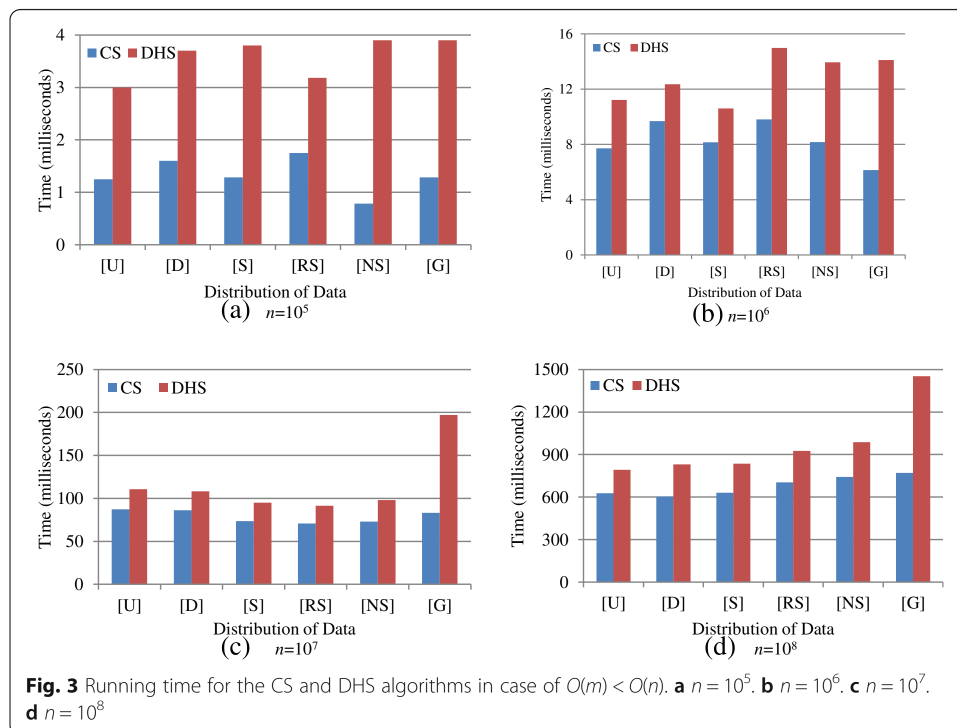


Fig. 3 Running time for the CS and DHS algorithms in case of $O(m) < O(n)$. **a** $n = 10^5$. **b** $n = 10^6$. **c** $n = 10^7$. **d** $n = 10^8$

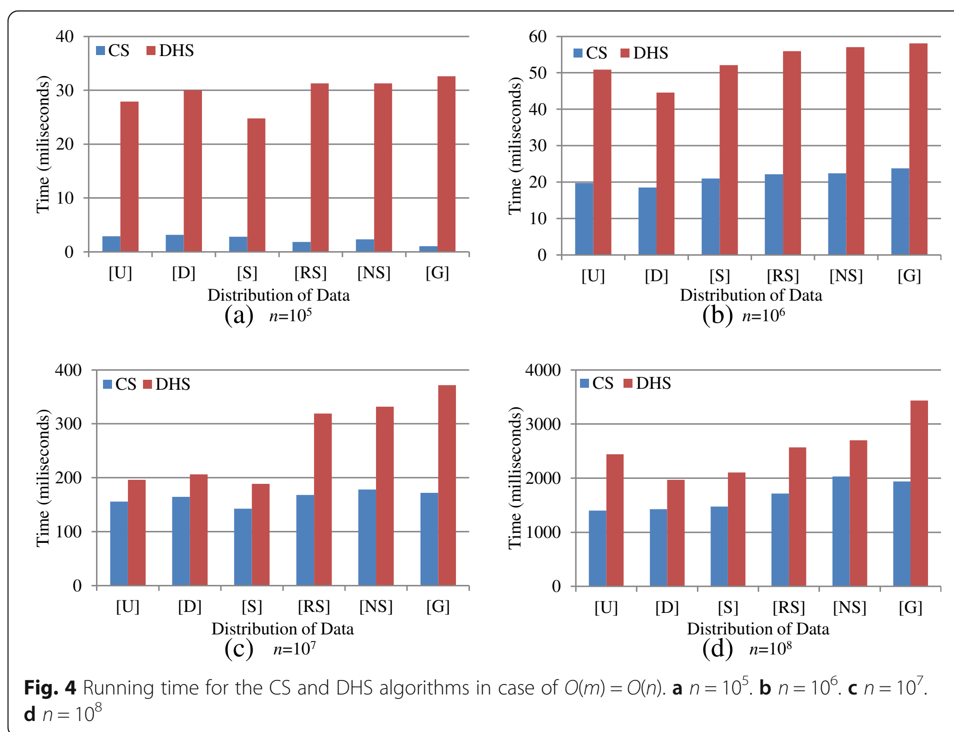


Fig. 4 Running time for the CS and DHS algorithms in case of $O(m) = O(n)$. **a** $n = 10^5$. **b** $n = 10^6$. **c** $n = 10^7$. **d** $n = 10^8$

certain type of data distribution. Fig. 3 illustrates the running times for the CS and DHS algorithm in the case of $O(m) < O(n)$ and shows that the running time for the CS algorithm is faster than for the DHS algorithm for all values of n and benchmarks. The difference in running time between the algorithms varies from one type of data distribution to another. For example, the running time for the CS algorithm using the six benchmarks are 7.7, 9.7, 8.2, 9.8, 8.2, and 6.1 milliseconds; while the running times for the DHS algorithm using the same benchmarks are 11.2, 12.4, 10.6, 15, 13.9, and 14.1 milliseconds in the case of $n = 10^6$. In general, the maximum difference between the two algorithms occurs in the case of a Gaussian distribution

Similarly, Fig. 4 illustrates the running times for the CS and DHS algorithm in the case of $O(m) = O(n)$ and shows that the running time for the CS algorithm is faster than for the DHS algorithm for all values of n and benchmarks.

Table 1 lists data pertaining to the performance improvements of the CS algorithm in two points of view (i) range of improvements, and (ii) mean of improvements. In the case of a range of improvements, we fix the size of the array n and calculate the percentage of improvement for each benchmark. Then, we record the range of improvements from the minimum to maximum values as in the second and fourth columns in Table 1 for

Table 1 Range of improvements for the CS and DHS algorithms

n	$O(m) < O(n)$		$O(m) = O(n)$	
	Range of improvements	Average improvements	Range of improvements	Average improvements
10^5	45–80%	62.5%	88.7–96.8%	91.9%
10^6	21.5–56.5%	34.7%	58.5–60.7%	59.9%
10^7	20–57%	28.3%	20–53.8%	35.4%
10^8	21–47%	28.1%	24.8–43.5%	33.6%

$O(m) < O(n)$ and $O(m) = O(n)$, respectively. In the case of the mean of improvements, we take the mean value for the percentage of improvements for all data distributions, as in the third and fifth columns. The results of applying these measurements are as follows.

1. In the case of $O(m) < O(n)$, the CS algorithm performed 45–80%, 21.5–56.5%, 20–57%, and 21–47% faster than the DHS algorithm for $n = 10^5$, 10^6, 10^7, and 10^8, respectively. For example $n = 10^8$, the percentage of improvements for CS algorithm for data distribution: [U], [D], [S], [RS], [NS], and [G], are 21%, 27.2%, 24.4%, 23.9%, 24.9%, and 47%, respectively. Therefore, the range of improvements for the CS algorithm is 21–47% when $n = 10^8$. Additionally, based on the percentage of improvement calculated for each data distribution and a fixed value of n, we can calculate the mean of improvements which are equal to 62.5%, 34.7%, 28.3%, and 28.1% for $n = 10^5$, 10^6, 10^7, and 10^8, respectively. For example, $n = 10^8$, the mean of improvements is 28%.
2. In the case of $O(m) = O(n)$, the CS algorithm performed 88.7–96.8%, 58.5–60.7%, 20–53.8%, and 24.8–43.5% faster than the DHS algorithm for $n = 10^5$, 10^6, 10^7, and 10^8, respectively. For example $n = 10^7$, the percentage of improvements for CS algorithm for data distribution: [U], [D], [S], [RS], [NS], and [G], are 20.5%, 20%, 24.4%, 47.4%, 46.3%, and 53.8%, respectively. Therefore, the range of improvements for CS algorithm is 20–53.8% when $n = 10^7$. Similarly, we can compute the mean of improvements which are equal to 91.9%, 59.9%, 35.4%, and 33.6% for $n = 10^5$, 10^6, 10^7, and 10^8, respectively.

From previous results, the CS algorithm performed 38.4% and 55.2% faster than the DHS algorithm for $O(m) < O(n)$ and $O(m) = O(n)$, respectively. Therefore, the percentage of improvement for the CS algorithm was roughly 46% on the average for all cases studied.

Conclusions

The sorting problem is to rearrange the elements of a given array in increasing order. This problem is important in a variety of computer science applications, and it is used as a subroutine in many computer applications. In this work, we studied the complexity analysis and measured performance of the double hashing sort (DHS) algorithm. The results of this study are (1) the previous complexity analysis of the DHS algorithm was not accurate; (2) we calculated the corrected analysis of this algorithm based on the relation between size of the input array n and domain of the input elements m; (3) there is a previous sorting algorithm called counting sort algorithm that is faster than the DHS algorithm in the case of $O(m) \leq O(n)$ from theoretical and practical points of view; and (4) our experimental studies are based on six benchmarks; the percentage of improvement was roughly 46% on the average for all cases studied.

Abbreviations

CS: Counting sort; D: Duplicates; DHS: Double hashing sort; G: Gaussian; NS: Nearly sorted; RS: Reverse sorted; S: Sorted; U: Uniform distribution

Acknowledgements

None

Funding

None

Availability of data and materials
The datasets used and/or analysed during the current study are available from the corresponding author on reasonable request.

Author's contributions
The author read and approved the final manuscript.

Authors' information
Hazem Bahig received the B.Sc. degree in Pure Mathematics and Computer Science from Ain Shams University, Faculty of Science in 1990. He also received M.Sc. and Ph. D. degrees in Computer Science in 1997 and 2003 respectively from Computer Science Division, Ain Sham University, Cairo, Egypt. Also, he is currently working in the College of Computer Science and Engineering, Hail University, KSA. His current research interests include high performance computing, design and analysis of algorithms and e-learning systems for algorithms.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 1 November 2018 Accepted: 16 December 2018
Published online: 04 April 2019

References

1. Graefe, G.: Implementing sorting in database systems. ACM Comput. Surv. **38**(3), 10 (2006)
2. Abam, M., Berg, M.: Kinetic sorting and kinetic convex hulls. Comput. Geom. **37**(1), 16–26 (2007)
3. Ezra, E., Mulzer, W.: Convex hull of points lying on lines in O(n log n) time after preprocessing. Comput. Geom. **46**(4), 417–434 (2013)
4. Kim, D.: Sorting on graphs by adjacent swaps using permutation groups. Comput. Sci. Rev. **22**, 89–105 (2016)
5. Shao, M., Lin, Y., Moret, B.: Sorting genomes with rearrangements and segmental duplications through trajectory graphs. BMC Bioinform. **14**(Suppl 15), S9 (**2013**)
6. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms. 3rd ed. MIT Press, Cambridge, England (2009)
7. Knuth, D.: The art of computer programming, vol. 3: sorting and searching, 2nd edn. Addison-Wesley, Reading (1973)
8. Goodrich, M.: Randomized shellsort: a simple data-oblivious sorting algorithm. J. ACM. **58**(6), 27 (2011)
9. Aumüller, M., Dietzfelbinger, M.: Optimal partitioning for dual-pivot quicksort. ACM Trans. Algorithms. **12**(2), 18 (2016)
10. Brodal, G., Fagerberg, R., Moruz, G.: On the adaptiveness of quicksort. ACM J. Exp. Algorithmics. **12**, 3.2 (2008)
11. Cook, C., Kim, D.: Best sorting algorithm for nearly sorted lists. Commun. ACM. **23**(11), 620–624 (1980)
12. Diekert, V., Wei, A.: QuickHeapsort: modifications and improved analysis. Theory Comput. Syst. **59**(2), 209–230 (2009)
13. Mohammed, A., Amrahov, S., Celebi, F.: Bidirectional conditional insertion sort algorithm; An efficient progress on the classical insertion sort. Futur. Gener. Comput. Syst. **71**, 102–112 (2017)
14. Wickremesinghe, R., Arge, L., Chase, J., Scott Vitter, J.: Efficient sorting using registers and caches. J. Exp. Algorithm. **7**, 9 (2002)
15. Cole, R., Ramachandran, V.: Resource Oblivious Sorting on Multicores. ACM Trans. Parallel Comput. **3**(4), 23 (2017)
16. Stehle, E., Jacobsen, H.: A memory bandwidth-efficient hybrid radix sort on GPUs. In: SIGMOD '17 Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, Illinois, USA, pp. 417–43 (2017)
17. Elmasry, A., Hammad, A.: Inversion-sensitive sorting algorithms in practice. ACM J. Exp. Algorithmics. **13**, 11 (2009)
18. Franceschini, G., Geffert, V.: An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. J. ACM. **52**(4), 515–537 (2005)
19. Omar, Y., Osama, H., Badr, A.: Double hashing sort algorithm. Comput. Sci. Eng. **19**(2), 63–69 (2017)
20. Osama, H., Omar, Y., Badr, A.: Mapping Sorting Algorithm, pp. 48–491. SAI Computing Conference 2016, London (2016)

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com