Homogeneous and Non-homogeneous Algorithms

Ioannis Paparrizos

Abstract Motivated by recent best case analyses for some sorting algorithms and based on the type of complexity we partition the algorithms into two classes: *homogeneous* and *non-homogeneous* algorithms.[1] Although both classes contain algorithms with worst and best cases, homogeneous algorithms behave uniformly on all instances. This partition clarifies in a completely mathematical way the previously mentioned terms and reveals that in classifying an algorithm as homogeneous or not best case analysis is equally important with worst case analysis.

Key words Algorithm analysis • Algorithm complexity • Algorithm classification

1 Introduction

In the 1970s and 1980s a lot of discussion was going on regarding the right use of the asymptotic symbols O, Θ and Ω used to analyze algorithms and compare their theoretical efficiency. Some researchers use these symbols to denote the rate of growth of functions and others to denote sets of functions; see relevant comments in [3, 10, 13]. Following the approach of using the asymptotic symbols as sets of functions we partition the class of algorithms into two non-empty subclasses: *homogeneous* and *non-homogeneous* algorithms. Both classes are wide. They contain iterative and recursive algorithms. Although both classes contain

[1]This paper was also presented at local proceedings of PCI'09 [Paparrizos, Homogeneous and Non-Homogeneous Algorithms (2009)].

I. Paparrizos (✉)
Computer Science Department, Columbia University, New York, NY, USA
e-mail: jopa@cs.columbia.edu

A. Migdalas et al. (eds.), *Optimization Theory, Decision Making, and Operations Research Applications*, Springer Proceedings in Mathematics & Statistics 31,
DOI 10.1007/978-1-4614-5134-1_17, © Springer Science+Business Media New York 2013

algorithms with worst and best cases, homogeneous algorithms behave uniformly on all instances of the problem being solved. The partition clarifies in a completely mathematical way the terms of algorithm, worst and best case complexity, the only difference between them being the sets of instances they referred to.

This classification of algorithms was triggered by recent theoretical result concerning best case analysis of some heapsort algorithms [2, 4–8, 20] and [21]. Also, computational results indicate that best case analysis might have practical value too, see, for example, [7] and [21]. Our results indicate that in order to classify an algorithm as homogeneous or not the complexity of the exact, up to a set of functions defined by the asymptotic symbol Θ, best case and worst case must be computed. When the classification is accomplished the analysis of the complexity of the algorithm is complete, indicating, from a theoretical point of view, that best case analysis is equally important with the worst-case analysis.

The term inhomogeneity has been used by Nadel [17] who characterizes the imprecision of an analysis of an algorithm in terms of the difference $\Delta_C = c_w - c_b$ between the worst and best case complexity, where C is a proper measure of complexity. In particular, for the sorting problem, C is the number of comparisons. Using various combinations of disorder parameters, Nadel [17] partitions the set of instances in big, medium, small, tiny and singleton subclasses and computes the inhomogeneity in each subclass. Other relevant results for other problems are presented in [11, 14–16, 18]. Our approach is different in the sense that the set of algorithms is partitioned and not the set of instances of the problem.

In the next section we formally describe the two classes of algorithms. Some details regarding the algorithm classification are presented in Sect. 3. Recursive and divide and conquer homogeneous and non-homogeneous algorithms are discussed and some side results are also presented in the last section.

2 Description of the Two Classes

We derive our results using the Random Access Machine (RAM) model in which every elementary operation such as addition, subtraction, multiplication, and division of two numbers, comparison of two numbers, reading and writing a number in the memory, calling a function, etc., is executed in constant time. It is well known that all constant functions belong to the set $\Theta(1)$. Recall that $\Theta(g(n))$ denotes a set of functions defined as follows:

Definition 1. Given a function $g(n)$ we denote by $\Theta(g(n))$ the set of functions $t(n)$ for which there exists constants $a > 0$ and $b > 0$ and a positive integer n_0 such that

$$bg(n) \leq t(n) \leq ag(n) \tag{1}$$

for every $n \geq n_0$.

All functions used in this paper denote time and therefore they are positive. The argument n denotes the dimension of the problem and, hence, it is a positive integer.

The sets of functions $O(g(n))$ and $\Omega(g(n))$ are similarly defined. Simply, in the definition of $O(g(n))$ the left inequality of (1) is missing, while in the definition of $\Omega(g(n))$ the right one. Observe that $\Theta(g(n))$ is strictly contained in the sets $O(g(n))$ and $\Omega(g(n))$. As a result the assumption that the basic operations are executed in $\Theta(1)$ time (instead of $O(1)$ or $\Omega(1)$ time) provides a more precise algorithm analysis.

It is well known that the symbol Θ considered as a binary relation between functions, is reflexive, symmetric and transitive and therefore it partitions the set of functions into disjoined classes. In other words, if $f(n)$ and $g(n)$ are two different functions, then either $\Theta(f(n)) = \Theta(g(n))$ or $\Theta(f(n)) \cap \Theta(g(n)) = \oslash$. In particular the following two results are well known.

Theorem 1. *If $f(n) \in \Theta(g(n))$, then $\Theta(f(n)) = \Theta(g(n))$.*

Theorem 2. *The sets $\Theta(1)$ and $\Theta(n)$ are disjoint.*

Given a computational problem we denote the set of instances of dimension n by $I(n)$. Consider now an algorithm A solving the problem under consideration. The time taken by algorithm A to solve instance i of dimension n is denoted by $t_A(i,n)$. In algorithm analysis we try to describe in a nice way the set of time functions

$$S = \{t_A(i,n) : i \in I(n)\}$$

One way to do this is via the sets of functions defined by the asymptotic symbols O, Θ, Ω. We are completely satisfied if we can determine a function $g(n)$ such that

$$S \subseteq \Theta(g(n)). \tag{2}$$

Once again, observe that we use the set $\Theta(g(n))$ which is strictly contained in the sets $O(g(n))$ and $\Omega(g(n))$, and therefore the description of set S is more precise. This preference though leads us naturally to the following definition.

Definition 2. An algorithm is *homogeneous* if there exists a function $g(n)$ such that relation (2) holds. Otherwise, the algorithm is *non-homogeneous*.

Theorem 3. *The class of algorithms is partitioned into two non-empty and disjoined subclasses, the subclasses of homogeneous and non-homogeneous algorithms.*

Proof. Let U be the class of all algorithms, H the class of homogeneous and NH the class of non-homogeneous algorithms. It is obvious from Definition 2 that

$$H \cap NH = \oslash \text{ and } H \cup NH = U.$$

It remains to show that $H \neq \oslash$ and $NH \neq \oslash$. This proof is done by providing a simple algorithm for each class.

Algorithm 1: MIN

```
1: a ← T(1)
2: for j = 2 → n do
3:     if T(j) < a then
4:         a ← T(j)
5:     end if
6: end for
```

Firstly, consider the problem of finding the smallest among n given numbers stored as elements of an array T.

The algorithm *min* (Algorithm 1) solves this problem and is homogeneous. Indeed assuming that an element of an array is reached in constant time $\Theta(1)$ in the computational model of constant times, it is easy to conclude that

$$t_{\min}(i,n) \in \Theta(n)$$

for every instance $i \in I(n)$. Hence, algorithm min is homogeneous and $H \neq \oslash$.

Secondly, consider the following problem. Given an array T of n elements sorted in increasing order, i.e.

$$T(j) \leq T(j+1) \text{ for } i = 1,2,\ldots,n-1$$

and a number x, sort all elements of T and the number x in increasing order. This problem is solved by the algorithm *insert* (Algorithm 2).

Denote by i_b the instance $T = [1,2,3,\ldots,n-1,n]$ and $x = n+1$. When algorithm *insert* is applied on instance i_b, the while loop is executed once and hence,

$$t_{\text{insert}}(i_b,n) \in \Theta(1). \tag{3}$$

Denote now by i_w the instance $T = [1,2,3,\ldots,n-1,n]$ and $x = 0$. When algorithm *insert* is applied on instance i_w, the while loop is executed $\Theta(n)$ times and therefore

$$t_{\text{insert}}(i_w,n) \in \Theta(n). \tag{4}$$

This is so because the first two assignments of the pseudo code insert are executed in $\Theta(1)$ time and each execution of the while loop takes $\Theta(1)$ time. We show now that there is no function $g(n)$ such that relation (2) holds. This in turn shows that algorithm insert is non-homogeneous. Suppose, on the contrary, that such a function $g(n)$ does exist. By relation (2) we conclude that

$$t_{\text{insert}}(i_b,n) \in \Theta(g(n)) \text{ and } t_{\text{insert}}(i_w,n) \in \Theta(g(n)). \tag{5}$$

By Theorem 1 and relations (3) and (4) we conclude that

$$\Theta(t_{\text{insert}}(i_b,n)) = \Theta(1) \text{ and } \Theta(t_{\text{insert}}(i_w,n)) = \Theta(n). \tag{6}$$

Algorithm 2: INSERT

```
1: j ← n
2: T(n+1) ← x
3: while j ≥ 1 and T(j) > T(j+1) do
4:     temp ← T(j)
5:     T(j) ← T(j+1)
6:     T(j+1) ← temp
7:     j ← j−1
8: end while
```

Combining Theorem 1 and relations (5) we conclude that

$$\Theta(t_{\text{insert}}(i_b, n)) = \Theta(t_{\text{insert}}(i_w, n)) = \Theta(g(n)) \tag{7}$$

Finally, from relations (6) and (7) we conclude that $\Theta(1) = \Theta(n)$, which contradicts Theorem 2. This completes the proof of the Theorem.

In the proof of Theorem 3 we used two simple algorithms to show that the classes of homogeneous and non-homogeneous algorithms are non-empty. In fact both classes are wide and include recursive and iterative algorithms. The class of non-homogeneous algorithms includes plenty of iterative algorithms. The great majority of recursive and divide and conquer algorithms are homogeneous. Among the exceptions is the well-known recursive sorting algorithm quick sort [12] and Euclid's algorithm for computing the greatest common divisor of two numbers.

3 Algorithm Classification

The instances i_b and i_w used in Theorem 3 are the well-known best and worst cases, respectively. We call i_b *minimum time instance* and i_w *maximum time instance*. More precisely, we give the following definition.

Definition 3. An instance i is a *minimum (maximum)* time instance for an algorithm A, if the total number of elementary operations executed when algorithm A is applied on it is the *minimum (maximum)* possible.

The analysis so far and particularly algorithm *min* used in the proof of Theorem 3 might mislead someone to conclude that homogeneous algorithms do not contain minimum and maximum time instances. This is not correct. A striking example of an iterative homogeneous algorithm containing minimum and maximum time instances is the well-known Floyd's classical algorithm [9] for building an initial heap. A heap is a data structure introduced in [22] to develop an efficient general iterative sorting algorithm known today as heapsort. A recursive homogeneous algorithm containing worst and best cases is the well-known algorithm in [1], which computes order statistics in linear time.

Some algorithms are obviously homogeneous. If this is not clear for a new algorithm with unknown complexity, using Definition 3 we can set

$$S_b = \{\ i_b : i_b \in I(n) \text{ is a minimum time instance}\},$$
$$S_w = \{\ i_w : i_w \in I(n) \text{ is a maximum time instance}\}.$$

In the worst (best) case analysis of an algorithm we try to determine a set $\Theta(g(n))$ ($\Theta(f(n))$) containing the set S_w (S_b) and say that the worst (best) case complexity of the algorithm is $\Theta(g(n))$ ($\Theta(f(n))$). Observe the similarities among the worst and best case complexities of a non-homogeneous algorithm and the complexity of a homogeneous algorithm. In particular, the only difference is the set of instances on which they are referred to. Therefore, all these complexities should be described by sets of the form $\Theta(g(n))$.

It is now of interest to determine the complexity of a non-homogeneous algorithm, i.e, to find a set of functions including set S. Since a set of the form $\Theta(g(n))$ does not exist, we generalize Definition 1 as follows.

Definition 4. Given two (proper) functions $f(n)$ and $g(n)$ we denote by $\Theta(f(n), g(n))$ the set of functions $t(n)$ for which there exist constants $a > 0$ and $b > 0$ and a positive integer n_0 such that

$$bf(n) \leq t(n) \leq ag(n)$$

for $n \geq n_0$.

It is easy to see that $\Theta(f(n), g(n)) = \Omega(f(n)) \cap O(g(n))$. It is also easy to see that the sets $\Theta(0, \infty) = \Omega(0) = O(\infty)$ include always set S. However, in order to be as precise as possible, we are always looking for a minimal set containing set S. In the case of non-homogeneous algorithms we are seeking the minimal set $\Theta(f(n), g(n))$ containing set S. Obviously, the set $\Theta(f(n), g(n))$ is minimal if there exist worst and best case instances i_w and i_b such that $t(i_w, n) \in \Theta(g(n))$ and $t(i_b, n) \in \Theta(f(n))$, respectively. Recall that the set $\Theta(1, n)$ describing the complexity of algorithm *insert* in Theorem 3 is minimal. Observe also that the classification of an algorithm as homogeneous or not is not possible unless the set $\Theta(f(n), g(n))$ describing its complexity is minimal. As the set $\Theta(f(n), g(n))$ is described by best and worst case complexities, both complexities are equally important from the theoretical point of view.

4 Additional Results and Discussion

We mentioned earlier that homogeneous algorithms contain worst and best cases. Hence, the average complexity of a homogeneous algorithm is easily defined. Clearly, the mean time of the algorithm on a random instance is,

$$t(n) = \frac{\sum_{i \in I(n)} t(i, n)}{|I(n)|}$$

where $|I(n)|$ denotes the number of elements of set $I(n)$. If the complexity of the homogeneous algorithm is $\Theta(g(n))$, it is natural to expect that $t(n) \in \Theta(g(n))$. Indeed, this is the case.

Theorem 4. *The average complexity of a homogeneous algorithm of complexity* $\Theta(g(n))$*, is also* $\Theta(g(n))$.

Proof. Let t(n) be the expected time to solve a random instance. Then

$$t(n) = \frac{\sum_{i \in I(n)} t(i,n)}{|I(n)|} \in \frac{\sum_{i \in I(n)} \Theta(g(n))}{|I(n)|} = \frac{|I(n)|\Theta(g(n))}{|I(n)|} = \Theta(g(n))$$

and the proof is complete. □

Observe that this result is independent of the distribution of the instances.

So far we focused our attention on iterative algorithms. Recursive algorithm can be homogeneous and non-homogeneous too. But how recursive homogeneous and non-homogeneous algorithms look like? A recursive or divide and conquer algorithm makes a fixed number of calls to itself. Therefore, if each call is made on a problem with fixed dimension, the algorithm is homogeneous provided the work required to solve all subproblems dominates the remaining work. On the contrary, if the dimensions of the subproblems on which calls are made are not fixed and depend on the instance, the algorithm might very well be non-homogeneous. Recall that this is the case for the algorithm quicksort [12]. A recursive or divide and conquer algorithm can be non-homogeneous if the number of calls to subproblems is not fixed and depends on the instance. This is the case for Euclid's algorithm computing the greatest common divisor.

Acknowledgments We thank an anonymous referee for useful suggestions and for bringing to our attention the reference [17].

References

1. Blum, M., Floyd, R., Pratt, V., Rivest, R., Tarjan, R.: Time bounds for selection. J. Comp. Syst. Sci. **7**(4), 448–461 (1973)
2. Bollobas, B., Fenner, T.I., Frieze, A.M.: On best case of heapsort. J. Algorithms **20**, 205–217 (1996)
3. Brassard, G.: Crusade for a better notation. ACM Sigact News **17**(1), 60–64 (1985)
4. Ding, Y., Weiss, M.A.: Best case lower bounds for Heapsort. Computing **49**, 1–9 (1992)
5. Dutton, R.: Weak-heapsort. BIT **33**, 372–381 (1993)
6. Edelkamp, S., Wegener, I.: On the performance of weak heasort, STACS. Lecture Notes in Computer Science, pp. 254–266. Springer, Berlin (2000)
7. Edelkamp, S., Stiegeler, P.: Implementing heapsort with $nlogn - 0.9n$ and quicksort with $nlogn + 0.2n$ comparisons. ACM J. Exp. Algorithmics (JEA) **7**(1), 1–20 (2002)
8. Fleischer, R.: A tied lower bound for the worst case of bottom-up heapsort. Algorithmica **11**, 104–115 (1994)
9. Floyd, R. Algorithm 245: treesort 3. Comm. ACM **7**, 701 (1964)

10. Gurevich, Y.: What does $O(n)$ mean? ACM Sigact News **17**(4), 61–63 (1986)
11. Haralick, R.M., Elliot, G.L.: Increase tree search efficiency for constraint satisfaction problems. Artif. Intell. **14**, 263–313 (1980)
12. Hoare, A.: Quicksort. Comp. J. **5**, 10–15 (1962) s
13. Knuth, D.: Big omicron and big theta and big omega. ACM Sigact News **8**(2), 18–23 (1976)
14. Nadel, B.A.: The consistent labeling problem and its algorithms: Towards exact-case complexities and theory-based heuristics. Ph.D. dissertation, Department of Computer Science, Rutgers University, New Brunswick, NJ, May (1986)
15. Nadel, B.A.: The complexity of constraint satisfaction in Prolog. In: Proceedings of the 8th National Conference Artificial Intell. (AAAI‘90)pp. 33–39, Boston, MA, August 1990. An expanded version is available as Technical Report CSC-89-004, Department of Computer Science, Wayne State University, Detroit, MI (1989)
16. Nadel, B.A.: Representation selection for constrain satisfaction: a case study using n-queens. IEEE Expert **5**(3), 16–23 (1990)
17. Nadel, B.A.: Precision complexity analysis: a case study using insertion sort. Inf. Sci. **73**, 139–189 (1993)
18. Nudel, B.A.: Solving the general consistent labeling (or constraint satisfaction) problem: two algorithms and their expected complexities. In: Proceedings of the 3rd National Conference Artificial Intell. (AAAI‘83), pp. 292–296, Washington, DC, Aug (1983)
19. Paparrizos, I.: Homogeneous and non-homogeneous algorithms. In: Proceedings of the 13th Panhellenic Conference on Informatics (PCI’09), September (2009)
20. Schaffer, R., Sedgwick, R.: The analysis of heapsort. J. Algorithms **15**, 76–100 (1993)
21. Wang, X.D., Wu, Y.J.: An improved heapsort algorithm with $nlogn - 0.788928n$ comparisons in the worst case. J. Comp. Sci. Tech. **22**(6), 898–903 (2007)
22. Williams, J.W.J.: Algorithm 232: heapsort. Comm. ACM **6**, 347–348 (1964)