Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

An Alternative Approach to the Efficiency of Recursive Merge Sort*

Tibor Ásványi

Eötvös Loránd University, Faculty of Informatics
Budapest, Hungary
asvanyi@inf.elte.hu

Abstract

The *time complexity* (also called *asymptotic running time* or *operational complexity*) $\Theta(n \lg n)$ of merge sort is usually calculated by solving the recurrence

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1, \\ \Theta(1) & \text{if } n = 1, \end{cases}$$

where n is the length of the sequence of keys to be sorted, $T(n)$ is either the best-case or the worst-case asymptotic running time of the algorithm, $\Theta(n)$ is that of division + sorted merge, and $\Theta(1)$ is that of the base case [1, 2].

In this paper we invent an alternative approach: We analyze the structure of the tree of recursive calls, consider its depth and estimate the number of steps [8] of computation at the different levels of that tree. Compared to the equation above we use a more strict notation [5, 6, 7] and argue about its scientific and didactic advantages in efficiency analysis of algorithms in general.

Keywords: algorithm, merge sort, recursion, operational complexity, asymptotic running time, efficiency analysis, education

MSC: 68P05, 68P10, 68P20, 68Q25

CEUR-WS.org/Vol-2650/paper1.pdf

1. Introduction

In the following sections of this paper first we introduce our time complexity measure (section 2) which is quite traditional (see [8]) but following [5, 6] we try to avoid abuse of notation. Next we make clear some notational conventions (section

*Thanks to the Eötvös Loránd University, Faculty of Informatics for financial support.
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

3). Then we discuss our version of the algorithm of merge sort (section 4). Next we present our calculation of the time complexity of this merge sort (section 5). Finally we argue about the educational and scientific advantages of our method of calculation compared to solving a recurrence of the above style (Section 6).

2. Operational complexity of programs

Let $MrT(n)$ and $mrT(n)$ be the maximum and minimum running time of some program where n is the size of its input. Thus $MrT, mrT : \mathbb{N} \to \mathbb{P}$ where $\mathbb{P} = \{x \in \mathbb{R} : x > 0\}$. The problem is that we cannot speak of the running time of it, because we do not know the computing environment. We count the *number of steps* of the algorithm instead.

In this paper we define the **steps** of an algorithm as its **subprogram invocations**[1] and its **loop iterations**. Counting these steps we get the appropriate information about the running time of the program while omitting constant factors which are unknown because we do not know the programming environment [8].

We will be interested in the maximum number of steps denoted as $MT(n)$ (Maximum Time complexity) and in the minimum number of steps denoted as $mT(n)$ where n is the size of the input data structure, in case of a sorting algorithm n is the length of the input array or list. Thus $MT, mT : \mathbb{N} \to \mathbb{N}$.

While we omit constant factors it is enough to give good estimations of functions $MT(n)$ and $mT(n)$. These estimations may be functions of type $\mathbb{N} \to \mathbb{R}$ and they may be negative for small sizes of the input. Consequently the following sets of functions are introduced traditionally.

Definition 2.1. $f : \mathbb{N} \to \mathbb{R}$ is
- *asymptotically nonnegative*, **iff** $\exists N \in \mathbb{N}, \forall n \geq N : f(n) \geq 0$
- *asymptotically positive*, **iff** $\exists N \in \mathbb{N}, \forall n \geq N : f(n) > 0$

Definition 2.2. Assume that f, g and h are asymptotically nonnegative functions.
$O(g) = \{f : \exists d > 0, \exists N \in \mathbb{N}, \forall n \geq N : 0 \leq f(n) \leq d * g(n)\}$
$\Omega(g) = \{f : \exists c > 0, \exists N \in \mathbb{N}, \forall n \geq N : 0 \leq c * g(n) \leq f(n)\}$
$\Theta(g) = O(g) \cap \Omega(g)$
$h + \Theta(g) = \{f : \exists p \in \Theta(g) : f = h + p\}$

We can say,
$f \in O(g)$ means that function g is an asymptotic upper bound of function f,
$f \in \Omega(g)$ means that function g is an asymptotic lower bound of function f
and $f \in \Theta(g)$ means that functions f and g are asymptotically equivalent (see corollary 2.3.b). $f \in \Theta(g)$ also means that the asymptotic order of f is g.

Using these notions we can give appropriate estimations of functions $MT(n)$ and $mT(n)$, as it is illustrated in Section 5.

[1] Surely we can omit nonrecursive calls and this omission is optional.

From the definitions above we can easily derive the following useful consequences:

Corollary 2.3. *Assume that $f, g, h : \mathbb{N} \to \mathbb{R}$ are asymptotically nonnegative.*
(a) $f \in \Theta(g) \wedge g \in \Theta(h) \Longrightarrow f \in \Theta(h)$
(b) $f \in \Theta(g) \iff g \in \Theta(f)$

Corollary 2.4. *Assume that $f : \mathbb{N} \to \mathbb{R}$ is asymptotically nonnegative and $g : \mathbb{N} \to \mathbb{R}$ is asymptotically positive.*

$$f \in O(g) \iff (\exists \psi : \mathbb{N} \to \mathbb{R}), \exists d > 0, \exists N \in \mathbb{N}, \forall n \geq N :$$
$$d * g(n) + \psi(n) \geq f(n) \quad \wedge \quad \lim_{k \to \infty} \frac{\psi(k)}{g(k)} = 0$$
$$f \in \Omega(g) \iff (\exists \varphi : \mathbb{N} \to \mathbb{R}), \exists c > 0, \exists N \in \mathbb{N}, \forall n \geq N :$$
$$c * g(n) + \varphi(n) \leq f(n) \quad \wedge \quad \lim_{k \to \infty} \frac{\varphi(k)}{g(k)} = 0$$

Theorem 2.5. $MrT \in \Theta(MT)$ *and* $mrT \in \Theta(mT)$.

Proof. Let we have maximum k processors where k is a constant.

Let a subprogram call as a step consist of the call and return of the subprogram and all the statements in the subprogram, including the process of exiting from directly embedded loops, but excluding the iterations of those loops and excluding the run of the directly embedded subprogram invocations.

Let a loop iteration as a step consist of the evaluation of the condition of the loop – when this condition is true – followed by performing the statement part of the loop, including the process of exiting from directly embedded loops, but excluding the iterations of those loops and excluding the run of the directly embedded subprogram invocations.

Thus we covered the text of the whole program with a finite number disjoint stages as steps. (The whole program is considered a special subprogram here.) No step contains another loop iteration or recursion, although these may be embedded into the step. Consequently each step has a maximal and a minimal running time. Let M be the maximum of the maximums and let m be the minimum of the minimums. Therefore $(m/k) * MT(n) \leq MrT(n) \leq M * MT(n)$. Thus $MrT(n) \in \Omega(MT(n)) \cap O(MT(n)) = \Theta(MT(n))$. Similarly $(m/k) * mT(n) \leq mrT(n) \leq M * mT(n)$. Thus $mrT(n) \in \Omega(mT(n)) \cap O(mT(n)) = \Theta(mT(n))$. □

Thus it is enough to calculate the asymptotic order of $MT(n)$ and $mT(n)$, that of $MrT(n)$ and $mrT(n)$ will be the same respectively. (See corollary 2.3.) Consequently we have the following corollary.

Corollary 2.6. $(MT \in \Theta(h) \iff MrT \in \Theta(h)) \wedge (mT \in \Theta(g) \iff mrT \in \Theta(g))$

Remark 2.7. Clearly, any definitions of $MT(n)$ and $mT(n)$ suffice, provided that theorem 2.5 remains true. For example, we can omit (some of the) nonrecursive calls, if it is more convenient for us.

3. Notations

We suppose that an array consists of a pointer and a so-called array object where the pointer refers to the object. An array object contains the length of the array object and its elements.

$A : \mathcal{T}[n]$ means that A is a pointer referring to an array object with element type $\mathcal{T}$ and length n. If we write $A : \mathcal{T}[n]$ on a formal parameter list, it specifies pointer A of type $\mathcal{T}[]$ and n is just a short notation for the length of the (actual parameter) array: n can be omitted here, if it is not needed. $A : \mathcal{T}[n]$ can also be a declaration statement. Then it declares pointer A of type $\mathcal{T}[]$, creates the array object of n elements and assigns its address to A. We suppose that this array object is automatically deleted when the block containing it is finished. Arrays are indexed from 0. $A[u..v)$ represents the sequence $\langle A[u], \ldots, A[v-1] \rangle$

The size of a binary tree t is its number of nodes $|t|$, the empty tree is $\oslash$, the number of internal nodes of t is $i(t)$, the number of its leaves is $l(t)$, where $|t| = i(t) + l(t)$. The height of t is $h(t)$ where $h(\oslash) = -1$. If $t \neq \oslash$, $t.left$ and $t.right$ are its left and right subtrees.

4. Merge sort

Merge sort was invented by John von Neumann in 1945 [3, 4] (see Figure 1). It uses the *divide and conquer* approach. Given a sequence of keys to be sorted, in this algorithm we have two cases:

The empty sequences and those consisting of a single item are already sorted; but we half the longer sequences, sort the half-sequences with the same method and merge the sorted parts in a sorted way. (See Figure 1.)

The interface procedure of merge sort is given in Figure 2. And its recursive subroutine can be found in Figure 3. With the choice of $m := \lfloor \frac{u+v}{2} \rfloor$, $A[u..m)$ and $A[m..v)$ have the same length, provided that the length of $A[u..v)$ is even number; and $A[u..m)$ is shorter by one than $A[m..v)$, if the length of $A[u..v)$ is odd number, because

$$length(A[u..m)) = m - u = \left\lfloor \frac{u+v}{2} \right\rfloor - u =$$

$$\left\lfloor \frac{u+v}{2} - u \right\rfloor = \left\lfloor \frac{v-u}{2} \right\rfloor = \left\lfloor \frac{length(A[u..v))}{2} \right\rfloor$$

The pseudocode of sorted merge is in Figure 4. Local array $Z : \mathcal{T}[d]$ is needed so that the output does not overwrite the input. Notice that a trivial solution would copy both halves of $A[u..v)$ to temporal arrays before merge, but it is enough to copy $A[u..m)$ to $Z[0..d)$ and this is done by the first loop.

The actual merge is done by the second and third loops of the procedure: each time we write into $A[k]$, we read from ($Z[i..d)$ and) $A[j..v)$, so it is enough to prove that $k < j$:

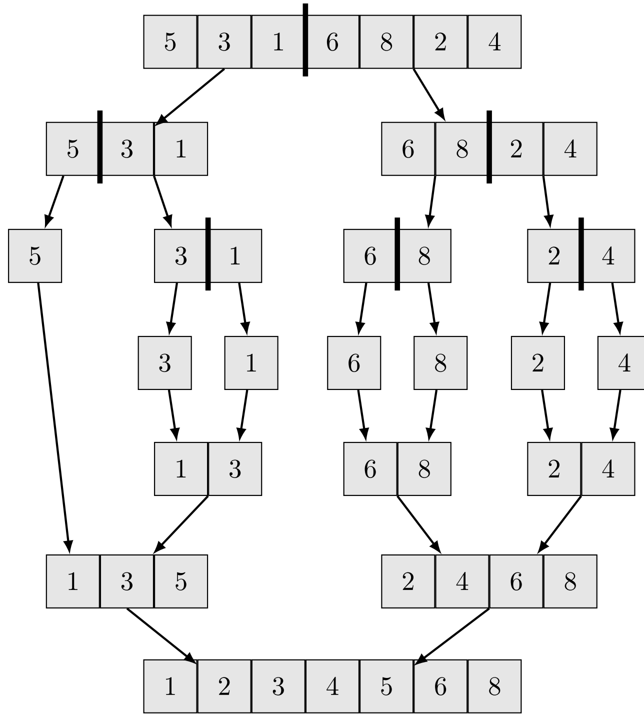

Figure 1: Illustration of merge sort

procedure MERGESORT($A : \mathcal{T}[n]$) ▷ sort the whole array A
  MSORT($A, 0, n$) ▷ which means sorting $A[0..n)$
end procedure ▷ $mT(n), MT(n) \in \Theta(n \lg n)$

Figure 2: Interface procedure of merge sort

When we overwrite $A[k]$, i items have been copied from $Z[0..d)$ and $j-m$ items from $A[m..v)$ to $A[u..v)$, altogether $k-u$ items. In addition, $i < d = m-u$ comes from the conditions of these loops. Thus

$$k - u = i + j - m < m - u + j - m$$

$$k - u < j - u$$

$$k < j$$

Now we explain while the actual merge is divided into the second and third loops. The second loop runs while both of $Z[0..d)$ and $A[m..v)$ contains some item(s) to be copied to $A[u..v)$.

```
procedure mSort(A : 𝒯[]; u, v : ℕ)                ▷ sort subarray A[u..v)
    if u < v − 1 then                            ▷ if length(A[u..v)) ≥ 2 then
        m := ⌊(u+v)/2⌋
        mSort(A, u, m)                           ▷ length(A[u..m)) ≥ 1
        mSort(A, m, v)                           ▷ length(A[m..v)) ≥ 1
        merge(A, u, m, v)
    end if
end procedure
```

Figure 3: Recursive subroutine of merge sort

If the second loop stops with $i = d = m - u$, we have copied all the content of $Z[0..d)$ to $A[u..v)$ and this is $d = m - u$ items, while we have copied $j - m$ items form $A[m..v)$ to $A[u..v)$. Altogether we have copied $k - u$ items there. Thus

$$k - u = (m - u) + (j - m)$$

$$k - u = j - u$$

$$k = j$$

This means that the elements of $A[j..v)$ are already in place and the third loop does nothing as needed.

If the second loop stops with $j = v$, we have copied all the content of $A[m..v)$ to $A[u..v)$ and the remainder of $Z[0..d)$ is copied to the end of $A[u..v)$ as efficiently as possible.

(Notice that the stability of merge sort is ensured by condition $Z[i] \leq A[j]$ of the second loop.)

5. The time complexity of merge sort

The time complexity analysis of sorting algorithms is often based on counting key comparisons. Our method which counts steps [8] is more general. It is useful, even if we have nothing to do with key comparisons.

Let $MT(n)$ and $mT(n)$ be the maximal and minimal number of steps of algorithm mergeSort$(A : \mathcal{T}[n])$. Considering figures 2 and 3, $MT(0) = mT(0) = 2$ is trivial. Thus in the rest of this section we can suppose that $n \geq 1$. Consequently no subarray determined by the algorithm will be empty.

In this section first we estimate the number of steps of subroutine merge. Then we analyze the structure of the call-tree of recursive procedure mSort. Thus we can determine the number of subroutine invocations excluding merge calls + give a good estimate of the number of steps in all of the merge calls. Adding these two we will have a lower bound of $mT(n)$ and an upper bound of $MT(n)$ of merge sort, and with corollary 2.4 we will establish the asymptotic order of its running time.

```
procedure MERGE(A : T[]; u, m, v : ℕ)
                                   ▷ sorted merge of A[u..m) and A[m..v) into A[u..v)
    d := m − u
    Z : T[d]
    for i := u ; i < m ; i++ do                      ▷ copy A[u..m) into Z[0..d)
        Z[i − u] := A[i]
    end for
                                   ▷ sorted merge of Z[0..d) and A[m..v) into A[u..v)
    k := u                                                        ▷ copy into A[k]
    i := 0 ; j := m                                              ▷ from Z[i] or A[j]
    while i < d ∧ j < v do
        if Z[i] ≤ A[j] then
            A[k++] := Z[i++]
        else
            A[k++] := A[j++]
        end if
    end while
    while i < d do                          ▷ copy Z[i..d) to the end of A[u..v)
        A[k++] := Z[i++]
    end while
end procedure                   ▷ l ≤ mT(l) ≤ MT(l) ≤ 2l where l = v − u + 1
```

Figure 4: Pseudocode of sorted merge

5.1. The number of steps of procedure MERGE

In order to calculate the operational complexity of procedure MERGE(A, u, m, v) first we introduce the notation $l = v - u$. Procedure MERGE is called, iff $u < v - 1$, i.e. $l \geq 2$. Let $mT_{\text{MERGE}}(l)$ and $MT_{\text{MERGE}}(l)$ be the minimum and maximum number of steps of performing procedure MERGE(A, u, m, v), respectively.

Remember that a *step* is a *subroutine call* or an *iteration of a loop.* We have a single procedure call now + the $\lfloor l/2 \rfloor$ iterations of the first loop + the iterations of the second and third loops: Minimum the $\lfloor l/2 \rfloor$ items in array Z must be copied back to A, which means $\lfloor l/2 \rfloor$ iterations of the second loop and no iteration of the third loop. If the second and third loop copies all the items, then it is the maximal l iterations of these loops together. Consequently

$mT_{\text{MERGE}}(l) \geq 1 + \left\lfloor \frac{l}{2} \right\rfloor + \left\lfloor \frac{l}{2} \right\rfloor \geq \left\lceil \frac{l}{2} \right\rceil + \left\lfloor \frac{l}{2} \right\rfloor = l$ and
$MT_{\text{MERGE}}(l) \leq 1 + \left\lfloor \frac{l}{2} \right\rfloor + l \leq 2l$, thus

$$l \leq mT_{\text{MERGE}}(l) \leq MT_{\text{MERGE}}(l) \leq 2l$$

(Therefore $mT_{\text{MERGE}}(l), MT_{\text{MERGE}}(l) \in \Theta(l)$.)

5.2. The call-tree of recursive procedure mSort

Let us consider figure 3. The number of mSort calls is clearly equal to the size $|T|$ of the call-tree T of mSort. The leaves of T correspond to case $u = v - 1$. Thus $l(T) = n$. ($n \geq 1$ is the length of array A which is being sorted.) And T is *strictly binary tree* according to the next definition.

Definition 5.1. t is *strictly binary tree*, **iff** each internal node of t has two children.

The next consequence comes by mathematical induction on $l(t)$.

Corollary 5.2. *If $t \neq \oslash$ is a strictly binary tree, then $i(t) = l(t) - 1$. (Thus $|t| = 2 * l(t) - 1$.)*

Thus mergeSort is invoked first + $|T| = 2n-1$ = the number of mSort calls.

Corollary 5.3. *The number of subroutine invocations excluding* merge *calls* $= 2n$.

Now we are going to prove that T is nearly complete, so the depth of T is $\Theta(\lg n)$. First we give the necessary definitions.

Definition 5.4. t binary tree is
- *leaf-balanced*, **iff** for each internal node of t, the number of leaves of its two subtrees can differ maximum by 1.
- *complete*, **iff** it is strictly binary and each of its leaves are at the same level.
- *nearly complete*, **iff** t is empty, or removing its lowest level we receive a complete tree.

Corollary 5.5. *Tree T is* leaf-balanced *(because* mSort *divides the actual subarray in a balanced way and the number of leaves in both parts is equal to the length of the part).*

If $t \neq \oslash$ is *complete* binary tree with height h, then at its zeroth (root) level there is 2^0 node, at the its first level 2^1 nodes and so on, on its (last) level h, it has 2^h nodes, altogether $|t| = 2^{h+1} - 1$.

Corollary 5.6. *If t is* nearly complete *nonempty binary tree with height h, then $2^h \leq |t| \leq 2^{h+1} - 1$. Thus $h = \lfloor \lg |t| \rfloor$.*

Theorem 5.7. *If t is a leaf-balanced strictly binary tree, then t is also* nearly complete.

Proof. We can suppose $t \neq \oslash$. Use mathematical induction on h. If $h = 0$, then t consists of a single node, and t is nearly complete. If the statement is true for heights $\leq h$, consider case $h(t) = h + 1$. Then $t.left$ and $t.right$ are leaf-balanced strictly binary trees with heights $\leq h$, so they are nearly complete. We have two cases. (1) $l(t.left) = l(t.right) \Rightarrow |t.left| = |t.right| \Rightarrow h(t.left) = h(t.right) \Rightarrow$ t is also nearly complete. (2) We can suppose that $l(t.left) + 1 = l(t.right) \Rightarrow$ $|t.left| + 2 = |t.right| \Rightarrow h(t.left) = h(t.right) \vee h(t.left) + 1 = h(t.right)$. Case $h(t.left) = h(t.right)$ is trivial. In case $h(t.left) + 1 = h(t.right)$ there are two leaves at the lowest level of the strictly binary, nearly complete $t.right$, and $t.left$ is complete. Thus t is also nearly complete. ☐

Corollary 5.8. $\lfloor \lg n \rfloor \le h(T) = \lfloor \lg |T| \rfloor = \lfloor \lg(2n-1) \rfloor \le \lfloor \lg(2n) \rfloor = \lfloor \lg n \rfloor + 1$.

5.3. The number of steps of all the MERGE invocations

In this section we give upper and lower estimates of the number of steps of all the MERGE invocations. First notice that the MERGE-calls correspond to the *internal nodes* of T defined at the beginning of subsection 5.2. Let $MT_{merges}(n)$ and $mT_{merges}(n)$ be the maximal and minimal number of steps of all the MERGE invocations where n is the length of the input array of MERGESORT.

First we give an **upper estimate of** $MT_{merges}(n)$. Based on corollary 5.8, the internal nodes (i.e. merge-calls) of T are maximum at levels $0..\lfloor \lg n \rfloor$ of T. Let $MT_{merges(i)}(n)$ be the maximal number of steps of all the MERGE invocations at some level i of T excluding its lowest level. Clearly $MT_{merges(i)}(n) \le$ the sum of the $MT_{\text{MERGE}}(l)$ values of all the MERGE calls at level i (see subsection 5.1). And the subarrays $A[u..v)$ of these MERGE calls are disjoint. We also know from subsection 5.1 that $MT_{\text{MERGE}}(l) \le 2l$ where $l = v-u$. Thus with the distributive rule of addition and multiplication of numbers we have $MT_{merges(i)}(n) \le 2n$. Therefore $MT_{merges}(n) \le (\lfloor \lg n \rfloor + 1) * 2n$.

Now we give a **lower estimate of** $mT_{merges}(n)$**.** Because T is a nearly complete tree, excluding its lowest two levels all the nodes of T are internal nodes. Based on corollary 5.8, all the nodes of T are internal nodes (with MERGE-calls) minimum at levels $0..(\lfloor \lg n \rfloor - 2)$ of T. Considering such a level i, procedure MERGE is called in each node of this level and the whole array A is covered by the disjoint $A[u..v)$ subarrays of the MERGE-calls. Let $mT_{merges(i)}(n)$ be the minimal number of steps of all the MERGE invocations at this level i. Clearly $mT_{merges(i)}(n) \ge$ the sum of the $mT_{\text{MERGE}}(l)$ values of all the MERGE calls at level i (see subsection 5.1). We also know from subsection 5.1 that $mT_{\text{MERGE}}(l) \ge l$ where $l = v-u$. Thus with the distributive rule of addition and multiplication of numbers we have $mT_{merges(i)}(n) \ge n$. Therefore $mT_{merges}(n) \ge (\lfloor \lg n \rfloor - 1) * n$.

5.4. The number of steps of procedure MERGESORT

We finish our calculations on the efficiency of MERGESORT in this section. Based on corollary 5.3 we have $MT(n) = 2n + MT_{merges}(n)$ and $mT(n) = 2n + mT_{merges}(n)$. Thus $MT(n) \le 2n + (\lfloor \lg n \rfloor + 1) * 2n$. Consequently $MT(n) \le 2n * \lg n + 4n$. Similarly $mT(n) \ge 2n + (\lfloor \lg n \rfloor - 1) * n \ge 2n + (\lg n - 2) * n$. Therefore $mT(n) \ge n * \lg n$. Summarizing our results we have

$$n * \lg n \le mT(n) \le MT(n) \le 2n * \lg n + 4n.$$

The first two inequalities and definition 2.2 of $\Omega(g)$ imply $mT(n), MT(n) \in \Omega(n * \lg n)$. We have $\lim_{k\to\infty} 4k/(k * \lg k) = \lim_{k\to\infty} 4/(\lg k) = 0$. Using corollary 2.4 we receive $mT(n), MT(n) \in O(n * \lg n)$. With definition 2.2 of $\Theta(g)$ we can conclude

$$mT(n), MT(n) \in \Theta(n * \lg n).$$

6. Critical note on the notation of recurrence on $T(n)$

As we mentioned in the Abstract of this paper, the runtime complexity of merge sort is traditionally calculated by solving the recurrence [1, 2]

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1, \\ \Theta(1) & \text{if } n = 1, \end{cases}$$

Our problem is that this notation is not well defined. It uses a notation like $f = h + \Theta(g)$ or $f = O(g)$ where $f, g, h : \mathbb{N} \to \mathbb{R}$ are asymptotically non-negative. And in this notation "$=$" means sometimes "$\in$", other times "$\subseteq$" and it may mean even *equality*. For example, $2n+3 = \Theta(n)$ means $2n+3 \in \Theta(n)$, $O(n) = O(n * \lg n)$ means $O(n) \subseteq O(n * \lg n)$ and $\Theta(2n+3) = \Theta(n)$ means that the two sets are equal.

Each time it is used – maybe in many steps of a long proof – one has to decide intuitively about its exact meaning.

This notation is often used, because it makes proofs *shorter*, but it also makes proofs *unclear*, so we believe that it does not serve *purposes of teaching*. A student must be careful not to derive *false* consequences like the following one.

$$1 = O(1) \wedge 2 = O(1) \Rightarrow 1 = 2$$

This is a grotesque case of **abuse of notation**.

We invented an alternative method instead. We analyzed the call-tree of the recursive program, and performed elementary, but mathematically exact calculations.

7. Summary

The calculation of the computational complexity of some recursive algorithms is traditionally based on the *recurrences* like that above. Their meaning may be intuitively clear but it is mathematically unclear. Thus we proposed a calculation on the efficiency of merge sort which is based strictly on the notions introduced and on the analysis of the call-tree of the recursive part of the algorithm.

Further work may go in this direction or in the direction of well defined recursive formulas like in [5, 6].

Acknowledgements. Thanks to my campus (Eötvös Loránd University, Faculty of Informatics) for financial support. And thanks to my student, *Kristóf Umann* for making Figure 1.

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to Algorithms (Third Edition), *The MIT Press* (2009).

[2] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press* (2013).

[3] Goldstine, H.H., Neumann, J. von, Planning and coding of problems for an electronic computing instrument, Part II, Volume 2, reprinted in *John von Neumann Collected Works, Volume V: Design of Computers, Theory of Automata and Numerical Analysis, Pergamon Press, Oxford, England, pp. 152-214.* (1963)

[4] Knuth, Donald, "Section 5.2.4: Sorting by Merging". Sorting and Searching. *The Art of Computer Programming. 3 (2nd ed.). Addison-Wesley. pp. 158–168.* ISBN 0-201-89685-0. (1998).

[5] Neapolitan, Richard E., Foundations of Algorithms (Fifth Edition), *Jones & Bartlett Learning* (2015).

[6] Shaffer, Clifford A.. A Practical Introduction to Data Structures and Algorithm Analysis, Edition 3.1 (C++ Version), *Virginia Tech, Blacksburg* (2011).

[7] Tarján, Róbert Endre, Data Structures and Network Algorithms, *Bell Laboratories* (1983).

[8] Weiss, Mark Allen, Data Structures and Algorithm Analysis, *Addison-Wesley* (2013).