Some comments on the complexity of Mergesort

Zvi M. Kedem

Contents

1 Introduction

2 Our formula and some questions

2.1 Our formula

2.2 Why do we have an upper bound and not a more precise complexity?

2.3 What are the constants $c_1$ and $c_2$?

2.4 Why $\leq$ and not $=$?

2.5 How to solve for $T(n)$?

2.6 What is the complexity and not just an upper bound on it?

2.7 What do we do with $n$ that is not a power of 2?

1 Introduction

We beat Mergesort to death to learn some generally useful concepts.

2 Our formula and some questions

2.1 Our formula

In this writeup we are interested in the complexity of mergesort algorithm. Formally we will derive an upper bound $O$ in order to use our discussion for other algorithms. We will however fix this easily at the end to show that we have in fact $\Theta$. I am covering the material somewhat differently from the textbook.

The complexity of Mergesort is expressed recursively by

$$T(n) \leq \begin{cases} 
  c_1 & \text{for } n = 1 \\
  c_2n + 2T(n/2) & \text{for } n > 1 \text{ that is a power of 2,}
\end{cases} \quad (1)$$
where $c_1 > 0$ and $c_2 > 0$ are some integer constants.

We immediately have some questions:

1. Why do we have an upper bound and not a more precise complexity?
2. What are the constants $c_1$ and $c_2$?
3. Why $\leq$ and not $=$?
4. How to solve for $T(n)$?
5. What is the complexity and not just an upper bound on it?
6. What do we do with $n$ that is not a power of 2?

These questions will be addressed next.

2.2 Why do we have an upper bound and not a more precise complexity?

Let’s remind ourselves of the situation in bubble sort. For a given value of $n$, there could be cases when we make no swaps at all, i.e., when the sequence is properly sorted, or many swaps, when it is badly “unsorted.” So, sometimes a swap takes place and sometimes is does not. Therefore it is not meaningful to compute the precise number of operations that will hold for all input sequences of length $n$. Therefore, we may start with trying to compute an upper bound on the complexity.

Such a “variety” for different sequences of length $n$ may or may not occur for Mergesort, but for full generality let’s assume that this may happen.

2.3 What are the constants $c_1$ and $c_2$?

We know that

1. Some constant number of operations is needed for the case $n = 1$. We use $c_1$ to denote a number that we are sure is at least as big as the number of operations needed in that case.
2. Some number of operations is needed for merging two sorted sequences each of length $n/2$. We know from the description of our procedure for merging (not discussed here) that this uses a number (maybe “the number” in our case but let’s not worry about this) of operations that is linear in $n$. We use $c_2$ to denote a number that we are sure is big enough so that $c_2 n$ is at least as big as the number of operations needed for the merging of the two sequences of length $n/2$ each.

2.4 Why $\leq$ and not $=$?

This follows from the discussion in subsection 2.3.
2.5 How to solve for $T(n)$?

Look at the figure on page 159 of the book. Pretend that $c_1 = 3$ and $c_2 = 7$. The total work obtained there is $7n \log_2 n = 3n$. Translating it back to our constants, the total work is $c_2 n \log_2 n + c_1 n$. However, this is an upper bound because in (1) we have $\leq$ and not $=,$ so we are computing upper bounds and not the actual values. (We could flesh out the discussion a little more, but we will not do it here.) Therefore

$$T(n) \leq c_2 n \log_2 n + c_1 n,$$  \hspace{1cm} (2)

$$T(n) = O(n \log_2 n)$$

Note 1. We use the standard mathematical notation in which (number) in the text refers to the displayed equation with that number. While reading this aloud we replace, e.g., “(1)” by “equation 1.”

2.6 What is the complexity and not just an upper bound on it?

The natural way to proceed is to compute a lower bound on it. From the structure of the algorithm we know that $c_1 \geq 1$ and $c_2 \geq 1$. Therefore, we can write an equation complementary to (1), that is

$$T(n) \geq \begin{cases} 
1 & \text{for } n = 1 \\
1 \cdot n + 2T(n/2) & \text{for } n > 1 \text{ that is a power of 2.}
\end{cases}$$

Again, referring to the figure on page 159 of the book, we obtain

$$T(n) \geq n \log_2 n + n.$$  \hspace{1cm} (3)

Combining (2) and (3), we get

$$n \log_2 n + n \leq T(n) \leq c_3 (n \log_2 n + n)$$

for some constant $c_3$, for example $c_3 = \max(c_1, c_2)$.

Therefore,

$$T(n) = \Theta(n \log_2 n + n) = \Theta(n \log_2 n).$$

2.7 What do we do with $n$ that is not a power of 2?

In this case, $n/2$ is not an integer. First, we need to specify the algorithm for such a case. This is pretty straightforward. Instead of splitting a sequence of length $n$ into two sequences each of length $n/2$ we split it into two sequences one of length $\lfloor n/2 \rfloor$ (the floor of $n/2$) and one of length $\lceil n/2 \rceil$
(the ceiling of \( n/2 \)). Therefore a general length \( n \) we have.

\[
T(n) \leq \begin{cases} 
  c_1 & \text{for } n = 1 \\
  c_2 n + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) & \text{for } n > 1,
\end{cases}
\]

(4)

Now to computing \( T(n) \),

1. Painful to solve (4), though not impossible. We can try various ways.

2. We could say that solving a bigger problem is not easier than a smaller problem. Therefore we claim that for \( m \) between two powers of 2 the complexity is between the complexities for the two powers. To be more precise, let \( n \) be a power of 2 and \( n < m < 2n \). As \( 2n \) is also a power of 2,

\[
T(n) \leq T(m) \leq T(2n).
\]

(5)

But

\[
T(2n) = \Theta(2n \log_2 2n) = \Theta(n \log_2 n).
\]

(6)

(Why? Because \( 2n \log_2 = 2n(1 + \log_2 n) \leq 3n \log_2 n \) for \( n \geq 2 \).) Therefore from (6) we see that

\[
T(n) = \Theta(n \log_2 n) \quad \text{for any } n > 1.
\]

But this argument is not quite sufficient as in general a bigger problem can be easier to solve than a smaller one.

**Example 1.** We want to determine whether an integer \( n \) is a prime. Is the number \( 2^{74,207,281} - 1 \) a prime? Difficult to answer, in fact this is a number of 22,338,618 digits, which happens to be the largest prime number known. However the larger number \( 2^{74,207,281} \) is, of course, divisible by 2 and therefore not a prime, ◦ Example 1

However, for Mergesort we can show that (5) holds. This can be proved by induction. Instead, we will sketch a different argument using an example, just to avoid some annoying, though simple, notation.

**Example 2.** We will show that

\[
T(4) \leq T(6) \leq T(8).
\]

Roughly speaking, we prove this by showing that a bigger problem can solve a smaller problem with the efficiency of the bigger problem. We are not doing it completely correctly because we want to introduce reduction, but the idea is good.

(a) \( T(4) \leq T(6) \). We need to sort a sequence of length 4, \( x = x_1, x_2, x_3, x_4 \). We sort the
sequence of length 6, \(x_1, x_2, x_3, x_4, \infty, \infty\). This is a sequence of length 6 so use \(T(6)\) operations. Once we have this sequence sorted, we take the first 4 elements to solve the problem of size 4. Therefore a problem of size 4 does not require more operations than a problem of size 4. We showed (not quite correctly) that \(T(4) \leq T(6)\).

(b) \(T(6) \leq T(8)\). Argument similar to the one previous one.