University of Stavanger, Norway

DAT600 - Assignment 1 Report

by Filip B. Gotten & Rasmus Øglænd

Contents

1 Counting the steps

In this task, we have implemented the four sorting algorithms—Insertion Sort, Quicksort, Merge Sort, and Heapsort—in Python. Our primary focus is on counting the steps involved in the non-constant time operations during the execution of these algorithms.

The objective is to gain insights into the dynamics of running time concerning the input size, denoted as variable N. Our emphasis lies in understanding the relationship between the number of steps and the growth of the input size, steering clear of intricate details of constant-time operations.

The counted steps are then compared with the expected steps, calculated by multiplying the algorithm's running time by a unique constant factor c which is found through trial and error. These comparisons enable us to establish the asymptotic equivalence between the implementations and the expected steps, providing a comprehensive assessment of the accuracy of the running time.

Finally, we verify whether the expected steps yield a curve that aligns with the counted steps. Additionally, we investigate a value for c that consistently positions the expected curve ahead of the counted steps curve. Meeting these criteria serves as a confirmation of the correctness of the time complexity for each algorithm.

The code can be found at the following repository: `https://github.com/filigott/DAT600-2024`.

Insertion sort

From Figure 1.1, it's evident that the algorithm adheres to the asymptotic running time of $O(n^2)$. The counted steps, corresponding to different input sizes, exhibit an exponential growth, closely trailing the expected graph generated from n^2c, where $c = 0.22$.

Unlike the insertion sort implementation provided by the teacher, which doubled the sorted list for each new iteration, our input lists were consistently random. This led to an increased count of steps, necessitating the use of a higher value for c to compensate for the differences.

However, the adjustment of c becomes inconsequential in the context of our task. The primary goal here is to affirm the asymptotic shape while identifying any suitable value for c that satisfies the criteria for the expected graph.

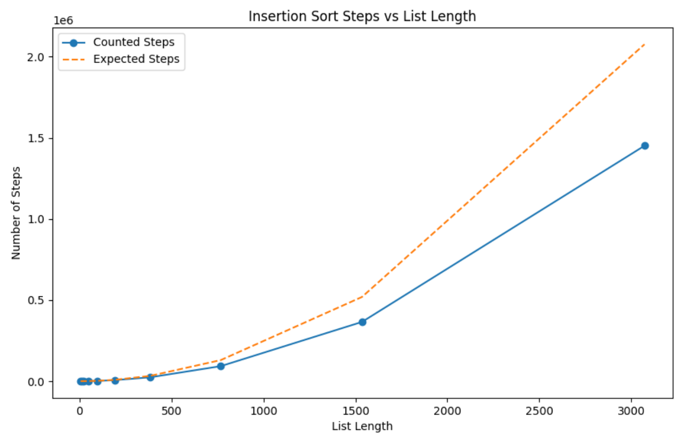

Figure 1.1: Caption

Quicksort

Figure 1.2 reveals that our algorithm adheres to the expected steps curve of $n \log(n)$, exhibiting a non-linear, increasing rate of growth. Notably, this growth is not as rapid as observed in Figure 1.1.

By employing a similar methodology, we identified a value ($c = 1.22$) that aligns with the expected red graph in the figure. This result indicates that the algorithm's performance mirrors the asymptotic shape of the counted steps, with the expected graph consistently positioned slightly ahead.

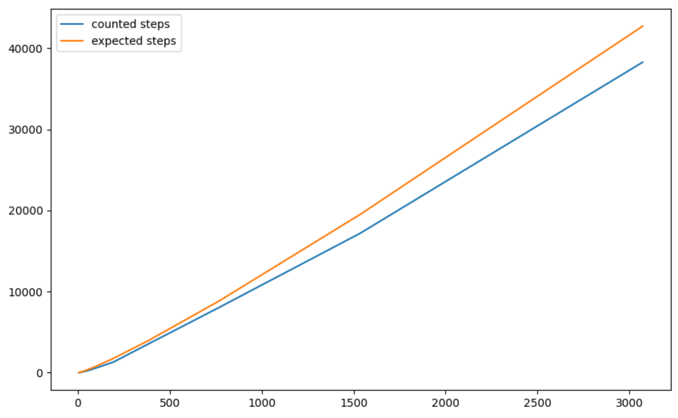

Figure 1.2: Caption

Merge Sort

Figure 1.3 echoes a narrative similar to Quicksort, owing to the anticipated running time of $n\log(n)$ with a value of ($c = 1.05$). Interestingly, we discovered that the counted steps for Merge Sort consistently yielded the same count, aligning precisely with the expected steps when $c = 1$.

This consistency in counted steps can be attributed to the intrinsic behavior of the algorithm and our specific implementation. It also sheds light on potential optimizations, particularly in the merging process, where lists that are already sorted could be leveraged for efficiency. This observation underscores the possibilities for refining the algorithm's implementation.

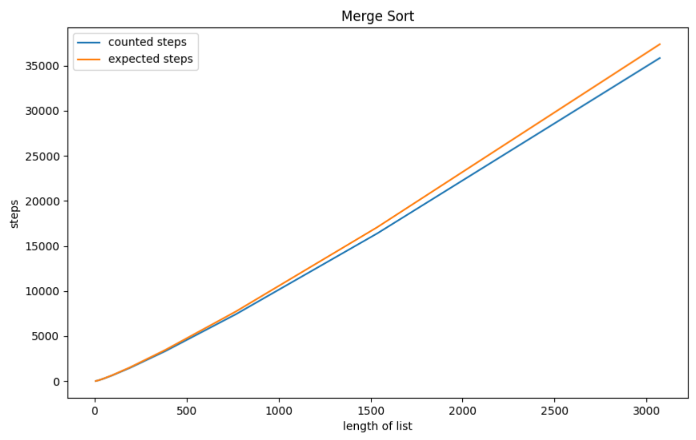

Figure 1.3: Caption

Heapsort

Figure 1.4 once again displays a similar graph to the other sorting algorithms with the expected running time of$n\log(n)$. As we always used random numbers, the graph don't showcase an asymptotic shape more similar to the insertion sort which can happen in the worst case for heap sort. In this case the value found $c = 1.65$.

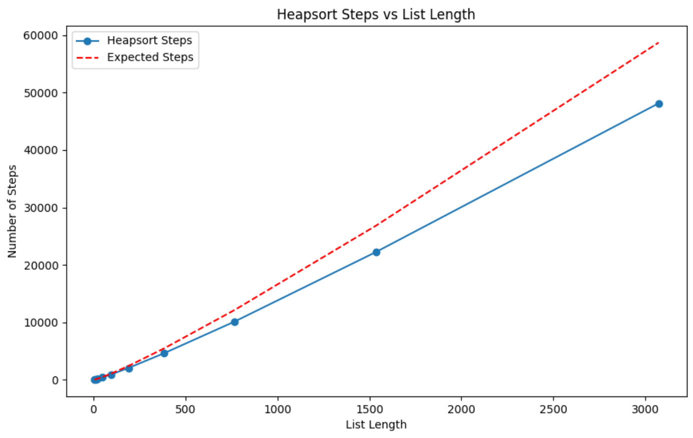

Figure 1.4: Caption

2 Compare true execution time

We decided to measure the execution time for insertion sort because this is the slowest sorting algorithm and the differences in execution time are greatest here. As we can see from the two figures, 2.1, 2.2, Go performs the list sorting much faster than Python.

We should note that every list is randomized, meaning that the Go and Python versions are sorting different lists for each length. However, the time difference in execution time is very noticeable even in this low sample size.

As we can see from the Figures, Go performs insertion sort for approximately 40.000 elements at the same time as Python does for 3000.

Even though execution time and counted steps are two different measurements, we still see similar shapes on the generated graphs, which corroborates the given time complexity of insertion sort.

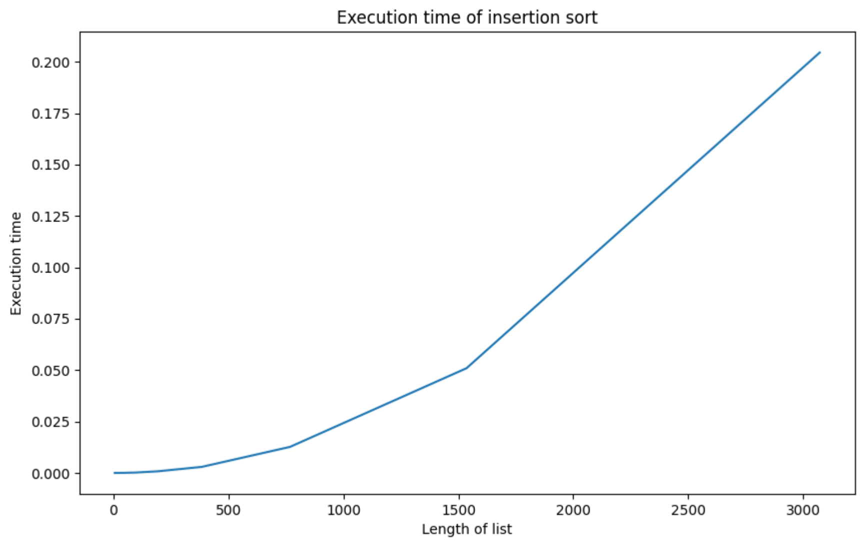

Figure 2.1: Measured execution time for Python implementation with doubling input size

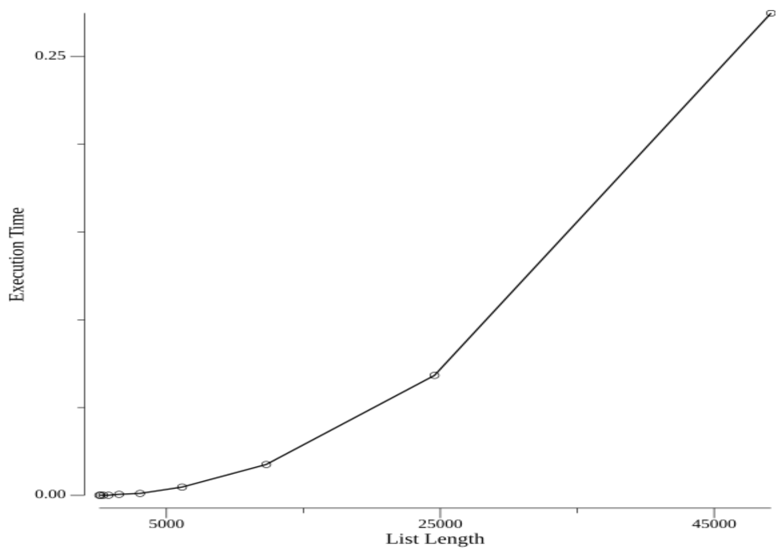

Figure 2.2: Meassured execution time for Go implementation with doubling input size, with more data points to properly showcase the expected graph

3 Basic Proofs and Time Complexity Analysis

Proof 1: $(n+a)^b = \Theta(n^b)$

Proving $(n+a)^b = O(n^b)$:

We want to show that there exist positive constants c and n_0 such that for all $n \geq n_0$, $(n+a)^b \leq c \cdot n^b$.

Let's simplify the expression $(n+a)^b$:

$$\begin{aligned}(n+a)^b &= \left((1+\frac{a}{n})n\right)^b \\ &= n^b\left(1+\frac{a}{n}\right)^b.\end{aligned}$$

Now, since b and a are constants, we can choose n_0 large enough so that $\left|1+\frac{a}{n}\right| \leq 2$ for all $n \geq n_0$. Therefore,

$$(n+a)^b \leq 2^b \cdot n^b$$

So, we can choose $c = 2^b$ and n_0 such that for all $n \geq n_0$, $(n+a)^b \leq c \cdot n^b$. This proves $(n+a)^b = O(n^b)$.

Proving $(n+a)^b = \Omega(n^b)$:

We want to show that there exist positive constants c' and n_0' such that for all $n \geq n_0'$, $(n+a)^b \geq c' \cdot n^b$.

Let's simplify the expression $(n+a)^b$:

$$\begin{aligned}(n+a)^b &= \left((1+\frac{a}{n})n\right)^b \\ &= n^b\left(1+\frac{a}{n}\right)^b.\end{aligned}$$

Now, since b and a are constants, we can choose n_0' large enough so that $\left|1+\frac{a}{n}\right| \geq \frac{1}{2}$

for all $n \geq n_0'$. Therefore,

$$(n+a)^b \geq \left(\frac{1}{2}\right)^b \cdot n^b$$

So, we can choose $c' = \left(\frac{1}{2}\right)^b$ and n_0' such that for all $n \geq n_0'$, $(n+a)^b \geq c' \cdot n^b$. This proves $(n+a)^b = \Omega(n^b)$.

Conclusion:

Since we've shown both $(n+a)^b = O(n^b)$ and $(n+a)^b = \Omega(n^b)$, we can conclude that $(n+a)^b = \Theta(n^b)$.

Proof 2: $\frac{n^2}{\log(n)} = o(n^2)$

To show that $\frac{n^2}{\log_2(n)} = o(n^2)$, we need to prove that the ratio of these two functions approaches zero as n goes to infinity. The definition of $f(n) = o(g(n))$ is that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Let's apply this to the given expression:

$$\lim_{n\to\infty} \frac{\frac{n^2}{\log_2(n)}}{n^2}$$

Canceling out n^2:

$$\lim_{n\to\infty} \frac{1}{\log_2(n)}$$

Now, as n goes to infinity, the fraction approaches zero.

$$\lim_{n\to\infty} \frac{1}{\log_2(n)} = 0$$

Therefore, we have shown that $\frac{n^2}{\log_2(n)} = o(n^2)$.

Proof 3: $n^2 \neq o(n^2)$

To show that n^2 is not $o(n^2)$, we need to demonstrate that the limit of the ratio $\frac{n^2}{n^2}$ does not approach zero as n goes to infinity. The definition of $f(n) = o(g(n))$ is that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Let's apply this to n^2:

$$\lim_{n\to\infty} \frac{n^2}{n^2}$$

Simplify the expression by canceling out n^2:

$$\lim_{n\to\infty} 1$$

The limit is equal to 1, not zero. This means that n^2 is not $o(n^2)$.

4 Divide and Conquer Analysis

Problem Statement

The following recurrence equation gives the running time of Karatsuba's multiplication algorithm:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

The objective is to determine the Big-O notation for the running time using the Master Theorem method and the Recursion Tree method.

Master Theorem Method

To analyze the time complexity by applying the Master Theorem, we have to consider the form of the Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Looking at the given recurrence, we see that in this scenario $a = 3$, $b = 2$, and $f(n) = \Theta(n)$. The subsequent step involves examining the cases outlined by the Master Theorem to determine the specific case that aligns with our scenario. The definitions are as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Which in simpler terms means:

1. In the first case, if the function $f(n)$ is significantly smaller than $n^{\log_b a}$ with a positive gap, the time complexity is $\Theta(n^{\log_b a})$. This implies that the dominating term is $n^{\log_b a}$, and other components are considered relatively "small."
2. In the second case, when $f(n)$ grows at a comparable rate to $n^{\log_b a}$, an additional factor of $\log n$ is introduced. The resulting time complexity becomes $\Theta(n^{\log_b a} \log n)$.

This case accounts for scenarios where both components contribute substantially to the overall complexity.

3. In the third case, if $f(n)$ outgrows $n^{\log_b a}$ with a positive gap, and specific conditions regarding $af\left(\frac{n}{b}\right)$ are met, then $f(n)$ becomes the dominant term. The time complexity is $\Theta(f(n))$, signifying that $f(n)$ governs the overall growth rate.

To check which case matches, we have to determine the following value:

$$\log_b a = \log_2 3 \approx 1.585$$
$$n^{\log_b a} = n^{1.585}$$

Continuing with the Master Theorem analysis:

1. **Case 1:** if $f(n) = O(n^{\log_b a-\epsilon})$

   Let's consider $\epsilon = 0.585$ for this case. We need to check if $f(n) = O(n^{1.585-0.585})$ is true, which equals $O(n^1)$. This is true since the found upper bound is identical to $f(n)$ which also is $O(n^1)$, meaning Case 1 of the Master Theorem applies.

2. **Case 2:** if $f(n) = \Theta(n^{\log_b a})$

   In this case, $f(n) = \Theta(n^{1.585})$ does not match the form $f(n) = \Theta(n^{\log_b a})$, due to $1.585 > 1$ so Case 2 of the Master Theorem does not apply.

3. **Case 3:** if $f(n) = \Omega(n^{\log_b a+\epsilon})$ and $af\left(\frac{n}{b}\right) \leq cf(n)$

   Let's consider $\epsilon = 0.415$. We need to verify if $f(n) = \Omega(n^{1.585+0.415})$ is true, which equals $\Omega(n^2)$. Additionally, we need to check if $3f\left(\frac{n}{2}\right) \leq cf(n)$ for some constant $c > 0$. Considering the given recurrence, $3f\left(\frac{n}{2}\right) = 3 \cdot \Theta\left(\frac{n}{2}\right) = \Theta(n)$. Since $\Theta(n) \leq c\Theta(n)$ does not hold for any constant $c < 1$, Case 3 of the Master Theorem does not apply.

In conclusion, applying the Master Theorem to analyze the recurrence relation $T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$, we find that Case 1 is applicable when setting $\epsilon = 0.585$. The upper bound $O(n^1)$ for $f(n)$ aligns with the given recurrence, establishing Case 1 as the valid choice. In contrast, Cases 2 and 3 do not apply. Hence, the time complexity of Karatsuba's multiplication algorithm is governed by Case 1, yielding $T(n) = \Theta(n^{1.585})$. This implies that the Big O time complexity remains unchanged: $T(n) = O(n^{1.585})$.

Recursion Tree Method

The Recursion Tree method provides a visual representation of the recursive calls in the algorithm, offering insights into its behavior. For the given recurrence relation $T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$, we can construct a recursion tree to illustrate the recursive decomposition of the problem.

Starting with the root node representing $T(n)$, each level of the tree corresponds to a recursive call. In this case, each node branches into three children due to $a = 3$, representing the three recursive calls with the size reduced to $\frac{n}{2}$ due to $b = 2$. The cost associated with each level, represented by $\Theta(n)$, signifies the work done at each recursive step.

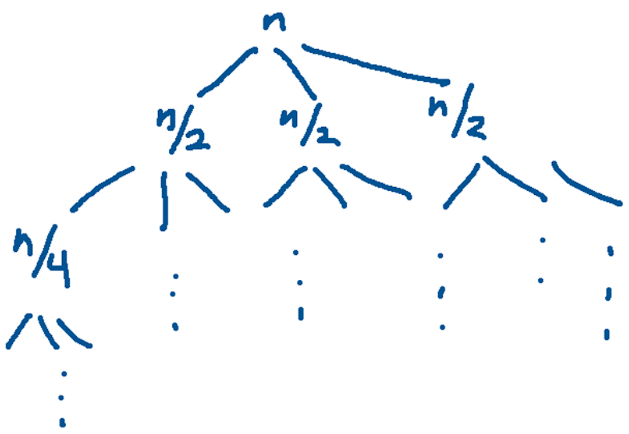

Figure 4.1: Karatsuba's algorithm Recursion Tree

The recursion tree continues until we reach subproblems of size 1, at which point the recursion stops. The total work done is the sum of the costs at each level of the tree.

Analyzing the tree structure, we observe that at each level, the cost is multiplied by a factor of 3. The height of the tree is $\log 2n$, as the problem size is divided by 2 at each level until reaching the base case.

The overall time complexity is determined by summing the costs at each level of the tree. Since each level contributes a factor of 3, and there are $\log_2 n$ levels, the time complexity is $O(3^{\log_2 n})$ which is asymptotically equivalent to $O(n^{\log_2 3})$.

In conclusion, utilizing the Recursion Tree method provides a visual perspective on the recursive structure of the algorithm. The resulting time complexity is consistent with the analysis obtained through the Master Theorem, confirming that $T(n) = \Theta(n^{1.585})$.