DAT600: Algorithm Theory Assignment 1:

By: Erlend Kulander Kvitrud

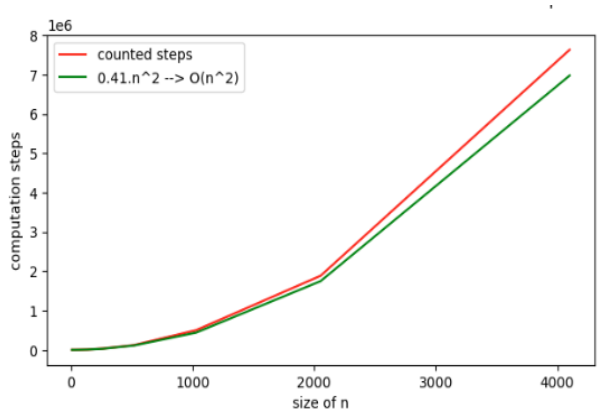
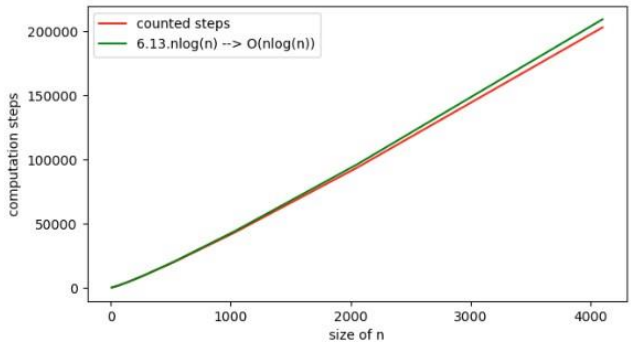
All code referenced in this report can be found on the following repo:
https://github.com/ErlendKK/DAT600-Assignment1

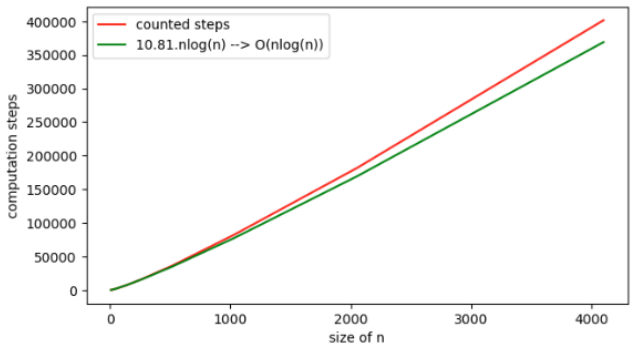
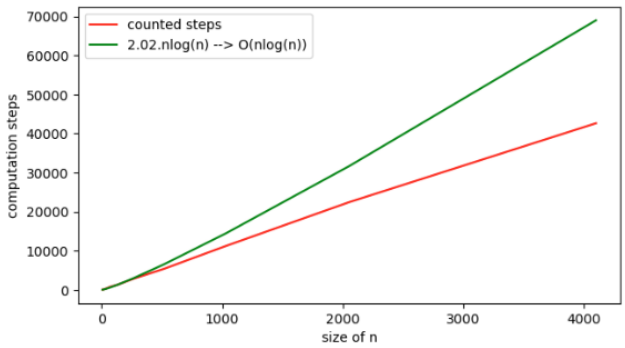
Problem 1

I based my answer to the question on the provided Jupiter Notebook «insertion-sort.ipynb», with some minor modifications. The main modification was to randomize the input arrays for each iteration using «B = [random.randint(1, 10) for _ in range(5 + 2**i)]», to ensure no artificial gains in efficiency stemming from the list being partially sorted at the start of each iteration.

My implementation of quicksort furthermore initially threw a recursionError, and I was only able to make it work for the number of N's I wanted (roughly 4000) by implementing sys.setrecursionlimit(10000). This is generally considered bad practice, as it can potentially cause stack overflow, but - at least on my laptop - it runs fine.

For insertion sort, the algoritm was already written, so I continued using this one. For the other three algorithms, my implementations are based on implementations I learned from the codecademy course «Learn Data Structures and Algorithms with Python».

Insertion sort	Merge sort
	

Heapsort	Quicksort
	

The figures above shows that Insertion sort, Merge sort, and Heapsort matches the references lines well.

For Quicksort, by contrast, the counted number of steps deviate quite a bit from the reference line. This might be caused by the reference being an approximation, which might well overestimate the actual number of required steps. Alternatively, it could be because of optimization that the interpreter is doing behind the scenes. Quicksort is known to be cache-friendly and, the interpreter might be able to leverage this in ways that are hard to predict. It could of course also be because of some mistake I made in my implementation of the algorithm, or a mistake in the way I count steps.

The figures also show that:

- heapsort and merge-sort required roughly the same order of magnitude number of operations
- insertion-sort required roughly an order of magnitude more operations than heapsort and merge-sort
- quicksort requires roughly an order of magnitude fewer operations than heapsort and merge-sort.

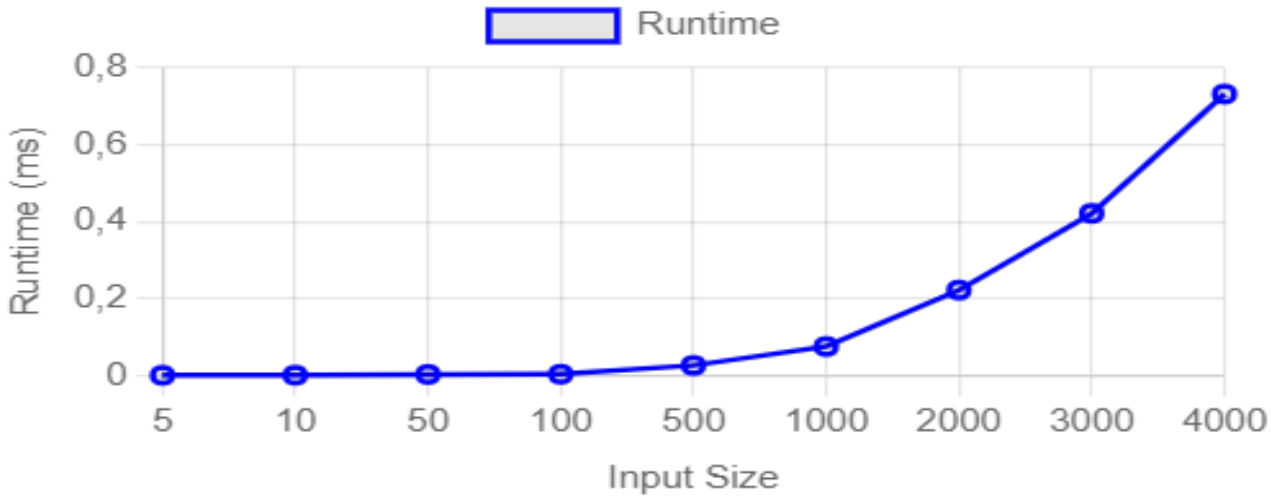
Task 2

I wrote the quicksort algorithm in Javascript and plotted runtime vs input size in canvas using chart.js. All inputs were randomly generated integers ranging from 50 to 4000:

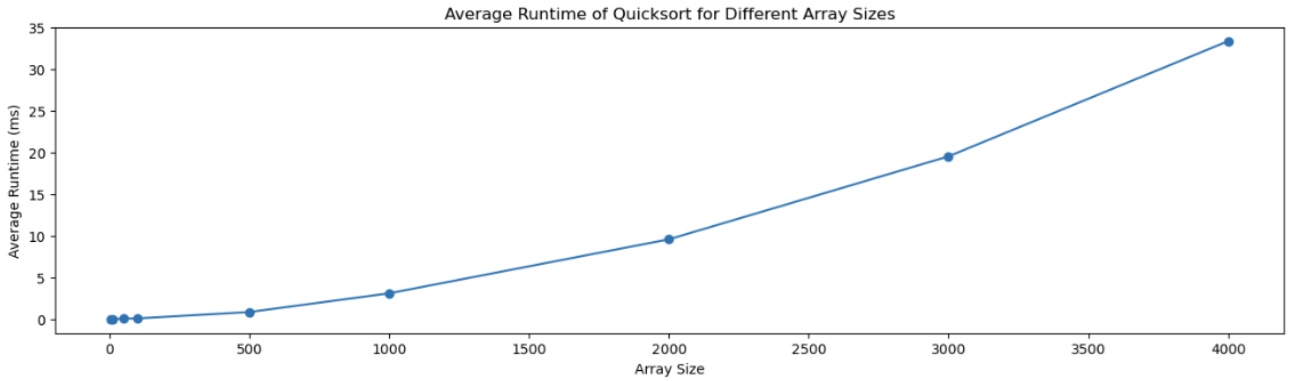
const inputSizes = [5, 10, 50, 100, 500, 1000, 2000, 3000, 4000];

The Javascript code was run in Chrome Version 120.0.6099.201. The python code was run in anaconda 24.1.0

Because actual runtime can vary depending on e.g., whatever else the computer is doing at the moment of testing, individual runtimes can fluctuate wildly. To account for this, I called the algorithm 1000 times for each input size, and logged their average run times. This resulted in the following graph:



For python meanwhile, the results looked like this:



As the graphs show, the average running times in Python was roughly one and a half order of magnitudes greater than those in Javascript. I am not entirely sure why the discrepancy is so great, but one contributing factor could be that, while Python remains an interpreted language, modern browsers run Javascript with JIT compilation and optimizer compilers which can significantly speed up the execution of code. This is particularly true for functions that are called repeatedly, as the JIT compiler can optimize these functions during runtime. Consequently, running the same function with identical-length arrays of integers between 0 and 10 might give misrepresentative picture of how fast this function would run in a browser environment under normal circumstances.

Task 3

3.1 Proposition 1 { (n + a)^b = Θ(n^b) }

We can prove this proposition, by applying Theorem 3.1 from Cormen et al (2022: 56):
f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Omega(g(n)).

In other words, we can show that f(n) = Θ(g(n)) is true by showing that both f(n) = O(g(n)) and f(n) = Ω(g(n)) are true.

In this case; we need to show:

I) $(n + a)^b = O(n^b)$;
II) $(n + a)^b = \Omega(n^b)$;

Condition I is true if there exist constants c_1 and n0 such that $0 <= (n + a)^b <= c_1 * n^b$ for all n >= n0.

Condition II is true if there exist constants c2 and n0 such that $0 <= c2 * n^b <= (n + a)^b$ for all n >= n0.

We prove these conditions by defining c_1 and c_2, in ways that per definition must makes the inequalities become true for some $n > n_0$, and then find the corresponding n_0 by solving the resulting inequality equations for n.

Condition I

For this condition, we can define $c_1 = 2^b$;
$=> c_1 * n^b = 2^b * n^b = (2n)^b$.

The expression (2n)^b will be greater than or equal to $(n + a)^b$ if 2n >= n + a
if a <= 0, this will be true for all n.
If a > 0, this will be true for all n >= a.
In general it will be true for n >= |a|

$=> (n + a)^b = O(n^b)$ and condition I is met

Condition II

For this condition, we can define $c_2 = (1/2)^b$
$=> c_2 * n^b = (1/2)^b * n^b = (n/2)^b$

This expression $(n/2)^b$ will be less than or equal to $(n + a)^b$ if n/2 <= n + a
if a > 0, this is true for all n > 1
if a < 0, this is true for n >= 2 |a|
In general it will be true for n > 2 |a|

$=> (n + a)^b = Omega(n^b)$ and condition I is met

=> The proposition $(n + a)^b = \Theta(n^b)$ is true

3.2 Proposition 2 { $n^2 / lg(n) = o(n^2)$ }

According to Cormen (2022: 60) we can prove any proposition of the form f(n) = o(g(n)) by showing that the 2 following criteria are met:

I) the limit of f(n) / g(n) approaches 0 as n approaches infinity.

II) the functions, f(n) and g(n), must be asymptotically nonnegative.

In this case:
f(n) = n^2 / lg(n)
g(n) = n^2

Condition I:
The expression becomes:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{n^2}{\log(n)}}{n^2}$$

which simplifies into:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1}{\log(n)}$$

which decreases monotonically as n increases and thus approaches 0 as n approaches infinity.

=> the first condition is met

Condition II:
n^2 is nonnegative for every real number, n
=> g(n) is asymptotically nonnegative.

log(n) is nonnegative for any n > 1
=> f(n) must be asymptotically nonnegative.
=> the second condition is met

=> The proposition n^2 / lg(n) = o(n^2) is thus true.

3.3 Proposition 3 { n^2 != o(n^2) }
For the proposition to be true (i.e. for n^2 to be != o(n^2)), we need to show at least one of these conditions to be false:

II) the limit of f(n) / g(n) approaches 0 as n approaches infinity.
II) the functions, f(n) and g(n), must be asymptotically nonnegative.

In this case:
f(n) = g(n) = n^2

Condition II
As shown for the previous proposition, both functions are asymptotically nonnegative.
=> Condition II is met.

Condition I:
The expression becomes:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{n^2}$$

Because both numerator and denominator grow at the same rate; the bound does not approach 0.
=> the first condition is not met.

=> The proposition n^2 != o(n^2) is thus true.

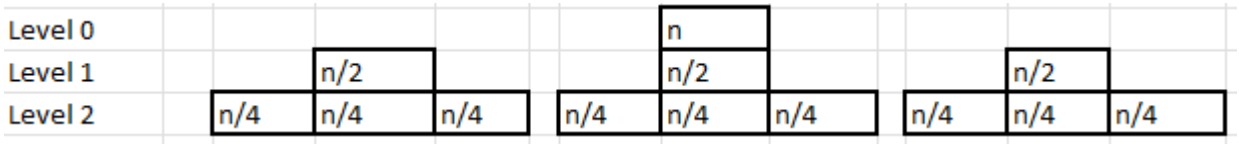
Task 4

This task evaluates the expression "T(n) = 3T(n / 2) + Θ(n)".

The first term, "3", indicates that each recursive call divides the problem into three subproblems. The second term, "T(n/2) indicates that each subproblem has time-complexity = n / 2 (or half that of the previous recursive step). The last term, "Θ(n)", represents the work done outside of the recursive calls.

4.1 Recursion Tree Method:

The following image shows the first levels of the recursive tree (plotted in Excel).



At each level the problem is broken down into 3 subproblems, each half the size of its parent-problem.

At level 0, the cost is equal to Θ(n).
At level 1, the cost is equal to 3Θ(n/2)
At level 2, the cost is equal to 3^2Θ(n/4)

For each subsequent level, the following patterns will hold:

- the factor outside of Θ equals that of level i – 1 times 3 => 3^i
- The factor inside of Θ equals that of level i – 1 divided on 2 => n/ 2^i

We thus see the following overall pattern: For each subsequent level, i, the cost is 3^i times Θ(n/ 2^i)

Since we're dividing the problem size by 2 at each level, this continues for $\log_2(n)$ levels, at which point, the factor inside of Θ equals 1.

We can thus solve the problem directly by expressing the total cost as follows:

$$T(n) = \sum_{i=0}^{\log_2(n)} 3^i \Theta\left(\frac{n}{2^i}\right)$$

Here, the dominant term is $3^{\log2(n)}$ Θ(1), which simplifies to Θ($3^{\log2(n)}$).
Finally we can use formula 3.21 in Cormen et al (2022: 66)

$$a^{\log_b c} = c^{\log_b a}$$

to further simplify this to Θ($n^{\log2(3)}$) which is roughly equal to Θ($n^{1,59}$)

4.2 Master theorem

To apply the master problem, we start by defining the constants a and b such that:

T(n) = aT(b /n) + f(n);

In this case:
a= 3
b = 2
f(n) = Θ(n)

We then compare f(n) to the "watershed function", $n^{logb(a)}$, which in this case is $n^{log2(3)}$. The way this function is used to calculate the total cost of the algorithm depends on which of the following 3 scenarios that is true. As summarized by Cormen et al (2022: 103):

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

In this case, we see that $n^{log2(3)}$ is approximately equal to $n^{1,59}$. It thus follows that there exists a number, epsilon, such that f(n) = O($n^{1,59 - epsilon)}$), namely 0,59 (or to be more exact: log2(3) - 1).

It thus follows that T(n) = $\Theta(n^{log2(3)})$.

This is the same answer as we got solving the problem directly from the recursive tree.

Sources

Cormen, T,H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2022). Introduction to Algorithms, Fourth Edition. The MIT Press. ISBN: 9780262046305