Dat600 Assignment 1

Martin Sævareid Lauritsen, Abel Segaye, Azhar Ahmed

05 February 2026

1 Task 1 counting steps

Note: To better follow along with Task 1 and Task 2 I recommend looking at the jupyter notebook present at the Github there you will also find full the full codebase for this assignment[1]

Task 1 was about implementing the 4 specified sort methods: *Insertion-[2], Merge-[3], Heap-[4] and Quicksort[5].* These sorting methods were implemented and developed with the help of understanding their functionality from their respective GeeksforGeeks topics page.

1.1 Sorting model implementations

Listing 1: Insertion sort implementation

```
def InsertionSort(list):
    stepCount = 0

    for i in range(1, len(list)):
        key = list[i]
        j = i - 1
        stepCount += 1

        while j ≥ 0 and list[j] > key:
            list[j+1] = list[j]
            j -= 1
            stepCount += 1

        list[j+1] = key

    return list, stepCount
```

Listing 2: Merge sort implementation

```
def MergeSort(list):
    # stepCount as an int was not correctly calculated because of ...
        recurssion
    stepCount = [0]

    def merge_sort(list):
        if len(list) ≤ 1:
            return list

        middle = len(list)//2
        left = merge_sort(list[:middle])
        right = merge_sort(list[middle:])

        return merge(left, right)

```

```
    def merge(left, right):
        res = []
        i = j = 0

        while i < len(left) and j < len(right):
            stepCount[0] += 1
            #print(left[i], right[j])
            if left[i] < right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
        res.extend(left[i:])
        res.extend(right[j:])
        return res

    merge_sort(list)
    return list, stepCount[0]
```

Listing 3: Heap sort implementation

```
def HeapSort(list):
    stepCount = [0]

    def heapify(arr, n, i):
        largest = i

        #  left index = 2*i + 1
        lI = 2 * i + 1

        # right index = 2*i + 2
        rI = 2 * i + 2

        # If left child is larger than root
        if lI < n and arr[lI] > arr[largest]:
            largest = lI

        # If right child is larger than largest so far
        if rI < n and arr[rI] > arr[largest]:
            largest = rI

        # If largest is not root
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]  # Swap

            # Recursively heapify the affected sub-tree
            heapify(arr, n, largest)
        stepCount[0] += 1

    n = len(list)

    # Build heap (rearrange array)
    for i in range(n // 2 - 1, -1, -1):
        heapify(list, n, i)
```

```
34
35      # One by one extract an element from heap
36      for i in range(n - 1, 0, -1):
37
38          # Move root to end
39          list[0], list[i] = list[i], list[0]
40
41          # Call max heapify on the reduced heap
42          heapify(list, i, 0)
43          stepCount[0] += 1
44
45      return list, stepCount
```

The heap sort implementation provided is heavily influenced by the GeeksforGeeks implementation [4]

Listing 4: Quick sort implementation

```
1  def QuickSort(list):
2      stepCount = [0]
3      def quick_sort(array):
4          if len(array) ≤ 1:
5              return array
6
7          pivot = array[len(array) // 2]
8          left = [x for x in array if x < pivot]
9          middle = [x for x in array if x == pivot]
10         right = [x for x in array if x > pivot]
11
12         stepCount[0] += len(array) - 1
13
14         return quick_sort(left) + middle + (right)
15
16     quick_sort(list)
17     return list, stepCount[0]
```

1.2 Execution and Visualization

These methods are then ran over a total of 250 iterations with a varying length n. For a deeper look into the execution and plotting I recommend looking at the source code found in the github repo [1]

Looking at figure 1 its pretty obvious that both the red and blue execution follow the general trend that was expected based on the asymptotic running time shown in the task description

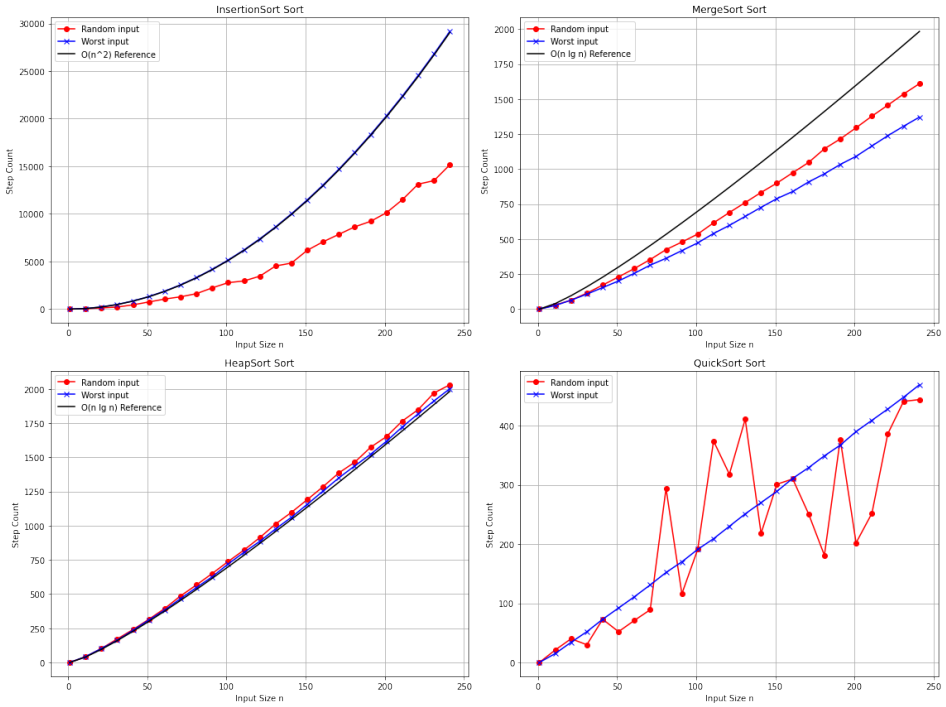


Figure 1: Figure displaying the steps taken by each implementation plotted against a length n

2 Task 2 compare true execution time

The chosen programming languages to implement _Insertion Sort was: *Python* and *Go(lang)* For the python implementation that can be found earlier in subsection 1.1

2.1 Code implementations

Listing 5: Insertion sort implementation in Go

```
func Insertion_sort(arr []int) (array []int) {
    for i := 0; i < len(arr); i++ {
        key := arr[i]
        j := i - 1

        for j ≥ 0 && arr[j] > key {
            arr[j+1] = arr[j]
            j--
        }

```

```
11            arr[j+1] = key
12        }
13        return arr
14    }
```

2.2 Observations

Looking at the differences shown between Go and Python in the table 1 its easy to see that Go is faster, for the run collected from Go the insertion sort true time only reaches about 1ms at n=2000, while in Python's case insertion sort approaches 1ms at n=100 and is at 3.7ms at n=250, while we are still operating below one second this increases fast with time and for process intensive tasks where BigO approaches or even exceeds O^2 Go can provide a significant improvement in true time performance

	n	**Python Time (ms)**	**Go Time (ms)**
0	50	0.1452	0.0178
1	100	0.6638	0.0037
2	250	3.7222	0.0165
3	500	11.4937	0.0608
4	750	18.5840	0.2230
5	1000	28.0561	0.3699
6	2000	98.0868	1.0235
7	5000	609.6821	19.1793

Table 1: Comparison of Python and Go Execution Times

3 Task 3 Basic proofs

Note: This section is taken from my digital notebook. These excerpts were a bit hard to place in the PDF, but can all be found in the report as well as on GitHub. Excerpts belonging to this task are shown in Figure **??**.

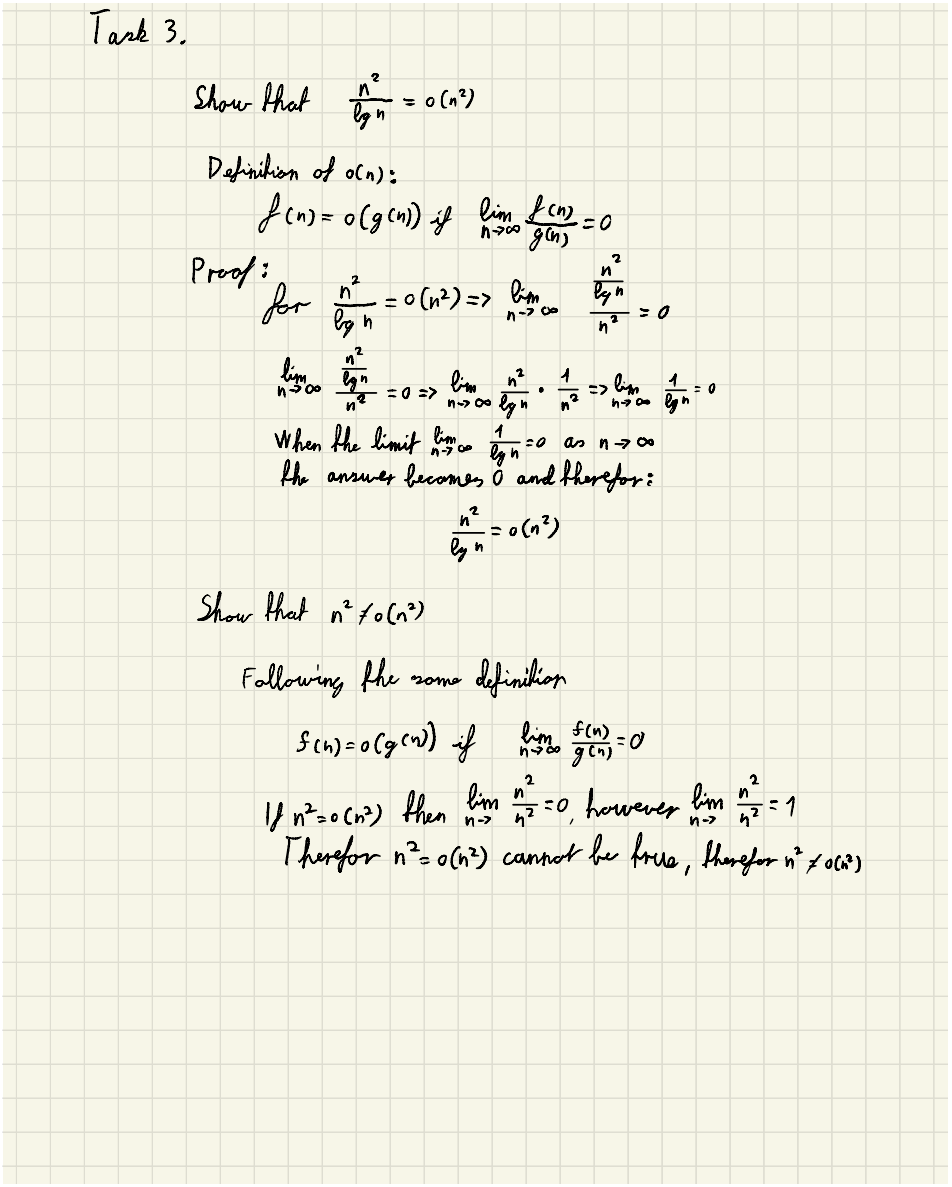


Figure 2: Task 3

4 Task 4 Divide and Conquer analysis

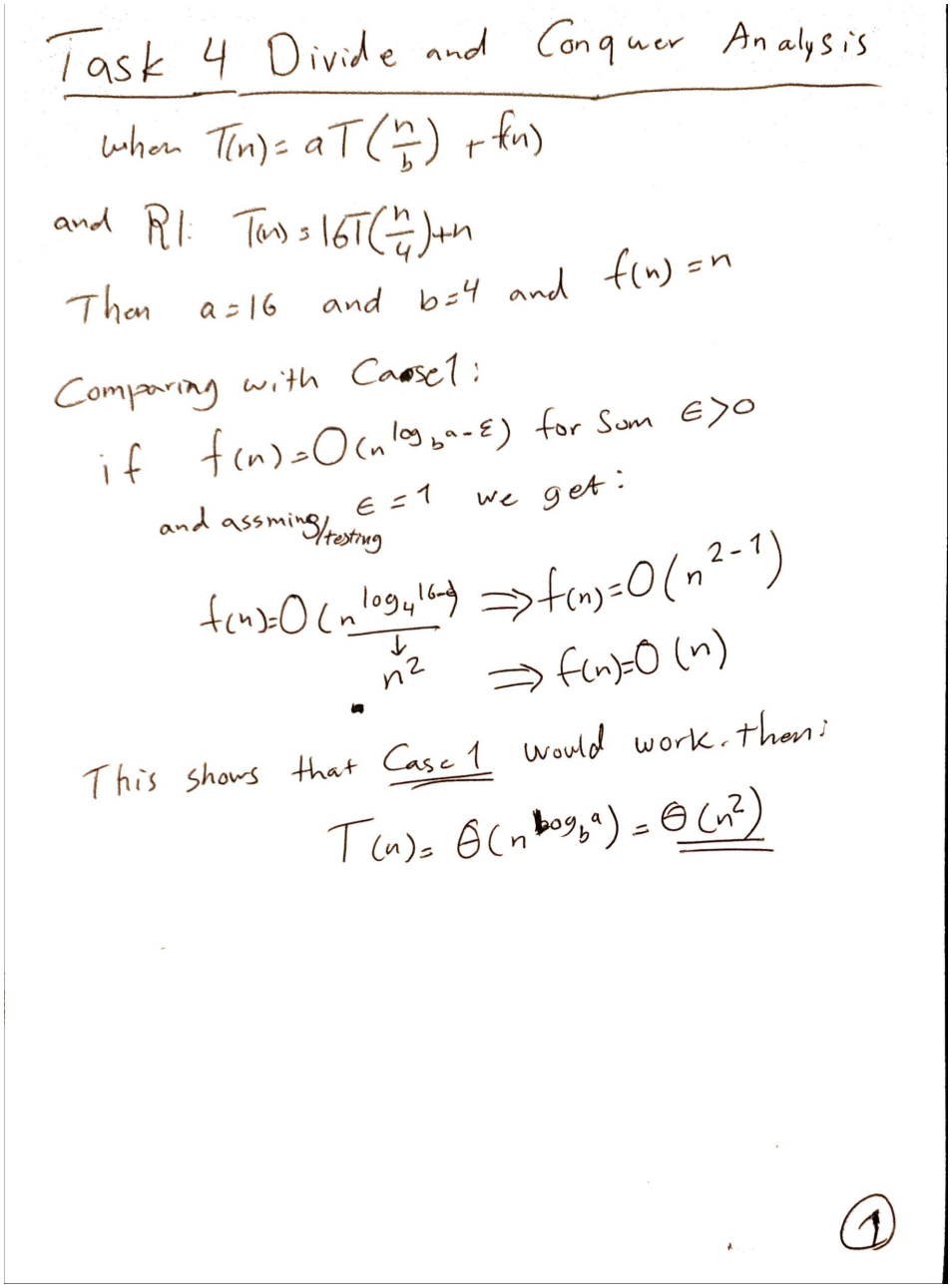
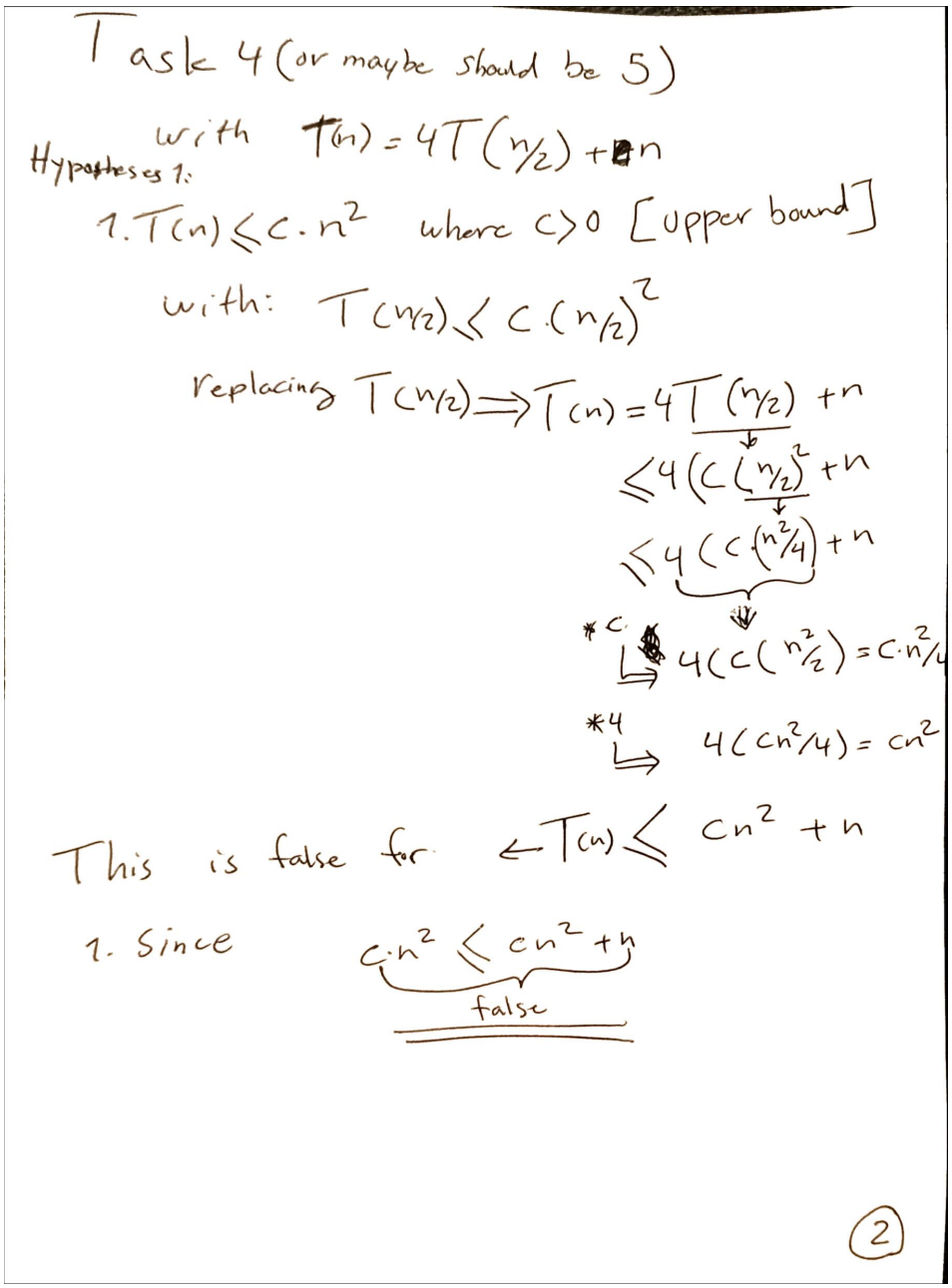


Figure 3: Task 4 calculations

5 Task 5


Task 4 (or maybe should be 5)
with $T(n) = 4T(n/2) + n$
Hypotheses 1:
1. $T(n) \leq c \cdot n^2$ where $c > 0$ [upper bound]
with: $T(n/2) \leq c(n/2)^2$
Replacing $T(n/2) \Rightarrow T(n) = 4T(n/2) + n$
$\leq 4(c(n/2)^2 + n$
$\leq 4(c(n^2/4) + n$
*c $4(c(n^2/2) = c \cdot n^2/4$
*4 $4(cn^2/4) = cn^2$
This is false for $\leftarrow T(n) \leq cn^2 + n$
1. Since $c \cdot n^2 \leq cn^2 + n$
false
2

Task 4 (or 5)

Hypothesis 2:

$T(n) \geq cn^2$ where $c > 0$ [lower bound]

$T(n) = 4 \cdot (T(n/2) + 2$

and $T(n/2) \geq c(n/2)^2$

$\Rightarrow T(n) \geq \underbrace{4(c(n/2)^2)}_{cn^2} + n$

$T(n) \geq cn^2 + n$

OR simply

$\underbrace{T(n) \geq cn^2}_{\text{True}}$ and hypothesis 2 works

Hypothesis 3:

$T(n) \leq (cn^2 - b \cdot n)$ where $c > 0$ and $b > 0$

and

$T(n/2) = c(n/2)^2$

$\Rightarrow T(n) = 4T(n/2) + n$

$\leq 4(c(n/2)^2 - b(n/2)) + n$

$= 4\left(\frac{cn^2}{4} - \frac{bn}{2}\right) + n$

$= cn^2 - 2bn + n$

$= cn^2 - (2b - 1)n.$

(3)

$T(n) \leq cn^2 - (2b-1)n \leq (cn^2 - bn)$

Since we can $n = 1$ for proofing:

$$c1^2 - (2b-1)1 \leq c1^2 - b1)$$

Removing Constants:

$$\Rightarrow -(2b-1) \leq -b$$

$$\Rightarrow 2b-1 \geq b$$

$$+b \Rightarrow b \geq 1$$

meaning as long as $b \geq 1$

$T(n) \leq cn^2 - (2b-1)n \leq cn^2 - bn$

is valid and hypothesis 3 also works

(4)

References

[1] Martin Sævareid Lauritsen. "Dat600 Assignment 1". In: (2025). URL: https://github.com/MslRobo/Dat600Repo.

[2] GeeksforGeeks. "Insertion Sort Algorithm". In: (Jan. 2025). URL: https://www.geeksforgeeks.org/insertion-sort-algorithm/.

[3] GeeksforGeeks. "Merge Sort – Data Structure and Algorithms Tutorials". In: (Jan. 2025). URL: https://www.geeksforgeeks.org/merge-sort/.

[4] GeeksforGeeks. "Heap Sort – Data Structures and Algorithms Tutorials". In: (Jan. 2025). URL: https://www.geeksforgeeks.org/heap-sort/.

[5] GeeksforGeeks. "Quick Sort". In: (Feb. 2025). URL: https://www.geeksforgeeks.org/quick-sort-algorithm/.