# Homework #3 — Fall 2021

Jared Dyreson
California State University, Fullerton

November 19, 2021

## Contents

**1 General Information** **2**
1.1 Contributors . . . . . 2
1.2 Repository . . . . . 2

**2 Algorithm 1 — Pattern Sorting** **3**
2.1 Pseudocode . . . . . 3
2.2 Mathematical Analysis . . . . . 3

**3 Algorithm 2 — Merging Arrays** **4**
3.1 Pseudocode . . . . . 4
3.2 Mathematical Analysis . . . . . 4

# 1 General Information

## 1.1 Contributors

1. Jared Dyreson — jareddyreson@csu.fullerton.edu — CWID: 889546529

## 1.2 Repository

The repository for this project can be found here. Please consult the README.md file for more information regarding the execution of the program.

# 2 Algorithm 1 — Pattern Sorting

## 2.1 Pseudocode

```
function swap(a: int, b: int):
    c: int = a
    a = b
    b = c

function pattern_sorting(container: list[int], pattern: list[int]):
    for key in pattern:
        for i in range(0, container.size()):
            for j in range(i, container.size()):
                if(key == container[i]):
                    swap(container[i], container[j])
```

## 2.2 Mathematical Analysis

In this function, we need to rearrange the array based on a pattern that contains at least one element in the aforementioned array. To begin, you need to iterate over all the keys to sort once, which will be linear time $O(n)$. For this routine, you will need a nested for loop to check the current index to all of the neighboring indices. That inner loop takes $O(n^2)$. This sorting algorithm takes directly from bubble sort, where the swap takes place when we meet the criteria. In this instance, the criteria is the left hand side of the array's index being equal to the current value in the pattern. In terms of memory usage, the only other external data container is a single integer in the swap method. Other than that, the program uses no extra space when sorting. We can come to the conclusion that since the outermost loop of $O(n)$ needs to run $O(n^2)$ times per iteration, the running time complexity is $O(n^3)$.

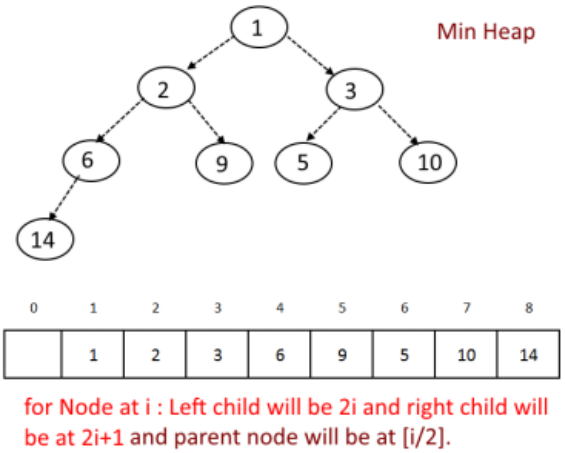
# 3 Algorithm 2 — Merging Arrays

## 3.1 Pseudocode

```
function merge(input: list[list[int]]):
    heap = createHeap(int, list[int], lessThanComp) # O(n)

    # nested loops are O(log(n) * n^2)
    for row in input:
        for element in row:
            heap.insert(element)

    external: list[int] = []  # O(1)

    while heap is not empty: # O(log(n))
        external.append(heap.pop())
```

## 3.2 Mathematical Analysis

In this function, we need to merge a collection of arrays into one large array and the resultant must be sorted in descending order. To achieve this, we are going to employ a min heap and it uses a priority queue with a greater than comparator to sort them. Here is an example of what min heap looks like; the least most element is at the top and removal of the root will have the least most element bubble to the top.




First, creating a min heap will take $O(n)$ time and will use a list to store all the values in the tree. Then, we need to iterate over the collection of arrays that contain the data we need to sort. This takes $O(n^2)$ time because of the way the data is stored. If the data was stored in

a contiguous data structure, this would only take $O(n)$ time but will still result in around the same amount of time as the number of elements hasn't changed. Each insertion inside the min heap will take $O(\log(n))$ time. Lastly, extracting the data from the heap will take $O(\log(n))$ per instance, it should be around $O(n)$ because we're completely emptying the heap. Overall, the time complexity for this function should be $O(\log(n) * n^2)$.