DSA Tutorial Week 4 - Sorting

Find All Source Code on the DSA Tutorial Github: DSA-Tutorial-Exercises

---

Problem 0 - Preparation *!examine before the tutorial!*

Illustrate the execution of **merge-sort** ,**heap-sort** and **quicksort** on an input array containing 1,7,4,6,2,8,3,5. For the quicksort illustration, select the middle element as pivot.

Use a diagram of a tree to clearly visualize the steps of the algorithms. Check your answers on the website https://visualgo.net/en/sorting. By running the input through each of the algorithms on the interactive website.

A. Draw a tree diagram to illustrate the execution of **merge-sort, heap-sort and quick-sort**. You can check your answers using https://visualgo.net/en/sorting. Use the following int array as input: [17,4,6,2,8,3,5]

B. Take another look at the algorithm that you optimized in week 1.
Here is an algorithm that for any given array calculates the sum of its unique elements:

```
// Finds the sum of all unique elements in an array
public static int sumOfUniqueElements(int[] array) {
    int sum = 0;
    for (int i = 0; i < array.length; i++) {
        boolean isUnique = true;
        for (int j = 0; j < array.length; j++) {
            if (i != j && array[i] == array[j]) {
                isUnique = false;
                break;
            }
        }
        if (isUnique) {
            sum += array[i];
        }
    }
    return sum;
}
```

Input: An array of integers (array), e.g.: {-1, 2, -1, 2, 3}
Output: The sum of all unique elements in the array, e.g.: 4 (-1+2+3)
Process: The algorithm iterates through each element of the array (i). For each element, it checks whether this element appears elsewhere in the array (j). If a duplicate is found, the element is not considered unique. If no duplicates are found, the element is added to the sum.

Given what you've learnt this week about sorting, can you apply a new optimization to this algorithm? In that case, what will its worst-case complexity be?

Problem 1 - Heaps

Consider the array **`A = [29, 18, 10, 15, 20, 9, 5, 13, 2, 4, 15]`**.

In order to solve the problems below, it may help to consider drawing this array as a max-heap in the form of a binary tree. For any given node, the left child is located at index ***2 i + 1***. The right child is located at index ***2i + 2***, where ***i*** is the index of the node in the array.

a. Does **`A`** satisfy the *max-heap* property? If not, fix it by swapping two elements.

b. Using array **`A`** (possibly corrected), illustrate the execution of the heap-extract-max algorithm, which extracts the max element and then rearranges the array to satisfy the max-heap property. For each iteration or recursion of the algorithm, write the content of the array **`A`**.

Problem 2 - Pancake Flipping

You are in front of a stack of pancakes of different diameters. Unfortunately, you cannot eat them unless they are sorted according to their size, with the biggest one at the bottom. To sort them, you are given a **spatula** that you can use to split the stack in two parts and then flip the top part of the stack. Write the pseudocode of a function **`sortPancakes`** that sorts the stack.

The *i-th* element of array pancakes contains the diameter of the *i-th* pancake, counting from the bottom. The sortPancakes algorithm can modify the stack only through the **`spatulaFlip`** function whose interface is specified below.

```
// Flips over the stack of pancakes from position pos and returns the result
int[] spatulaFlip(int pos, int[] pancakes);

// Returns all pancakes in sorted order
int[] sortPancakes(int[] pancakes) {
```

1. ***After writing the pseudocode***, implement both the `spatulaFlip` and `sortPancakes` function in a programming language of your choice.
2. What algorithm that you know does the flipping sort resemble?
3. An example implementation can be found in the “answers” directory. Though try to implement it yourself first.
4. Test your implementation with various **arrays of pancake sizes** to ensure it correctly sorts them.

Pointer: Notice that you can move a pancake at **position x** to **position y**, without modifying the positions of the order of the other pancakes, using *a sequence of spatula flips*.

Problem 3 - Divide and Conquer

Given an array **A** and a positive integer **k**, the selection problem amounts to finding the largest element **x** ∈ **A** such that at most **k-1** elements of **A** are less than or equal to **x**, or nil if no such element exists. A simple way to implement it is as follows:

```
SimpleSelection(A, k)
      if k > A.length
            return nil
      else sort A in ascending order
            return A[k-1]
```

Write *another* algorithm named **QuickSelect** that solves the selection problem ***without first sorting*** **A**. Use a divide-and-conquer strategy that “divides” **A** using one of its elements.

Use a divide-and-conquer approach similar to QuickSort, but instead of sorting the entire array, recursively partition the array to find the k-th smallest element.

Also, illustrate the execution of the algorithm on the following input by writing its state at each main iteration or recursion.

```
A = [29, 28, 35, 20, 9, 33, 8, 9, 11, 6, 21, 28, 18, 36, 1]
k = 6
```

After writing the pseudocode for **QuickSelect**, implement it in Java.

Reflection

- Does QuickSelect have a different worst-case time-complexity than QuickSort? How about the average case or expected case?
- Which algorithm will perform the task quicker do you think and why?