Assignment 2 — Divide and Conquer

Due: October 11th

A divide and conquer algorithm divides a problem into smaller subproblems, solves those subproblems, and then combines the resulting subsolutions into a solution to the original problem. Depending on the original problem, this could reduce the number of subproblems that must be solved, or it could merely reorganize the order in which they are solved.

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/oUe8CGsr`

Required Files: `report.pdf`, `compile.sh`, `run.sh`

Optional Files: `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs`

Part 1: Finding Missing Elements

Suppose that you are given a sorted sequence of multiple integers, within which exactly one of the possible integers (that is, the integers between the first and last elements) is missing. For example:

$$\left(\quad 1 \quad , \boxed{2 \quad , \quad 4} \, , \quad 5 \quad , \quad 6 \quad , \quad 7 \quad , \quad 8 \quad \right)$$

Note that, because the sequence is sorted, any successive elements must appear consecutively. As a result, the missing element could be found naïvely in linear time simply by iterating over the elements of the sequence, checking each element to see if it differs by 1 from those immediately before and after.

Part 2: A Binary-Search-Based Approach

Further suppose you know that, within the given sequence, all elements are distinct. Exactly one element within the possible range does not appear; no element appears two or more times. For example:

$$(-2, -1, 0, 1, 3, 4, 5, 6, 7, 8, 9)$$

In the above sequence, the missing element is 2. In this situation, it should be possible to discard subproblems which cannot possibly be missing an element, in a manner similar to that of binary search.

Design and analyze (but do not yet implement) a divide and conquer algorithm to find the missing element within a sorted sequence of multiple integers, where elements must be distinct. Your algorithm should have complexity $O(\log n)$, having avoided a substantial amount of potential work.

Part 3: A Merge-Sort-Based Approach

Now suppose, as a generalization of the above examples, that elements need not be distinct, and could potentially appear two or more times. For example:

$$(-2, -1, 0, 1, 3, 3, 4, 5, 5, 5, 6)$$

In the above sequence, the missing element remains 2. However, in order to find it, it is no longer possible to determine (at least not generally and in constant time) which subproblems must necessarily contain all possible elements. As a result, all subproblems must be solved, in a manner similar to that of merge sort. Such an algorithm would have complexity $O(n)$, having achieved no improvement over a naïve algorithm.

You now have two approaches for solving this problem: one that *sometimes* works in logarithmic time, and one that *always* works in linear time. Recall that when two functions are added together, the larger eventually dominates the sum: $O(\log n) + O(n) = O(n)$. That is to say, there is no asymptotic penalty to simply trying the logarithmic algorithm first[1].

Put another way, the linear algorithm was going to recurse on all subproblems regardless, so there is no penalty to using the logarithmic algorithm's approach in deciding *which* subproblem to recurse on first. If the logarithmic algorithm can find the missing element, you have solved the problem in $O(\log n)$ time. If it cannot, you can fall back on the linear algorithm, thereby solving the problem in a combined $O(n)$ time.

In your programming language of choice (see Assignment 1), implement a divide and conquer algorithm to find the missing element of a given sequence:

- Your implementation must first attempt to solve the problem in binary-search-based logarithmic time.
- Your implementation must resort to merge-sort-based linear time if and only if that first attempt fails.

In order to incorporate the linear approach, your implementation will likely deviate from your logarithmic pseudocode designed in Part 2. You may assume that all sequences will contain multiple comma-separated integers, sorted in ascending order, such that exactly one of the possible integers does not exist as an element.

For example, the second of the above sequences would be represented as:

```
-2, -1, 0, 1, 3, 4, 5, 6, 7, 8, 9
```

Your program must accept as a command line argument the name of a file containing a sequence as described above, then print to `stdout` the missing element.

For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
2
```

Your program will be tested using `diff`, so its printed output must match *exactly*.

Part 4: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `report.pdf` — Pseudocode for an efficient, divide and conquer algorithm to find missing elements as described in Part 2, along with proof of correctness and analysis of complexity.
- `compile.sh` — A Bash script to compile your submission (even if it does nothing), as specified.
- `run.sh` — A Bash script to run your submission, as specified.

The following files are optional:

- `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.rs` — The source code of a working program to find missing elements, as specified.

Any files other than these will be ignored.

[1] Provided that your logarithmic implementation can detect its failure and avoid returning false positives. Although all given sequences will be missing exactly one element, your implementation might recurse on subsequences that are not missing any.