CMPS 6610 Problem Set 03

In this assignment we'll explore further sequence, map, reduce, scan, and divide and conquer algorithms.

To make grading easier, please place all written solutions directly in `answers.md`, rather than scanning in handwritten work or editing this file.

All coding portions should go in `main.py` as usual.

Part 1: Searching Unsorted Lists

As we know, the binary search algorithm takes as input a sorted list of length n and a specified key and is able to find it (or conclude that it is not in the list) in $O(\log n)$ time. Let's consider a slightly different problem in which we are given an unsorted list L with a key x, and we want determine whether x is in L. For each part below, design an algorithm using the prescribed sequence operation. Note that you can preprocess the list as needed.

1a) Use `iterate` to implement the `isearch` stub, and check that your code passes the test cases given by `test_isearch` (feel free to add additional cases).

.
.

1b) What is the work and span of this algorithm?

enter answer in `answers.md`

.
.

1c) Now, use `reduce` to implement the `rsearch` stub. Test it with `test_rsearch`.

.
.

1d) What is the work and span of the resulting algorithm, assuming that `reduce` is implemented as specified in the lecture notes?

enter answer in `answers.md`

.
.

1e) Finally, let's consider another implementation of `reduce` as given by `ureduce` in `main.py`. That is, if you replace `reduce` from part b) with `ureduce` then there should be no difference in output. However, what is the work and span of the resulting algorithm for `rsearch`?

enter answer in `answers.md`

.
.

Part 2: Data Deduplication

In a distributed (i.e. "cloud") setting we may have collections of data that contain duplicates. Service providers want to save space on their storage hardware by storing only unique copies of data elements. Let's design algorithms for this task, both in a single list and in a distributed setting.

2a. List deduplication Suppose you are given a list A of n unsorted elements with duplicates. Design an algorithm and provide a SPARC specification for a function `dedup` that takes A as an argument and returns the distinct elements of A (preserving order). Analyze the work and span of your algorithm.

.
.

2b. Deduplication in a network Imagine now that we have a collection of lists $A_0, \ldots, A_m$, where each list has n elements. In the distributed setting all we care about is identifying the unique elements, without regard to the order in which they appear in the input lists. Design an algorithm and provide a SPARC specification for a function `multi-dedup` that takes A as an argument and returns the distinct elements of A (preserving order). Analyze the work and span of your algorithm and compare it to the work and span from part a) above.

.
.

2c. Sequence operations Are any of our sequence operations useful for either of these problem settings? If so, which operations are useful and why? If not, why do they not help us?

.
.

Part 3: Parenthesis Matching

A common task of compilers is to ensure that parentheses are matched. That is, each open parenthesis is followed at some point by a closed parenthesis. Furthermore, a closed parenthesis can only appear if there is a corresponding open parenthesis before it. So, the following are valid:

- `( ( a ) b )`
- `a () b ( c ( d ) )`

but these are invalid:

- `( ( a )`
- `(a ) ) b (`

Below, we'll solve this problem three different ways, using iterate, scan, and divide and conquer.

3a. iterative solution Implement `parens_match_iterative`, a solution to this problem using the `iterate` function. **Hint**: consider using a single counter variable to keep track of whether there are more open or closed parentheses. How can you update this value while iterating from left to right through the input? What must be true of this value at each step for the parentheses to be matched? To complete this, complete the `parens_update` function and the `parens_match_iterative` function. The `parens_update` function will be called in combination with `iterate` inside `parens_match_iterative`. Test your implementation with `test_parens_match_iterative`.

.
.

3b. What are the recurrences and corresponding asymptotic expressions for the work and span of this solution?

enter answer in `answers.md`

.
.

3c. Using `scan` Implement `parens_match_scan` a solution to this problem using `scan`. **Hint**: We have given you the function `paren_map` which maps `(` to `1`, `)` to `-1` and everything else to `0`. How can you pass this function to `scan` to solve the problem? You may also find the `min_f` function useful here. Implement `parens_match_scan` and test with `test_parens_match_scan`

.
.

3d. Assume that any `map`s are done in parallel, and that we use the most efficient implementation of `scan` (that uses contraction) from class. What are the recurrences for the work and pan of this solution?

enter answer in `answers.md`

.
.

3e. A Divide-and-Conquer Solution** Implement `parens_match_dc_helper`, a divide and conquer solution to the problem. A key observation is that we *cannot* simply solve each subproblem using the above solutions and combine the results. E.g., consider '(()))', which would be split into '(((' and ')))', neither of which is matched. Yet, the whole input is matched. Instead, we'll have to keep track of two numbers: the number of unmatched right parentheses (R), and the number of unmatched left parentheses (L). `parens_match_dc_helper` returns a tuple (R,L). So, if the input is just '(', then `parens_match_dc_helper` returns (0,1), indicating that there is 1 unmatched left parens and 0 unmatched right parens. Analogously, if the input is just ')', then the result should be (1,0). The main difficulty is deciding how to merge the returned values for the two recursive calls. That is, if (i,j) is the result for the left half of the list, and (k,l) is the output of the right half of the list, how can we compute the proper return value (R,L) using only i,j,k,l? Try a few example inputs to guide your solution, then test with `test_parens_match_dc_helper`.

.
.

3f. Assuming any recursive calls are done in parallel, what are the recurrences and corresponding asymptotic expressions for the work and span of this solution?

enter answer in `answers.md`

.
.