CMPS 2200 Assignment 3

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

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. (4pts) 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. (4pts) What is the work and span of this algorithm?

Enter answer in `answers.md`

.
.

1c. (4pts) Now, use `reduce` to implement the `rsearch` stub. Test it with `test_rsearch`.

.
.

1d. (4pts) What is the work and span of the resulting algorithm, assuming that `reduce` is implemented as specified in the lecture notes?

.
.

1e. (4pts) 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`?

.
.

Part 2: Document indexing

A key component of search engines is a data structure called an **inverted index** which maps each word to the list of documents it appears in.

Assume we have three documents with ids 0,1,2:

```
[
    ('document one is cool is it', 0),
    ('document two is also cool', 1),
    ('document three is kinda neat', 2)
]
```

then an inverted index would be

```
[('also', [1]),
 ('cool', [0, 1]),
```

```
 ('document', [0, 1, 2]),
 ('is', [0, 1, 2]),
 ('it', [0]),
 ('kinda', [2]),
 ('neat', [2]),
 ('one', [0]),
 ('three', [2]),
 ('two', [1])]
```

To implement this in map-reduce, we will implement our own map and reduce functions, similar to `recitation-04`.

The map function `doc_index_map` is already complete. E.g.

```
>>> doc_index_map(('document one is cool is it', 0))
    [('document', 0), ('one', 0), ('is', 0), ('cool', 0), ('is', 0), ('it', 0)]
```

The reduce function is also implemented, but it has a bug:

```
>>> doc_index_reduce(['is', [0,0,1,2]])
    ('is', [0,0,1,2])
```

The problem is that document ids are duplicated in the final output (e.g., `0` in the above example).

While of course we could just fix `doc_index_map` to not emit duplicates, we will instead modify the `doc_index_reduce` function. We will do so with the help of another function `dedup` which takes in two sorted, deduplicated lists and returns their concatenation without any duplicates:

```
>>> dedup([1,2,3], [3,4,5])
[1,2,3,4,5]
```

2a. (3pts) Implement `dedup` **in constant time** and test it with `test_dedup`.

2b. (3pts) Modify the `doc_index_reduce` function to use both `dedup` and `reduce`. Test it with `test_doc_index_reduce`.

.
.
.
.
.
.

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. (4pts) 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. (4pts) What are the recurrences for the Work and Span of this solution? What are their Big Oh solutions?

enter answer in `answers.md`

.
.

3c. (4pts) scan solution 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. (4pts) Assume that any `map`s are done in parallel, and that we use the efficient implementation of `scan` from class. What is the Work and Span of this solution? You don't need to give recurrences, but do state the runtimes of the relevant sequences primitives (e.g, map, scan, reduce) to justify your answer.

Enter answer in `answers.md`

.
.

3e. (4pts) 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. E.g., 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. (4pts) Assuming any recursive calls are done in parallel, what are the recurrences for the Work and Span of this solution? What are their Big Oh solutions?

Enter answer in `answers.md`

.
.