Homework 3

Carl Eastlund

Due **Mon., Mar. 25** at **9:00pm**.

Collaboration Policy: Your work on this assignment must be your own. You *may not* copy files from other students in this class, from people outside of the class, from the internet, or from any other source. You *may not* share files with other students in this class.

You *may* discuss the problems, concepts, and general techniques used in this assignment with other students, so long as you do not share actual solutions.

If you are in doubt about what you *may* and *may not* do, ask the course instructor before proceeding. If you violate the collaboration policy, you will receive a zero as your grade for this entire assignment and you will be reported to OSCCR (`northeastern.edu/osccr`).

As usual, provide the code for your solution in `solution.rkt` and provide mathematical arguments in `solution.tex` and `solution.pdf`.

This assignment deals with strings and characters. For programming with strings in Racket, you may find the following functions useful:

- `(string-length str)`, which produces the length of the string `str`.
- `(string-ref str i)`, which produces the `i`th character of `str`, indexed from 0.
- `(string->list str)`, which produces a list of the characters contained in the string `str`.
- `(list->string chars)`, which produces a string from a list of characters `chars`.
- `(char=? a b)`, which reports whether characters `a` and `b` are the same.
- `(char->integer c)`, which produces the Unicode code-point number corresponding to the character `c`.
- `(integer->char i)`, which produces the character corresponding to the Unicode code-point number `i`.

1. Implement `(difference one two)` which reports, for any two strings `one` and `two`, the minimum number of operations that can be used to transform `one` into `two`. The operations to consider are as follows:

   - **delete**, which deletes a single character from `one`
   - **insert**, which inserts a single character into `one`
   - **replace**, which replaces a single character in `one`
   - **swap**, which swaps two adjacent characters in `one`

   Note: Once we insert, replace, or swap a character, we are not allowed to change it again by deleting, replacing, or swapping it. This seemingly arbitrary restriction actually simplifies the solution, because it means once we process part of a string, we never have to backtrack to consider changing it again.

   For example, the difference between `"cat"` and `"bat"` is 1 because it takes a single **replace** operation to replace `"c"` with `"b"`. The difference between `"pot"` and `"top"` is 3 because we may either delete the starting `"p"`, swap `"o"` with `"t"`, and insert a new `"p"` at the end; or we may insert a `"t"` at the front, swap `"p"` with `"o"`, and delete tha `"t"` at the end. The options for turning `"pot"` into `"top"` via a sequence of 3 swaps are not allowed, because we cannot swap any given character in the string more than once.

   (a) Implement a straightforward recursive solution called `difference/recursive`. Analyze its running time. If its running time is exponential or worse, you need only find a lower bound in the sense of Ω.

   (b) Identify the set of recursive subproblems that `difference/recursive` must solve.

   If memoization will improve the running time of your implementation, then write a memoized version of your solution called `difference/memoized` and analyze its running time. Your code will be graded based on the relationship between `difference/memoized` and `difference/recursive`; they should be the same other than the changes needed to implement memoization.

   Otherwise, if memoization will not improve the running time of your implementation, then argue why not.

   (c) Identify the set of recursive subproblems that your solution chooses among at each recursive step.

   If a greedy choice of just one of those subproblems will suffice without recursively considering the others, then write a greedy version of your solution called `difference/greedy` and analyze its running time. Argue that the greedy choice is optimal. Base the greedy version on `difference/memoized` if you have written it, and `difference/recursive` otherwise.

   (d) Define `difference` to call the most efficient version of the algorithm you have written thus far.

2. Implement (shared one two) which reports, for any two strings one and two, the length of their longest common substring. For instance, the longest common substring of "abracadabra" and "bric-a-brac" is "brac", which has length 4.

   (a) Implement a straightforward recursive solution called shared/recursive. Analyze its running time. If its running time is exponential or worse, you need only find a lower bound in the sense of Ω.

   (b) Identify the set of recursive subproblems that shared/recursive must solve.

   If memoization will improve the running time of your implementation, then write a memoized version of your solution called shared/memoized and analyze its running time. Your code will be graded based on the relationship between shared/memoized and shared/recursive; they should be the same other than the changes needed to implement memoization.

   Otherwise, if memoization will not improve the running time of your implementation, then argue why not.

   (c) Identify the set of recursive subproblems that your solution chooses among at each recursive step.

   If a greedy choice of just one of those subproblems will suffice without recursively considering the others, then write a greedy version of your solution called shared/greedy and analyze its running time. Argue that the greedy choice is optimal. Base the greedy version on shared/memoized if you have written it, and shared/recursive otherwise.

   (d) Define shared to call the most efficient version of the algorithm you have written thus far.