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 `"cab"` and `"abc"` is 2 because we can remove `"c"` at the front and insert it again at the end. We cannot use swap twice to achieve the same length—`"c"` with `"a"` and then `"c"` with `"b"`—because we cannot swap the `"c"` more than once. *Note:* The previously posted example with `"top"` and `"pot"` was in error; the difference is 2: replace `"t"` with `"p"` and `"p"` with `"t"`.

   Your solution should traverse the two given strings from left to right, considering one operation at a time and advancing along the strings as the operations use up one and produce the other.

   (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. Once again, your code will be graded on the relationship between `difference/greedy` and its predecessor.

   (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.

   Your solution should consider all possible positions for a shared substring between the two inputs.

   (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. Once again, your code will be graded on the relationship between `shared/greedy` and its predecessor.

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

3. Implement `(compress str)`, which reports the minimum number of bits necessary to represent the characters in the string `str` using a simple compression system. In this system, every character is represented by a unique sequence of bits. These sequences are determined by a binary tree whose leaves are the unique characters used in the string. The path to each character determines its encoding: the sequence contains a 0 bit for every left branch and a 1 bit for every right branch. The function `compress` must account for all possible encodings, meaning all possible binary trees containing the set of characters in the string.

   For example, the minimum number of bits to encode `"tart"` is 6: one bit for `"t"` and two each for `"a"` and `"r"`, meaning a tree with `"t"` on one side and a subtree with `"a"` and `"r"` on the other. The encodings `"t"=0/"a"=10/"r"=11` and `"t"=1/"a"=00/"r"=01` are examples.

   Your solution should start with a sequence of leaves corresponding to the unique characters in the string, and build up a total encoding by repeatedly choosing which of the trees so far to combine into a new node.

   (a) Implement a straightforward recursive solution called `compress/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 `compress/recursive` must solve.

   If memoization will improve the running time of your implementation, then write a memoized version of your solution called `compress/memoized` and analyze its running time. Your code will be graded based on the relationship between `compress/memoized` and `compress/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 `compress/greedy` and analyze its running time. Argue that the greedy choice is optimal. Base the greedy version on `compress/memoized` if you have written it, and `compress/recursive` otherwise. Once again, your code will be graded on the relationship between `shared/greedy` and its predecessor.

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