Assignment: 9
Due: Tuesday, March 27th, 9:00 pm
Language level: Intermediate Student with Lambda
Allowed recursion: No explicit recursion allowed
Files to submit: dangerous.rkt, dangerous-bonus.rkt
Warmup exercises: HtDP Without using explicit recursion: 9.5.2, 9.5.4
Practice exercises: HtDP 20.2.4, 24.3.1, 24.3.2

- Make sure you read the OFFICIAL A09 post on Piazza for the answers to frequently asked questions.
- All helper functions must be defined locally or written as lambda expressions.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
1. Perform the assignment 9 questions using the online evaluation “Stepping Problems” tool linked to the course web page and available at


   The instructions are the same as those in assignment 3; check there for more information if necessary.

2. Sometimes organizations monitor written work for "dangerous speech". In this question you will write a program which will analyse text for dangerous speech.

   For this question you may NOT use direct recursion. Rather, you must use functional abstraction and abstract list functions. Any helper functions must be encapsulated in local. Place your solutions in dangerous.rkt

   (a) Write a function string-has-ci? which consumes two arbitrary strings. It produces true if the first string is contained in the second, using case-insensitive matching, and produces false otherwise.

   For example:

   (check-expect
     (string-has-ci? "tide" "I find Tide Pods make my dishes so clean!") true)

   (check-expect
     (string-has-ci? "tide" "So scary! tidal waves!") false)

   (check-expect
     (string-has-ci? "tide" "Antidepressants for everybody!") true)

   **Hint:** Use build-list in your solution (and other ALFs will be helpful too).

   (b) Write a function generate-flagger which consumes a list of strings that represent dangerous speech, and produces a function. The strings in the list consumed must be non-empty strings. The produced function consumes a symbol (which must be 'detect or 'count-unique) and a string. The produced function has the following behaviour:

   - If the symbol is 'detect, the produced function produces true if the consumed string contains dangerous speech, and false otherwise.

   - If the symbol is 'count-unique, the produced function produces the number of occurrences of distinct elements of the dangerous speech list. It is possible that the appearance of multiple elements from the dangerous speech involve overlapping text in the string consumed by the generated function. The last example demonstrates this where "ar" and "arr" are both counted, but they appear in the same part of the string "Yes, I am a weekend warrior."

   The dangerous speech is tested in a case-insensitive manner. You may assume that all strings in the list of dangerous speech are distinct.
For example:

```racket
(define coop-flagger
  (generate-flagger (list "petition" "CECA" "staff salaries" "co-op fee" "waterlooworks")))
(define staff-flagger
  (generate-flagger (list "union" "revolt" "wages" "justice" "solidarity" "workers")))
(define pirate-flagger
  (generate-flagger (list "plank" "scurvy" "ar" "arr" "matey" "aye" "ye" "swab")))
```

```racket
(check-expect (coop-flagger 'detect "The icecaps are melting!") true)
(check-expect (coop-flagger 'detect "The coop fee is too high") false)
(check-expect (coop-flagger 'detect "It goes to STAFF SALARIES!") true)
(check-expect (staff-flagger 'detect "Tide pods are revolting") true)
(check-expect (coop-flagger 'count-unique "The coop fee is too high!") 0)
(check-expect (staff-flagger 'count-unique "It is time to revolt! Revolt! Our wages are too low!") 2)
(check-expect (pirate-flagger 'count-unique "Yes, I am a weekend warrior.") 3)
```

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

3. **5% Bonus**: Extend `generate-flagger` to produce an additional function. This function consumes two parameters: the symbol `censor` and a string. It produces a function which consumes an additional character. This function produces a string that is identical to the consumed string, except that every instance of dangerous speech in the string is replaced by the character. For example, given the definition of `pirate-flagger` above:

```racket
(check-expect ((pirate-flagger 'censor "Yes, I am a weekend warrior." )
"Yes, I am a weekend warrior.") #\-)
(check-expect ((pirate-flagger 'censor "--s, I am a weekend w---ior." )
"--s, I am a weekend w---ior.")
```

The same restrictions apply as above. Place your solution in `dangerous-bonus.rkt`.
**Enhancements:** Reminder—enhancements are for your interest and are not to be handed in.

Consider the function \((\text{euclid-gcd})\) from slide 7-16. Let \(f_n\) be the \(n\)th Fibonacci number. Show that if \(u = f_{n+1}\) and \(v = f_n\), then \((\text{euclid-gcd} \ u \ v)\) has depth of recursion \(n\). Conversely, show that if \((\text{euclid-gcd} \ u \ v)\) has depth of recursion \(n\), and \(u > v\), then \(u \geq f_{n+1}\) and \(v \geq f_n\). This shows that in the worst case the Euclidean GCD algorithm has depth of recursion proportional to the logarithm of its smaller input, since \(f_n\) is approximately \(\phi^n\), where \(\phi\) is about 1.618.

You can now write functions which implement the RSA encryption method (since Racket supports unbounded integers). In Math 135 you will see fast modular exponentiation (computing \(m^e \mod t\)). For primality testing, you can implement the little Fermat test, which rejects numbers for which \(a^{n-1} \not\equiv 1 \pmod n\), but it lets through some composites. If you want to be sure, you can implement the Solovay–Strassen test. If \(n - 1 = 2^d m\), where \(m\) is odd, then we can compute \(a^m \pmod n\), \(a^{2m} \pmod n\), \(a^{4m} \pmod n\), \ldots , \(a^{n-1} \pmod n\). If this sequence does not contain 1, or if the number which precedes the first 1 in this sequence is not \(-1\), then \(n\) is not prime. If \(n\) is not prime, this test is guaranteed to work for at least half the numbers \(a \in \{1, \ldots , n-1\}\).

Of course, both these tests are probabilistic; you need to choose random \(a\). If you want to run them for a large modulus \(n\), you will have to generate large random integers, and the built-in function \(\text{random}\) only takes arguments up to 4294967087. So there is a bit more work to be done here.

For a real challenge, use Google to find out about the AKS Primality Test, a deterministic polynomial-time algorithm for primality testing, and implement that.

Continuing with the math theme, you can implement the extended Euclidean algorithm: that is, compute integers \(a,b\) such that \(am + bn = \gcd(m,n)\), and the algorithm implicit in the proof of the Chinese Remainder Theorem: that is, given a list \((a_1, \ldots , a_n)\) of residues and a list \((m_1, \ldots , m_n)\) of relatively coprime moduli \((\gcd(m_i,m_j) = 1 \text{ for } 1 \leq i < j \leq n)\), find the unique natural number \(x < m_1 \cdots m_n\) (if it exists) such that \(x \equiv a_i \pmod{m_i}\) for \(i = 1, \ldots , n\).