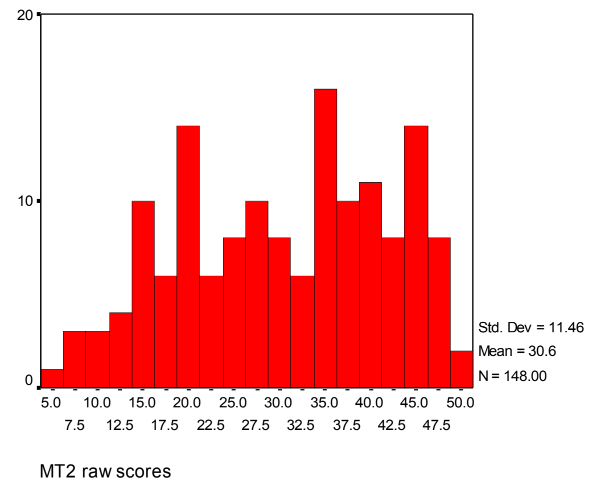
Fall 2005 CS3
Midterm 2
Solutions and Standards

Here is a point breakdown for the exam as a whole, out of 50 points. There were no "perfect" scores, either at the high or the low ends…



Problem (6 / 6 points). Remove-letter

Consider a procedure `remove-letter` that takes two inputs, a letter and a sentence, and returns the sentence with all occurrences of the letter removed. For example:

`(remove-letter 'e '(here is a sentence with e in it)`	➔	`(hr is a sntnc with "" in it)`
`(remove-letter 'e '(not any within))`	➔	`(not any within)`
`(remove-letter 'e '())`	➔	`()`

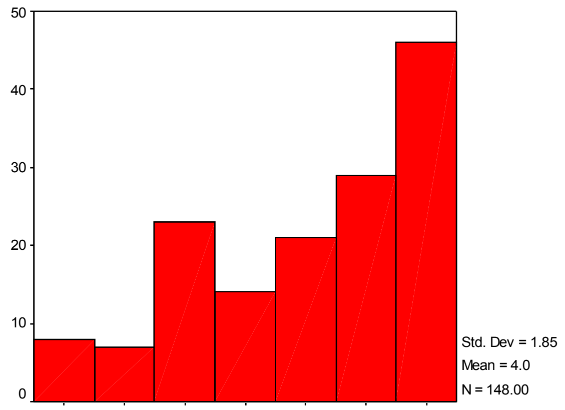
Part A: Write `remove-letter` without using any explicit recursion (i.e., use higher order functions instead)

There were three common solutions:

```
;; solution 1:
(define (remove-letter ltr sent)
   (every (lambda (wd)
                 (remove-letter-from-word ltr wd))
          sent))
(define (remove-letter-from-word ltr wd)
   (keep (lambda (ltr-from-wd)
             (not (equal? ltr ltr-from-wd))
          wd))


;;solution 2:
(define (remove-letter ltr sent)
   (every (lambda (wd)
             (keep (lambda (ltr-from-wd)
                      (not (equal? ltr ltr-from-wd))
                   wd))
          sent))


;;solution 3:
(define (remove-letter-from-word ltr wd)
   (accumulate word
               (every (lambda (ltr-from-wd)
                         (if (equal? ltr ltr-from-wd)
                             ""
                             ltr-from-wd))
                      wd))
```

The scoring rubric was as follows:

- -2 for each missing `lambda`
- -1 for unnecessary code
- -2 for the wrong HOF
- -3 if missing inner function



MT2_1A

Part B: Write `remove-letter` <u>without using higher-order functions</u> (i.e., use recursion instead).

This problem had two nice solutions:

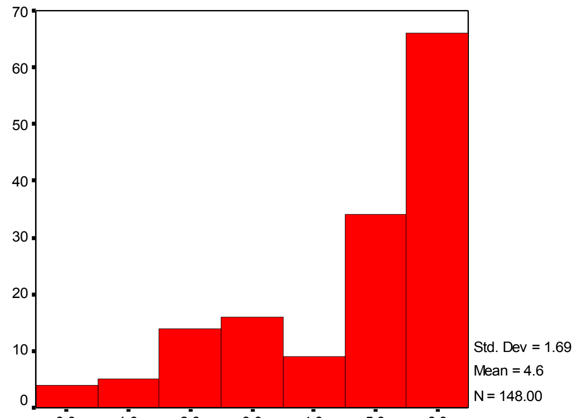
```
;; solution 1:
(define (remove-char char sent-or-word)
```

Instructional Login ______________________

```
  (cond ((empty? sent-or-word) sent-or-word)
        ((sentence? sent-or-word)
         (se (remove-char (first sent-or-word))
             (remove-char (bf sent-or-word))))
        ((equal? (first sent-or-word) char)
         (remove-char (bf sent-or-word)))
        (else (word (first sent-or-word)
                    (remove-char (bf sent-or-word))))))

;; solution 2:
(define (remove-char char sent)
  (if (empty? sent)
      sent
      (se (remove-char-from-word (first sent))
          (remove-char (bf sent)))))
(define (remove-char-from-word char wd)
  (cond ((empty? wd) wd)
        ((equal? (first wd) char)
         (remove-char-from-wd (bf wd)))
        (else (word (first wd)
                    (remove-char-from-word (bf wd))))))
```

The scoring rubric was as follows:

- -½ for using `cond` instead of `if`
- -1 for defining `letter?` to be `equal?`
- -1 for nested `if`s or `cond`s
- -1 for all other style issues (style is capped at -1)
- -2 no recursive call
- -1 for bad base case
- -1 if call `remove-char-from-word` instead of `remove-char`
- -0 for using `member?` in `remove-char` (it is not needed!)
- -1 misc errors



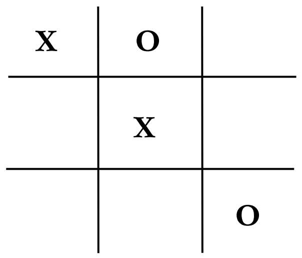
MT2_1B

Instructional Login ____________________

Problem (10 points): Not just a ticky-tack question

In tic-tac-toe, a pivot is an open square that identifies a winning move through the generation of a fork. In `ttt.scm`, the pivot procedure takes a sentence of triples and a player, and returns a sentence of pivots. The code in `ttt.scm` is reproduced in an appendix at the end of this exam.

For the board `b` equal to "x o _ _ x _ _ _ o", for example:



`(pivots (find-triples b) 'x)`	→	`(4 7)`
`(pivots (find-triples b) 'o)`	→	`()`

Rewrite `pivots` without using higher order procedures (i.e., using only recursion). You can use procedures defined in `ttt.scm` as long as those procedures don't use higher order functions. (You may use `appearances`).

Make sure to name your helper procedures and parameters well. You only need to comment when you think it necessary to help explain the intent of your procedure.

Here are some procedures you can use *without* writing them:

`keep-my-singles` takes a sentence of triples and a player and returns a sentence of triples that satisfy `my-single?` (that is, triples with two empty squares and one square filled by the player):

`(keep-my-singles (find-triples b) 'x)`	→	`("4x6" x47 "3x7")`
`(keep-my-singles (find-triples b) 'o)`	→	`("78o" "36o")`

`explode-all` takes a sentence of words and returns a sentence with each word "exploded" into single-letter words:

`(explode-all '(bob joe))`	→	`(b o b j o e)`
`(explode-all '(25o 7o9))`	→	`(2 5 o 7 o 9)`

There were two main ways to solve this question, and both involved easier algorithms than that used in the book. (A few of you tried to duplicate the algorithm in the book, a got into trouble. Remember, chapter 10 in Simply Scheme comes before recursion, and would have changed quite a bit had recursion been used).

Both solutions involved a main helper procedure within which to do the recursion. Several of you didn't do this and ran into trouble when figuring out what to use for triples when making the recursive call.

The first solution uses recourses down the list returned by `explode-all`. Below is a tail-recursive solution (most of you did an embedded solution):

```
(define (pivots triples me)
   (pivots-helper (explode-all (keep-my-singles triples me))
                  '()))

(define (pivots-helper squares current-pivots)
  (cond ((empty? squares) current-pivots)
        ((or (not (number? (first squares)))
             (not (member? (first squares) (bf squares)))
             (member? (first squares) current-pivots))
           (pivots-helper (bf squares) current-pivots))
        (else (pivots-helper (bf squares)
                             (se (current-pivots)
                                 (first squares))))))
```

Note that there are two recursive cases: when a pivot is found or when one isn't. To find a pivot involves checking to see that the current square is a number (to ignore the "`x`"s and "`o`"s that will be in the sentence), checking that the current square appears again later in the sentence, and checking that we haven't already found this square. We took off 2 points each if you didn't do the first or second check correctly. We didn't take off any if you didn't do the last check—technically there was one rare case where pivots-helper would see the same square three times, but it wouldn't affect the rest of the program. With the embedded version of this code, it was very hard to check for this third case!

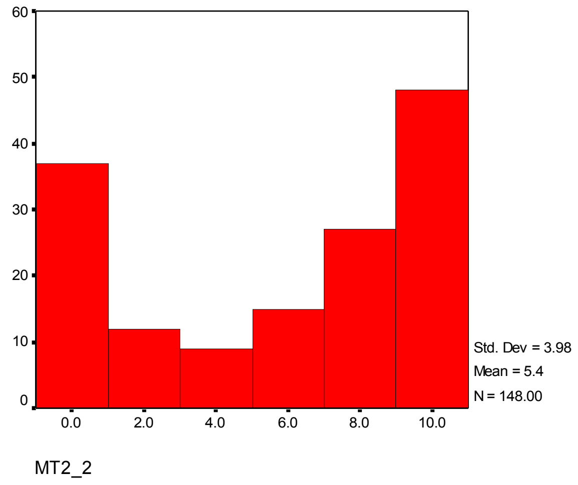
Other points were taken off for not doing the other recursive case or base case correctly, not setting up your helper right, or not calling `explode-all` or `keep-my-singles` or `pivots` itself correctly.

The other main solution, although less common, involved checking each of the squares in the tic-tac-toe board (i.e., 1 through 9) to see if that was a pivot (i.e., to see if it was in the exploded list twice or more times). This could be done embedded (without the drawback above) or tail recursively, or even without recursion. Here is an embedded solution:

```
(define (pivots triples me)
   (pivots-helper (explode-all (keep-my-singles triples me))
                  9))

(define (pivots-helper squares current-square)
  (cond ((<= current-square 0) '())
        ((>= (appearances current-square squares) 2)
         (se (pivots-helper squares (- current-square 1))
             current-square))
        (else (pivots-helper squares (- current-square 1)))))
```

Note that there is no need to check for `number?` here, so the point allocation was somewhat different for this solution.

The shape of the grading results below indicate that most of you either "got it" or "didn't", with little room in between… Those of you that didn't work on the tic-tac-toe materials in lab were certainly at a disadvantage.



Problem (3 / 2 / 4 points): This is random

STk has a procedure `random` which is somewhat different than other procedures you have seen. Each time it is called, it returns a *different* random number. Random takes one argument, which specifies the upper bound on the random number that it will return:

`(random 10)`	➔	6	*`(random 10)` will return a number between 0 and 9. This time it was 6.*
`(random 10)`	➔	0	*Next time it was called, it returned 0. This might have been 6, though, since it is random!*
`(random 10)`	➔	3	*And again…*

Here is a buggy attempt to write a procedure to analyze at a bunch of random numbers:

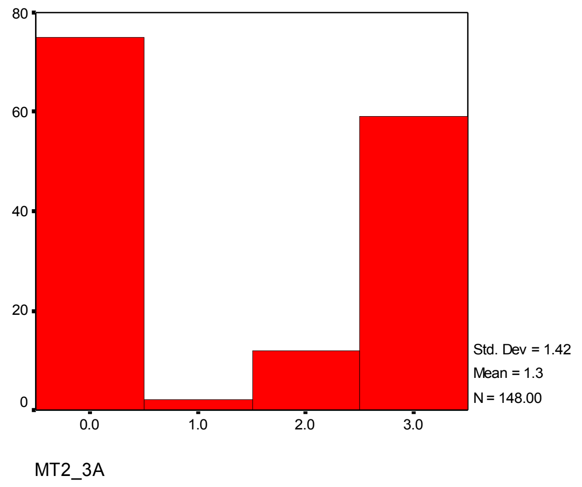
```
;; check random runs (random val) n times, and returns the
;;   minimum and maximum values
(define (check-random val n)
   (check-random-helper val n 0 0))

;; this version doesn't work!  ...because random changes each time
(define (check-random-helper val n cur-min cur-max)
   (if (<= n 0)
       (se cur-min cur-max)
       (check-random-helper
              val
              (- n 1)
              (if (< (random val) cur-min)
                  (random val)
                  cur-min)
              (if (> (random val) cur-max)
                  (random val)
                  cur-max)
              )))
```

Part A (3 points):. This version is buggy because `random` is called several times instead of once per each of the `n` cycles—remember, each call to random may return a different value! Fix check-random-helper *without* defining any additional procedures.

The solution to this problem was to use a `let` statement that sets an initial (single) call to `(random val)` to a local variable:

```
(define (check-random-helper val n cur-min cur-max)
   (if (<= n 0)
      (se cur-min cur-max)
      (let ( (new-random (random val)) )
         (check-random-helper
               val
               (- n 1)
               (if (< new-random cur-min)
                  new-random
                  cur-min)
               (if (> new-random cur-max)
                  new-random
                  cur-max)
               )))
```

This problem was graded, for the most part, as all or nothing. The idea is to recognize the problem, that is we need to get a single value of (random val) and use it multiple times. The only way to do this is a `let` statement. Students who used a `let` generally got full credit, with many losses for improper syntax (generally one point off) or incorrect placing of the let (placing the let inside of the recursive call cost 1.5 points). Remember, a let returns a value like all other statements and this value is the result of evaluating its body. Students who did not use a let recieved no credit.



Part B (2 points). This version is buggy as well because, even with a correct fix to *Part A*, it always returns `0` as the minimum value of the set of random numbers. Fix this for *all possible* cases by modifying the `check-random` procedure (not the helper procedure) below:

The right solution was

```
(check-random-helper val n (- val 1) 0))
```

The important thing to recognize here is that for each of n repetitions, the currently generated random number will be between 0 and (val – 1), and will be compared to `cur-min` and `cur-max` in order to determine whether the current random is the new `cur-min` or `cur-max`.

For instance, `cur-min` will be updated only when the current random value is less than `cur-min`. An initial value of 0 means this will never happen! Starting values of `(- val 1)`, `val` , `(+ 1 val)`, and even `(* val 42)` promise to produce correct results for all cases: basically, anything bigger than (val – 1).

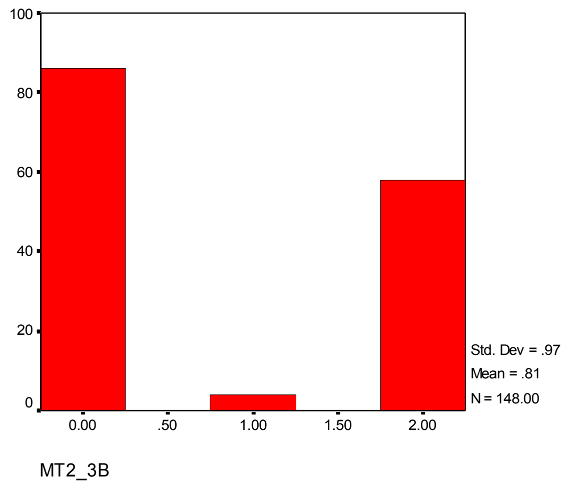
Many students tried

```
 (check-random-helper val n (random val) (random val)))
```

which is either dead wrong or an attempt to simulate the first iteration of the check-random helper. The second case recieved full credit in the following form:

```
;;we will tabulate the results of the first random,
;;check-random-helper need now perform (- n 1) repetitions
(check-random-helper val (- n 1) (random val) (random val)))
```

even though there is now one too many calls to `random`. One point was taken off if you did not sufficiently change the `cur-min` value to fix the problem or if you introduced an error by changing `cur-max`.



Part C (4 points). Write `get-random-elements` which takes a sentence and returns the first `n` elements of the sentence, where `n` is a random number. For instance:

```
(get-random-elements '(this is a fine day)) can return either
        (),
        (this),
        (this is),
        (this is a),
        (this is a fine), or
        (this is a fine day) ,
```

Remember, `(random n)` returns an integer anywhere from `0` up to `n-1`. For instance,

`(random 5)` can return either `0`, `1`, `2`, `3`, or `4`.

You may use either recursion or higher-order procedures. You may use helper procedures.

The most popular approach first gets a random number to decide how many words to keep, and then calls a helper procedure:

```
(define (get-random-elements sent)
   (get-first-elements sent (random (+ 1 (count sent)))))

;;given a sentence (sent) and a number (num),
;; returns a sentence with the first num elements of sent
(define (get-first-elements sent num)
   (if (= num 0)
       '()
       (se (first sent)
       (get-first-elements (bf sent) (- num 1)))))
```

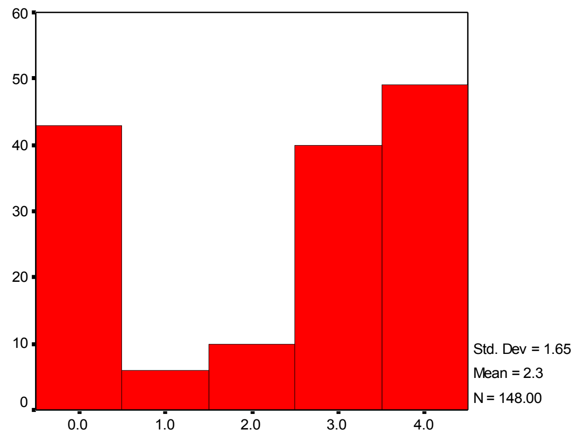
Instructional Login ______________

Another solution might use the higher order procedure `repeated` to remove a random number of words from the end of the sentence:

```
 (define (get-random-elements sent)
    ((repeated bl (random (+ 1 (count sent)))) sent))
```

A common error was being off-by-one. Your solution must allow for returning the empty sentence and the sentence in its entirety. This means that to decide how many words to keep or how many words to throw away, you should consider `0, 1, . . . , (count sent)`. In order to get these possibilities from the random procedure, you must call `(random (+ 1 (count sent)))`. This error cost one point.

Some solutions were EXTRA random. The problem states that you must return the first n elements of the sentence. A procedure that returned some random subset of the sentence lost at least 3 points.



MT2_3C

Problem (3 / 6 points): It was a dark and mysterious recursion…

Consider the recursive procedure `gather` that takes a sentence of at least two single-character words (i.e., letters such as 'a', 'b', etc.):

```
;; sent-of-ltrs is a sentence of at least 2 words that are single
;;   letters
(define (gather sent-of-ltrs)
   (cond ((empty? sent-of-ltrs) '())
         ((empty? (bf sent-of-ltrs))
          (se (first sent-of-ltrs)))
         ((equal? (first (first sent-of-ltrs))
                  (first (bf sent-of-ltrs)))
          (gather (se (word (first sent-of-ltrs)
                            (first (bf sent-of-ltrs)))
                      (bf (bf sent-of-ltrs)))))
```

Instructional Login ________________________

```
        (else
         (se (first sent-of-ltrs)
             (gather (bf sent-of-ltrs))))))
```

Part A (3 points). What will `(gather '(a b b b c d d))` return?

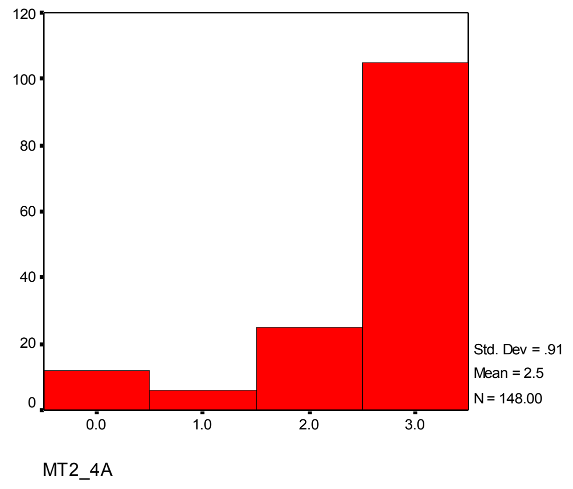
This expression returns `(a bbb c dd)`.

-1: A common mistake was to return `(a bb b c dd)`. Note that in the third case of `gather`, if the first two letters are equal, they are formed into a word and passed into `gather` again. The letters are not extracted until the next letter is different.

-2: Another mistake was to return the argument as is: `(a b b b c d d)`.

-3: No points were given for the answer: `(a b c d)`. No letters from the argument should have been removed.

Some of you put down `'(a bbb c dd)` with a quote in front. No points were taken off for this. When we ask for a return value, make sure you put down whatever `STK` would return, which is *without* the quote.



Part B (6 points). Write `gather-hof`, which behaves the same as `gather` but uses no explicit recursion.

Solution:

```
(define (gather-hof sent)
   (accumulate
       (lambda (new so-far)
           (cond ((not (sentence? so-far))
                  (if (equal? new so-far)
                      (se (word new so-far))
                      (se new so-far)) )
                 ((equal? new (first (first so-far)))
                  (se (word new (first so-far)) (bf so-far)) )
                 (else (se new so-far) )) )
       sent) )
```

-2: If you did not take into consideration that `so-far` could be either a word or a sentence. This is similar to the trick we used in the `diagonal` homework with the `position` procedure.

A few of you did a cool trick where the `so-far` variable is always `sentenced`. This guarantees that `so-far` would consistently be a sentence, eliminating the need to check for it being a word or sentence, or the case of taking the `bf` of a word. In this case, an `if` with two cases would be sufficient:

```
…(lambda (new so-far)
     (if (equal? new (first (first (se so-far))) )
         (se (word new (first so-far)) (bf (se so-far)) )
         (se new so-far) ) )
```

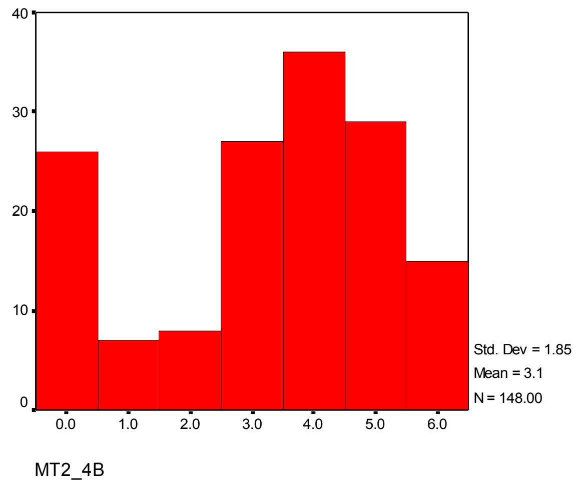
-2: If you did not use `accumulate` correctly.

-1: If the equality check was on only `(first so-far)`. We must use `(first (first so-far))` here. Consider the case of `so-far` being `(bb c d)`: it is just the letter `b` that we want to compare `new` to, not the entire word `bb`. Also, consider the case of `so-far` being `(b c d)`: the letter `b` would still be returned from `(first (first so-far))` because `(first 'b)` is still `b`.

-.5 to -2: If the sentence/return value of `lambda` is not correct. The number of points taken off depended on how far your answer deviated from the correct solution. One point was taken off if the return value was only `(se so-far)`, discarding new if they were not equal. One point was also taken off if `(se (word new (first so-far)))` was returned, forgetting about `(bf so-far)`.

Some of you kept the `cond` (the first two cases) from the given recursion code, and called `accumulate` in the `else` case. This was unnecessary! The problem guarantees that the argument given to `gather` is a sentence of at least two single-character words. No points were taken off for this in this particular problem. However, if you used a `cond`, but did not correctly include the call to `accumulate`, say, put it into `else`, one point was taken off for this.

REMEMBER: `wd` is not the same as `word` A good number of you still make the procedure call (**`wd`** `new so-far)`. Make sure you know that `wd` is **not** a procedure. The procedure you want to use is `word`. SPELL IT OUT! If you were to type it into `STk`, you would get an unbound variable error because `wd` does not exist! No points were taken off for this, but we might in the future!



Problem (9 points): Does money grow on tree recursions?

Consider a set of three coins: a penny, worth 1 cent; a nickle, worth 5 cents; and a dime, worth 10 cents. Write a procedure named `possible-amounts` which takes a number `n`, and returns a sentence of all the possible amounts that any `n` coins of these three types can make. For instance

`(possible-amounts 1)`	➔	`(1 5 10)`
`(possible-amounts 2)`	➔	`(2 6 11 10 15 20)` *(This includes two pennies, a penny and a nickel, a penny and a dime, two nickels, a nickel and a dime, and two dimes)*
`(possible-amounts 3)`	➔	`(3 7 12 11 16 21 15 20 25 30)`

Fill in the blanks to make the definition of `possible-amounts` work correctly:

```
(define *coin-amounts* ______'(1 5 10)__________)

(define (possible-amounts n)
   (pa-helper *coin-amounts* n))

(define (pa-helper coins n)
   (cond ((<= n 1) _______coins_______)             ;; base case 1
         ((empty? coins) ______'()___________)      ;; base case 2
         (else (se (add-coin-to-every                ;; recur case 1
                       (first coins)
                       (pa-helper coins (- n 1)))
                   (pa-helper                        ;; recur case 2
                       __________(bf coins)____________
```

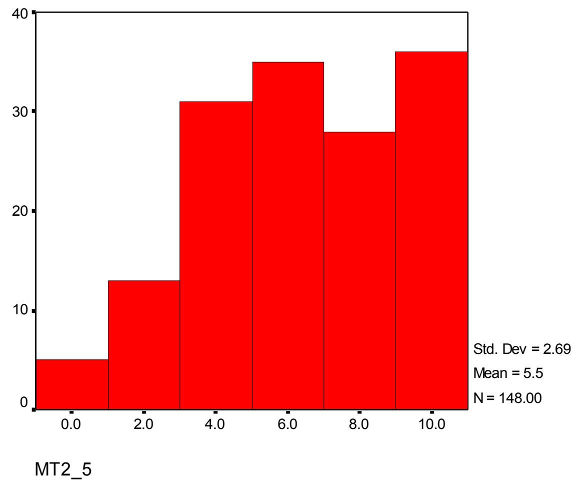
```
                                    ________________n________________ )))))

;; add coin to each element of sent
(define (add-coin-to-every coin sent)
   (every (lambda (num)
               (+ coin num))
         sent))
```

Most of you did well on this problem, which is heartening; this is a rather tricky tree recursion. There are two different recursive cases: the first adds the first coin type to every solution involving this set of coin types with one fewer to use, while the second finds all the solutions where the same number of coins is used but without the first coin type.

Both recursive cases are used each time through, which makes a tree recursion. Since we are counting down both the number of coins to use and the types of coins, there will be base cases for each. When there is only one coin to use, the solution might be any of the available coins. When there are no available coins, no matter how many you are supposed to use, the answer will be the empty set.

You might try playing with this problem in STk to understand it more fully!

The first blank was worth 1 point, the other 4 blanks were worth 2 points each.


Std. Dev = 2.69
Mean = 5.5
N = 148.00
MT2_5