Evidence Packet from Instructor Brian-Thomas Rogers

University of Illinois Springfield | Academic Integrity Council

Academic Integrity Violation Report Form

(see *UIS Academic Integrity Procedures* at http://www.uis.edu/academicintegrity/policy/)

Use this form to report your assessment of the level of the academic integrity violation and the proposed resolution.

Date 12/7/2020 Faculty Name Brian-Thomas Rogers Department Computer Science

Email broge2@uis.edu Phone 217-206-8165 Campus Address UHB 3107

Course CSC 388: Programming Languages Section C Term Fall 2020

Student Name Christian Foy UIN 656531050

REQUIRED FIRST STEP ☑ I have submitted the **Prior Violation Discovery Form** to the Provost's Office and have learned:

☐ student has no prior violation ☑ student has violated previously in manner similar to this new violation

☐ student has violated previously in a manner different from this new violation.

Alleged Policy Violation (see http://www.uis.edu/academicintegrity/policy/ for definitions) Date of violation 12/2/2020

☑ Plagiarism ☐ Misrepresentation ☐ Unauthorized Access ☑ Cheating ☐ Interference ☐ Facilitation

Brief description of violation: [please also attach copies of evidence to this form]

The student submitted a solution to a problem from a different section of the class that I have not given. While there is one similarity between the assignments they are different in very fundamental ways. I have attached both assignments, with the permission of the professor from the other section, to show the differences. I have also attached the hint given by the professor which contains part of the solution to his assignment which is close to what the student submitted as well as a solution to my assignment for comparison.

cfoy.zip - Students Submission
Homework 3_2016.pdf - The assignment description from the other professor.
HW3 Hints_2016.docx - The hint given by the other professor
Assight3_closest.docx - My assignment this semester.
fa2020_teacher_solution.rkt - The solution to my assignment.

RESOLUTION: Pick one. For all Tiers, student is required to review *UIS Academic Integrity Policy* as part of resolution.

Tier 1 Written Warning with Learning Plan See AI Procedures for list of possible learning tasks to address the violation.

To be used when faculty member and student agree that the infraction will be most effectively addressed by the student learning more about academic integrity and how to avoid violations.

☐ I have met with the student and the student has accepted responsibility for the infraction and understands that future violations of academic integrity may result in sanctions. The student and I have agreed on the following learning plan to address the problem [describe here]:

RECEIVED
DEC 07 2020
OFFICE OF
THE PROVOST/VCAA

Tier 2 Agreement with Sanction See AI Procedures for list of possible sanctions.

To be used when student has one or more prior Tier 1 findings of academic integrity violations; or, if the alleged violation appears intentional or is of such severity as to merit a more severe sanction.

☐ I have met with the student and the student has accepted responsibility for the infraction. The student and I have agreed on the following sanction(s) [list here]:

Tier 3 Referral to Hearing

To be used when student has two or more prior findings of an academic integrity violation or when the alleged violation is an act so egregious that it may merit suspension or expulsion from the university. Tier 3 is also used in those circumstances when the faculty and student are unable to come to a common understanding of the student's responsibility involving a lower level (Tier 1 or Tier 2) allegation. Tier 3 may also be requested by the student to contest a charge.

☐ Student Requests a Hearing ☑ Faculty Member Requests a Hearing

Signatures:

Faculty Name: Brian-Thomas Rogers Faculty Signature: Brian Rogers Date 12/7/2020

Student Name: _____ Student Signature: _____ Date _____

STUDENT: By signing this form, you agree to the above action, you acknowledge the UIS Academic Integrity Policy, and you certify that you are fully aware of its contents.

Please send the completed and signed form to the Provost's Office electronically via the PEAR secure email system (go.uis.edu/PEAR) to lslom3@uis.edu, or in a sealed envelope addressed to Academic Integrity, PAC 525 to ensure security and confidentiality of student information. The information from this form will be entered into the Academic Integrity database.

CSC388 Online Programming Languages

Homework 3

(due by midnight on Wednesday, May 4th)

HW3 must be submitted electronically. Your submission must include 2 files:

(a) The source code in R5RS,

(b) 1-2 page description of your algorithm and its complexity as an ASCI text document, MS Word document, or a PDF file. Since the program you have to write is short, the description of the algorithm and the complexity analysis carry significant weight in your grade.

Pack all your files in a zip file. Use the following naming conventions. If your name is John Smith, then your file name must be jsmith.zip. Homeworks which are not properly named or packed will receive 0 points.

Write a function *plus-minus* in Scheme that takes a list L of one or more positive integers (duplicate are possible) and other auxiliary numeric parameters of your choice. Zero is not included in the list. The list L is flat, i.e., it does not contain sublists. The function *plus-minus* returns #t if it is possible to place plus or minus signs in front of each positive integer (including the first one) in the list so that the resulting arithmetic expression equals one of the positive integers in the list L. Otherwise, the function returns #f.

Keep in mind that in order to form a valid arithmetic expression, a plus or a minus sign must be inserted in the front of each positive integer, including the first one. Plus and minus signs must not be inserted between the digits of the integers, i.e., an integer cannot be split into two or more integers. The resulting arithmetic expression must equal one of the original positive integers in the list, i.e., a positive integer without a sign in front of it.

It is up to you to choose the auxiliary parameters that *plus-minus* takes. All auxiliary parameters must be numeric (not lists) and should have initial values set to zero. For example, if L is '(1 2 3) and if you decide to use two additional auxiliary parameters, *plus-minus* must be called as follows:

```
(plus-minus '(1 2 3) 0 0)
```

If there are three auxiliary parameters, then the function must be called:

```
(plus-minus '( 1 2 3) 0 0 0 )
```

and so on.

Note that the function must be called *plus-minus*. Other names will not be accepted. The list must precede the auxiliary parameters, which are initially set to zeroes.

Examples (for the sake of illustration, two auxiliary parameters are used):

```
(plus-minus '(1)0 0)  returns #t.
(plus-minus '(1 2)0 0)  returns #t.
(plus-minus '(1 2 3)0 0)  returns #t.
(plus-minus '(7 1 5 2) 0 0)  returns #t.
(plus-minus '(27 6 12 11)0 0)  returns #f.
(plus-minus '(34 15 17 25 35) 0 0)  returns #f.
```

The whole solution must be packed in one recursive function *plus-minus* which must look as follows:

```
(define plus-minus (lambda (<list L followed by parameters of your
     choice initially set to zero>)
        (cond
          ...
)))
```

In other words, you have to choose your auxiliary parameters and define **ONE COND statement.** Inside COND, you can use ONLY the following constructs:

- `null?`
- `cond`
- `car`
- `cdr`
- `else`
- `+`
- `-`
- `=`
- `#t`
- `#f`
- `plus-minus`
- the names of your parameters
- numeric constants
- parentheses

You cannot use a construct if it is not listed above. The use of built-in functions is not allowed. You cannot define or call any other function with the exception of *plus-minus*. In other words, your code must define and use only one function, *plus-minus,* which must be defined using the constructs listed above.

If your program uses a construct not on the list, then it is not a solution to this homework.

It is not allowed to bypass the homework requirements by packing several functions in one function definition using a numeric flag. In other words, for one value of the numeric flag the function does one thing and for other value of the flag the function does something else.

Here is an example of packing several function using a numeric flag as a second parameter:

```
(define plus-minus (lambda (list flag)
   (cond
      ((= flag 0) the function implements one type of functionality)
      ((= flag 1) the function implements a second type of functionality)
      (else the function implements a third type of functionality )
   )))
```

In other words, the flag is not related to the problem at hand and it used for the sole purpose of packing different functionalities in one function definition. This is not allowed.

Your report MUST include a brief explanation of the auxiliary function parameters.

Plagiarism

Plagiarism is defined as copying or receiving materials from a source or sources and submitting this material as one's own without acknowledging the particular debts to the source (quotations, paraphrases, basic ideas), or otherwise representing the work of another as one's own, is not allowed. Collaboration or group work, usually evidenced by unjustifiable similarity, is not permitted. The homework will be subject to screening by software programs designed to detect evidence of plagiarism or collaboration. Keep in mind that posting a message (including anonymous messages) about the homework on the Internet (blog, software development forums, etc.) is plagiarism. I will be monitoring the Web for the next couple of weeks and taking actions if such a message is posted.

Any student (or a group of students) accused of a violation of academic integrity will receive an F for the course.

While HW2 enumerates all possible sums, HW3 must enumerate all possible sums and all possible targets. In Java, this can also be done by adding few more recursive lines to the Java recursive method, which I posted previously in the solution to HW2. Since our goal is to solve the problem with one function in Scheme, try to stay away from Java and focus entirely on Scheme. Functional programming requires a new way of thinking, which is different from imperative programming. This is why simulating imperative programming in Scheme is not a good idea. Moreover, it can confuse you further. Try to think in terms of functions and try to develop basic skills for working with functions. Such skills are indefensible not only in functional programming but also in all other languages.

Half of your code in HW3 comes from HW2. Your first task is to translate the recursive method from the solution to HW2 into Scheme. It is easier and more convenient to use the shorter version of the recursive method:

```
public static boolean recursiveSplit(Node current, int sum, int target){
                if (current == null) return target == sum;
                return( (recursiveSplit(current.next, sum + current.data, target)) ||
                        (recursiveSplit(current.next, sum - current.data, target)));
        }
```

Make sure that you translate the code above correctly to Scheme. You can test it by calling the function on short lists for different possible targets in the list. Sum must be initially set to zero. The correct translation to Scheme requires that you understand the logic of the code and know how COND, CAR, CDR, and = work.

The main idea of the code above is that it makes two recursive calls, one after another. The function succeeds if one of the recursive calls succeeds. Since you cannot use OR in Scheme, you have to make the recursive calls one after another. Make the first recursive call. If it succeeds, the function succeeds. Otherwise, make the second recursive call. If it succeeds, the function succeeds. If the second recursive call fails, the function fails.

Once you translate this to Scheme, you will be very close to HW3 solution. In other words, your HW3 function takes three parameters: the list, the sum, and the target. Initially, the sum and the target are set to zero.

A couple of students asked if they can use COND statements inside the main COND statement. The answer is yes, you can use as many COND statements as you want in the main COND statement. According to the homework description:
"Inside COND, you can use ONLY the following constructs:
- `null?`
- cond
..."
That is, nested COND's are allowed.

The purpose of HW2 was to reinforce your knowledge of (or to teach you) recursive enumeration. HW3 asks you to apply recursive enumeration again (to enumerate all possible targets this time). If you understand the bigger idea behind HW2, you should not have any problems with HW3.

Solving HW2 and HW3 is an exercise in programming logic. Once you figure out the programming logic, the coding is simple.

My advice is to read carefully once again the tips and the solution to HW2. If you read them, you will see that recursive enumeration works by:
- reading all elements from the list recursively, and
- exploring all possibilities for each element.

I am going to iterate once again on these two concepts. What does it mean to read all elements from the list recursively? It means that you need to read the first element, (car list), and make a recursive call on the rest of the list, (cdr list). In other words, you do not need to process each and every element from the list. It suffices to tell Scheme how to process the first element, and Scheme will recursively process all remaining elements from (cdr list) in the same way. current.data in the Java solution corresponds to (car list) and current.next corresponds to (cdr list). The base case of recursion is when we have read all elements. Then, what remains from the list is the empty list.
What does it mean to explore all possibilities for each element? It means that you need to describe all possible cases for (car list). In HW2, our goal was to enumerate all possible sums, and therefore, there were two cases for each element: to be added to the current sum or to be subtracted from the current sum. Each of these cases was coded as a separate recursive call. One of the recursive calls must succeed for the whole function to succeed. Initially, we start with sum equal to zero and generate an exponential number of cases (2 for each element, 2^N in total), in which each element has been both added to and subtracted from the sum.

What are the possibilities for each element in the enumeration of all possible targets? The answer is simple: each element can be the target or cannot be the target. In other words, the enumeration of all possible targets requires that you make two recursive calls: one in which the current element is the target and a second recursive call in which the current element is not the target. Since we are enumerating both sums and targets, we need four recursive calls in total. One of these four recursive calls needs to succeed for the whole function to succeed.

The function starts with a zero sum and an "empty" target (initially set to zero) and generates all possible combinations of targets and sums by subtracting from and adding each element to the sum, and using each element as a target or not using each element as a target.

Here is the solution to HW2 in Scheme:

```
(define plus-minus(lambda(list sum target)
            (cond
              ((null? list) (= sum target))
              ((plus-minus (cdr list) (+ sum (car list)) target) #t)
              (else (plus-minus (cdr list) (- sum (car list)) target)))))
```

You need to add few more lines to get the solution to HW3.

CSC 388- Programming Languages
Assignment 3

Fall 2020

Assignment 3 must be submitted to Canvas. The code must be in the **R5RS dialect** of Scheme. See "Racket Installation Directions" on Canvas for more details. You will also be required to write an analysis of your solution. You can find more information under the Submission section in this document.

Goal

The purpose of this assignment is to give you exposure to a new language, new paradigm, and new way of thinking about problems. The syntax is simple enough that the focus should be on the problem. The syntax is simpler than most imperative languages with the few rules it does have.

Problem

Write a function **closest** in Scheme that takes a list of at least size 1 of positive integers, duplicates are possible, and a target value that is either 0 or a positive integer, return the closest positive value or, 0 if target is 0, without going over the target after placing either a plus or minus symbol in front of every value. If no such value can be made, then return -1. There may be more than 1 combination which results in the closest value or the value itself. The function should just return a single value.

Suppose the list is (1 2 3) then the possible permutations of placing a plus or minus are

```
+1+2+3 = 6
+1+2-3 = 0
+1-2+3 = 2
+1-2-3 = -4
-1+2+3 = 4
-1+2-3 = -2
-1-2+3 = 0
-1-2-3 = -6
```

It is up to you to choose the auxiliary parameters that **closest** takes. (Auxiliary parameters are parameters used to help find the solution. The list parameter is mandatory and thus not an auxiliary parameter.) All auxiliary parameters must be numeric (not lists) and should have initial values set to zero except for the target. For example, if L is '(1 2 3) and you decide to use two additional auxiliary parameters, **closest** should be called as follows:

```
(closest '(1 2 3) 0 0)
```

Suppose the second auxiliary variable is the target variable and we want to check for a target of 7 then the call would look as follows

```
(closest '(1 2 3) 0 7)
```

CSC 388- Programming Languages
Assignment 3

Fall 2020

If there are three auxiliary parameters, then may be called:

```
(closest '(1 2 3) 0 0 0)
```

The only auxiliary variable which does not need to be 0 is target.

Examples

Example 1: '(1 2 3 4) with target 0. Returns 0. There are a few combinations. One of those combinations is +1-2-3+4 = 0.

It should not matter the order of the values. For instance,

'(3, 1, 4, 2) with target 0 should still return 0. A possible combination is +3-1-4+2 = 0.

Example 2: '(3 4) with target 2 should return 1. The only combination is -3 + 4 = 1.

Example 3: '(7 1 5 9 3) with target 13 should return 13. The only combination is +7-1+9-5+3 = 13.

Example 4: '(6 10 4 14 20 17 7 3 3 1) with target 0 should return -1. There are no combinations which are positive and do not go over 0.

Example 5: '(20 14 15 20 13) with target 5 should return 2. The only combination is -20+14+15-20+13 = 2.

Example 6: '(413 337 228 402 476 480 497) with target 36 should return -1. There are no combinations which are positive and do not go over 36.

Code

The whole solution must be packed into one recursive function called **closest** which must look as follows

```
(define closest (lambda (<list L followed by your auxiliary
parameters>)
                  (cond
                        …your code…
)))
```

In other words, you must choose your auxiliary parameters and define ONE COND statement.

Criteria

You must follow the given criteria. Not doing so will result in heavy point reduction.

First, you must only define the one function. No additional functions may be defined.

Second, you are restricted to the following constructs

- `null?`
- `car`
- `cdr`

CSC 388- Programming Languages
Assignment 3

Fall 2020

- `else`
- `+`
- `-`
- `=`
- `>`
- `>=`
- `<`
- `<=`
- `closest`
- `if`
- `user defined names (for the names of your parameters)`
- `integer literals`
- `parentheses`

You cannot use a construct not listed above. In other words, you cannot call any other function except for **closest** and what is listed above.

Submission

For this assignment, you must submit 2 items.

1. Your R5RS Scheme solution as a .rkt file. You can do this in Dr. Racket with File → "Save Definitions As". Call it closest.rkt.
2. A 1 - 2 page document with a description of your algorithm and its complexity. The document should be either an ASCII text file, MS Word file, or a PDF.

Pack all your files in a zip file. Use the following naming convention. If your name is John Smith, then your file name must be jsmith.zip. Zipped files which are not properly named or packed correctly will receive 0 points.

Plagiarism

Plagiarism is defined as copying or receiving materials from a source or sources and submitting this material as one's own without acknowledging the debts to the source (quotations, paraphrases, basic ideas), or otherwise representing the work of another as one's own, is not allowed. Collaboration or group work, usually evidenced by unjustifiable similarity, is not permitted. The homework will be subject to screening by software programs designed to detect evidence of plagiarism or collaboration. Keep in mind that posting a message (including anonymous messages) about the homework on the Internet (blog, software development forums, etc.) is plagiarism. I will be monitoring the Web for the next couple of weeks and taking actions if such a message is posted.

Any student (or a group of students) accused of a violation of academic integrity will receive an F for the course. You have been warned.

CSC 388- Programming Languages
Assignment 3

Fall 2020

Additionally, you can find the academic honesty policy for the department with this link: https://csc.uis.edu/honesty

As well as the University policy here:

https://www.uis.edu/academicintegrity/policy/

PROGRAMMING LANGUAGES ASSIGNMENT 3

Institutional Affiliation

Instructor

Course code: CSC 388

Date of submission

Algorithm description

This is a recursive algorithm. It solves the problem because it generates all possible Ls (Ls) for all possible targets. The algorithm either recursively adds or subtracts the first element in the current list (i.e., car list) to the L and then either uses or does not use the first element in the current list as the target – against which the L is compared for equality. In other words, it recursively adds and subtracts (car list) from L – thus, performing a depth-first, pre-order traversal on a binary tree – for each list possibility – i.e., using or not using (car list) as a target.

The pseudocode for the algorithm is:

closest (list L target)
;(base case)
if list is empty, test if L equals target
; (recursive cases)
if target equals zero, closest(initialize target with (car list) to prevent false positives)
closest(add (car list) to L but do not use (car list) as target)
closest(subtract (car list) from L but do not use (car list) as target)
closest(add (car list) to L and use (car list) as target)
closest(subtract (car list) from L and use (car list) as target)

For example, if we were to extract a binary tree traversal produced for the list {27 6 12 11} irrespective of target usage and take the target values against which the L is compared for each node:

The table extracted represents the execution of the four main recursive calls:

- (closest (cdr list) (+ L (car list)) target)
- (closest (cdr list) (- L (car list)) target)
- (closest (cdr list) (+ L (car list)) (car list))
- (closest (cdr list) (- L (car list)) (car list))

Sums	2		-20
Targets	27	17	27
	11		11
	12	19	12
	11		11
	6	25	6
	11		11
	12	27	12
	11		11

So, for example, in this part of the table:

- Recursive call (1) is made which produces an L==2 by adding (car list) to L and target==27 by not using (car list) as the target.
- Next, recursive call (2) results in L==-20 by subtracting (car list) from L and target==27 by not using (car list) as the target.
- In call (3), however, we return to adding (car list) to L but now also use (car list) as the target, resulting in L==2 and target==11.
- Finally, in call (4) we subtract (car list) from L and use (car list) as the target, resulting in L==-20 and target==11.

Complexity Analysis

The worst case scenario occurs when no leaf L matches any list element, and as a result, all leaves must be traversed for all elements. Each leaf resulting from exploring the possibilities of adding and subtracting (car list) to or from the L is compared against each leaf resulting from exploring the possibilities of using or not using (car list) as the target. As a result, the Big O notation is O(2n)(2n), which reduces to O(22n), and ultimately to O(2n).

CSC 388 Programming Languages – Homework #3

Camron Khan

Algorithm Description

Pseudocode

plus-minus (list sum target)

;===base case===

if list is empty, test if sum equals target

; ===recursive cases===

if target equals zero, plus-minus(initialize target with (car list) to prevent false positives)

plus-minus(add (car list) to sum but do not use (car list) as target)

plus-minus(subtract (car list) from sum but do not use (car list) as target)

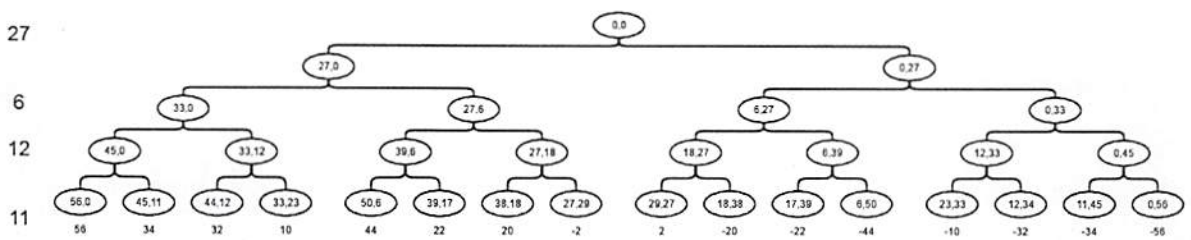
plus-minus(add (car list) to sum and use (car list) as target)

plus-minus(subtract (car list) from sum and use (car list) as target)

Explanation

This algorithm solves the problem because it generates all possible sums for all possible targets. The algorithm either recursively adds or subtracts the first element in the current list (i.e., *car list*) to the sum and then either uses or does not use the first element in the current list as the target – against which the sum is compared for equality. In other words, it recursively adds and subtracts (car list) from sum – thus, performing a depth-first, pre-order traversal on a binary tree – for each list possibility – i.e., using or not using (car list) as a target.

For example, below is the binary tree traversal produced for the list {27 6 12 11} irrespective of target usage:



The above figure illustrates the binary tree traversal for list {27 6 12 11}

And here, you can see the target values against which the sum is compared for each leaf:

The above figure shows the order in which target values are produced as the binary tree is traversed

In the previous table, each numbered block represents the execution of the four main recursive calls:

1. (plus-minus (cdr list) **(+ sum (car list))** **target**)
2. (plus-minus (cdr list) (- sum (car list)) **target**)
3. (plus-minus (cdr list) **(+ sum (car list))** **(car list)**)
4. (plus-minus (cdr list) (- sum (car list)) **(car list)**)

So, for example, in block #17 of the table:

1. Recursive call (1) is made which produces a **sum==2** by adding (car list) to sum and **target==27** by not using (car list) as the target.
2. Next, recursive call (2) results in sum==-20 by subtracting (car list) from sum and **target==27** by not using (car list) as the target.
3. In call (3), however, we return to adding (car list) to sum but now also use (car list) as the target, resulting in **sum==2** and **target==11**.
4. Finally, in call (4) we subtract (car list) from sum and use (car list) as the target, resulting in sum==-20 and **target==11**.

Here is the resulting trace output:

```
>(plus-minus (mcons 27 (mcons 6 (mcons 12 (mcons 11 '()))))) 0 27)
> (plus-minus (mcons 6 (mcons 12 (mcons 11 '())))) 27 27)
> >(plus-minus (mcons 12 (mcons 11 '()))) 33 27)
> > (plus-minus (mcons 11 '()) 45 27)
> > >(plus-minus '() 2 27)
< < <#f
> > >(plus-minus '() -20 27)
< < <#f
> > >(plus-minus '() 2 11)
< < <#f
> > (plus-minus '() -20 11)
< < #f
```

Algorithm Complexity Analysis

The worst case scenario occurs when no leaf sum matches any list element, and as a result, all leaves must be traversed for all elements. Each leaf resulting from exploring the possibilities of adding and subtracting (car list) to or from the sum is compared against each leaf resulting from exploring the possibilities of using or not using (car list) as the target. As a result, the Big O notation is $O(2^n)(2^n)$, which reduces to $O(2^{2n})$, and ultimately to $O(2^n)$.

Teacher Solution 2020

```
;Fall 2020 Solution
;Brian-Thomas Rogers
(define closest (lambda (lst value target)
                  (cond
                    ((null? lst) (if (<= value target) value -1))
                    ((> (closest (cdr lst) (+ value (car lst)) target)
                        (closest (cdr lst) (- value (car lst)) target))
                     (closest (cdr lst) (+ value (car lst)) target))
                    (else (closest (cdr lst) (- value (car lst)) target)))))
```

Christian Solution 2020

```
#lang racket

; About

;Course: CSC 388
;Assignment: Programming Languages Assignment 3
;Date: 2-12-2020

; Algorithm


(define closest(lambda(list L target)
       (cond
          ;===base case===
          ((null? list) (= L target))  ;if list is empty, test if L equals target
          ; ===recursive cases===
          ;if target equals zero, initialize target with (car list) to prevent false
positives
          ((= target 0) (closest list L (car list)))
          ;add (car list) to L but do not use (car list) as target
          ((closest (cdr list) (+ L (car list)) target) #t)
          ;subtract (car list) from L but do not use (car list) as target
          ((closest (cdr list) (- L (car list)) target) #t)
          ;add (car list) to L and use (car list) as target
          ((closest (cdr list) (+ L (car list)) (car list)) #t)
          ;subtract (car list) from L and use (car list) as target
          (else (closest (cdr list) (- L (car list)) (car list))))))


;=============================================
; Testing
;=============================================
(#%require racket/trace)
(trace closest)

(define list1 (list 1))
(define list2 (list 1 2))
(define list3 (list 1 2 3))
(define list4 (list 8 2 6 1))
(define list5 (list 27 6 12 11))
(define list6 (list 37 26 22 21 30))
```

```
(closest list5 0 0)
```

Khan Solution 2016

```
;=============================================
; About
;=============================================
;Author: Camron Khan
;Course: CSC 388
;Assignment: Homework 3
;Date: 5-1-2016


;=============================================
; Algorithm
;=============================================

(define plus-minus(lambda(list sum target)
   (cond
     ;===base case===
     ((null? list) (= sum target))  ;if list is empty, test if sum equals target
     ; ===recursive cases===
     ;if target equals zero, initialize target with (car list) to prevent false
positives
     ((= target 0) (plus-minus list sum (car list)))
     ;add (car list) to sum but do not use (car list) as target
     ((plus-minus (cdr list) (+ sum (car list)) target) #t)
     ;subtract (car list) from sum but do not use (car list) as target
     ((plus-minus (cdr list) (- sum (car list)) target) #t)
     ;add (car list) to sum and use (car list) as target
     ((plus-minus (cdr list) (+ sum (car list)) (car list)) #t)
     ;subtract (car list) from sum and use (car list) as target
     (else (plus-minus (cdr list) (- sum (car list)) (car list))))))


;=============================================
; Testing
;=============================================
(#%require racket/trace)
(trace plus-minus)

(define list1 (list 1))
(define list2 (list 1 2))
(define list3 (list 1 2 3))
(define list4 (list 7 1 5 2))
(define list5 (list 27 6 12 11))
(define list6 (list 34 15 17 25 35))

(plus-minus list5 0 0)
```