Rigoberto Quiroz

5/10/19

CS2302 1:30 PM – 2:50 PM

Lab8 Report

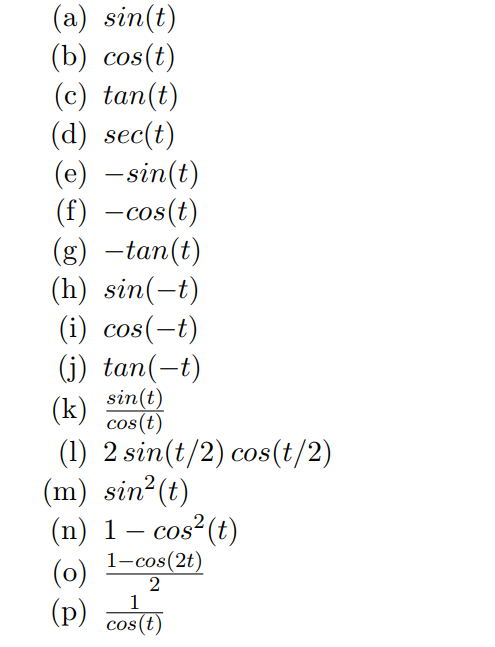
**Description:**

For this lab I had compare various trig functions to see if they are equal to each other, since some trig functions might have some points that are the same, we must check for random values, so we can know for sure that both functions are the same. For the second part of the algorithm, given a set of random values or set values, we must design a backtracking algorithm to solve the partition problem we have to meet certain conditions:

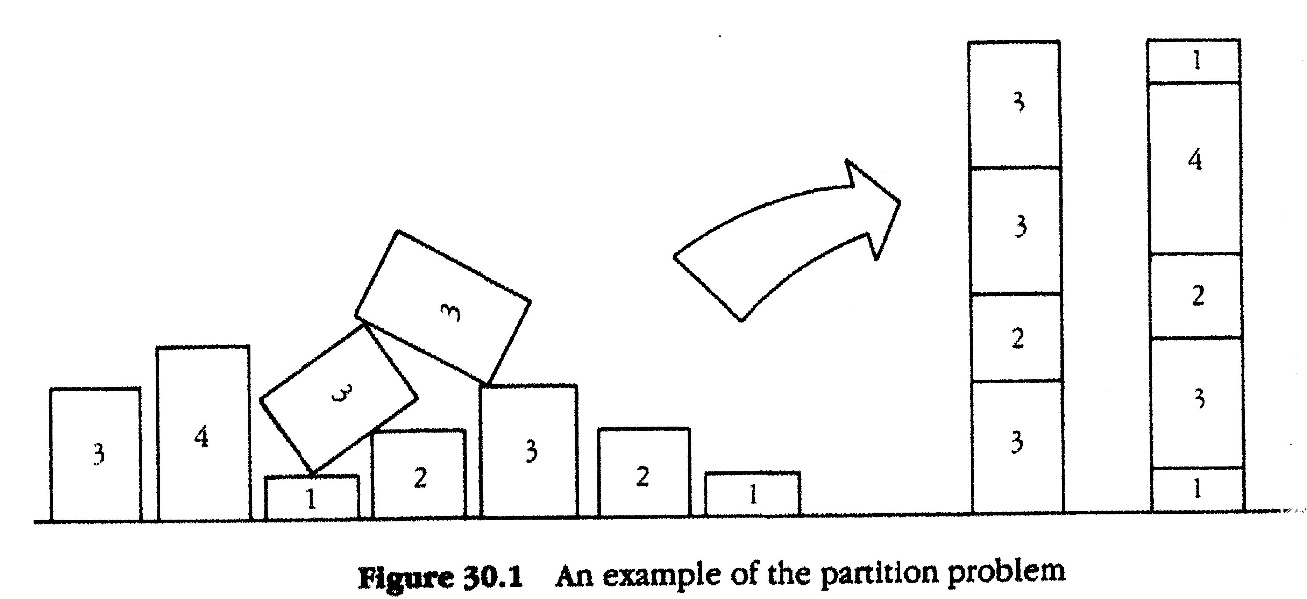
* Union of subsets must be S Original Set
* Disjunction of both subsets must be empty, or sets must not have the same value.
* The sum of each subset must be equal.)

Once these conditions have been met we will display both subsets.

List of Trig Functions to compare:



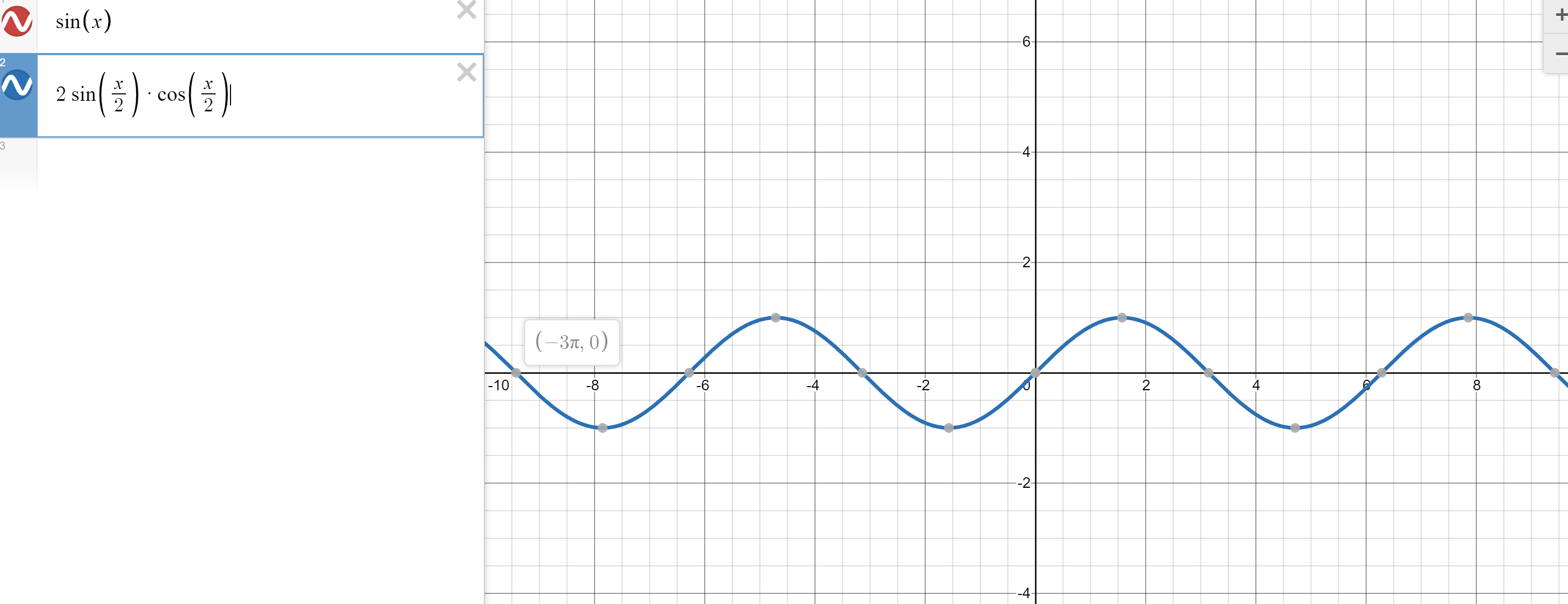
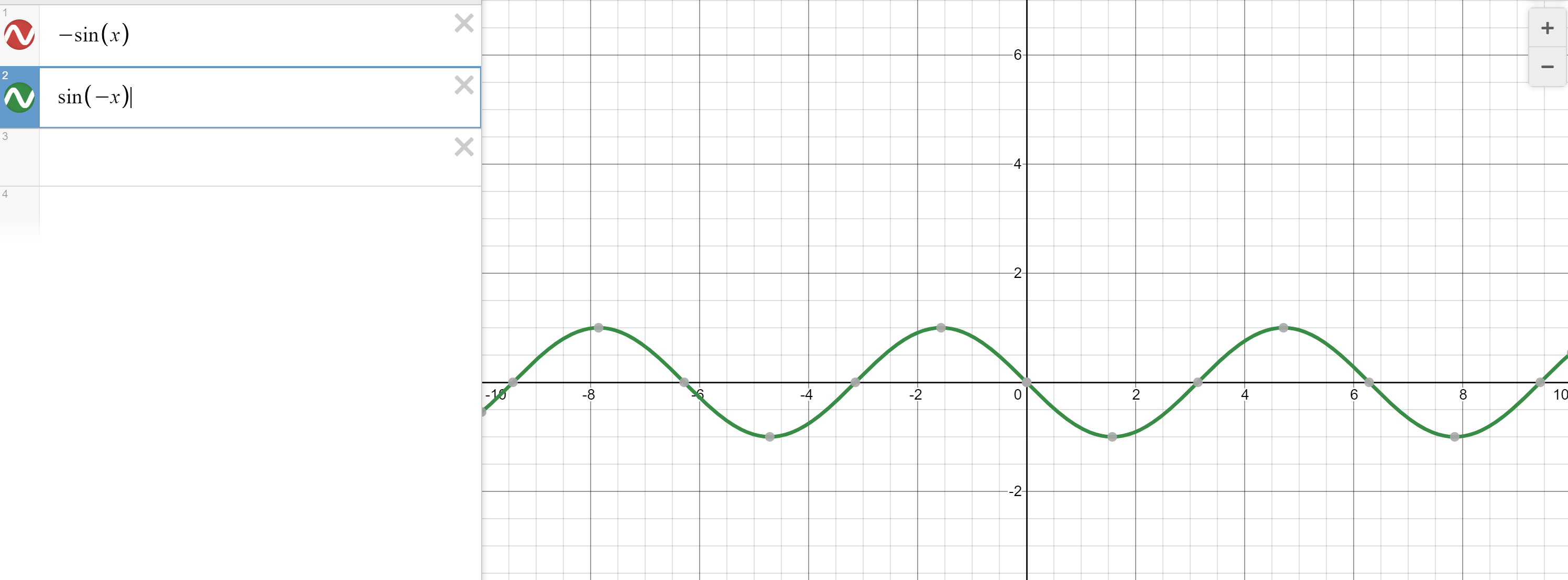
Partition Problem:



The way I was able to solve the randomization algorithm was by making a list of all the trig functions that we are going to compare, then make a nested loop that will compare each of the trig function (including itself) with 1000 difference test cases to see if they are the same, if they aren’t then we will mark them as false. For the partition problem (backtracking), I first found the sum of the set and divided that by 2 and set that as our goal. Once that is done we will recursively check all possible subsets, we will stop until we find two subsets that meet our goal. After we have generated both of subsets we will check for the union and disjunction and see if they qualify. If they pass both test, then we have solved the partition problem.

**8.1 randomized\_equal method:**

This method will take in f, list of trig functions (strings), how many times will be testing our functions and the precision of our similarities. First, we will create a list of list that will store all the functions (ans) that are equal to each other, then we will traverse f (list of trig functions) using a nested loop. Since we are testing our functions multiple times we will also have a loop that will control that. Once we are in the inner most loop (1000 test cases). We will use the Math library to evaluate the functions with each other. If the precision is not meet, then we will insert that into our ans list as False and break from loop since we no longer must check for those functions. We will repeat the same process for all the functions. Once all the loops have been executed we will return our Boolean List of Lists and print in out to show which function are equal.

* Examples of Trig Functions being equal:   
  
* 

**8.2 backtracking\_Partition:**

This method will take in, S our set containing all values, S1, a copy of our set. Last, an index, will be used to traverse the list and goal, what both subsets will add up to. First, we will check if our goal is met, (if goal is 0) if it is not we will continue with our next conditions otherwise we will return True and 2 empty sets. Next, we will check if our goal is negative or if last (index) is negative (We check last because in Python if we receive a negative index, we will check the last index and so on). If this condition is met, then we will return False and 2 empty sets. Next, we will store the return value of our recursion calls into three variables, a (Boolean value), SS (Set1), S2(Set2). Once we have those value we will check if ‘a’ is true, if it is then we will append the value of S into our SS list, set the index of last in S1 to 0 and copy those values into our second list (we will not copy the 0) This is done so that we can maintain the List with all the values. Then we will return True, SS, and S2. If ‘a’ is False, then make a recursive call but shifting our last index. The reason why we have a copy, S1, is because we are making changes into that list, if we were to update our original list then we would end up destroying our original list and its values.

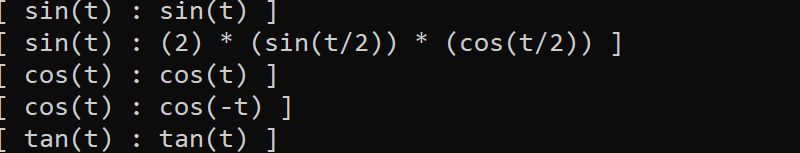
**8.2.A checkDisjunction method:**

This method will receive the original list, set 1 and set 2 (subsets created from backtracking\_Partition method). We will check subset has a lower length than then the other, then we will traverse the bigger list checking that values that are in subset 1 do not appear in subset 2, if we find matching values we will return False (These subsets will not solve the partition problem). If no values match then we will return True.

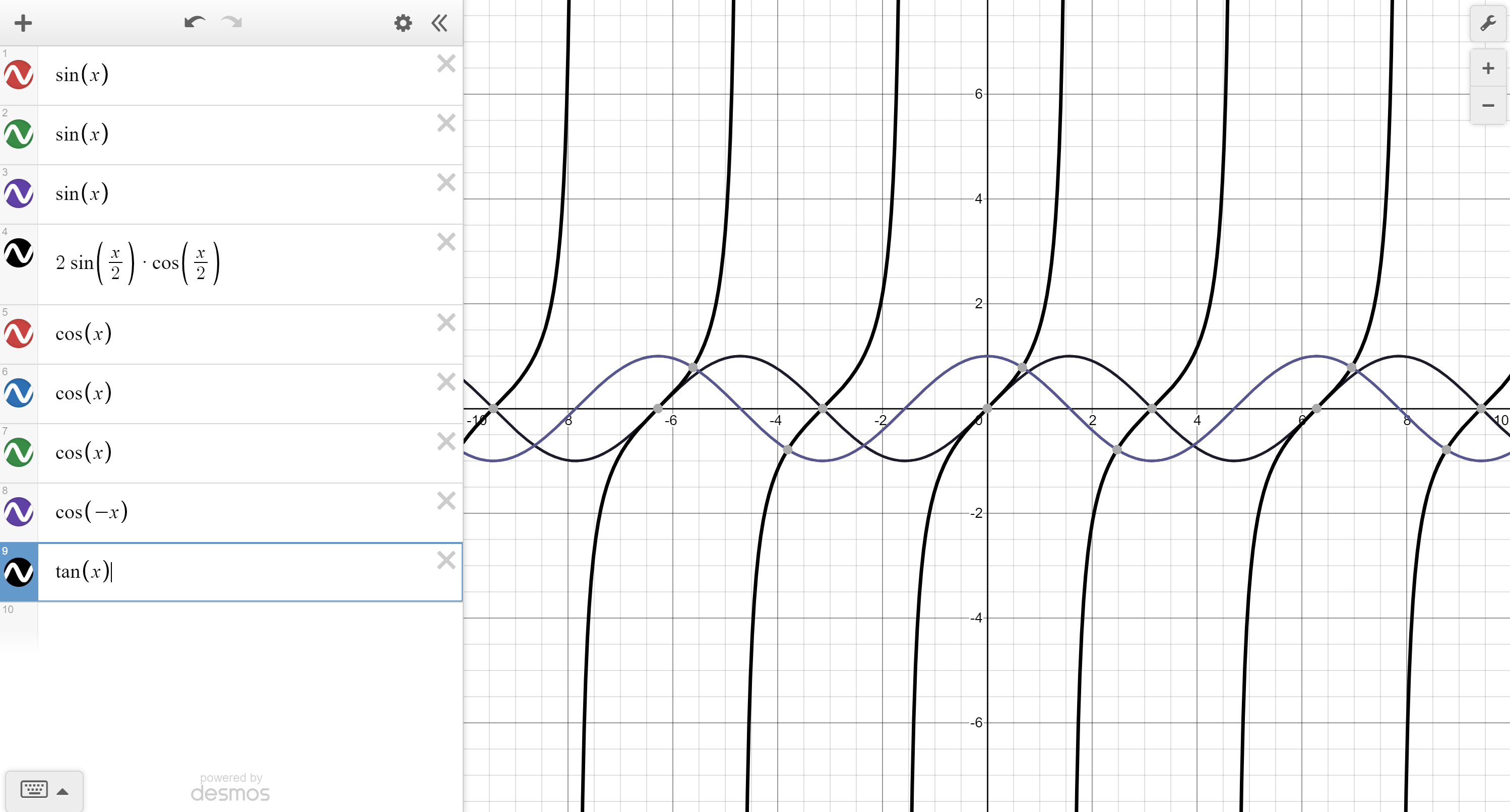
**8.2.B checkUnion Method:**

This method will receive the original list, set 1 and set 2 (subsets created from backtracking\_Partition method). We will first check if the added length of both subsets matches the length of the original. If it does then we will check that both sets values match to that of the original set. If the lengths do not match then we will return False, meaning that either we are missing values or that we have extra values in our subsets. Since we cannot only check for the lengths of the sets, because we can have the same amount of values, but the values can be different, we have to check each value.

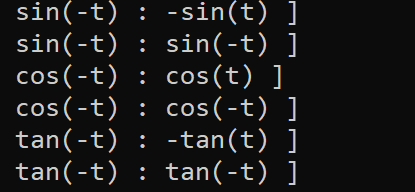
**Test Cases: (Randomized Algorithm)**

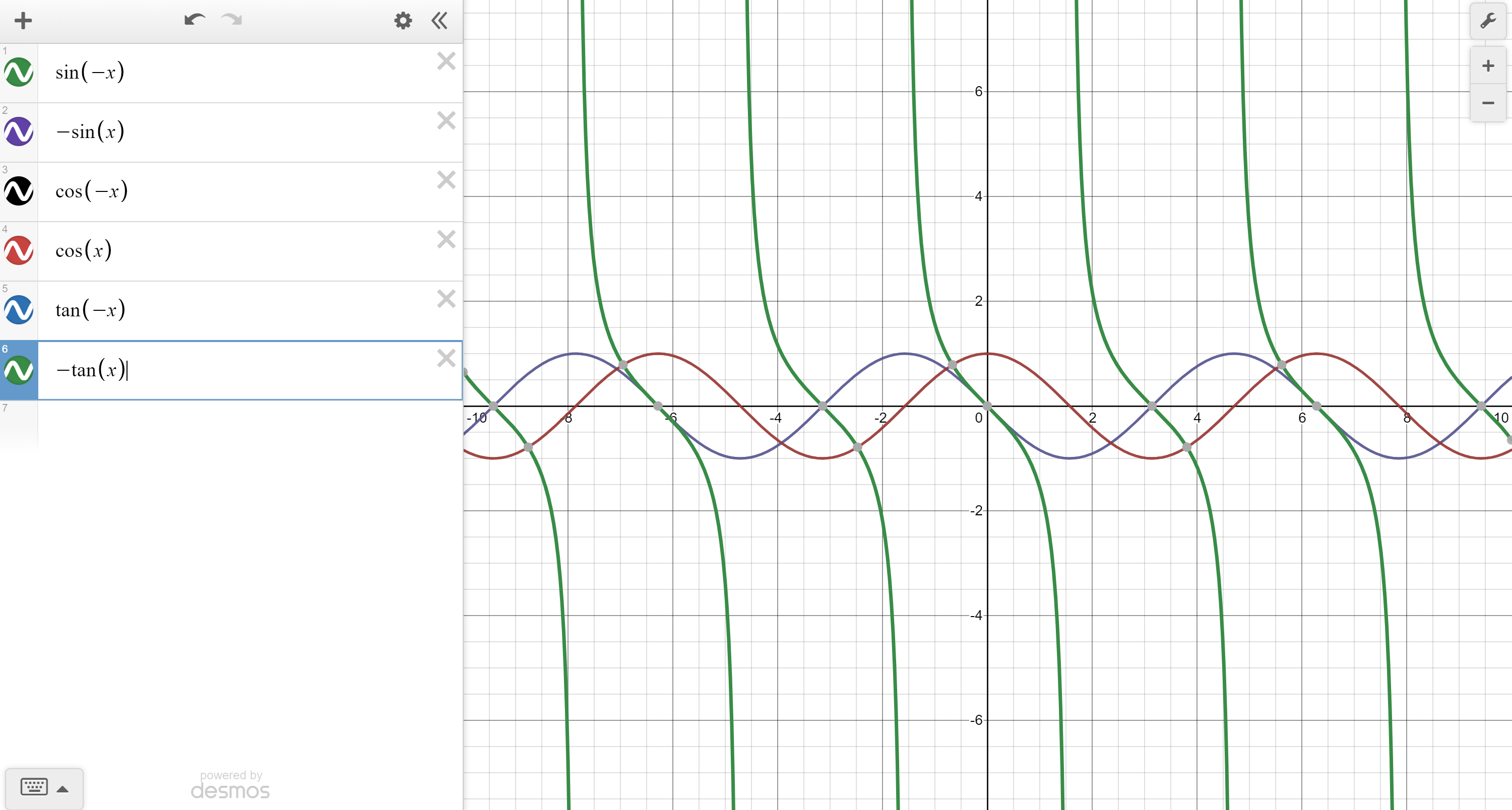


**Graphical Comparisons:**

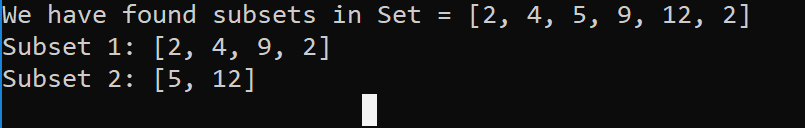


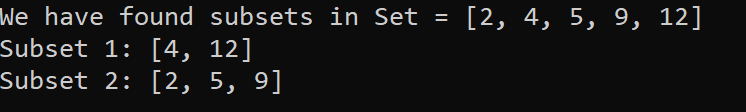
As you can only three graphs are being represented: sin(x), cos(x) and tan(x). Even though these values are being compared with other values we only see three graphs meaning that they are equal as shown by the graph.

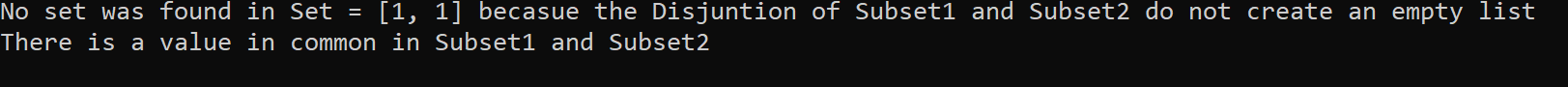


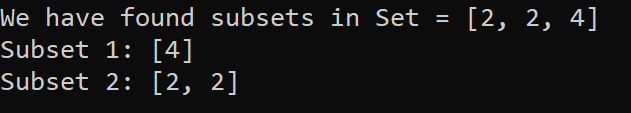


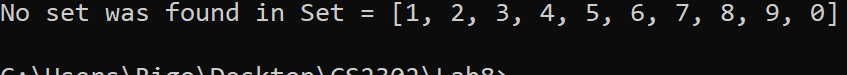
**(Backtracking Algorithm):**



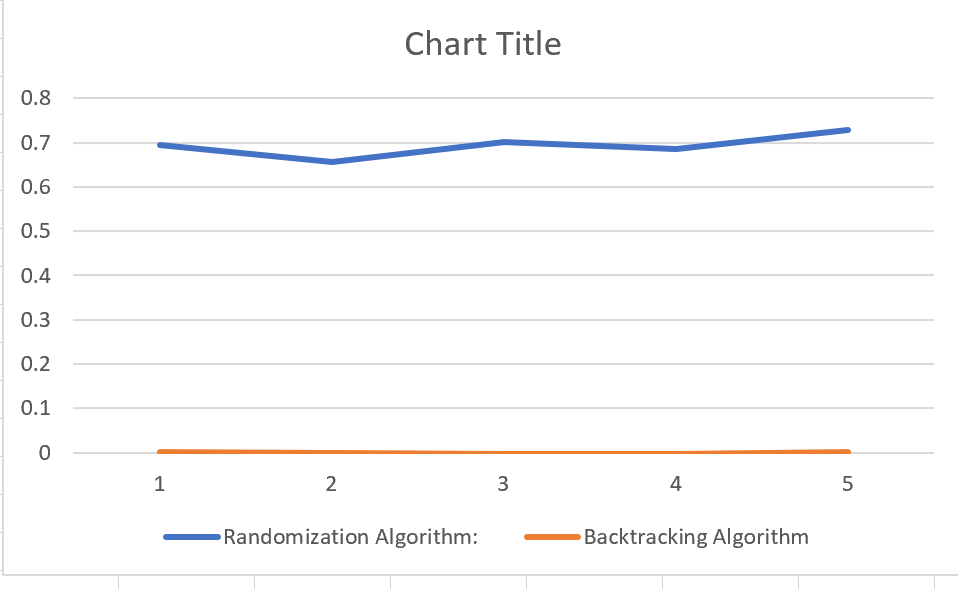








**Running Times:**



**Note:**

As you will notice the running time for the backtracking algorithm is extremely fast, being able to compute the answer extremely quickly. The same could be said with the randomization algorithm, running in a n^2 + 1000 time or just n^2 time.

**Summary:**

What I learned from this lab or what I got out of it was a deep understanding on how randomization and backtracking design algorithms work. How efficient and powerful these designs can be to output the data. I also learn how the partition problem worked and how when worked with a backtracking algorithm it can compute it extremely fast.

**Appendix:**

# Author: Rigoberto Quiroz

# Section: 1:30 PM - 2:50 PM

# This program will get a list of trig functions, then we will compare them to see

# which are the same, they are the same then we will print them out otherwise

# we will skip it. We will check each function 1000 times with values between

# pi and - pi. For the second part we will enter a list of values, then the program

# will try to find a subset following the partition problem. If we do then we will

# print the subsets otherwise will print the error message displaying why there

# was no partition problem.

# libraries

import random

import numpy as np

from math import \*

import time

# randominzed algoritm

def radomized\_equal(f,tries=1000,tolerance=0.0001):

ans = [[] for i in range(len(f))]

for i in range(len(f)):

ans[i] = [True for i in range(len(f))]

for i in range(len(f)):

for j in range(len(f)):

for x in range(tries):

t = random.uniform(-pi, pi)

f1 = eval(f[i])

f2 = eval(f[j])

if np.abs(f1-f2)>tolerance:

ans[i][j] = False

break

# returns lists of same functions

return ans

# Backtracking

def backtracking\_Partition(S,S1,last,goal):

# if we reach our goal

if goal == 0:

# return True and 2 empty lists

return True,[],[]

# otherwise return False and 2 empty lists (if we surpass our goal or index)

if goal < 0 or last < 0:

return False,[],[]

# saves boolean and lists

a, SS, S2= backtracking\_Partition(S,S1,last-1,goal-S[last])

if a:

# if we have a true value we will append it to a list

SS.append(S[last])

S1[last] = 0

S2 = []

for i in range(len(S1)):

if S1[i] != 0:

S2.append(S1[i])

# return Lists

return True, SS, S2

else:

# goes back and checks other posiible lists

return backtracking\_Partition(S,S1,last-1,goal)

def checkDisjunction(S,S1,S2):

# checks which is a subset of which

if len(S1) < len(S2):

# checks if we do not have repeated values

for k in S1:

if k in S2:

return False

else:

for k in S2:

if k in S1:

return False

return True

def checkUnion(S,S1,S2):

# checks if all the values are still in the subsets

if len(S1) + len(S2) != len(S):

return False

setSum = 0

subsetSum = 0

for k in S1:

if k not in S:

return False

for k in S2:

if k not in S:

return False

return True

# trig identites to compare

startR = time.time()

trigIdentities = ['sin(t)', 'cos(t)', 'tan(t)','1/cos(t)','-sin(t)', '-cos(t)'

, '-tan(t)', 'sin(-t)', 'cos(-t)', 'tan(-t)', '(sin(t))/(cos(t))', '(2) \* (sin(t/2)) \* (cos(t/2))'

, 'sin(t)\*sin(t)', '1-((cos(t))\*(cos(t)))', '(1-cos(t)\*cos(t))/2', '1/(cos(t))']

comparisons = radomized\_equal(trigIdentities)

endR = time.time()

# prints info

print('Tirg Functions that equal each other:')

for i in range(len(comparisons)):

for j in range(len(comparisons[i])):

if comparisons[i][j]:

print('[',trigIdentities[i],':', trigIdentities[j],']')

print('Time for random:',endR-startR)

# set to check

startB = time.time()

S = [2,4,5,9,12]

S1 = [S[i] for i in range(len(S))]

sum = 0

for i in range(len(S)):

sum += S[i]

possibleSubset, subset1, subset2= (backtracking\_Partition(S,S1,len(S)-1,sum/2))

S1 = 0

S2 = 0

endB = time.time()

if possibleSubset:

# prints if we had a subsets or not

union = checkUnion(S,subset1,subset2)

disjunction = checkDisjunction(S,subset1,subset2)

if union and disjunction:

print('We have found subsets in Set =',S)

print('Subset 1:', subset1)

print('Subset 2:', subset2)

else:

if union is False:

print('No set was found in Set =',S, 'becasue the Union of Subset1 and Subset2 are not subsets of S')

else:

print('No set was found in Set =',S, 'becasue the Disjuntion of Subset1 and Subset2 do not create an empty list')

print('There is a value in common in Subset1 and Subset2')

else:

print('No set was found in Set =',S)

print('time for backtacking:', endB-startB)