Laboratory #8 Randomization and Backtracking

Ivan Gastelum

Data Structures 2302

Professor Olac Fuentes

TA: Nath Anindita

May 13th 2019

**INTRODUCTION:**

Algorithm design techniques are very important in developing a code that can solve a specific problem. Throughout time, computer scientist, software engineers, and developers have come up with these algorithm design methods in to approach any problem and get to a specific solutions that’s either efficient or concrete. In this laboratory, we used to Algorithm design techniques to solve two specific problems: Randomization, and Backtracking. Randomization is a technique that uses random values, characters or string in order to compare or calculate and come up with a solution for it, one famous problem is finding equalities among algebraic expressions. How can we make a computer find if 2x + 2 equals 2(x+1)?

Second technique used in this laboratory is backtracking, which is a method that is followed by a reversed way of solving a problem, firstly thinking of the results and going backwards until we can input some values. One common problem that Backtracking can be used to solve is that we have a sets of numbers and a value goal, and a code needs to check if there exists a subset which sum is equal to the given goal value.

So, below one problem from Randomization and one from backtracking are defined in this lab:

(Randomized algorithms) Write a program to ”discover” trigonometric identities. Your program should test all combinations of the trigonometric expressions shown below and use a randomized algorithm to detect the equalities. For your equality testing, generate random numbers in the −π to π range.

1. sin(t) (b) cos(t) (c) tan(t) (d) sec(t) (e) −sin(t) (f) −cos(t) (g) −tan(t) (h) sin(−t) (i) cos(−t) (j) tan(−t) (k) sin(t) cos(t) (l) 2 sin(t/2) cos(t/2) (m) sin2 (t) (n) 1 − cos2 (t) (o) 1−cos(2t) 2 (p) 1 cos(t)

(Backtracking) The partition problem consists of determining if there is a way to partition a set of integers S into two subsets S1 and S2 such that P S1 = P S2. Recall that S1 and S2 are a partition of S if and only if S1∪S2 = S and S1∩S2 = {}. Write a function that solves the partition problem using backtracking. If a partition exists, your program should display it; otherwise it should indicate that no partition exists. For example, if S = {2, 4, 5, 9, 12}, your program should output the partition S1 = {2, 5, 9} and S2 = {4, 12} and if S = {2, 4, 5, 9, 13} your program should indicate that no partition exists.

**PROPOSED DESIGN SOLUTION AND IMPLEMENTATION:**

***Randomization:***

For randomization I used the given functions “equal(f1,f2)” as a base in order to find the equality of trigonometric expressions, the method consists of a for loop that loops 1000 times, and constantly checks the result by inputting random numbers into the expression at a given range until it either checks if the difference in the results are less than a given tolerance.

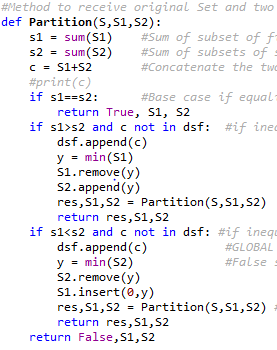
Now, since all trigonometric functions needed to be compared with each other, a O(n^2) with a double nested loop needed to be development for this type of work. So, taking what I learned from the previous material, I created an adjacency matrix of size the length of the list that was storing the entire trigonometric functions. This way I could traverse through the entire adjacency matrix and store the Boolean values once the equal method is called at each comparison. By the end of the nested loop, I could simply return the adjacency matrix.

***Backtracking:***

For the second problem involving backtracking, two subsets in which if concatenated, they will results the original given set of numbers, should return true if these subsets are equal in their sums, otherwise return false. To find a solution using backtracking I was trying to find a way to recursively stop at some if its not possible to have mixture of subsets that are not equal to each other. As a result, there should be two base cases, one for the True case, one for the False case, and the second base case was a bit challenging.

I first defined the method ‘Partition’ that receives the original set S, and two other sets which are the first and second half of the original set. At this point I created two conditions besides the base case for True conditions: if the sum of the first subset is greater than the sum of the second one, a way for me to think of a solutions is to constantly modify the sub-lists by removing a value and appending it to the other list. So if one list is greater than the other, the minimum value of the list would be removed and inserted to the other one, and we keep doing this method until we find an equality, in this case, the base case will hit True, and stop recursion.

Now, this work was done by me only to make it work for the True cases, and it worked. However, this rarely happens, we will often have False conditions. So, I think of a way of keeping track of the lists that have been compared by appending the list into a new list that I named ‘dsf’, a global list. This list of lists, will keep track of every single combination that has been done, and with a simple condition that if we have repeated a list, we will simply hit false and stop. So, I inserted along the two if conditions a second condition that check if the concatenate list of S1 and S2 are in the dsf, we will hit false and return False.



**EXPERIMENTAL RESULTS:**

|  |  |  |
| --- | --- | --- |
|  | Running Time Complexity (miliseconds) | |
| #Trials | Randomization | Backtracking |
| 1 | 2265 | 15.62 |
| 2 | 2249.934 | 0 |
| 3 | 2249.936 | 62.5 |
| 4 | 2218.688 | 0 |
| 5 | 2267.996 | 2.99 |

**Fig1. Shows the running time complexities of the two problems done in 10 trials**

**CONCLUSION:**

According to the results seen in the table above, we can conclude that backtracking problem, solves the problem at a fast pace, and by analyzing the big oh notation we can conclude that it’s a O(log n ) at the worst cases. Meanwhile, comparing trigonometric functions by randomizations has a big oh notation of O(n^3) since the main method contains two nested loops and inside the loops we have another method which running time is O(n).

**APPENDIX:**

