CS 241 Final Project - Subarray Sum

Adam Rawashdeh & Nick Mustrat

May 2023

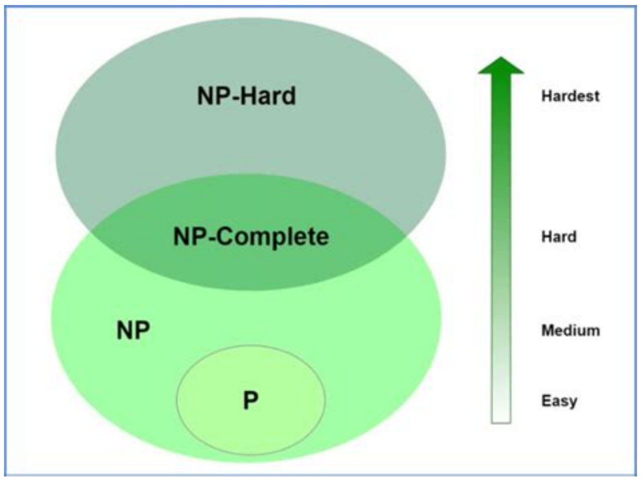
Summary

1. Background

For our final project, we chose to undertake the Sub-array Sum problem. ”Given a list of positive integers L and target number T, can you write an algorithm that runs in polynomial time and returns true if there exists a set of numbers in L that sum to T and false if no such set exists?”

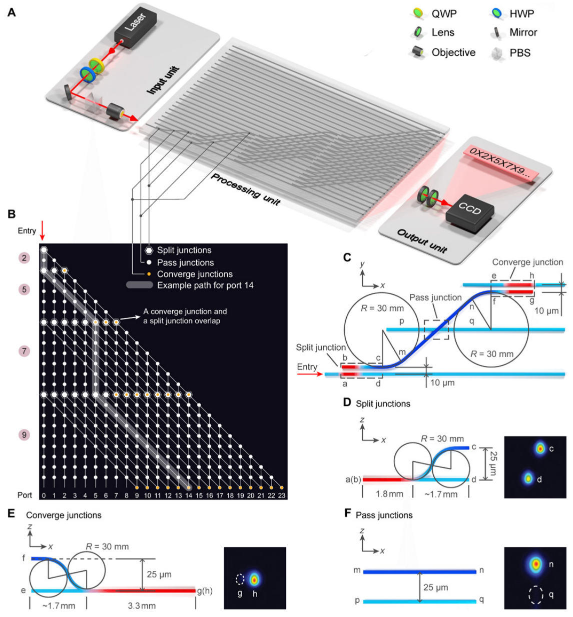
As it turns out, the Subset Sum Problem is a well known decision problem in computer science. A decision problem is posed as a yes-or-no question, meaning can or can it not be done. The difficulty in this problem lies within the desire to achieve polynomial time. With that being said, the Subset Sum Problem falls into the realm of computational complexity theory. This classifies problems into different time-complexity classes based on how demanding the task is. Two important classes are known as both P & NP.

P is known as ’polynomial time’. It refers to the class of problems that can provide some solution within polynomial time in regard to the input size. In other words, P includes the set of problems for which there exists efficient algorithms in order to find *a* solution. NP, on the other hand, is known as ’non-deterministic polynomial time’. This refers to the class of problems that can be *verified* within polynomial time. By verified, we of course mean that the given solution is checked for correctness. The subset sum problem itself falls within this very conundrum, being relegated as both ’NP-Complete’ & ’NP-Hard’. ’NP-Hard’ refers to the set of only the hardest problems in NP whereas ’NP-Complete’ is a special subset of said problems.



Even with recent advancements made within computer science, it is difficult to say if this type of problem can be solved with modern computers. If an efficient algorithm were to be created to solve this particular problem, other NP-Complete problems would benefit from this breakthrough. That is to say a snowball effect of problem-solving would occur thanks to this coveted algorithm. This would truly be a monumental breakthrough in all of computer science. Such an occurrence would benefit many practical scenarios in the realms of resource allocation, scheduling and cryptography.

Something that is rather interesting to note is that there is currently a scalable photonic computer making an attempt to solve the Subset Sum Problem. The reason as to why photons are being utilized to solve this NP-hard problem is that photons have high propagation speed, low detectable energy levels, and a strong robustness that may be able to withstand such an endeavor. This research currently being done within China is a pursuit to prove the advantages of photonic computers as opposed to conventional computers. These are machines classified as 'non-Von Neumann' architecture, consisting of an input unit, an output unit, and a processing unit. The processing unit is juxtaposed with the central processing unit of a typical computer utilized by a software engineer.



As interesting as this is, we must stay on topic and not discuss whether $P = NP$, or the use of photon computers. The Subset Sum Problem has many iterations, also classified as NP-Complete, such as all of the inputs being positive, negative, a mix of both, $T = 0$, or $T = \frac{1}{2}$ the sum of all inputs. This is also known as the partition problem. For this particular problem, we will strictly be discussing a list of positive integers L aimed to achieve target number T. If this is indeed possible to run while in polynomial time, then the function would return 'True'. If no such subset exists then it would simply return 'False'. As simple as this sounds, given the context of the difficulty of this problem it is certainly no small feat.

2. Applications

In order to solve this problem there exists a variety of ways to approach it. Here we will discuss some techniques that we considered worthy of implementation:

1. Dynamic Programming

The reason we include only dynamic programming and not recursion is because dynamic programming is an optimization of recursion. Dynamic programming is a method in which it efficiently solves a problem consisting of several or more sub-problems. How dynamic programming works is that the results of these sub-problems are stored until that specific answer is required for the problem at large. This is the same case for the 'Sub-array Sum' which consists of overlapping sub-problems in the form of many possible subsets. It is this reason for which both recursion and dynamic programming are capable methods of approach.

2. Referencing Previous Attempts

Looking at old methods and previous ways on how other programmers went about solving this problem is important because it will introduce to us different ideas that we otherwise would not have considered. We can use information from what we learned and try to forward that knowledge into what we already have implemented. This is a powerful device for learning different approaches to problem solving. With new eyes come new angles, and with newly introduced problems at this level of difficulty fresh ideas are always welcome. An example of an attempt is the 'Meet-in-the-Middle' method. Although this is not polynomial time, it is a useful solution for finding a valid subset.

3. Evaluating All Possible Solutions

Of course, this is probably the most inefficient way to solve this problem. This approach would take into account every single situation and cause the code to run for a long time. The larger input we feed to the function, the longer time it would take to generate a result. In actuality, this particular issue is the backbone of the Subset Sum Problem. Even so, it is still a valid attempt. Once a way to evaluate for a smaller input size has been developed, the code may be reworked and improved upon in order to efficiently solve for the larger ones. As a result, this code would not necessarily need to consider every situation. One example of this is to begin with ordering the list and removing any values that are already larger than that of T.

Mentioned in this section are but a few of techniques that we have considered as beneficial when it comes to developing a solution the Subset Sum Problem. Of course there are already countless methods to go about it that have already been implemented. In this report, we wish to be original with our work to the best of our abilities.

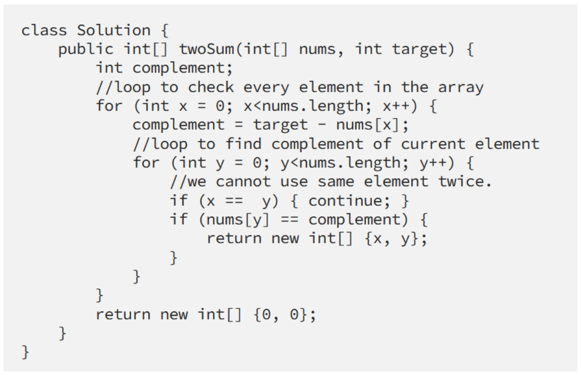
3. Our Thoughts

When first reading this problem our initial thoughts were a LeetCode problem called 'Two Sum'. The question is stated as follows:

"Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order."

The difference is that in Sub-array Sum we are *simply* suppose to return 'True' or 'False' in regard to whether or not there exists a set that adds up the target. Once again, this algorithm *must* run in polynomial time.

The time complexity of the 'Two Sum' code is O(n^2). The solution is as follows:



```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int complement;
        //loop to check every element in the array
        for (int x = 0; x<nums.length; x++) {
            complement = target - nums[x];
            //loop to find complement of current element
            for (int y = 0; y<nums.length; y++) {
                //we cannot use same element twice.
                if (x ==  y) { continue; }
                if (nums[y] == complement) {
                    return new int[] {x, y};
                }
            }
        }
        return new int[] {0, 0};
    }
}
```

How this code works is that while iterating and inserting elements into the hash table, it looks back to check if the current element's complement already exists. If it exists, then a solution has been found. In turn, the indices are returned.

What makes the Sub-array Sum more difficult is that it can be any 'N-sized' list our algorithm searches through. The quite *literal* million dollar question is if we can solve it. Based on all of our research, the likelihood is most definitely *not* in our favor. Even so, we will certainly try out best.

Individual Work

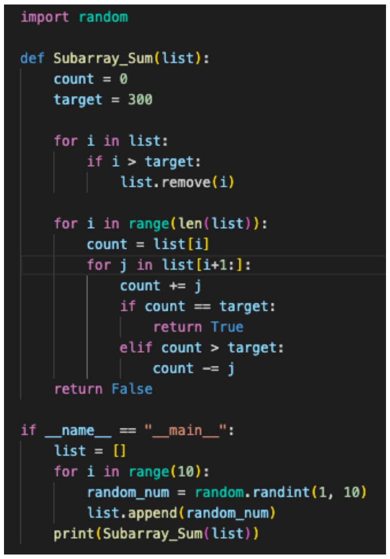
What I'm Attempting

My initial approach to this problem was to do something dynamic program related. However, my knowledge on that topic was very weak and I decided to shift towards a more greedy algorithm type of approach. I believe that way will be less efficient for that fact that it won't consider if a current result is right or wrong. However, it's the idea that came to my head so I went for it.

Result

Now the moment we've all been waiting for. My perfect, not so perfect, code.

Taadaa

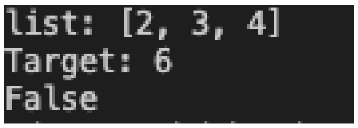


```
import random

def Subarray_Sum(list):
    count = 0
    target = 300

    for i in list:
        if i > target:
            list.remove(i)

    for i in range(len(list)):
        count = list[i]
        for j in list[i+1:]:
            count += j
            if count == target:
                return True
            elif count > target:
                count -= j
    return False

if __name__ == "__main__":
    list = []
    for i in range(10):
        random_num = random.randint(1, 10)
        list.append(random_num)
    print(Subarray_Sum(list))
```

Steps:
1. First we pass in a list as an argument.
2. Next we initialize two variables count which will keep count of our current sum and target which is the value we want to reach.
3. After we make a for loop to remove any numbers in the list that are greater than the target. This is so we can avoid those as options and potentially make our code faster.
4. Finally, we make a nested loop where our outer loop will be our starting point and the inner loop will go until either it reaches the target or if it's at the end of the list. Each time the inner loop finishes we move our outer loop up by 1 and check to see if it'll work with the new set. We than have an if statement to see if our count is equal to target and if so we return true. Than we have an else if statement to see if our count is bigger than our target and if so we minus that value from the count and continue on because there are still potential numbers we haven't tried. (Note: we could have changed the code to sort the list and once we get a count bigger than the target we could've broken out of that loop and went on to our next set. However, that might cause an issue with the potential sets and keeping it random felt like the better option)

Lack of Results

The real issue with this code is one pretty big thing. It doesn't take into account every possible combination. It works with what it haves keeps going until its no longer possible to create the target. So what do I mean by that exactly. Say if we have a list [2,3,4] and our target was 6. It will start off with adding 2 with 3 giving us 5. It will than check 5 with 4 and than realize its bigger than target so the combination is bad. Now it never will take into account just the subset of [2,4] because after it goes through all and see's it doesn't work, the program will assume that 2 is a bad apple in the batch and move on to the next set. Here is the output:



```
list: [2, 3, 4]
Target: 6
False
```

Conclusion

Overall, this project was fun, but stressful. Not because it was hard, wait I take that back because still at the moment it's unsolvable, but that wasn't the goal of this project. The goal was to take what we know and to try and take a stab at it. While I didn't end up with a million dollars and lifetime of fame, I got something not better than but still valuable: Knowledge. I learned a ton from this project and the class in general. It was a nice way to test the information I learned and to see how I can display my progress.