CMSC 142 T2L — Comparative Analysis
Github Repository: https://github.com/jgbanaag/CMSC-142-Project

Members:

Almira, Anton
Alviar, John Angelo
Banaag, Jullianne
Pendon, Rainier
Villamin, Jan Neal Isaac

Subset Sum Problem

The subset sum problem discussed in CMSC 142 is defined as follows. Given a set `S` of integers whose size is `n`, find subset/s whose elements sum to a target sum `X` or determine if such subset exists. There are three different versions of the problem we are interested in. The first version is enumerating all subsets of `S` whose elements sum to `X`, the second version is finding if there exists a solution, and the last version is printing one solution.

Solution 1: Backtracking

We can solve this version by enumerating all combinations of integers from `1` to `n`, treating these integers as indices of the array a that contains an order of the elements of `S`. This solution can be implemented using the backtracking code that enumerates all combinations of `k` integers from `n` integers for all `1 <= k <= n`. For each enumerated subset, we print this subset if the sum of its elements is `X`. The explanation of the code for generating the subsets can be found in this document.

Analysis based on Properties of the Solution

Enumerating all subsets of S is bijective to enumerating integers from `0` to `2^n - 1`. Each integer `x` from `0` to `2^n - 1` can be represented with `n` bits. The `i`-th bit of `x` can represent whether the `i`-th element `a[i]` is part of the subset or not. If `i`-th bit is `1`, then `a[i]` is part of the subset, and `0` otherwise. For instance, the case `n = 5, x = 00101` represents the subset `{a[0], a[2]}`, because the `0`-th bit and 2nd bit are both set while the other bits are not. Since there are `2^n` of these integers, then there are `2^n` subsets of `S`.

Another way to think about this is that `a[i]` can be part of the subset or not. So for each element, there are two possible cases, `a[i]` is in the subset and `a[i]` is not in the subset. There are `n` elements and for each independent event of taking `a[i]` or not, there are two ways, so there are `2 * 2 * 2 * ... * 2 = 2^n` possible subsets.

Now, for each subset, we print its elements which is in the worst case, O(n). So, a possible upper bound for the time complexity is O(2^n) * O(n) = O(2^n * n).

We can actually prove that this is a tight bound if we are forced to print the solutions. Note that if we have a subset of length k, we are only printing k elements. So, T(n) = sum[k = 0...n] (k * C(n, k)). We can prove that this sum evaluates to 2^(n - 1) * n.

Note that C(n, k) = C(n, n - k). Adding opposite terms evaluates to k * C(n, k) + (n - k) * C(n, n - k) = n * C(n, k). If n is even, then there is a solo term n/2 * C(n, n/2) in the middle, otherwise, there are exactly (n + 1)/2 opposite pairs. We can evenly split these sum pairs into two: n * C(n, k) = n/2 * C(n, k) + n/2 * C(n, k) = n/2 * C(n, k) + n/2 * C(n, n - k).

So, the sum is actually sum[k = 0...n] (n/2 * C(n, k)) = n/2 * sum[k = 0...n] C(n, k). But sum[k = 0...n] C(n, k) is just the number of subsets in a set of length n which is just 2^n. So, T(n) = n/2 * sum[k = 0...n] C(n, k) = n/2 * 2^n = 2^(n - 1) * n = O(2^n * n).

Now, if we don't enumerate all subsets and we just want to find a subset or decide if a subset exists, then we can reduce the time complexity to O(2^n) because we only need to keep track of the sum. When we add an element or backtrack, we update the sum at the same time by adding a[i] or subtracting a[i]. Thus, the time complexity will only be O(2^n) because there are 2^n subsets.

Note that the time complexity of the best, average, and worst case are the same for the enumeration problem.

While deciding the problem, the best case is O(n) if the first subset is empty or O(1) if the target sum is 0 (if empty sets are allowed). If trivial cases are ignored, then it is O(n) since we still need to process the input array. The worst case is when there are no subsets whose elements sum up to X.

For the average case, we consider that the probability of each subset to be chosen is equally likely. Since there are 2^n subsets, then the average time complexity for the decision problem is E[T(n)] = (1 + 2 + 3 + ... + 2^n) / 2^n = (2^n * (2^n + 1) / 2) / 2^n = (2^n + 1) / 2 = O(2^n). Now if the chosen subset is to be printed, the analysis will be slightly more complex. Suppose the chosen subset of length

k is to be printed. Note that this printing will only be done once. The probability that the subset is chosen is P = 1/2^n. The probability that the subset has a length of k is C(n, k) / 2^n = C(n, k) * P. By linearity of expectation, the average time complexity is the average time it takes to choose the subset before stopping the search plus the average time it takes to print the solution. Thus, E[T(n)] = S * P + [sum[k = 0...n] (k * C(n, k))] * P = (2^n (2^n + 1) / 2) / 2^n + 2^(n - 1) * n / 2^n = (2^n + 1) / 2 + n / 2 = O(2^n).

Based on the Actual Implementation
The implementation uses backtracking. We use n stacks and fill them up with possible options. We only fill up options that are greater than the previous selected integer. The space complexity of our implementation is O(n^2) because we maintain an n by n option table.

The time complexity however is still relatively the same. Note that each candidate that we push in a stack will always be part of a distinct subset later on and will be popped when we backtrack from the integer. This comes from the fact that everytime we backtrack, we generate a subset. Since each element are processed two times, then processing the combinations is approximately 2 * 2^n steps which is still O(2^n). Printing the solution happens for each combination. Therefore, the same analysis can be applied, the time complexity is O(2^n * n). If the version is just deciding or finding a single subset, then we can optimize by processing the sum everytime we consider a new top of stack or we backtrack, so the optimized time complexity is O(2^n).

Solution 2: Dynamic Programming
The dynamic programming solution uses recurrence relation to solve the version of determining whether a subset whose sum is X exists in S. Let dp[i][j] be 1 if there exists a subset of the first i elements of S whose sum is X and 0 otherwise. We process i from 1 to n and view this as building the subset of S. The recurrence relation is dp[i][j] = dp[i - 1][j] | dp[i - 1][j - a[i]]. Note that dp[i - 1][j] represents not adding a[i] to the subset so the sum remains to be j. On the other hand, dp[i - 1][j - a[i]] represents adding a[i] to the subset and the previous sum before adding a[i] is j - a[i]. In other words, if it is already possible to form the sum j using the previous i - 1 elements, we can also form the sum j using the current i elements. If it is already possible to form the sum j - a[i] using the previous i - 1 elements, then we can form the sum j using the current i elements. Hence, the recurrence relation is dp[i][j] = dp[i - 1][j] | dp[i - 1][j - a[i]] for all 0 <= j <= X for all 1 <= i <= n. The answer for the problem is dp[n][X].

Implementation
We can implement this using recursion, but this is essentially the same as our brute force approach since from `f(i, j)` we will call `f(i - 1, j)` and `f(i - 1, j - a[i])` leading to an exponential runtime. If memoization was done, however, then this will lead to a dynamic programming approach runtime. However, we can do slightly better.

To implement this in pseudo-polynomial runtime we use a bottom-up dynamic programming approach. We populate an `n` by `X` table by going through each `i`, and for each `i`, enumerate all possible sums `j` from `0` to `X`.

Finding a Single Solution
To find a single solution, we can maintain `p[j]` where `p[j] = a[i]` where `i` is minimized and `dp[i][j] = 1` and `dp[i][j - a[i]] = 1`. In other words, `p[j]` is the first ever element `a[i]` scanned that can serve as the last term added to a subset to get `j`. Initially, `p[j] = -1` and we only process it when at a certain point in time `(i, j)` – it is possible to get `a[i]` to form `j`. To reconstruct an answer after filling up the table, we start a variable `s = X`. We append `p[s]` to the subset and update `s := s - p[s]`. Do this until `s` becomes `0`.

This means that we take the first ever possible last term for the sum `s` and recursively do this for sum `s - p[s]`. The way we update `p` guarantees that the indices of the elements of the reconstructed subset are distinct from each other. We can show this by a mixture of proof by contradiction and mathematical induction. Suppose at some point, there are two sums `j1, j2` such that `j1 < j2` and `p[j1] = a[i]` and `p[j2] = a[i]`. The update `p[j1] := a[i]` and `p[j2] := a[i]` only happens at the time we process `i` because prior to `i`, `p[j1] = p[j2] = -1` and they will never get updated after `i`. However, since it is already possible to form the sum `j2`, then `p[j2], p[j2 - p[j2]], ..., p[j2 - p[j2] - p[j2 - p[j2]] - ...]` are already updated prior to `i`. That is, we already identified a subset to form `j2` prior to `i`. Hence, `j2` will never lead to `j1` since `p[j1]` and `p[j2]` are updated at the same `i`. To complete the induction, simply consider the base case of `i = 1`.

Time and Space Complexity Analysis
The time complexity is then `O(n * X)` for the best, average, and worst cases since we always fill up the table. For each i, we enumerate all possible sums from `0` to `X`. The space complexity is also `O(n * X)` since we are maintaining an `n` by `X` table.

Comparison between Backtracking and Dynamic Programming

Each approach has its own advantages and disadvantages. For the backtracking approach, it allows us to enumerate all subsets whose sum is `X` and those whose sum is not `X`. On the other hand, the dynamic programming approach only allows us to determine if a subset exists and find one solution. Furthermore, the dynamic programming approach operates on the assumption that the range of the possible sums of subsets of `S` is enumerable in the range `0` to `X`. If the sums can be real numbers, then it is not always impossible to enumerate or iterate through them. The implementation of the dynamic programming approach further assumes that the elements of the subset are non-negative integers to be able to use possible sums as column indices of the table. Another assumption is that `n * X` fits into the limitations on the memory space of the program running the algorithm.

Furthermore, we are under the assumption that the elements are non-negative. This assumption allows us to have an optimal substructure combined with the first assumption: `dp[i][j]` only depends on states with smaller `i` and at most `j`. That is, we can build solutions using `i` elements by inferring from solutions using fewer elements. The reason that we can infer from lesser or equal sums is that `a[i] >= 0` implies that `j - a[i] <= j`.

However, if a backtracking approach was used, it is still possible to determine all subsets without these assumptions since we exhaustively search subsets and get actual sums of the subsets. What we are interested in is the subsets instead of focusing on assumptions on the sum. If the elements are, say, real numbers, positive, zero, or negative, we can still sum subsets of them and check if the result is equal to `X`.

The main advantage of dynamic programming approach is that it is usually more efficient than the backtracking approach. We say usually, because it is more efficient when the number of possible sums are reasonable. With processors nowadays, the backtracking approach can work in a reasonable amount of time within `n <= 26`. This is because `2^26 <= 10^8` which runs within seconds in a decent processor. However, if `n * |set of all possible sums| <= 10^8`, and the sum is enumerable, then a dynamic programming approach can work better when `n` is larger (such as thousands and more).

To summarize, here is the comparison of the two approaches.

Approach	**Backtracking**	**Dynamic Programming**
Main assumptions	Taking the sum of elements in a subset and	The possible sums are enumerable (or given) and

		comparing it with X can be done accurately.	its range is not too large. The elements are non-negative.
What can it solve effectively?		Enumerating all subsets whose elements sum to X (and those that don't).	Deciding if there exists a subset whose sum is X or finding a single solution.
Time complexity	Best	O(2^n * n) (or O(2^n) if printing is not required or if the problem is only the decision problem or finding a single solution)	O(n * X)
	Average	O(2^n * n) (or O(2^n) with the same reason)	O(n * X)
	Worst	O(2^n * n) (or O(2^n) with the same reason)	O(n * X)
Space complexity		O(n^2) due to an n by n option table. A different implementation can reduce this to O(n) such as enumerating subsets in lexical order with an increasing number of elements.	O(n * X) due to an n by X table. It can be further optimized to O(X) using different tricks (such as using alternating tables of size X or by processing sums from X to 0, updating dp[j]).
Possible input constraints with the assumption that T(n) = 10^8 is roughly one second		n <= 26	n <= 5000, X <= 5000 n <= 10^7, X <= 10 In general, n * X <= 10^8.
Main advantages over the other		More flexible and scalable since we can relax the assumptions made by the dynamic programming approach.	If assumptions are satisfied, it is better for larger values of n with a reasonable number of possible sums.
Implementation complexity		Simple in some ways since the only goal is to find a way to enumerate all subsets.	Simple in some ways since we are essentially only iterating over a 2D array. Error prone since base

		cases, sentinel/dummy values, and transitions are critical.
Intuition	More intuitive and you can generate the code without much thought in the solution.	Identifying the subproblem and defining the recurrence relation is not as intuitive.