CS 31: Algorithms (Spring 2019): Lecture 6

Date: 9th April, 2019

Topic: Dynamic Programming 1: Subset Sum

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.

The next few lectures we study the method of dynamic programming. To set the idea behind dynamic programming, let's revisit what we did for divide-and-conquer (D&C). In D&C, given an *instance* of a problem, we broke it into *instances* of *smaller* sub-problems. We recursively solved the sub-problems. Then we *combined* the *solutions* of the sub-problems, often in a quite clever way, to obtain a *solution* to the original problem.

In Dynamic Programming (DP), the situation is similar but subtly different. Here, given an *instance* of a problem, we consider the *solution* to the problem. Of course, we don't *know* the solution, but we know it *exists*, and we abstractly think of it.

Next, we need to observe that this *solution* actually can be broken up into *solutions to smaller subproblems*. If we can't observe this, then perhaps dynamic programming can't solve our problem. This is so crucial that the textbook CLRS has given it a name – *optimal substructure*.

Do you see the subtle difference between what D&C does and what DP does? D&C breaks a problem into two; DP needs the solution to break into two. Of course, all this is way too abstract to appreciate right now. Let's reinforce this with examples.

1 Subset Sum

SUBSET SUM
Input: Positive integers $a_1, \ldots, a_n$, Target positive integer B.
Output: Decide whether there is a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = B$? If YES, return the subset.

What is a naive algorithm for the Subset Sum problem? Seems like one needs to go over all the subsets of $\{1, 2, \ldots, n\}$ – which takes $\Omega(2^n)$ time. Not good. Subset Sum is one of the poster child problems for Dynamic Programming. Let's see how it works.

Given the instance $(a_1, \ldots, a_n; B)$ of Subset Sum, let us assume there is a set S of these numbers which sum to B. Fix this set S in your mind. Can we "break" this set S into subsets which are solutions to "smaller Subset Sum instances"?

What are smaller subset-sum instances? It could be small in two ways: (1) fewer number of a_i's, that is, n, or (2) smaller B. We see below that the solution S indeed "breaks" into solutions to smaller problems.

- Suppose someone tells us that the number a_n (this is a number we read from the input) is **not in** the set S. Then, we can say $S = S \setminus a_n$ is a solution to the Subset Sum instance $(a_1, \ldots, a_{n-1}; B)$.
- Suppose now someone tells us that the number a_n is **in** the set S. Then, we can say $S \setminus a_n$ (a subset of S) is a solution to the Subset Sum instance $(a_1, \ldots, a_{n-1}; B - a_n)$.

Good, this is progress. S must satisfy one of the two cases above, and in each case a very particular subset of S is a solution to a smaller subproblem. The subproblems are different in each case, but nevertheless in both cases they are *smaller*. In the first case, n goes down; in the second case, both n and B go down.

Next, we see that the *converse* is true: in each of the two subproblems, a solution to the subproblem can be made into a solution we care about. Indeed,

- if one gives us a subset T which is a solution to $(a_1, \ldots, a_{n-1}; B)$, then the same T is a solution to $(a_1, \ldots, a_n; B)$ as well;
- if one gives us a subset T which is a solution to $(a_1, \ldots, a_{n-1}; B - a_n)$, then $T + a_n$ is a solution to $(a_1, \ldots, a_n; B)$ as well.

We have now observed the optimal substructure in action. The main idea was to fix this abstract solution S and branch on one of the two conditions (one of which must occur): $a_n \in S$ or $a_n \notin S$. Given a solution to any of these cases, we can build up a solution to the original problem. (This is kinda like the combine step of divide-and-conquer but usually much more straightforward)

Once you have gone over the above in your head, it is time to make things ***precise using correct definitions***. Here is the definition useful for the subset sum problem; let us make it, and then we can go over why we made this definition.

Definition: For any integer $0 \leq m \leq n; 0 \leq b \leq B$, we define $F(m, b) = 1$ if there exists a subset $S \subseteq \{1, 2, \ldots, m\}$ such that $\sum_{i \in S} a_i = b$.

Why is this the definition we are making at this point? Well firstly, we desire to find the answer $F(n, B)$ (we also want to find the subset if $F(n, B) = 1$, but baby steps). Second we observe that the above argument converted the "search among $a_1, \ldots, a_n$" to instances "searching amng $a_1, \ldots, a_{n-1}$". This *linear ordering* is actually very important: this allows us to pass the first parameter m which searches among $a_1, \ldots, a_m$. Finally, why did we pass "b" as a parameter to the function? Again, this is because in the observation we made above, we note that in one of the subproblems the target parameter changed from B to $B - a_n$. This is the reason we passed the target into the function.

Ok, so what good is this definition? It is actually super useful because it allows us to convert the "English intuition" of optimal substructure mentioned above into a recurrence relation, as we show below. And why are recurrence relations useful? Because they *immediately* lead to recursive algorithms. These recursive algorithms will be "disastrous" just

like how the recursive algorithm for Fibonacci numbers was. But we can convert them into bottom-up iterative good algorithms.

Claim 1.

- (Base Cases.) $F(m, 0) = 1$ for all $0 \leq m \leq n$. $F(0, t) = 0$ for all $t > 0$. $F(m, t) = 0$ for any $t < 0$.

- (Recurrence.) For any $m \geq 1, b > 0$; we have

$$F(m, b) = \max\left(F(m-1, b), F(m-1, b-a_m)\right)$$

Proof. The base case (often) follows from the definition. $F(m, 0) = 1$ because the empty subset of $\{1, 2, \ldots, m\}$ always sums to 0. Since all a_i's are positive, $F(m, t) = 0$ for all $t < 0$ since we can't get a negative number by adding positive numbers. $F(0, t) = 0$ if $t > 0$, since the only subset of the empty set is the empty set which sums to 0.

Let us now prove the recurrence equality. We do this by proving two inequalities.

($\leq$): The proof is precisely the English description above written rigorously. The skeleton of this proof will form the structure of most dynamic programming arguments we will see.

If $F(m, b) = 0$, then the inequality follows since the RHS is ≥ 0. So suppose $F(m, b) = 1$. That is, there is a subset $S \subseteq \{1, 2, \ldots, m\}$ which sums to b. If $a_m \in S$, then $S \setminus a_m \subseteq \{1, 2, \ldots, m-1\}$ sums to $b - a_m$, implying $F(m-1, b-a_m) = 1$. If $a_m \notin S$, then $S \subseteq \{1, 2, \ldots, m-1\}$ sums to b, implying $F(m-1, b) = 1$. Since one of the two cases must hold; $F(m, b) \leq \max(F(m-1, b), F(m-1, b-a_m))$.

($\geq$): This is the "conversely" part of the argument we made above.

$F(m, b) \geq F(m-1, b)$ because if there is indeed a subset T of $\{1, 2, \ldots, m-1\}$ which sums to b, then T also a subset of $\{1, 2 \ldots, m\}$ that sums to b.

Similarly, $F(m, b) \geq F(m-1, b-a_m)$ because if there is indeed a subset T of $\{1, 2, \ldots, m-1\}$ which sums to $b - a_m$, then $T + a_m$ is also a subset of $\{1, 2 \ldots, m\}$ that sums to b.

□

Remark: *"Do we really need to always* prove *the recurrence equality?," is a common question here. Well, why not? For one you will then be sure that you are correct. The English explanation is necessary and is often OK. But a proof always is necessary to be* **sure** *you are not pushing things under the rug or pulling wool over your own eyes.*

Once we have obtained the recurrence, we are really almost done. The "clever" part is over. What remains is *systematic* to all DP algorithms and can be picked up with practice.

We start with the *disastrous* implementation by just recursively calling. I provide it below in red: *NEVER* show this in public (but writing it privately is a very good idea).

```
1: procedure RecSubSum(m, b):
2:     ▷ Returns 1 if there is a subset of a_1, ..., a_m that sums to exactly b.
3:     if b = 0 then:
4:         return 1
5:     if m = 0 and b > 0 then:
6:         return 0
7:     if b < 0 then:
8:         return 0
9:     return max (RecSubSum(m − 1, b), RecSubSum(m − 1, b − a_m) )
```

The above algorithm is correct (we are still solving the Yes-No question); but it is disastrous for the reason the recursive Fibonacci algorithm was terrible. But we saw how to fix that! And indeed, we fix the above RecSubSum analogously.

First we allocate space for a table. The dimensions correspond to the variables that are passed in the recurrence. The range is from the base-case to the point we are interested in.

```
1: procedure SubsetSum(B, a_1, ..., a_n):
2:     ▷ Says YES if there is a subset summing to B, otherwise N0
3:     Allocate space F[0 : n, 0 : B] ≡ 0
4:     F[m, 0] ← 1 for all m.
5:     F[0, b] ← 0 for all b > 0. ▷ Base Cases
6:     for 1 ≤ m ≤ n do:
7:         for 1 ≤ b ≤ B do:
8:             if b − a_m < 0 then: ▷ So we know F(m − 1, b − a_m) = 0 in this case
9:                 F[m, b] ← F[m − 1, b].
10:            else:
11:                F[m, b] ← max (F[m − 1, b], F[m − 1, b − a_m])
12:    ▷ At this point F[n, B] has the answer; if it is 1 there is a solution, otherwise not.
```

✍

Exercise:

- *Run the above algorithm* by hand *for the input* $(1, 2, 3, 4; 9)$.
- *Implement this above algorithm and see it in action.*

Recovery. The above algorithm works because the "table" $F[m, b]$ contains the function value $F(m, b)$. However, we need more : we need that when $F[n, B] = 1$, we need a subset S which sums to B. How do we find this?

One inefficient way to do this is that instead of $F[m, b]$ being 0 or 1, we actually also store a subset of $\{1, 2, \ldots, m\}$ summing to b in the case $F[m, b] = 1$. This blows up the

space required by a factor n since each table could contain $\Theta(n)$ elements. But we don't need this; since we have the full table $F[0:n, 0:B]$, we can use it to *read out* the subset which sums to B as follows.

We start with an empty subset and "counters" $m = n$ and $b = B$. We have $F[n, B] = 1$ (otherwise, we have answered NO). But since $F[n, B] = \max(F[n-1, B], F[n-1, B-a_n])$, one of these two must be 1. If $F[n-1, B] = 1$, then we add nothing to B and decrease n by 1. If $F[n-1, B-a_n] = 1$, then we add a_n to the subset and decrease B by a_n and n by 1. We proceed iteratively, maintaining the invariant that the total sum of the subset plus the "current B", that is b, equals the original B **and** $F[m, b] = 1$. In the end, we reach $m = 0$ and since $F[m, b] = 1$, we must have $b = 0$ (the only base case with $m = 0$ that evaluates to 1.) At this point the subset we have sums to exactly B. The full pseudocode for subset sum with recovery is given below giving us the following theorem.

Theorem 1. SUBSET SUM can be solved in time and space $O(nB)$.

Remark: *Is this a polynomial time algorithm? To answer this, we need to define clearly what a polynomial time algorithm is. An algorithm is polynomial time, if its running time $T(n)$ is, for large enough n, at most some fixed polynomial $p(n)$ where n is the* size *of the instance. We cheekily left out the size of the Subset Sum problem; the size after all is $\Theta(\log B + \sum_{i=1}^{n} \log a_i) = O(n \log B)$ since we can throw away any $a_i > B$. Now we observe that our running time $O(nB)$ is* exponentially *larger than the size of the problem; the B is the nub. As stated, the above algorithm is* **not a polynomial time algorithm.**

```
1:  procedure SUBSETSUM(B, a_1, ..., a_n):
2:      ▷ Says YES if there is a subset summing to B, otherwise NO
3:      Allocate space F[0 : n, 0 : B] ≡ 0
4:      F[m, 0] ← 1 for all m.
5:      F[0, b] ← 0 for all b > 0. ▷ Base Cases
6:      for 1 ≤ m ≤ n do:
7:          for 1 ≤ b ≤ B do:
8:              if b − a_m < 0 then: ▷ So we know F(m − 1, b − a_m) = 0 in this case
9:                  F[m, b] ← F[m − 1, b].
10:             else:
11:                 F[m, b] ← max (F[m − 1, b], F[m − 1, b − a_m])
12:     ▷ At this point F[n, B] has the answer; if it is 1 there is a solution, otherwise not.
13:     if F[n, B] = 0 then:
14:         return NO
15:     ▷ Recovery:
16:     m ← n; b ← B; S ← ∅.
17:     ▷ Invariant: ∑_{i∈S} a_i + b = B and F[m, b] = F[n, B] = 1
18:     while b > 0 do:
19:         if F[m − 1, b] = 1 then:
20:             m ← m − 1
21:             S ← S
22:             b ← b
23:         else: ▷ In this case, we must have F[m − 1, b − a_m] = 1
24:             m ← m − 1
25:             S ← S + m
26:             b ← b − a_m
27:         ▷ Check that the Invariant holds in both cases
28:     ▷ At this point, b = 0. Since invariants hold, we have ∑_{i∈S} a_i + 0 = B.
29:     return S
```

To recap, to design and analyze a dynamic program for the Subset Sum problem we had the following ingredients. This is going to be the steps in **all** dynamic programming algorithms. Indeed, for your problem set, I require you to write all of these.

1. *Definition*: A precise definition of the function which will be recursively represented. Clearly mention the parameters which you are interested in.
2. *The Base case:* The "small" values at which the function's value is known.
3. *The Recurrence Relation:* Clearly state the recurrence relation. Give an explanation of why it is correct.
4. *Proof:* To be absolutely sure, give a proof of the recurrence relation.

5. *Implemetation Pseudocode* Write the correct implementation of the recurrence a la Fibonacci using tables. Be sure that you are filling up the tables in the correct *order*. Often this is standard, but you will see some tricky examples.

6. *Recovery Pseudocode.* Write the code for recovery (when needed) by back-tracking on the table that you obtained. This may seem non-trivial, but it is actually straightforward after a little practice.

7. *Running Time and Space.* Write down the running time of and also space used by your algorithm.