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.

In the next few lectures we study the method of dynamic programming (DP). The idea is really *recursion* as in divide and conquer (D&C), but there are significant differences. In D&C, the smaller instances are often obtained in a "straightforward way" (cutting an array in the middle, splitting a polynomial, etc), and they are often disjoint and don't interact with each other. The creativity in D&C lies in combining the solutions to these smaller instances. In Dynamic Programming, the creativity actually lies in finding the smaller instances itself; indeed, in some sense, the smaller instances are created by actually looking at how their solutions will combine to give the solution to the original instance[1], and thus the combining-the-solutions step is easy. The smaller instances, however, heavily interact, and dynamic programming is efficient only when one can still argue the *total* number of smaller instances that are recursively solved does not explode. If one can set things up so, dynamic programming can often solve problems which, at first, may look rather impossible to solve. Let's give some more details before diving into concrete applications.

Let I be an *instance* of a problem we want to solve. We first abstractly imagine a solution S of I. Then, we need two things to happen. First, from S we can obtain "pieces", let's call them *solutionettes*, $S_1, S_2, \ldots$ such that (a) each solutionette S_j itself is the correct solution to a *smaller instance* I_j of the same problem, and (b) given any solutionettes $T_1, T_2, \ldots$ to the smaller instances $I_1, I_2, \ldots$, we can construct a solution T to the original instance I. This describes the "division" into smaller subproblems $I_1, I_2, \ldots$ and how to combine them. The second *key* things is to somehow show that the total *number* of smaller subinstances *ever* encountered is "small". This is often done by figuring out an *arrangement* of the possible smaller instances I_j (either in a line, or in a grid) and arguing the arrangement size is small. This is the *creative* part of DP: the *definition* of the smaller subinstances into a nice arrangement, and the *recursive* way in which solutions can be combined. Of course, all this is very abstract, and perhaps hard to follow. I suggest keep looking at examples and revisiting the above discussion often.

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? One can go over all the subsets of $\{1, 2, \ldots, n\}$ – which takes $O(2^n)$ time. Not great. Subset Sum is one of the poster child problems for Dynamic Programming. Let's see how it works.

Let us revisit the abstract idea discussed at the beginning of this lecture. Given the instance $I := (a_1, \ldots, a_n; B)$ of Subset Sum, assume there is a set S of these numbers which sum to B. Fix this set

[1]This may not make any sense now. It may in the third of fourth reading of these notes.

S in your mind. Can we "break" this set S into subsets which are solutions to "smaller instances of Subset Sum"?

How do we even start breaking a solution into smaller solutions? One thing perhaps to start with (for any problem) is just taking one element and removing it from the solution. Which element should we start with? Often starting with the "last" (in some order) or "first" is a good idea. We will often go with the last. In this case, we start by trying to remove a_n, the last element, from S.

- Suppose a_n was in S. Consider the set $T = S \setminus a_n$. Can we say whether T is a solution to some other, hopefully smaller, Subset Sum instance? A moment's thought tells YES: T is a solution to the instance $I_1 = (a_1, a_2, \ldots, a_{n-1}; B - a_n)$. If the elements in S sum to B, the elements of T sum to $B - a_n$. Moreover, T is a subset of the first $n - 1$ elements.

- But what is a_n was not in S. We can't even then "remove" a_n from S? How do we proceed? This is perhaps the a ha! moment. In this case, then, S *itself* is a solution to a smaller subinstance of Subset Sum. Which one? The instance $I_2 = (a_1, a_2, \ldots, a_{n-1}; B)$. The instance with the "last" element kicked out.

To summarize, we took our thought solution S of the instance I, and observed that in one case $S \setminus a_n$ is the solution for $I_1 = (a_1, a_2, \ldots, a_{n-1}; B - a_n)$, and in the other case, S itself is the solution for $I_2 = (a_1, \ldots, a_{n-1}; B)$. This gives us the way to obtain the two *smaller instances* I_1 and I_2 from the instance I.

Now let us try to see if we can achieve the two things we need to make dynamic programming work. We saw that a solution S to I implies solutions to I_1 and I_2 (indeed, that is how they were constructed). How about vice-versa? Indeed, it is simple. We see that

- 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.

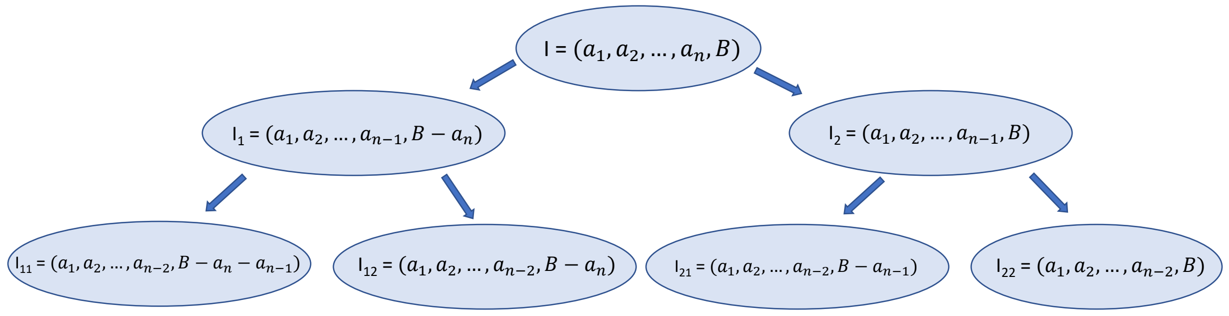
The argument for breaking the solution above was "reversible". Therefore, we have obtained our recursive substructure.

Next, we need to see whether these various subinstances *ever* seen when solving recursively are not too many in number. Why would that be? Well let us stare at the two instances obtained. Indeed, it may help to actually "draw out" the tree of instances obtained a little more for the pattern to emerge. See Figure 1 for the first two layers; I recommend drawing one more to make sure you understand the instances.

We see that unlike in the case of Fibonacci numbers, there is no "repeating balls" (at least in the first two-three layers). This could be disheartening. Don't be. Rather ask "how does a general ball (instance) look like?" in this tree. In this case the answer is it looks like $I' = (a_1, a_2, \ldots, a_m; b)$ for some integer $1 \le m \le n$, and some integer $b \le B$. Therefore, the number of such smaller instances is *not* that large. Indeed it is at most nB many (assuming B isn't that large). The fact that the smaller instances could be arranged as a $n \times B$ grid is the second key observation that convinces us dynamic programming would work for subset sum.

Concretely writing down a DP solution

The above discussion was trying to give an intuition how when faced with a problem one can come up with a dynamic programming solution. Writing it down, and finally getting the code from those thoughts can be tricky. But it actually mechanical, and I would like to show the steps.



I = $(a_1, a_2, \dots, a_n, B)$
I$_1$ = $(a_1, a_2, \dots, a_{n-1}, B - a_n)$
I$_2$ = $(a_1, a_2, \dots, a_{n-1}, B)$
I$_{11}$ = $(a_1, a_2, \dots, a_{n-2}, B - a_n - a_{n-1})$
I$_{12}$ = $(a_1, a_2, \dots, a_{n-2}, B - a_n)$
I$_{21}$ = $(a_1, a_2, \dots, a_{n-2}, B - a_{n-1})$
I$_{22}$ = $(a_1, a_2, \dots, a_{n-2}, B)$

Figure 1: The smaller instances for subset sum

1. ***Definition.*** When you figure out that the subinstances can be arranged in a nice order, you actually can concretely *define a recursive function* which will assist to write the final code. As we observed that a general sub instance is defined by the m and the b, the following definition emerges for Subset Sum.

   For any integer $0 \leq m \leq n; 0 \leq b \leq B$, define $F(m, b) = 1$ if there exists a subset $S \subseteq \{1, 2, \dots, m\}$ such that $\sum_{i \in S} a_i = b$, and 0 otherwise. We are interested in figuring out whether $F(n, B) = 0$ or 1.

   We are also interested in *finding the subset* if $F(n, B) = 1$, but for the time being let us focus on the "decision question."

   In general your definition should be parametrized by the "arrangement" you have discovered in your sub-instances, and contain the "essence" of each subinstance.

2. ***The Base Cases.*** Any recursive function must have base cases. These are the "small values" for which the value of the function is known. For Subset Sum, what are they? Here are some.

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

   What do they mean in English? $F(m, 0) = 1$ means there is a subset of the first m elements (even when $m = 0$) which sums to 0. Which is it? The empty set $\emptyset$. $F(0, t) = 0$ for $t > 0$ because there is no subset of the empty set which can sum to *more* than 0. Finally, if $t < 0$, then there is no subset which can add to t since a_i's are positive.

3. ***The Recurrence Relation:*** After the definition, this is the most important part of the dynamic programming solution. How does the recursive function get defined using smaller values? Again, this one obtains by noting how the solutions to the smaller instances give rise to a solution to the original instance. For Subset Sum, this is

   For any $m \geq 1, b > 0$; we have

   $$F(m, b) = \max\left(F(m-1, b), F(m-1, b - a_m)\right) \qquad \text{(SubsetSumRec)}$$

Once again, the English reason is that given solutions to $(a_1, \ldots, a_{m-1}; b)$ and $(a_1, \ldots, a_{m-1}; b - a_m)$, one can get the solution of $(a_1, \ldots, a_m; b)$. The latter has a solution if one of the two have a solution. Thus, we need to take an OR (which is the same as MAX).

4. ***Proof:*** Sometimes, English reasoning can be misleading. Best to *prove* the recurrence. For Subset Sum we have really already done so when we were trying to describe the idea behind the recurrence. We will repeat it again for good measure. The proof of the recurrence is where you really can precisely talk about the idea behind the algorithm.

   Proof of (SubsetSumRec). There are always two directions. One corresponds to go from a solution to the original instance to solutions to smaller instances. The other is vice-versa. 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, the "bigger instance" has a solution. We need to show one of the smaller instances has a solution.

   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; $F(m,b) \leq \max(F(m-1,b), F(m-1,b-a_m))$.

   (≥): In this case, we make precise the argument that if *any* of the smaller instances has a solution, then so does the bigger one.

   $F(m,b) \geq F(m-1,b)$ 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.

   □

5. ***Implemetation Pseudocode.*** The hard part is done! Now, we have to just implement the above recursive function using smart recursion. Just to belabor the point, let me first again give the 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).

```
procedure RecSubSum($m, b$):
    ▷ Returns 1 if there is a subset of $a_1, \ldots, a_m$ that sums to exactly $b$.
    if $b = 0$ then:
        return 1
    if $m = 0$ and $b > 0$ then:
        return 0
    if $b < 0$ then:
        return 0
    return max (RecSubSum($m-1, b$), RecSubSum($m-1, b-a_m$) )
```

The above algorithm is correct (we are still solving the decision version). But it is disastrous for the reason the recursive Fibonacci algorithm was terrible. However, we now know how to fix it. One could use memoization. I like to use tables, because the table will also help me *recover* the subset if the answer is 1.

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.

```
procedure SubsetSum($B, a_1, \ldots, a_n$):
    ▷ Says YES if there is a subset summing to B, otherwise NO
    Allocate space $F[0:n, 0:B] \equiv 0$
    $F[m, 0] \leftarrow 1$ for all $m$.
    $F[0, b] \leftarrow 0$ for all $b > 0$. ▷ Base Cases
    for $1 \le m \le n$ do:
        for $1 \le b \le B$ do:
            if $b - a_m < 0$ then: ▷ We know $F(m-1, b-a_m) = 0$ in this case
                $F[m, b] \leftarrow F[m-1, b]$.
            else:
                $F[m, b] \leftarrow \max(F[m-1, b], F[m-1, b-a_m])$
    ▷ At this point $F[n, B]$ has the answer; if it is $1$ there is a solution, otherwise not.
```

6. ***Recovery Pseudocode.*** 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 $F[n, B] = \max(F[n-1, B], F[n-1, B-a_n])$, *at least* one of these two must be 1. If $F[n-1, B] = 1$, then we decrease nothing from B and decrease n by 1. If $F[n-1, B-a_n] = 1$, then we add the index 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 pseudocode for the recovery is given below giving below. There is no need to write this part separately, and should be included with the previous.

```
procedure RecoverSubsetSum($F[0:n, 0:B]$):
    ▷ This is taking input the filled up table $F$ from previous routine. There is no need to write this separately, and ideally should be part of the same code.
    if $F[n, B] = 0$ then:
        return NO
    ▷ Recovery:
    $m \leftarrow n$; $b \leftarrow B$; $S \leftarrow \emptyset$.
    ▷ Invariant: $\sum_{i \in S} a_i + b = B$ and $F[m, b] = F[n, B] = 1$
    while $b > 0$ do:
        if $F[m-1, b] = 1$ then:
            $m \leftarrow m - 1$
            $S \leftarrow S$
            $b \leftarrow b$
        else:  ▷ In this case, we must have $F[m-1, b-a_m] = 1$
            $m \leftarrow m - 1$
            $S \leftarrow S + m$
            $b \leftarrow b - a_m$
        ▷ Check that the Invariant holds in both cases
    ▷ At this point, $b = 0$. Since invariants hold, we have $\sum_{i \in S} a_i + 0 = B$.
    return $S$
```

7. ***Running Time and Space.*** The final part is to analyze the running time and space required by the algorithm. For Subset Sum, we observe that the running time is dominated by the two for loops. Thus the total time is $O(nB)$.

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

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.

Remark: *Was the algorithm for* SUBSETSUM *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.**