Dynamic Programming: Subset Sum + Knapsack[1]

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 overlap with each other. The master theorem or the kitty method implies a decent running time. The creativity in D&C often lies in combining the solutions to these smaller instances.

In Dynamic Programming, the smaller instances *are not* "disjoint", and at first glance, the number of them seem to *explode*. The ***key*** in dynamic programming, or rather dynamic programming can only work, if we can somehow observe a "pattern" in these smaller instances to argue that in fact they don't explode, but rather, the *same* problem is being asked to solve repeatedly. Just like in Fibonacci number computations. Observing this pattern is the real creativity in dynamic programming. In my opinion, students struggle with dynamic programming because to observe this pattern one must look "under the hood" as to how the smaller sub-instances that are being called look like. Once one gets used to this, dynamic programming becomes easy.

Let me give some more abstract 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.

a. First, from S we can obtain "pieces", let's call them *solutionettes*, $S_1, S_2, \ldots$ such that (i) each solutionette S_j itself is the correct solution to a *smaller instance* I_j of the same problem, and (ii) 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. This is the ***recursive structure*** of the problem[2].

b. 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.

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

<u>SUBSET SUM</u>
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 $\{1, 2, \ldots, n\}$, and then check for every subset whether it sums to B. This takes $O(n2^n)$ time. Not great for $n > 50$. Subset Sum is a poster child problem for Dynamic Programming. Let's see how dynamic programming solves this.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2] The book by Cormen, Leiserson, Rivest, and Stein call it the "optimal substructure"

Let us revisit the abstract idea discussed at the beginning of this lecture. Let's fix an instance (input) I for the subset sum problem. It looks like $I := (a_1, \ldots, a_n; B)$ of Subset Sum. Now, suppose there is indeed 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 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 as follows.

- 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 if a_n was not in S? We can't "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 smaller instance? 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 $I_2 = (a_1, \ldots, a_{n-1}; B)$. There is no other case. This gives us the way to obtain the two *smaller instances* I_1 and I_2 from the instance I. Thus, we obtained the *recursive structure* of SUBSET SUM.

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? That is, given solutions to I_1 and I_2, can we construct solutions to I? Indeed, we can, and that is simple.

- if one gives us a subset T which is a solution to $I_1 = (a_1, \ldots, a_{n-1}; B - a_n)$, then $T + a_n$ is a solution to $(a_1, \ldots, a_n; B)$ as well.
- if one gives us a subset T which is a solution to $I_2 = (a_1, \ldots, a_{n-1}; B)$, then the same T 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.

Unlike in the case of Fibonacci numbers, we do not immediately see any "repeating balls" (at least in the first two-three layers). This could be disheartening. But don't be disheartened. Rather ask "how does a general ball (sub-instance) look like?" in this tree. Is it "succinctly" describable? In this case the answer is it looks like $I' = (a_1, a_2, \ldots, a_m; b)$ for some integer $1 \leq m \leq n$, and some integer $b \leq B$. Therefore, the number of such smaller instances is *not* that large. Indeed it is at most nB many. 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.

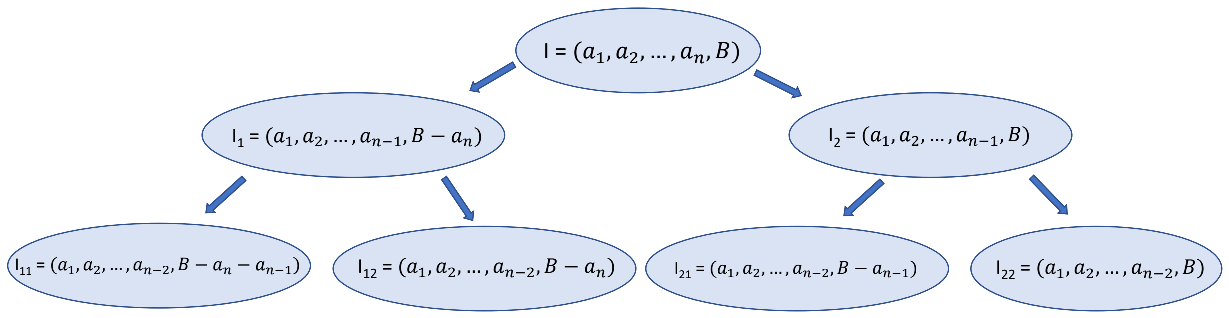
I = $(a_1, a_2, \ldots, a_n, B)$
I$_1$ = $(a_1, a_2, \ldots, a_{n-1}, B - a_n)$
I$_2$ = $(a_1, a_2, \ldots, a_{n-1}, B)$
I$_{11}$ = $(a_1, a_2, \ldots, a_{n-2}, B - a_n - a_{n-1})$
I$_{12}$ = $(a_1, a_2, \ldots, a_{n-2}, B - a_n)$
I$_{21}$ = $(a_1, a_2, \ldots, a_{n-2}, B - a_{n-1})$
I$_{22}$ = $(a_1,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

Figure 1: The smaller instances 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. You should not *write* the solution as above. Rather, the process of getting from "thought" to "pseudocode" is pretty mechanical, and follows in six-seven steps. At least in this class (and later on when needed), always write DP solutions in this way. After 8-9 such trials, you will see yourself becoming a master of DP!

a. ***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 $(a_1, \ldots, a_m; b)$ is defined by two things: the m and the b. Next, we write a definition to precisely say what we mean.

   For any integer $0 \leq m \leq n; 0 \leq b \leq B$, define $F(m, b) = 1$ if there exists a subset $S \subseteq \{1, 2, \ldots, m\}$ such that $\sum_{i \in S} a_i = b$, and is 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. Coming up with this definition is the ***first key*** step in your dynamic programming solution.

b. ***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.

c. ***The Recurrence Relation:*** After the definition, this is the ***second key*** step in your 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).

d. ***Proof:*** Sometimes, English reasoning can be misleading (as we will see later in the course). Therefore, we must formally *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[3].

Proof of (SubsetSumRec). There are ***always two directions*** in proving an equality. 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.

($\leq$): *In this case, we make precise the argument that a solution to the bigger instance leads to solution to one of the smaller instances.*
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" $(a_1, \ldots, a_m; b)$ 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; $F(m, b) \leq \max(F(m-1, b), F(m-1, b-a_m))$.

($\geq$): *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.

$\square$

e. ***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.

[3] At this point, I hope everyone feels the urge to learn LaTeX and use it. Hands will hurt if all this is to be handwritten

```
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. But we will use the bottom-up approach and use explicit tables, because the table will also help me *recover* the subset[4] 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: it's the base case we didn't hard-code.
                $F[m, b] \leftarrow F[m-1, b]$.
            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.
```

f. ***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

[4] I am not saying using memoization one can't; one can. To me everything being explicit just helps.

$F[n-1, B-a_n] = 1$, 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 $F[m-1, b-a_m] = 1$
            $S \leftarrow S + m$
            $b \leftarrow b - a_m$
            $m \leftarrow m - 1$
        ▷ 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 $S$
```

g. ***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.

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

e. *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 may see some tricky examples.
f. *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.
g. *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.**

Next, we see a cousin of the Subset Sum problem. It is the first example of an *optimization* problem. The Subset Sum problem was a *decision* problem, in that, the output was YES or NO (ok, so if YES we also wanted the subset). In optimization problems, the question is not whether a feasible solution exists, but more of among all *candidate feasible solutions* can you choose one which is *best* in a certain metric.

2 Knapsack Problem.

KNAPSACK
Input: n items; item j has profit p_j and weight w_j. A knapsack of capacity B. All of these are positive integers.
Output: Find the subset $S \subseteq \{1, 2, \ldots, n\}$ which maximizes $\sum_{j \in S} p_j$ and "fits" in the knapsack; that is, $\sum_{j \in S} w_j \leq B$.

Note the question "does there exist a subset which fits in the knapsack?" is trivial to answer. Yes, there is: the *empty set*. The interesting part is to figure out which among all candidate subsets, gives the largest profit. As in Subset Sum, the brute-force approach of going over all possible subsets that fit in the knapsack and choosing the best, is a time consuming affair. We want to do better via dynamic programming.

As in the Subset Sum case, let us fix an instance I of Knapsack $((p_1, w_1), \ldots, (p_n, w_n); B)$. Let us abstractly consider an optimal solution $S \subseteq \{1, \ldots, n\}$ for this problem (the subset marks the indices of the items picked). Can we break this S up into solutionettes for smaller instances of Knapsack? We will proceed *exactly* like in Subset Sum.

Let us focus on the "last" item n and ask whether it is in S or not.

- If it *is* in S, then I claim $S_1 = S \setminus n$ is an *optimal* (max profit) solution to a smaller instance of Knapsack. Can you see which one? It is $I_1 = ((p_1, w_1), (p_2, w_2), \ldots, (p_{n-1}, w_{n-1}); B - w_n)$. Why is S_1 the best solution in I_1? Well, if there was something better, then adding the nth item to that solution would give a better solution than S to the original instance I.
- If the nth item *is not* in S, then again S itself is the optimal (max-profit) solution to the smaller instance $I_2 = ((p_1, w_1), (p_2, w_2), \ldots, (p_{n-1}, w_{n-1}); B)$ of Knapsack. Again, if not, then a better solution for this smaller instance would be a better solution for the original instance.

One can also argue the *vice-versa* direction: given optimal solution to both I_1 and I_2, can we find the optimal solution to I? Can you guess how to do it? We will take the solution to I_1 and *add* the profit p_n of the nth item, and compare it to I_2 (when we don't add the nth item). And take the one that is better (gives more profit). The best of these two will be the best solution for I.

All the above discussion, again, is the thoughts going in our head which lead us towards the rigorous solution to the dynamic programming problem. At this point, we should perhaps draw the tree diagram for the recursive structure of the problem (as in Figure 1 in the previous lecture notes), and we will see as in Subset Sum, they arrange up in a grid. There are two parameters of interest: m, denoting the first m items, and b, the available size in the knapsack. After we do all this, it is time to venture on to the 7-fold path we laid down last time.

Before we do so, let me introduce another piece of notation which is going to be very useful for *arguing* about *optimization* problems. It is the notion of Cand which captures the collection of *candidate feasible* solutions to the smaller instance one is considering. For the Knapsack problem, since we know that m and b are the parameters of interest, we define the following:

$$\mathsf{Cand}(m,b) : \text{all possible subsets of } \{1,2,\ldots,m\} \text{ of items with total weight is} \leq b.$$

In English, $\mathsf{Cand}(m,b)$ are the candidate feasible solutions to the instance $((p_1,w_1),\ldots,(p_m,w_m);b)$. And by definition, the *best* (maximum profit) solution is the one giving the maximum value. For writing our recurrence, it will be this value that will be most important, and this is going to be the part of our definition. We will write a recurrence for $F(m,b)$ which is the maximum profit subset in $\mathsf{Cand}(m,b)$.

a. ***Definition:*** For any $0 \leq m \leq n$ and $0 \leq b \leq B$, let $\mathsf{Cand}(m,b)$ be all subsets $S \subseteq \{1,\ldots,m\}$ which fit in a knapsack of capacity b, that is, $\sum_{j\in S} w_j \leq b$. Define

$$F(m,b) = \max_{S\in\mathsf{Cand}(m,b)} \sum_{j\in S} p_j$$

We use shorthands $p(S) = \sum_{j\in S} p_j$ and $w(S) = \sum_{j\in S} w_j$ for brevity. We are interested in $F(n,B)$.

b. ***Base Cases:***
   - $F(0,b) = 0$ for all $0 \leq b \leq B$; an empty set gives profit 0.
   - $F(m,0) = 0$ for all $0 \leq m \leq n$; an empty set gives profit 0.

c. ***Recursive Formulation:*** As can be deduced from the discussion above, we assert for all $m \geq 1, b \geq 1$:

$$F(m,b) = \max\left(\, F(m-1,b),\ F(m-1,b-w_m) + p_m \,\right)$$

d. ***Formal Proof:*** As in Subset Sum, we need to show an equality. We do so by proving the two inequalities. In what follows, we first show that the left hand side (LHS) is $\leq$ the right hand side (RHS). Subsequently, we show LHS $\geq$ RHS. This proves LHS $=$ RHS. We will see that the set Cand will be useful in proving this.

   ($\leq$): Let S be the subset in $\mathsf{Cand}(m,b)$ such that $F(m,b) = p(S)$.

   Case 1: S doesn't contain item m. Then $S \in \mathsf{Cand}(m-1,b)$ and so $F(m-1,b) \geq p(S) = F(m,b)$, since $F(m-1,b)$ is the *maximum* over all sets in $\mathsf{Cand}(m-1,b)$.

   Case 2: S contains item m. Then $S\backslash m$ lies in $\mathsf{Cand}(m-1,b-w_m)$ and $p(S\backslash m) = p(S) - p_m = F(m,b) - p_m$. Thus, $F(m-1,b-w_m) \geq F(m,b) - p_m$, since $F(m-1,b-w_m)$ is the *maximum* over all sets in $\mathsf{Cand}(m-1,b-w_m)$.

($\geq$): Let S be the subset in $\mathsf{Cand}(m-1,b)$ such that $p(S) = F(m-1,b)$. Observe S also lies in $\mathsf{Cand}(m,b)$. Thus, $F(m,b) \geq p(S) = F(m-1,b)$ since $F(m,b)$ is the *maximum* over all sets in $\mathsf{Cand}(m,b)$.

Similarly, let S be the subset in $\mathsf{Cand}(m-1,b-w_m)$ such that $p(S) = F(m-1,b-w_m)$. Form $S' = S + m$. Note that $S' \in \mathsf{Cand}(m,b)$ since $w(S') \leq b$. Therefore, $F(m,b) \geq p(S') = F(m-1,b-w_m) + p_m$.

e. ***Pseudocode for computing*** $F[n,B]$ ***and recovery pseudocode:*** The pseudocode is one formed, as in Subset Sum, by the smart recursion idea on the above recurrence equality. The recovery process is also similar.

```
procedure KNAPSACK($B$,$(p_1,w_1),\cdots,(p_n,w_n)$):
    ▷ Returns the subset of items of type $1,\ldots,n$ which fits in knapsack of capacity $B$ and gives maximum profit.
    Allocate space $F[0:n,0:B]$
    $F[0,b] \leftarrow 0$ for all $0 \leq b \leq B$ ▷ Base Case
    $F[m,0] = 0$ for all $0 \leq m \leq n$. ▷ Base Case
    for $1 \leq m \leq n$ do:
        for $1 \leq b \leq B$ do:
            if $b - w_m \geq 0$ then :
                $F[m,b] \leftarrow \max(F[m-1,b], F[m-1,b-w_m] + p_m)$
                ▷ Note $F[m-1,b-w_m]$ is set before $F[m,b]$ in this ordering.
            else: ▷ Implicitly, in this case $F[m-1,b-w_m] = -\infty$
                $F[m,b] \leftarrow F[m-1,b]$
    ▷ $F[n,B]$ now contains the value of the optimal subset
    ▷ Below we show the recovery pseudocode
    $m \leftarrow n$; $b \leftarrow B$; $S \leftarrow \emptyset$.
    ▷ Invariant: $\sum_{j\in S} w_j + b \leq B$ and $F[m,b] + \sum_{j\in S} p_j = F[n,B]$
    while $m > 0$ do:
        if $F[m,b] = F[m-1,b]$ then:
            $m \leftarrow m-1$
        else: ▷ We know $F[m,b] = F[m,b-w_m] + p_m$.
            $S \leftarrow S + m$
            $b \leftarrow b - w_m$.
            $m \leftarrow m-1$
    return $S$
```

Note that in the recovery the invariant always holds and at the end since $F[0,k] = 0$, we have $p(S) = F[n,B]$.

f. ***Running time and space*** The above pseudocode take $O(nB)$ time and space where n is the number of items.