CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 8

September 30, 2019

Instructor: Sepehr Assadi

1 Introduction to Dynamic Programming

We will start our study of dynamic programming from this lecture. Our running example for this lecture is computing *Fibonacci numbers*, the sequence of numbers: $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$ (do you see the pattern?) The pattern in this sequence is that (except for the first two numbers), each number is obtained by summing the two numbers before it in the sequence. More formally, we can define the n-th Fibonacci number, denoted by F_n, as follows:

$$F_n = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise.} \end{cases}$$

A Recursive Algorithm

Considering Fibonacci numbers are defined recursively, we can perhaps compute F_n easily also using the same recursive formula:

`RecFibo`(n) :

1. If $n = 0$ or $n = 1$, return 1.
2. Otherwise, return `RecFibo`$(n-1)$ + `RecFibo`$(n-2)$.

The correctness of this algorithm is very easy to see. But what can be said about its running time?

Let us write a recurrence for the runtime of this algorithm. We use $T(n)$ to denote the runtime of `RecFibo`(n) (we omit the word 'worst case' because there is only one possible input for n not a whole set (as in sorting arrays of length n) that we may need to take a worst case over). Hence, $T(n) = T(n-1) + T(n-2) + O(1)$. Nnote that we can freely use equality here instead of inequality because again $T(n-1)$ and $T(n-2)$ is precisely the runtime of `RecFibo`$(n-1)$ and `RecFibo`$(n-2)$ not some upper bound that may not be an equality in general.

While we generally did not solve recurrences like the above one, one thing should be clear from this recurrence: $T(n)$ behaves very similarly to F_n itself and in fact $T(n) \geq F_n$. You may have seen in previous courses that $F_n \approx \phi^n$ where $\phi = (\sqrt{5}+1)/2 \approx 1.61\cdots$ is the so-called golden ratio. For our purpose, it suffices to say that $T(n) = \Omega(c^n)$ for some constant $c > 1$.

As we know, exponential functions grow very fast and thus the runtime of the above algorithm becomes *super slow* even for very small values of n. But why this recursive algorithm, which sounds so natural for this problem, runs this slow? The answer to this question turns out to be simple and has to do with the fact that the above algorithm is "forgetful" – we can see this by examining the chain of recursive calls in the algorithm in Figure 1 for the simple case when $n = 5$.

The problem is that the algorithm again and again computes the same value of `RecFibo`(i) for different choices of i as can be seen in Figure 1.

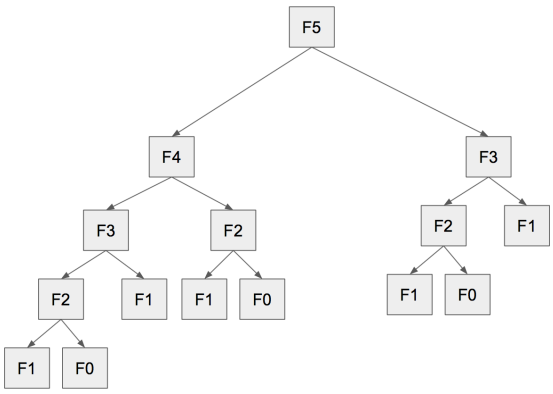

Figure 1: The recursion tree for `RecFibo`(5), i.e., for $n = 5$.

This is indeed problematic: why do we have to recompute the same function again and again? The answer is well we do not have to! We just have to be slightly more careful.

Memoization

We can speed up the computation of this recursive algorithm significantly by simply storing the computed answers in a table and make sure we do not recompute them again and again by consulting this table. Specifically, we can do as follows. We first pick an array $F[0:n]$ (for the purpose of this problem, it helps to set the first index of the array to be 0 instead of 1) and initialize it so that all its entries are written as 'undefined'. We then run the following algorithm:

`MemFibo`(n) :

1. If $F[n] \neq$ 'undefined', return $F[n]$.
2. If $n = 0$ or $n = 1$, let $F[n] = 1$.
3. Otherwise, let $F[n] =$ `MemFibo`$(n-1) +$ `MemFibo`$(n-2)$.
4. Return $F[n]$.

The logic of this algorithm is exactly the same as the logic of `RecFibo` and thus again there is nothing new to prove about the correctness of this algorithm. But what can we say about the runtime of this algorithm? Let us first check the recursion tree for this algorithm in Figure 2. In this new recursion tree, we are not going to recompute an already computed solution and thus for every value of i, there is only one entry for `MemFibo`(i) in this tree.

Analyzing the runtime of `MemFibo`(n) in general is also easy. There are in total n *subproblems* (or recursive calls) when computing the function `MemFibo`(n) (they are `MemFibo`(i) for $1 \leq i \leq n$). Each recursive call takes $O(1)$ time *on its own* ignoring the time needed that takes inside each inner recursion. Since we only compute each recursive call once, the total runtime is then $O(n)$. If you noticed, this is the same exact principle we used before when writing the recursion tree for each recurrence and computed the value of the tree – simply compute the total time spent on each call to the function and add them up to obtain the final runtime.

From an algorithmic point of view, we are already done! We obtained a fast algorithm for computing F_n as we wanted to. However, it is sometimes worth stating this algorithm in a different manner without involving the recursive calls. In other words, instead of relying on recursions to do the job for us when computing `MemFibo`(n), we can explicitly fill out the needed values in the table $F[n]$. The main benefit of this approach

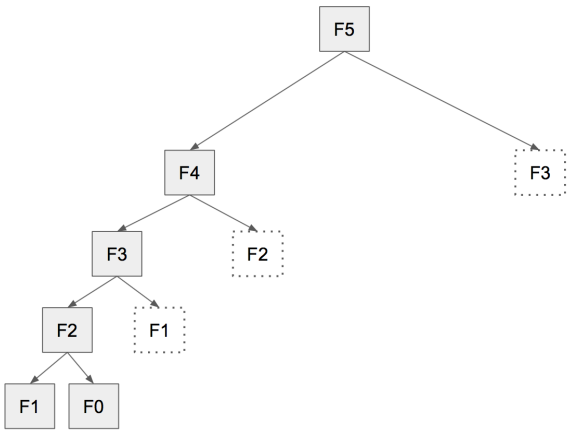

Figure 2: The recursion tree for `MemFibo`(5), i.e., for $n = 5$.

is that it makes understanding the runtime of the algorithm crystal clear (it also has some benefit in terms of reducing the runtime by some constant factor but we can ignore that for the purpose of this course).

Bottom-Up Dynamic Programming

We now replace the recursive calls in `MemFibo` with an *explicit* way of filling the table F it uses directly in a bottom-up fashion, i.e., by starting from the bottom of recursion tree (the base cases) and going to top (the final value we are interested in). The algorithm is as follows:

`DynamicFibo`(n):

1. Let $F[0 : n]$ be an empty array.
2. Let $F[0] = 1$ and $F[1] = 1$.
3. For $i = 2$ to n: let $F[i] = F[i-1] + F[i-2]$.
4. Return $F[n]$.

Again, you can verify (and prove) that the values in table F is exactly the same in both `DynamicFibo` and `MemFibo`. And again, this will imply the correctness of this new algorithm. As for the runtime? Well the algorithm consists of a single for-loop iteration with $\Theta(n)$ iterations, each involving $O(1)$ time, thus the total runtime is $O(n)$.

Concluding Remarks

Congratulations! We finished our first dynamic programming algorithm. All other dynamic programming algorithms we will see in this course follow the same exact pattern:

- Write a *recursive* formula for the problem we want to solve.
- Compute the recursive formula using a recursive function.
- Use a table to store the value of the recursive function (on each separate input) whenever we compute it once. After this, *never* compute that value of the function again and simply return the value from the stored table.

- To perform the step above, we can either use the memoization technique or the iterative approach by explicitly filling out this table. For the purpose of this course, **there is absolutely no difference between these two approaches**: both are considered valid dynamic programming solutions (even though for traditional reasons they are named differently) and both will result in the *asymptotically* same worst case runtime. The decision for using each of the two is ultimately up to you (and may vary from problem to problem) as each one has their own benefits and drawbacks. Generally speaking, memoization seems to be the more straightforward approach (at least to your Instructor) but is harder for analyzing the runtime, while bottom-up dynamic programming requires you to think carefully how the values of the function are obtained recursively from smaller subproblems (although this is generally not a hard task) but then the runtime analysis becomes very easy and straightforward.

  So to sum up this part, you can *always* use either the memoization or the bottom-up dynamic programming (the iterative approach for filling the table), whenever you decide, or asked, to write a dynamic programming algorithm for a problem.

As we saw so far, dynamic programming is simply a way of *speeding up* recursive algorithms by not recomputing the same value again and again. This avoiding re-computation part is rather mechanical and once you see a couple of examples, you will learn how to do this easily. The main part of the approach is however in designing a suitable *recursive formula/function* for the problem at hand and prove its correctness (even though for Fibonacci numbers this part was rather trivial). This is the part which we will focus on in the remainder of this lecture and the next couple of lectures.

2 The Knapsack Problem

We now consider a canonical example of the application of dynamic programming: the so-called knapsack problem defined as follows:

Problem 1. The knapsack problem is defined as follows:

- **Input:** A collection of n items where item i has a positive integer weight w_i and value v_i plus a knapsack of size W.
- **Output:** The maximum value we can obtain by picking a subset S of the items such that the total weight of the items in S is at most W, i.e.,

$$\max_{S \subseteq \{1,\ldots,n\}} \sum_{i \in S} v_i$$
$$\text{subject to} \quad \sum_{i \in S} w_i \leq W.$$

We will design a dynamic programming algorithm for this problem. As we stated before, the *main step* in doing so is to write a *recursive formula/algorithm* for the whole problem in terms of the answers to smaller subproblems. This is done in two steps:

(a) **Specification:** *Describe* the problem that you are designing the recursive formula for, *in coherent and precise English*. In this step, you are *not* writing *how* to solve that problem, but *what* is the problem you are trying to solve. At this point, you should also specify how the answer to the original question can be obtained *if* we have solution for this specification.

(b) **Solution:** Give a *recursive* formula or algorithm for the problem you described in the previous step by solving the *smaller instances* of the *same exact* problem. Remember that we *always* need to prove this recursive formula indeed computes the specification we described earlier for the problem.

Again, to emphasize, once we have the above, our task is almost done: the rest can all be done in a *mechanical* way by using the memoization technique or dynamic programming as we already saw for Fibonacci numbers (and will see shortly for the knapsack problem).

We now apply this method to the knapsack problem.

(a) **Specification:** For any integers $0 \leq i \leq n$ and $0 \leq j \leq W$, define:

- $K(i, j)$: the maximum value we can obtain by picking a subset of the first i items, i.e., items $\{1, \ldots, i\}$, when we have a knapsack of size j.

Note that at this step, we are only specifying, in plain English, the problem we are attempting to solve, not the way we are going to solve it. We should also specify one more thing: how does solving this problem can help us in answering the original question? By returning $K(n, W)$, we can solve the original problem. This is because $K(n, W)$ is the maximum value we can obtain by picking a subset of the first n items (hence all items), when we have a knapsack of size W (which is the original size of our knapsack).

(b) **Solution:** We now write a recursive formula for $K(i, j)$ as follows:

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1, j) & \text{if } w_i > j \\ \max\{K(i-1, j-w_i) + v_i, K(i-1, j)\} & \text{otherwise} \end{cases}.$$

Are we done then? Of course *not*! At this point, we have simply written some recursive formula for $K(i, j)$ but in no way it is clear that whether this formula does what it is supposed to do or not. We thus need to also prove that this is indeed a correct formula for computing $K(i, j)$.

Let us consider the base case of this function first: either when $i = 0$ or $j = 0$. In both cases, we have $K(i, j) = 0$ which is also the value we can achieve by using the first 0 items (i.e., no item at all or $i = 0$) or when the knapsack has no size (i.e., $j = 0$). So the base case of this function matches the specification.

We now consider the larger values of i and j. Suppose first $w_i > j$. In this case, $K(i, j) = K(i-1, j)$. This is correct because we cannot fit item i in a knapsack of size j and thus the best value we can achieve is by picking the best combination of items from the first $i-1$ items, which is captured by $K(i-1, j)$. Thus, whenever $w_i > j$, $K(i, j) = K(i-1, j)$ precisely captures the specification we had.

Finally, we have the case when $w_i \leq j$ (corresponding to the last line of the recursive formula). At this point, we have two options in front of us for maximizing the value of items:

(1) We either pick item i in our solution which leaves us with the first $i-1$ items remaining to choose from next and a knapsack of size $j - w_i$ but we also collected the value v_i. Hence, in this case, we can obtain the value of $K(i-1, j-w_i) + v_i$ (remember that $K(i-1, j-w_i)$ is the largest value we can get by picking a subset of the first $i-1$ items in a knapsack of size $j - w_i$ which is precisely the size of our knapsack after we pick item i in the solution).

(2) The other option would be to not pick i in our solution which leaves us with the first $i-1$ items remaining to choose from and a knapsack of the same size j (we do not collect any value in this case). This is captured by the value $K(i-1, j)$. But which of options (1) or (2) we should choose? Since our goal is to pick the *maximum* value we can get by picking a subset of first i items in a knapsack of size j (the definition of $K(i, j)$), we should also pick the option between (1) and (2) which gives us the *maximum value*; hence, $K(i, j) = \max\{K(i-1, j-w_i) + v_i, K(i-1, j)\}$ as computed by the function as well.

Are we done now? Absolutely yes! We proved that $K(i, j)$ as described by the recursive formula exactly matches the specification we provided before.

Before we move on, let us briefly mention that the proof above is an "induction in disguise": the induction hypothesis is that $K(i, j)$ matches the specification we provided and we simply followed an induction base proof followed by an induction step proof. However, we really do *not* need to each time explicitly call this an induction proof because in these scenarios, the induction hypothesis is always that the recursive function we wrote matches the specification we provided and the rest of the proof

is proving this anyway without explicitly mentioning it. So, to emphasize again, in order to prove the correctness of the recursive formula, we can simply explain the its logic and show that why this matches the specification we had in both the base case and the other remaining cases.

So we are done with the main step of the problem in designing a recursive formula for our problem. We can now turn this into a dynamic programming easily.

Memoization. We store a two-dimensional table $D[0:n][0:W]$ initialized with 'undefined' everywhere (again we index the table starting from 0 not 1) and use the following memoization algorithm: (note that we are not giving the parameters n, W as well as the arrays of weights and values to the function and instead think of them as some global parameters)

`MemKnapsack`(i, j):

1. If $D[i][j] \neq$ 'undefined', return $D[i][j]$.
2. If $i = 0$ or $j = 0$, let $D[i][j] = 0$.
3. Otherwise, if $w_i > j$, let $D[i][j] =$ `MemKnapsack`$(i-1, j)$;
4. Else, let $D[i][j] = \max\{$`MemKnapsack`$(i-1, j-w_i) + v_i,$ `MemKnapsack`$(i-1, j)\}$.
5. Return $D[i][j]$.

It is immediate to verify that `MemKnapsack`$(i, j) = K(i, j)$ (the recursive formula we defined above) for all valid choices of i, j. Hence, the final solution to our problem is to simply return `MemKnapsack`(n, W) $(= K(n, W)$ which we said earlier is the value we are interested in). There is nothing left to prove for the correctness of this algorithm: we already did it when proving $K(i, j)$ indeed matches its description which is the maximum value we can get by picking a subset of the first i items in a knapsack of size j.

What about the runtime of this algorithm? There are n choices for i and W choices for j so there are in total $n \cdot W$ subproblems. Each subproblem also, ignoring the time it takes to do the inner recursions, takes $O(1)$ time. Hence, the runtime of the algorithm is $O(nW)$.

At this point, we are done with our dynamic programming solution for the knapsack problem and it runs in $O(nW)$ time (again, remember that memoization is a form of dynamic programming). In the following, we show how to obtain the bottom-up dynamic programming solution as well but that is *not* necessary.

Bottom-Up Dynamic Programming. We use the following algorithm by filling up the table $D[i][j]$ of the previous recursive algorithm in a bottom-up fashion ourself, instead of letting the recursion do it for us!

`DynamicKnapsack`:

1. Let $D[0:n][0:W]$ be an empty two-dimensional array (we index the table starting from 0 here).
2. For $i = 1$ to n: let $D[i][0] = 0$.
3. For $j = 1$ to W: let $D[0][j] = 0$.
4. For $i = 1$ to n:
   (a) For $j = 1$ to W:
      i. If $w_i > j$, let $D[i][j] = D[i-1][j]$.
      ii. Else, let $D[i][j] = \max\{D[i-1][j-w_i] + v_i, D[i-1][j]\}$.
5. Return $D[n][W]$.

Again, we have to ensure that $D[i][j] = K(i, j)$ – unlike the case for memoization where this was straightforward, here it may not be the case anymore. However, this can be done in a careful manner also by making sure that whenever we apply the recursive rules (the lines in the inner for-loops), the values we are updating $D[i][j]$ from, i.e., $D[i-1][j-w_i]$ and $D[i-1][j]$, are already equal to their correct value $K(i-1, j-w_i)$ and $K(i-1, j)$. It is easy to verify that the runtime of this algorithms is $O(nW)$ also.

This finishes another dynamic programming solution for knapsack, this time using the bottom-up dynamic programming approach instead of the memoization technique we used before.

Final note: why not a greedy solution for knapsack? During today's lecture, a couple of you suggested the following greedy solution: sort the items based on the value v_i/w_i (i.e., the ratio of value over weight) and pick the items according to this order until we fill up the knapsack. Let us consider the following example that shows this algorithm does *not* necessarily finds the correct solution every time.

Suppose we have $n = 4$ items with weights $5, 5, 4, 4$ and values $5, 5, 3, 3$ and the total knapsack size of $W = 8$. The optimum solution for this instance is to pick the last two items which both fit the knapsack (as their total weight is $4 + 4 = 8 = W$) and have value $3 + 3 = 6$. On the other hand, if we sort the items based on their v_i/w_i values, the first two items have this ratio equal to 1 while the last two have ratio $3/4$; hence, we place either item 1 or 2 in the knapsack and at this point no other item can fit the knapsack. The returned solution would thus have value $5 < 6$ which means the algorithm does *not* find the optimum solution.

In general, *greedy algorithms rarely work correctly* (although there are cases that they do and we will study them later in the course).