All About Dynamic Programming

Dynamic programming is breaking down a problem into smaller sub-problems, solving each sub-problem and storing the solutions to each of these sub-problems in a dictionary/array/etc so each sub-problem is only calculated once.

It is both a **mathematical optimization** (https://en.wikipedia.org/wiki/Mathematical_optimization) technique and a computer programming technique.

As an algorithmic strategy, Dynamic Programming can often significantly improve the runtime of an algorithm. This can mean turning an exponential time (2^n) algorithm into a linear or quadratic time algorithm. This is especially important as we start covering more *Optimization* Problems, where we must find the best solution among many possible candidates.

Why Is Dynamic Programming Called Dynamic Programming?

Richard Bellman (https://en.wikipedia.org/wiki/Richard_E._Bellman) invented DP in the 1950s. **Bellman named it Dynamic Programming** (https://en.wikipedia.org/wiki/Dynamic_programming#History) because at the time, RAND (his employer), disliked mathematical research and didn't want to fund it. He named it Dynamic Programming to hide the fact he was really doing mathematical research.

Bellman explains the reasoning behind the term Dynamic Programming in his autobiography, Eye of the Hurricane: An Autobiography (1984, page 159). He explains:

> "I spent the Fall quarter (of 1950) at RAND.
>
> My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from?
>
> The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely.
>
> His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I

> choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

What are Sub-Problems?

Sub-problems are smaller versions of the original problem. Let's see an example. With the equation below:

```
1 + 2 + 3 + 4
```

We can break this down to:

```
1 + 2
3 + 4
```

Once we solve these two smaller problems, we can add the solutions to these sub-problems to find the solution to the overall problem.

Notice how these sub-problems breaks down the original problem into components that build up the solution. This is a small example, but it illustrates the beauty of Dynamic Programming well. If we expand the problem to adding hundreds or thousands of numbers it becomes clearer why we need Dynamic Programming. Take this example:

```
6 + 5 + 3 + 3 + 2 + 4 + 6 + 5
```

We have `6 + 5` twice. The first time we see it, we work out `6 + 5`. When we see it the second time we think to ourselves:

> "Ah, 6 + 5. I've seen this before. It's 11!"

In Dynamic Programming we store the solution to the problem, so we do not need to recalculate it. By finding the solutions for every single sub-problem, we can tackle the original problem itself.

Why Do We Store Partial Answers?

Let's see why storing answers to solutions make sense. We're going to look at a famous problem, ***Fibonacci sequence*** ***(https://www.mathsisfun.com/numbers/fibonacci-sequence.html)***. This problem is normally solved using **Divide and Conquer**.

There are 3 main parts to **Divide and Conquer** **(https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm)**:

1. **Divide** - Break the problem into smaller sub-problems of the same type.
2. **Conquer** - Solve the sub-problems recursively.
3. **Combine** - Combine all the sub-problems to create a solution to the original problem.

Dynamic programming avoids having to resolve problems in step 2.

Let us look at an example. The Fibonacci sequence is a sequence of numbers. It's the current number plus the previous number. We start at `1`.

```
1 + 0 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
```

In Python, this is:

```python
# Basic, unoptimized recursive fibonacci function
def fibonacci(n:int) -> int:
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Let's calculate `fibonacci(4)`. In an execution tree, this looks like:

We calculate `fibonacci(2)` twice, once in order to calculate `fibonacci(4)` and once to calculate `fibonacci(3)`. On bigger inputs (such as `fibonacci(10)`) the repetition builds up; without dynamic programming, the time complexity is 2^n. The purpose of dynamic programming is to not calculate the same thing twice.

Instead of calculating `fibonacci(2)` twice, we will store the solution somewhere and only calculate it once. We can do this either using **Memoization** or **Tabulation**.

- Memoization (Top-down): Recursively break down a problem, and whenever you see a solution store it in case it becomes useful later.
- Tabulation (Bottom-up): Iteratively fill up a table of ordered solutions, so you can refer to previous solutions.

Both approaches tend to be equivalent, although they often look very different in practice.

How to Apply Memoization

First, we will **memoize** our previous recursive solution by introducing a dictionary that serves as a [cache (https://en.wikipedia.org/wiki/Cache_(computing))](https://en.wikipedia.org/wiki/Cache_(computing)) .

```
# Memoized Fibonacci
previous_answers = {}
def fibonacci(n:int) -> int:
    if n == 0 or n == 1:
        return n
    # If we have seen the solution before...
    elif n in previous_answers:
        # Return that solution!
        return previous_answers[n]
    else:
        answer = fibonacci(n-1) + fibonacci(n-2)
        # If this is a new solution, we better remember it for later
        previous_answers[n] = answer
        return answer
```

Abusing a global variable can be risky, although it is technically safe to do. As long as we know the input `n` should *always* return the same value consistently ("deterministically"), then we do not have to worry about the cache being out-of-date or inaccurate. However, if you would like to ever avoid having that global state, you could wrap the code in a "closure", so that a local variable is available inside the actual function but not outside:

```
# "Closure" version of Memoized Fibonacci that avoids global state
def fibonacci(n:int) -> int:
    previous_answers = {}
    # Inner function will have access to previous_answers
    def fib_helper(n: int) -> int:
        if n == 0 or n == 1:
            return n
        elif n in previous_answers:
            return previous_answers[n]
        else:
            answer = fibonacci(n-1) + fibonacci(n-2)
            previous_answers[n] = answer
            return previous_answers[n]
    return fib_helper(n)
# But previous_answers will not exist outside of the function call!
```

Memoization is convenient because we can usually convert an existing recursive solution with minor additions. In fact, Python supports a special decorator for exactly this purpose:

```
# Memoized recursive fibonacci via a decorator

from functools import cache

@cache
def fibonacci(n:int) -> int:
    if n == 0 or n == 1:
        return n
```

```
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

How to Apply Tabulation

In order to tabulate a function, we need to have a data structure to match inputs to solutions, just like in memoization. However, we will intentionally fill this table iteratively starting from the smallest possible input and moving forward. Although we could technically use a dictionary here, we will instead use a list. Not only does this have a better worst case time complexity, but using a list (often of multiple dimensions) is more common for tabulation. In fact, tabulation pretty much requires that your input be describable as indices of a table.

```
# Tabulated fibonacci
def fibonacci(n):
    # Initially fill the list with zeros
    answers = [0 for _ in range(n+1)]
    # Set the base cases using multiple assignment
    answers[0], answers[1] = 0, 1
    # Iterate through the rest of the indices
    for i in range(2, n+1):
        # Calculate the current cell based on the previous two
        answers[i] = answers[i-1] + answers[i-2]
    # Answer ends up in the last cell
    return answers[n]
```

How to Identify Dynamic Programming Problems

Dynamic Programming cannot improve the solution to every problem. The question is then:

> "When should I solve a problem with dynamic programming?"

Before we even start to plan the problem as a dynamic programming problem, think about what the brute force solution might look like. Are sub steps repeated in the brute-force solution? If so, we try to imagine the problem as a dynamic programming problem.

Mastering dynamic programming is all about understanding the problem. List all the inputs that can affect the answers. Once we've identified all the inputs and outputs, try to identify whether the problem can be broken into subproblems. If we can identify subproblems, we can probably use Dynamic Programming.

Then, figure out what the recurrence is and solve it. When we're trying to figure out the recurrence, remember that whatever recurrence we write has to help us find the answer. Sometimes the answer will be the result of the recurrence, and sometimes we will have to get the result by looking at a few results from the recurrence.

Dynamic Programming can solve many problems, but that does not mean there isn't a more efficient solution out there. Solving a problem with Dynamic Programming feels like magic, but remember that dynamic programming is merely a clever brute force. Sometimes it pays off well, and sometimes it helps only a little.

How to Solve Problems using Dynamic Programming

Now we have an understanding of what Dynamic programming is and how it generally works. Let's look at to create a Dynamic Programming solution to a problem. We're going to explore the process of Dynamic Programming using the ***Weighted Interval Scheduling Problem (https://courses.cs.washington.edu/courses/cse521/13wi/slides/06dp-sched.pdf)*** .

Pretend you're the owner of a dry cleaner. You have n customers come in and give you clothes to clean. You can only clean one customer's pile of clothes (PoC) at a time. Each pile of clothes, i , must be cleaned at some pre- determined start time s_i and some predetermined finish time f_i.

Each pile of clothes has an associated value, v_i, based on how important it is to your business. For example, some customers may pay more to have their clothes cleaned faster. Or some may be repeating customers and you want them to be happy.

As the owner of this dry cleaners you must determine the optimal schedule of clothes that maximises the total value of this day. This problem is a re-wording of the *Weighted Interval scheduling problem*.

You will now see 4 steps to solving a Dynamic Programming problem. Sometimes, you can skip a step. Sometimes, your problem is already well-defined and you don't need to worry about the first few steps.

Step 1. Write the Problem out

Grab a piece of paper. Write out:

- What is the problem?
- What are the sub-problems?
- What would the solution roughly look like?

In the dry cleaner problem, let's put down into words the subproblems. What we want to determine is the maximum value schedule for each pile of clothes such that the clothes are sorted by start time.

Why sort by start time? Good question! We want to keep track of processes which are currently running. If we sort by finish time, it doesn't make much sense in our heads. We could have 2 with similar finish times, but different start times. Time moves in a linear fashion, from start to finish. If we have piles of clothes that start at 1 pm, we know to put them on when it reaches 1pm. If we have a pile of clothes that finishes at 3 pm, we might need to have put them on at 12 pm, but it's 1pm now.

We can find the maximum value schedule for piles `n-1` through to `n`. And then for `n-2` through to `n`. And so on. By finding the solution to every single sub-problem, we can tackle the original problem itself. The maximum value schedule for piles `1` through `n`. Sub-problems can be used to solve the original problem, since they are smaller versions of the original problem.

With the interval scheduling problem, the only way we can solve it is by brute-forcing all subsets of the problem until we find an optimal one. What we're saying is that instead of brute-forcing one by one, we divide it up. We brute force from `n-1` through to `n`. Then we do the same for `n-2` through to `n`. Finally, we have loads of smaller problems, which we can solve dynamically. We want to build the solutions to our sub-problems such that each sub-problem builds on the previous problems.

2. Mathematical Recurrences

Mathematical recurrences (https://en.wikipedia.org/wiki/Recurrence_relation) are used to:

> Define the running time of a divide and conquer technique

Recurrences are also used to define problems. If it's difficult to turn your subproblems into a formula, then it may be the wrong subproblem.

There are 2 steps to creating a mathematical recurrence:

1: Define the Base Case

Base cases are the smallest possible denomination of a problem.

When creating a recurrence, ask yourself these questions:

> "What decision do I make at step 0?"

It doesn't have to be 0. The base case is the smallest possible denomination of a problem. We saw this with the Fibonacci sequence. The base was:

- If n == 0 or n == 1, return 1

It's important to know where the base case lies, so we can create the recurrence. In our problem, we have one decision to make:

- Put that pile of clothes on to be washed

or

- Don't wash that pile of clothes today

If n is 0, that is, if we have 0 PoC then we do nothing. Our base case is:

```
if n == 0, return 0
```

2: What Decision Do I Make at Step n?

Now we know what the base case is, if we're at step `n` what do we do? For each pile of clothes that is compatible with the schedule so far. Compatible means that the start time is after the finish time of the pile of clothes currently being washed. The algorithm has 2 options:

1. Wash that pile of clothes
2. Don't wash that pile of clothes

We know what happens at the base case, and what happens else. We now need to find out what information the algorithm needs to go backwards (or forwards).

> "If my algorithm is at step `i`, what information would it need to decide what to do in step `i+1`?"

To decide between the two options, the algorithm needs to know the next compatible PoC (pile of clothes). The next compatible PoC for a given pile, `p`, is the PoC, `n`, such that s_n (the start time for PoC n) happens after f_p (the finish time for PoC `p`). The difference between s_n and f_p should be minimised.

In English, imagine we have one washing machine. We put in a pile of clothes at 13:00. Our next pile of clothes starts at 13:01. We can't open the washing machine and put in the one that starts at 13:00. Our next compatible pile of clothes is the one that starts after the finish time of the one currently being washed.

> "If my algorithm is at step `i`, what information did it need to decide what to do in step `i-1`?"

The algorithm needs to know about future decisions. The ones made for PoC `i` through `n` to decide whether to run or not run PoC `i-1`.

Now that we've answered these questions, we've started to form a recurring mathematical decision in our mind. If not, that's also okay, it becomes easier to write recurrences as we get exposed to more problems.

Here's our recurrence:

```
OPT(i) = {  0, if i =0
            max(v[i] + OPT(next[i]), OPT(i+1)), if n > 1
         }
```

Let's explore in detail what makes this mathematical recurrence. `OPT(i)` represents the maximum value schedule for PoC `i` through to `n` such that PoC is sorted by start times. `OPT(i)` is our subproblem from earlier.

We start with the base case. All recurrences need somewhere to stop. If we call `OPT(0)` we'll be returned with `0`.

To determine the value of `OPT(i)`, there are two options. We want to take the maximum of these options to meet our goal. Our goal is the *maximum value schedule* for all piles of clothes. Once we choose the option that gives the maximum result at step `i`, we memoize its value as `OPT(i)`.

Mathematically, the two options - run or not run PoC `i`, are represented as:

```
v[i] + OPT(next[n])
```

This represents the decision to run PoC `i`. It adds the value gained from PoC `i` to `OPT(next[n])`, where `next[n]` represents the next compatible pile of clothing following PoC `i`. When we add these two values together, we get the maximum value schedule from `i` through to n such that they are sorted by start time if `i` runs.

Sorted by start time here because `next[n]` is the one immediately after `v[i]`, so by default, they are sorted by start time.

```
OPT(i + 1)
```

If we decide not to run `i`, our value is then `OPT(i + 1)`. The value is not gained. `OPT(i + 1)` gives the maximum value schedule for `i+1` through to `n`, such that they are sorted by start times.

3. Determine the Dimensions of the Memoization Array and the Direction in Which It Should Be Filled

The solution to our Dynamic Programming problem is `OPT(1)`. We can write out the solution as the maximum value schedule for PoC `1` through `n` such that PoC is sorted by start time. This goes hand in hand with "maximum value schedule for PoC `i` through to `n`".

From step 2:

```
OPT(1) = max(v[1] + OPT(next[1]), OPT(2))
```

Going back to our Fibonacci numbers earlier, our Dynamic Programming solution relied on the fact that the Fibonacci numbers for `` `0 `` through to `n - 1` were already tabulated. That is, to find `fibonacci(5)` we already tabulated `fibonacci(0)`, `fibonacci(1)`, `fibonacci(2)`, `fibonacci(3)`, `fibonacci(4)`. We want to do the same thing here.

The problem we have is figuring out how to fill out a tabulation table. In the scheduling problem, we know that `OPT(1)` relies on the solutions to `OPT(2)` and `OPT(next[1])`. PoC `2` and `next[1]` have start times after PoC `1` due to sorting. We need to fill our tabulation table from `OPT(n)` to `OPT(1)`.

We can see our array is one dimensional, from `1` to `n`. But, if we couldn't see that we can work it out another way. The dimensions of the array are equal to the number and size of the variables on

which `OPT(x)` relies. In our algorithm, we have `OPT(i)` - one variable, `i`. This means our array will be 1-dimensional and its size will be `n`, as there are `n` piles of clothes.

If we know that `n=5`, then our tabulation table might look like this:

```
answers = [0, OPT(1), OPT(2), OPT(3), OPT(4), OPT(5)]
```

`0` is also the base case. `memo[0] = 0`, per our recurrence from earlier.

4. Coding Our Solution

We need to go from our recurrence relation to actual code that can determine the set of jobs to take. Worded differently, we need to find the maximum profit with and without the inclusion of job[i]. Actually determining the jobs to take from that is a related, but difficult, problem.

First, let's define what a "job" is. As we saw, a job consists of 3 things:

```
# Class to represent a job (simple data-only class)
class Job:
    def __init__(self, start: int, finish: int, profit: int):
        self.start: int = start
        self.finish: int = finish
        self.profit: int = profit
```

From there, we define our desired function. Notice that it returns the optimal *profit* of the given jobs, not the set of jobs to take.

```
def schedule(jobs: list[Job]) -> int:
    pass
```

We ultimately need a data structure to hold our partial solutions. Since we are doing tabulation, this is likely to be a list of some kind. In this case, we need `n` storage slots, so we will create a list of this size initialized to zero.

```
def schedule(jobs: list[Job]) -> int:
    # Create an array to store solutions of subproblems. job_profits[i]
    # stores the profit for jobs till jobs[i] (including jobs[i])
    n = len(jobs)
    job_profits = [0 for _ in range(n)]
```

Our next step is to set the base case in the tabulation list. We sort the jobs by start time, and set table[0] to be the profit of job[0]. Since we've sorted by start times, the first compatible job is always job[0].

```
def schedule(jobs: list[Job]) -> int:
    ...
    jobs = sorted(jobs, key = lambda a_job: a_job.start)
    job_profits[0] = jobs[0].profit
```

Our next step is to fill in the entries using the recurrence we learnt earlier. To find the next compatible job, we're using Binary Search. In the full code posted later, it'll include this. For now, let's worry about understanding the algorithm.

If the next compatible job returns `-1`, that means that all jobs before the index, `i`, conflict with it (so cannot be used). `Inclprof` means we're including that item in the maximum value set. We then store it in table[i], so we can use this calculation again later.

```
def schedule(jobs: list[Job]) -> int:
    ...
    # Fill entries in table[] using recursive property
    for i in range(1, n):
        # Find profit including the current job
        profit_if_included = jobs[i].profit
        next_compatible_job_index = binary_search(jobs, i)
        if next_compatible_job_index != -1:
            profit_if_included += job_profits[next_compatible_job_index]
        else:
            pass # All previous jobs conflict with this one!

        # Store maximum of including and excluding the job
        job_profits[i] = max(profit_if_included, job_profits[i - 1])
```

Our final step is then to return the profit of all items up to `n-1`.

```
def schedule(jobs: list[Job]) -> int:
    ...
    return job_profits[n-1]
```

The full code can be seen here:
https://gist.github.com/acbart/c3c4cc7e6bb8275d071303dad0f82102 (https://gist.github.com/acbart/c3c4cc7e6bb8275d071303dad0f82102)

Time Complexity of a Dynamic Programming Problem

For our original problem, the Weighted Interval Scheduling Problem, we had n piles of clothes. Each pile of clothes is solved in constant time. The time complexity is:

```
O(n) + O(1) = O(n)
```

For Fibonacci, we were also able to get a linear time result. This does not mean that Dynamic Programming always produces linear time complexities, but it should help illustrate the huge gains possible with this approach.

About

This page was heavily adapted from the following blog post: **"What Is Dynamic Programming With Python Examples"** **(https://skerritt.blog/dynamic-programming/)** by Brandon Skeritt, published on December 31, 2019. I reduced out unnecessary content, fixed wording to be consistent with the terms of art, and cleaned up the Python examples.