

# Hao's Guide - Algorithm Crash Course - Dynamic Programming

---

## Introduction

Similar to greedy, dynamic programming are used when problems require minimisation and maximisation of an outcome. Hence dynamic programming algorithms are also **optimisation** problem (Dynamic programming shall be reference as DyP from now on, giggidy).

However, these problems have one unique difference to greedy. Greedy processes are done with no knowledge of previous decisions, but DyP is done with knowledge of *every* previous decisions. What does this mean you may ask. This will be made clear in the second example.

This document will explain the basics and the considerations of both these topics using question examples and a step-by-step worked-out thought process.

## Dynamic Programming

### What is it

In dynamic programming, unlike greedy, the best decisions are based on the previous decisions which each could have their own different outcome depending on what is decided. That sentence was very long and confusing, but lets explain with some examples.

### Dynamic Programming: Weird Money and Change

Remember in greedy method, we had the money example:

One need to is given \$16 in the least unit of currency. The units of currency available being \$10, \$5 and \$1. The greedy algorithm is simple: we start from the highest unit of currency at each point of the transaction until you completely pay off the \$16.

But now, let's change up this problem and you will see why greedy is not the be all and end all.

Suppose that the problem is the same, \$16 to give with the least units of currency. However, this time, because of bad government planning, the available currency is \$10, \$8, and \$1.

Have you seen the problem yet?

If you were to apply greedy, you would give 1 x 10\$ and 6 x \$1, totaling 7 units of currency. But instead you could have given 2 x \$8 currency which is clearly the least number of currency.

Oh no, you might be thinking: "Dynamic programming looks like its gonna be a pain in the ass, why did I do this course?". However, its actually not too tricky. All you have to do is 3 steps

1. Decide on the subproblem
2. Determine the base case(s)
3. Define the recursive case(s)

Yep, these are recursion problems.

## Dynamic Programming: The Hungry Frog

Let me demonstrate with simple example with my demonstration partner, *the frog*

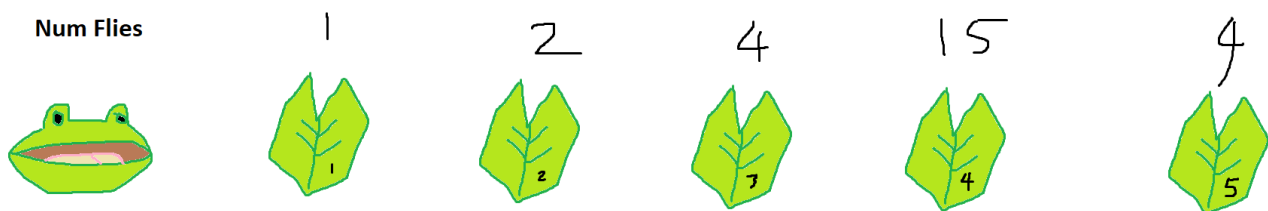
Suppose you have a frog on a lily pad. In front of the frog, are a line of lily pads, each with a number of flies available to consume on them. The frog want to consume the most number of flies of course, but can only jump forward 2 or 3 lily pads (I.E Lily pad 1 -> Lily pad 3 or Lily pad 4). There are N number of lily pads. In what order, should the frog jump such that it consumes the most flies?

Assumptions:

- You cannot jump backwards
- Frog starts at Lily pad 0, with 0 flies consumed

You might think, "Looks like greedy to me though." So let's test that hypothesis.

Suppose the number of flies are arranged in this order (Frog is on Lily pad 0):



Our greedy method is: "grab the most number of flies each time". So this would mean, between the two Lily pads we can pick, Lily pad 2 or Lily pad 3. Since Lily pad 3 has 4 flies, we go for that. Now we are on Lily pad 3, the next is Lily pad 5 or 6, and since Lily pad 5 has 4 flies compared to Lily pad 6's 2 flies, we jump to Lily pad 5. Total we consumed 8 flies.

Amazing, but you might have noticed you your frog just missed out on a fat buffet of 15 flies on Lily pad 4. But since you chose Lily pad 3, you could not jump to Lily pad 4 can eat 15 flies. You needed to make an ungreedy decision in the beginning to reap rewards later.

Now, before you go check the census date to drop the course, let's think for a moment how to solve this problem. How can we ensure that for each Lily pad we are making the most optimal solution for future.

Let's look at our 3 steps:

### Step 1: The subproblem:

So instead of saying what is the optimal amount of flies that can be consumed overall, let's just ask a smaller question:

What is the optimal amount of flies I can eat up to each lily pad?

Let's think about it:

- Lily pad 1: We can ignore Lily pad 1 since we can't get to it
- Lily pad 2: We can get 2 flies
- Lily pad 3: We can get 4 flies
- Lily pad 4: We can consume  $15 + 2 = 17$  flies suppose that we jump from Lily pad 2
- Lily pad 5: We can consume either 6 flies if we jumped from Lily pad 2 or 8 flies if we jumped from Lily pad 3, so the optimal amount of flies is 8.

But how do we figure out all the optimal case at each lily pad? How do we check? Exactly: Recursion

### Step 2: The base case:

Right now the idea of recursion still probably haven't clicked yet, so bare with me, I'm getting there. Trust me.

With any recursion we need a base case, since the frog is always starting on the Lily pad 0, where there are 0 flies. The base case is:

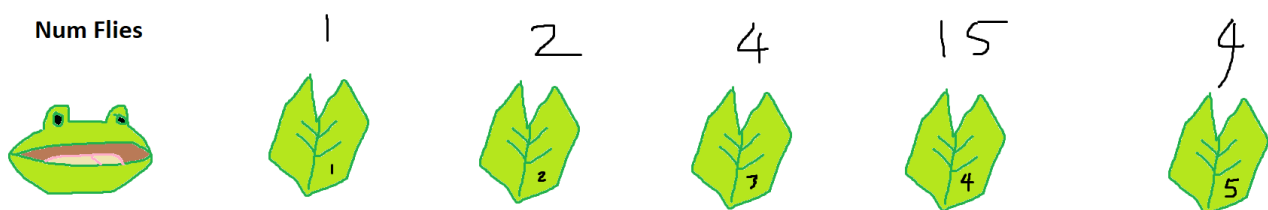
$\text{Lily pad}(0) = 0$

### Step 3: The recursive case

Here, you will finally see the build up with been edging towards.

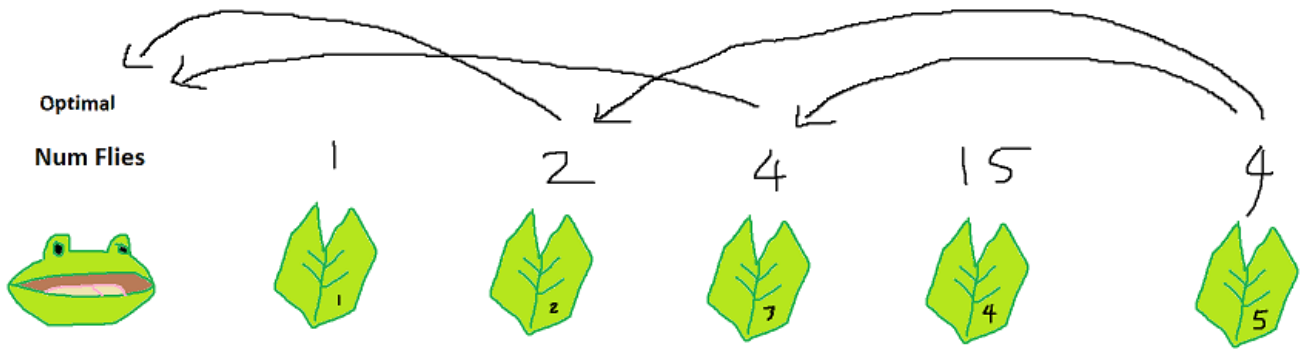
Let's ask a simple question, what is the optimal number of flies we can eat at Lily pad 5?

Diagram below, if you'd forgot:

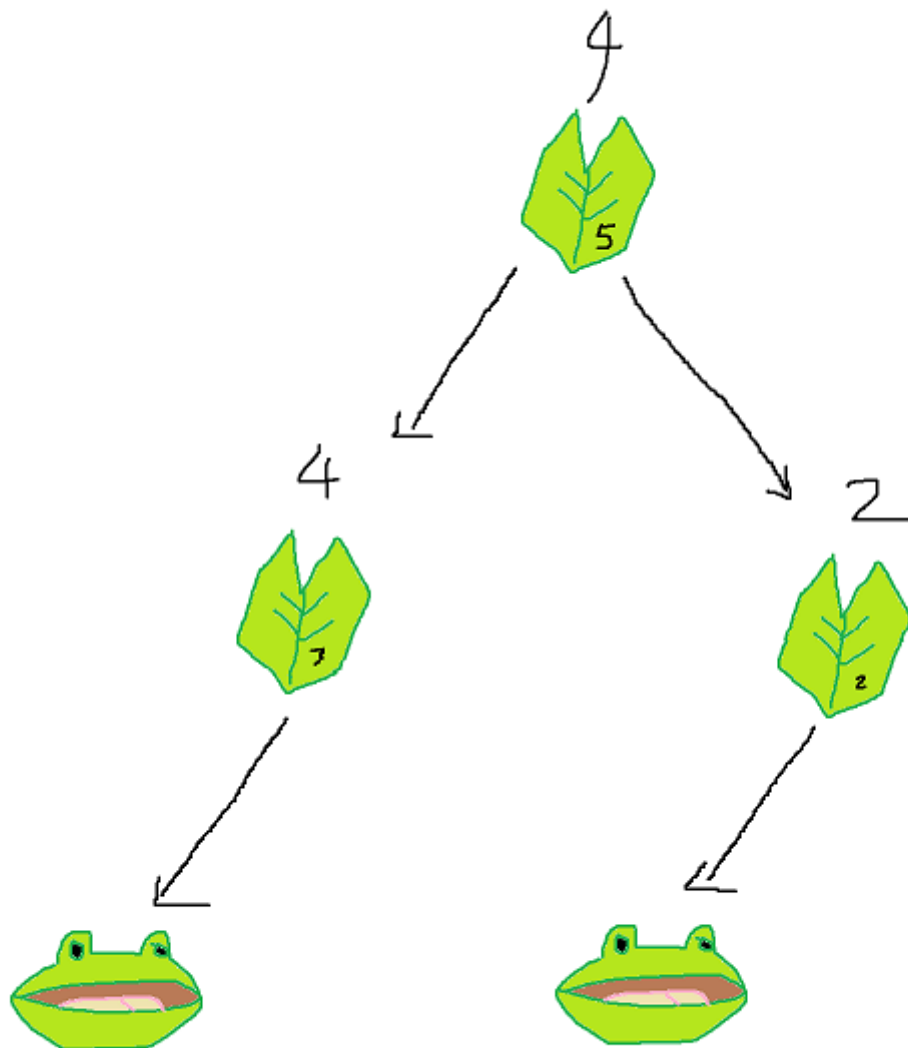


Well, it depends, we could have jumped from either Lily pad 2 or Lily pad 3. Okay, but which of these two is the better option, well obviously Lily pad 3, since it has more flies.

To visualise the previous statement:



or you could do it like this



Now do you see what I mean?

You can find the optimal amount of flies by generating using the previous Lily pad optimal flies.

Obviously, at this few lily pads, it's a bit redundant, but if you're dealing with a larger amount of lily pads, it becomes more apparent.

If you do the same recursive tree structure for every Lilypad, then you will find the most optimal amount of flies you can eat up to each one!

## Conclusion

So in words, your solution is:

Find the most optimal number of flies up to lilypad 0, by adding the most optimal of the two Lilypad before which can jump from. Store the recursive call back order of lilypad index to find the route to lead to the most optimal lilypad

In maths land, we will say:

$$\text{opt}(i) = f_i + \max\{\text{opt}(i - 2), \text{opt}(i - 3)\}$$

Where  $i$  is the lilypad index

## Notes:

- There can always be more base cases, depending on your circumstance
- The recursive cases are the hardest to reason. My advice: When in doubt, draw a tree.
- There can be a large number of recursive cases, it won't always so neatly become a binary tree.

# Good Luck

