EMERGING METHODS FOR EARLY DETECTION FOREST FIRES

TEAM ID:PNT2022TMID30386

DYNAMIC PROGRAMMING

Despite having significant experience building software products, many engineers feel jittery at the thought of going through a coding interview that focuses on algorithms. I've interviewed hundreds of engineers at Refdash, Google, and at startups I've been a part of, and some of the most common questions that make engineers uneasy are the ones that involve Dynamic Programming (DP).

Many tech companies like to ask DP questions in their interviews. While we can debate whether they're effective in evaluating someone's ability to perform in an engineering role, DP continues to be an area that trips engineers up on their way to finding a job that they love.

Dynamic Programming — Predictable and Preparable

One of the reasons why I personally believe that DP questions might not be the best way to test engineering ability is that they're predictable and easy to pattern match. They allow us to filter much more for preparedness as opposed to engineering ability.

These questions typically seem pretty complex on the outside, and might give you an impression that a person who solves them is very good at algorithms. Similarly, people who may not be able to get over some mind-twisting concepts of DP might seem pretty weak in their knowledge of algorithms.

The reality is different, and the biggest factor in their performance is preparedness. So let's make sure everyone is prepared for it. Once and for all.

7 Steps to solve a Dynamic Programming problem

In the rest of this post, I will go over a recipe that you can follow to figure out if a problem is a "DP problem", as well as to figure out a solution to such a problem. Specifically, I will go through the following steps:

How to recognize a DP problem

Identify problem variables

Clearly express the recurrence relation

Identify the base cases

Decide if you want to implement it iteratively or recursively

Add memoization

Determine time complexity

Sample DP Problem

For the purpose of having an example for abstractions that I am going to make, let me introduce a sample problem. In each of the sections, I will refer to the problem, but you could also read the sections independently of the problem.

Here are the rules:

1) You’re given a flat runway with a bunch of spikes in it. The runway is represented by a boolean array which indicates if a particular (discrete) spot is clear of spikes. It is True for clear and False for not clear.

Example array representation:

2) You’re given a starting speed S. S is a non-negative integer at any given point, and it indicates how much you will move forward with the next jump.

3) Every time you land on a spot, you can adjust your speed by up to 1 unit before the next jump.

4) You want to safely stop anywhere along the runway (does not need to be at the end of the array). You stop when your speed becomes 0. However, if you land on a spike at any point, your crazy bouncing ball bursts and it’s game over.

The output of your function should be a boolean indicating whether we can safely stop anywhere along the runway.

Step 1: How to recognize a Dynamic Programming problem

First, let’s make it clear that DP is essentially just an optimization technique. DP is a method for solving problems by breaking them down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, you simply look up the previously computed solution. This saves computation time at the expense of a (hopefully) modest expenditure in storage space.

Recognizing that a problem can be solved using DP is the first and often the most difficult step in solving it. What you want to ask yourself is whether your problem solution can be expressed as a function of solutions to similar smaller problems.

In the case of our example problem, given a point on the runway, a speed, and the runway ahead, we could determine the spots where we could potentially jump next. Furthermore, it seems that whether we can stop from the current point with the current speed depends only on whether we could stop from the point we choose to go to next.

That is a great thing, because by moving forward, we shorten the runway ahead and make our problem smaller. We should be able to repeat this process all the way until we get to a point where it is obvious whether we can stop.

Recognizing a Dynamic Programming problem is often the most difficult step in solving it. Can the problem solution be expressed as a function of solutions to similar smaller problems?

Step 2: Identify problem variables

Now we have established that there is some recursive structure between our subproblems. Next, we need to express the problem in terms of the function parameters and see which of those parameters are changing.

Typically in interviews, you will have one or two changing parameters, but technically this could be any number. A classic example of a one-changing-parameter problem is “determine an n-th Fibonacci number”. Such an example for a two-changing-parameters problem is “Compute edit distance between strings”. If you’re not familiar with these problems, don’t worry about it.

A way to determine the number of changing parameters is to list examples of several subproblems and compare the parameters. Counting the number of changing parameters is valuable to determine the number of subproblems we have to solve. It's also important in its own right in helping us strengthen the understanding of the recurrence relation from step 1.

In our example, the two parameters that could change for every subproblem are:

Array position (P)

Speed (S)

One could say that the runway ahead is changing as well, but that would be redundant considering that the entire non-changing runway and the position (P) carry that information already.

Now, with these 2 changing parameters and other static parameters, we have the complete description of our sub-problems.

Identify the changing parameters and determine the number of subproblems.

Step 3: Clearly express the recurrence relation

This is an important step that many rush through in order to get into coding. Expressing the recurrence relation as clearly as possible will strengthen your problem understanding and make everything else significantly easier.

Once you figure out that the recurrence relation exists and you specify the problems in terms of parameters, this should come as a natural step. How do problems relate to each other? In other words, let's assume that you have computed the subproblems. How would you compute the main problem?

Here is how we think about it in our sample problem:

Because you can adjust your speed by up to 1 before jumping to the next position, there are only 3 possible speeds, and therefore 3 spots in which we could be next.

More formally, if our speed is S, position P, we could go from (S, P) to:

(S, P + S); # if we do not change the speed

(S — 1, P + S — 1); # if we change the speed by -1

(S + 1, P + S + 1); # if we change the speed by +1

If we can find a way to stop in any of the subproblems above, then we can also stop from (S, P). This is because we can transition from (S, P) to any of the above three options.

This is typically a fine level of understanding of the problem (plain English explanation), but you sometimes might want to express the relation mathematically as well. Let’s call a function that we’re trying to compute canStop. Then:

canStop(S, P) = canStop(S, P + S) || canStop(S — 1, P + S — 1) || canStop(S + 1, P + S + 1)

Woohoo, it seems like we have our recurrence relation!

Recurrence relation: Assuming you have computed the subproblems, how would you compute the main problem?

Step 4: Identify the base cases

A base case is a subproblem that doesn’t depend on any other subproblem. In order to find such subproblems, you typically want to try a few examples, see how your problem simplifies into smaller subproblems, and identify at what point it cannot be simplified further.

The reason a problem cannot be simplified further is that one of the parameters would become a value that is not possible given the constraints of the problem.

In our example problem, we have two changing parameters, S and P. Let’s think about what possible values of S and P might not be legal:

P should be within the bounds of the given runway

P cannot be such that runway[P] is false because that would mean that we’re standing on a spike

S cannot be negative, and a S==0 indicates that we’re done

Sometimes it can be a little challenging to convert assertions that we make about parameters into programmable base cases. This is because, in addition to listing the assertions if you want to make your code look concise and not check for unnecessary conditions, you also need to think about which of these conditions are even possible.

In our example:

P < 0 || P >= length of runway seems like the right thing to do. An alternative could be to consider making P == end of runway a base case. However, it is possible that a problem splits into a subproblem which goes beyond the end of the runway, so we really need to check for inequality.

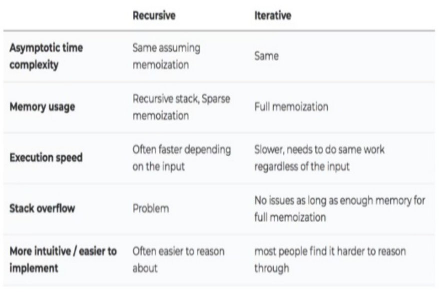
This seems pretty obvious. We can simply check if runway[P] is false.

Similar to #1, we could simply check for S < 0 and S == 0. However, here we can reason that it is impossible for S to be < 0 because S decreases by at most 1, so it would have to go through S == 0 case beforehand. Therefore S == 0 is a sufficient base case for the S parameter.

Step 5: Decide if you want to implement it iteratively or recursively

The way we talked about the steps so far might lead you to think that we should implement the problem recursively. However, everything that we’ve talked about so far is completely agnostic to whether you decide to implement the problem recursively or iteratively. In both approaches, you would have to determine the recurrence relation and the base cases.

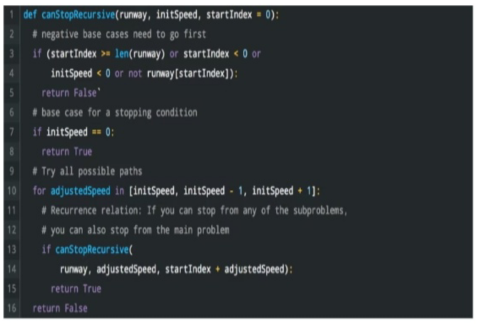
To decide whether to go iteratively or recursively, you want to carefully think about the trade-offs.



	Recursive	Iterative
Asymptotic time complexity	Same assuming memoization	Same
Memory usage	Recursive stack, Sparse memoization	Full memoization
Execution speed	Often faster depending on the input	Slower, needs to do same work regardless of the input
Stack overflow	Problem	No issues as long as enough memory for full memoization
More intuitive / easier to implement	Often easier to reason about	most people find it harder to reason through

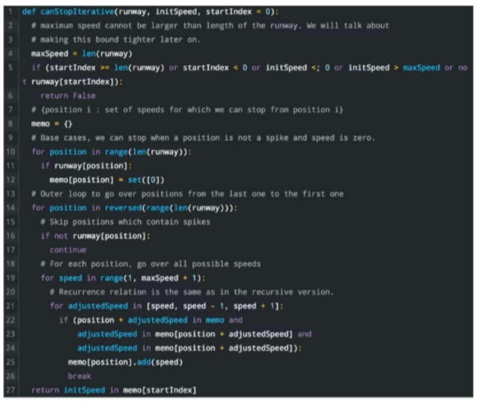
In our particular problem, I implemented both versions. Here is python code for that:

A recursive solution: (original code snippets can be found here)



```python
def canStopRecursive(runway, initSpeed, startIndex = 0):
  # negative base cases need to go first
  if (startIndex >= len(runway) or startIndex < 0 or
      initSpeed < 0 or not runway[startIndex]):
    return False`
  # base case for a stopping condition
  if initSpeed == 0:
    return True
  # Try all possible paths
  for adjustedSpeed in [initSpeed, initSpeed - 1, initSpeed + 1]:
    # Recurrence relation: If you can stop from any of the subproblems,
    # you can also stop from the main problem
    if canStopRecursive(
        runway, adjustedSpeed, startIndex + adjustedSpeed):
      return True
  return False
```

An iterative solution: (original code snippets can be found here)



```python
def canStopIterative(runway, initSpeed, startIndex = 0):
  # maximum speed cannot be larger than length of the runway. We will talk about
  # making this bound tighter later on.
  maxSpeed = len(runway)
  if (startIndex >= len(runway) or startIndex < 0 or initSpeed <; 0 or initSpeed > maxSpeed or no
t runway[startIndex]):
    return False
  # {position i : set of speeds for which we can stop from position i}
  memo = {}
  # Base cases, we can stop when a position is not a spike and speed is zero.
  for position in range(len(runway)):
    if runway[position]:
      memo[position] = set([0])
  # Outer loop to go over positions from the last one to the first one
  for position in reversed(range(len(runway))):
    # Skip positions which contain spikes
    if not runway[position]:
      continue
    # For each position, go over all possible speeds
    for speed in range(1, maxSpeed + 1):
      # Recurrence relation is the same as in the recursive version.
      for adjustedSpeed in [speed, speed - 1, speed + 1]:
        if (position + adjustedSpeed in memo and
            adjustedSpeed in memo[position + adjustedSpeed] and
            adjustedSpeed in memo[position + adjustedSpeed]):
          memo[position].add(speed)
          break
  return initSpeed in memo[startIndex]
```