← Back to Home

Two Pointers
Sliding Window
Intervals
Stack
Linked List
Binary Search
Heap
Depth-First Search
Breadth-First Search
Backtracking
Graphs
Dynamic Programming
- Fundamentals
- Solving a Question with Dyn...
- Counting Bits
- Decode Ways
- Maximal Square
- Unique Paths
- Longest Increasing Subseq...
- Word Break
- Maximum Profit in Job Sche...

Greedy Algorithms
Trie
Prefix Sum

← Fundamentals

Counting Bits →

Solving a Question with Dynamic Programming

This page breaks down how to solve a question with dynamic programming into a series of steps which ultimately leads to the "bottom-up" solution to the problem.

1. Find the Recurrence Relation
2. Identify the Base Case(s)
3. Write the Recursive Solution
4. Add Memoization
5. Convert to "Bottom-Up" DP
6. Further Optimization

DESCRIPTION (credit Leetcode.com)

You're a treasure hunter in a neighborhood where houses are arranged in a row, and each house contains a different amount of treasure. Your goal is to collect as much treasure as possible, but there's a catch: if you collect treasure from two adjacent houses, it triggers a neighborhood-wide alert, ending your hunt immediately.

Given an array `treasure` of non-negative integers, where each integer represents the amount of treasure in a house, write a function to return the maximum amount of treasure you can collect without triggering any alarms.

Example 1:
Input: treasure = : [3, 1, 4, 1, 5]
Best Haul: 12

Explanation: Collect from houses 0, 2, and 4 for a total of 3 + 4 + 5 = 12.

Why Dynamic Programming?

Let's first touch on why this question is a good candidate for dynamic programming.

Recall from the **Fundamentals** section that a problem is a good candidate for dynamic programming if:

- It has optimal substructure (it can be solved using recursion)
- It has overlapping subproblems (the same recursive call is made multiple times)

So let's break down why this problem has both of these properties.

Optimal Substructure

This problem has optimal substructure because the optimal solution for the first `i` houses can be calculated using the optimal solutions for the first `i - 2` and `i - 1` houses.

> In this case, the optimal solution refers to the maximum amount of treasure you can collect without triggering any alarms.

Say we are trying to calculate the optimal solution for the first 6 houses in array `treasure` below:

If we already know the optimal solution for the first 5 houses is 12:

And the optimal solution for the first 4 houses is 7:

Then we have two choices for including house 6:

We can choose to *take* from house 6 by adding its value to the optimal solution for 4 houses, which in this case gives us 9:

Or we can choose to *skip* house 6 by using the optimal solution for 5 houses, which is 12:

By taking the maximum of those values, we can calculate the optimal solution for the first 6 houses, which is 12.

We can then calculate the optimal solution for 7 houses in the same way, by using the optimal solution for 5 houses (which allows us to take from house 7), and 6 houses (which allows us to skip house 7).

> **General Tips for Identifying Optimal Substructure**
>
> Assume you already know the answer for a smaller version of the input. Then see if you can use that information to solve the problem for a larger input.
>
> If the input is an array, assume you have an answer to the first few elements of the array. If the input is a string, assume you have an answer for the first few characters. If the input is a number, assume you have an answer for a smaller number, etc.

Overlapping Subproblems

We can imagine that as our `treasure` array grows, we will end up using the optimal solution for the same subproblems multiple times. For example, we use the optimal solution for the first 5 houses to calculate both the optimal solution for the first 6 houses and the optimal solution for the first 7 houses.

We'll get a better visual of the overlapping subproblems in Step 3, but for now, we've identified this is a good candidate for dynamic programming because of the overlapping subproblems and optimal substructure properties.

We can now move to the first step:

1. Identify the Recurrence Relation

We can formalize the optimal substructure from above by defining a **recurrence relation** for the problem. A recurrence relation describes the answer to our problem in terms of answers to the same problem with smaller inputs.

We saw from above that the optimal solution for the first `i` houses can be calculated using the optimal solutions for the first `i - 2` and `i - 1` houses.

Let `dp(i)` be the optimal solution for the first `i` houses. Then our recurrence relation is:

```
dp(i) = max(dp(i - 1), dp(i - 2) + treasure[i - 1])
```

Make sure you are clear as to what the recurrence relation represents in terms of the input and the problem. Having a clear, consistent definition of what the recurrence relation represents will help you with every step of this process.

In this case, `dp(i)` refers to the optimal solution (most amount treasure) that can be collected from the *first* `i` *houses* of `treasure`, which is why we use `i - 1` to index into `treasure`. (In other words, to calculate the optimal solution for the first 6 houses, we need to consider `treasure[5]`, since treasure is 0-indexed.)

> **General Tips for Identifying the Recurrence Relation**
>
> Decide what parameters define the subproblem. In this case, the parameters are `i`, which represents the index of the last house in `treasure` that we are considering. In general, these parameters will be indexes (of strings or arrays) or lengths that can be used to describe the size of the subproblem.

2. Identify the base case(s)

By now, we have a clear understanding of what our recurrence relation represents. We should use that understanding to identify the base case(s) for our problem.

The base cases are the inputs for which we know the answer without having to use

The base cases are the inputs for which we know the answer without having to use the recurrence relation. They are typically the smallest possible values for our parameters.

In this case:

- If we have 0 houses, we can't collect any treasure, so `dp(0) = 0`.
- If we have 1 house, we can only collect the treasure from the first house, so `dp(1) = treasure[0]`.

> **General Tips for Identifying the Base Case(s)**
>
> Think empty or small here: empty strings, empty arrays, 0, or 1. Another way of thinking about `dp(0)` for our problem is: "how much treasure can we collect if the `treasure` array is empty?"

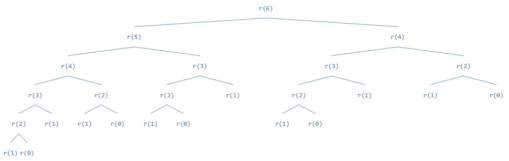
3. Write the Recursive Solution

Combine the recurrence relation and the base case(s) to write the recursive solution. This involves introducing a helper function with the same signature as the recurrence relation we defined in Step 1.

```python
def rob(treasure):
    if not treasure:
        return 0

    def rob_helper(i):
        if i == 0:
            return 0
        if i == 1:
            return treasure[0]

        skip = rob_helper(i - 1)
        take = rob_helper(i - 2) + treasure[i - 1]
        return max(skip, take)

    return rob_helper(len(treasure))
```

Overlapping Subproblems

With the recursive solution written out, we can clearly see the overlapping subproblems by visualizing the call tree of our recursive function. The call tree shows the function calls made when `treasure` has 6 houses:



r(3) is called a total of 3 times, each of which generates the same sequence of recursive calls.

Our recursive function has an exponential time complexity of $O(2^n)$, where `n` is the length of the `treasure` array. We can make it much more efficient by adding memoization.

4. Add Memoization

With our recursive solution written, adding memoization is a three-step process:

1. Initialize an empty dictionary to store the results of recursive calls. Eventually the keys of the dictionary will be the inputs to the recursive function, and the values will be the results of the recursive function.

Now, inside the body of the recursive function:

1. If the input to the recursive function is in the memoization dictionary, return the value stored in the dictionary. **Make sure you do this before making any recursive calls**.
2. If the input is not in the dictionary, calculate the value via recursion and store it in the dictionary before the return statement.

```python
def rob(treasure):
    if not treasure:
        return 0
    # initialize memoization dictionary
    memo = {}

    def rob_helper(i):
        # base cases
        if i == 0:
            return 0
        if i == 1:
            return treasure[0]

        # check and return from memo
        # BEFORE making any recursive calls
        if i in memo:
            return memo[i]

        # recurrence relation
```

```
        skip = rob_helper(i - 1)
        take = rob_helper(i - 2) + treasure[i - 1]
        result = max(skip, take)

        # store result in memo before returning
        memo[i] = result
        return result

    return rob_helper(len(treasure))
```

Since every recursive call needs to have access to the same dictionary, it's best to create the dictionary outside of the recursive function, and to define the recursive function inside the main function. This way, the dictionary is accessible to both the main function and the recursive function.

Adding memoization reduces the time complexity of our solution to `O(n)`, where `n` is the length of the `treasure` array. This is because we only need to calculate the optimal solution for each house once. When we need to calculate it again, we can just look it up in the memoization dictionary in `O(1)` time.

5. Convert to "Bottom-Up" DP

Finally, we can convert our memoized recursive solution to a bottom-up dynamic programming solution. A "bottom-up" dynamic programming solution uses iteration to build up to the final solution from base cases.

Recall from the fundamentals that the "bottom-up" approach is equivalent to starting from the leaf nodes of the memoized call tree and working our way up to the root node.

Step 1: Initialize an array to store the results of the subproblems.

In this case, `dp[i]` will represent the optimal solution for the first `i` houses, which means we need an array of size `len(treasure) + 1` to store the results of the subproblems.

In other words, if we are solving for an array size of 6, we need an array of size 7 to store the results of the subproblems for 0 to 6 houses. The final answer will be stored in `dp[6]`.

Step 2: Fill in the base cases in the array.

In this case, `dp[0] = 0` and `dp[1] = treasure[0]`. In other words, the optimal solution for 0 houses is 0, and the optimal solution for 1 house is the treasure at the first house.

Step 3: Start iterating to fill in the rest of the array, up to and including the final answer at `dp[len(treasure)]`. Use the recurrence relation to calculate each value.

```
def rob(treasure):
    if not treasure:
        return 0

    # initialize dp array
    dp = [0] * (len(treasure) + 1)

    # fill in base cases (dp[0] = 0 already)
    dp[1] = treasure[0]

    # iterate to fill in the rest of the array
    for i in range(2, len(treasure) + 1):
        # fill in dp[i] using the recurrence relation
        take = dp[i - 2] + treasure[i - 1]
        skip = dp[i - 1]
        dp[i] = max(take, skip)

    return dp[len(treasure)]
```

General Tips for Converting to Bottom-Up DP

Practice, practice, practice! As before, it's important to have a clear understanding of what the recurrence relation represents. This will help you with everything from filling in the base cases to the bounds of your loops, which can be a common source of off-by-one errors.

6. Further Optimization

For this question, there is room for further optimization. We can see that we only need to keep track of the two most recent values of `dp` to calculate the next value. In other words, once we've obtained `dp[4]`, we no longer need `dp[0]`, `dp[1]`, or `dp[2]` for the rest of the calculation.

This means we don't need to store the entire `dp` array, just the last two values, which we use to calculate the next value. This can reduce the space complexity of our solution from `O(n)` to `O(1)`.

```
def rob(treasure):
    if not treasure:
        return 0

    prev, curr = 0, treasure[0]

    for i in range(2, len(treasure) + 1):
        # calculate the next value of dp
        take = prev + treasure[i - 1]
        skip = curr
        prev, curr = curr, max(take, skip)
```

```
    return curr
```

> **General Tips for Further Optimization**
>
> You should think about further optimizations after you have a working bottom-up solution. Often times, the recurrence relation will give you a hint as to how you can reduce the space complexity of your solution. In this case, the recurrence relation only requires the two most recent values of `dp`, so we don't need to store the entire array.

Common Mistakes

It's important to keep the variables you use in your code consistent with how you defined the subproblems in the recurrence relation. This will help you avoid off-by-one errors and other bugs.

For example, we could have alternatively defined `dp(i)` to represent the optimal solution of all houses up to the `ith-index` in the `treasure` array. So `dp(0)` would represent the optimal solution for the first house, `dp(1)` would represent the optimal solution for the first two houses, and so on.

This simple change in definition has many downstream effects:

```
def rob(treasure):
    if not treasure:
        return 0
    if len(treasure) == 1:
        return treasure[0]

    # Initialize dp array
    dp = [0] * len(treasure)

    # Base cases
    dp[0] = treasure[0]
    dp[1] = max(treasure[0], treasure[1])

    # Fill dp array
    for i in range(2, len(treasure)):
        dp[i] = max(dp[i-1], dp[i-2] + treasure[i])

    return dp[len(treasure) - 1]
```

- The base cases change: `dp[0] = treasure[0]` and `dp[1] = max(treasure[0], treasure[1])`
- The recurrence relation is different: `dp(i) = max(dp(i - 1), dp(i - 2) + treasure[i])`
- The bounds of the iteration change.

The fact that a simple change in definition can have downstream effects in so many parts of the code is why it's so easy to make off-by-one errors or index out of bounds errors as you work through dynamic programming problems.

So remember: the more clearly you can define your subproblems in terms of the inputs and the problem, the easier it is to keep everything consistent. Keep this in mind as you work through dynamic programming problems that follow.

Next: Counting Bits →

VB **vishal bajoria**
Created at: 9/1/2024, 10:22 PM

I got a kick out of how 'robbing a house' was rebranded as a 'treasure hunt'—that had me laughing! But seriously, the explanation of dynamic programming was perfect. The way you converted it from top-down to an optimized bottom-up approach is brilliant. Big thanks for that!