Dynamic Programming (Pre-workshop 8)

Disclaimer / How to use this document

To become comfortable with dynamic programming (DP), we need practice. This tutorial contains the explanations for 15 ro so DP problems, ordered from easy to hard, and introducing all the central ideas of DP progressively. No previous experience with DP is needed. The problems are meant to provide practice: the tutorial will be more useful if you try to solve the problems before looking at the solution. We took problems from leetcode and provided the links so that you can practice there.

And yet, the problems are not the important part. Practice is not just about solving / remembering specific problems. It is about developing and getting comfortable with a plan / approach / template that we can follow under pressure in an interview. Luckily, all DP problems have enough similarities that we can follow the same steps for all of them. Before getting to the practice problems, I present a general template to approach DP problems (sections Template 1: Problem Solving, and Template 2: Implementation). Then, we'll see how it applies to each practice problem.

You can take this template as is, or, better, adjust it to fit your preferences/workflow. Use the practice problems to practice using it, memorize it, and refine it. Take note of the key things to remember and the common mistakes you make, so that each time you employ the template you get better at it.

The coding examples use Python. We try to not use advanced language features and document it with comments, so it should be understandable even for people who are not familiar with Python. In any case, coding is only a part of DP. Most of the discussion around how to approach DP problems will be language agnostic.

Finally, we improve the slides based on feedback from the fellows. Your feedback is appreciated! Besides the standard workshop feedback form, you can send questions or suggestions directly to nil.mamano@gmail.com. All the code samples pass the test cases on leetcode, so there shouldn't be any bugs, but let us know if you see one.

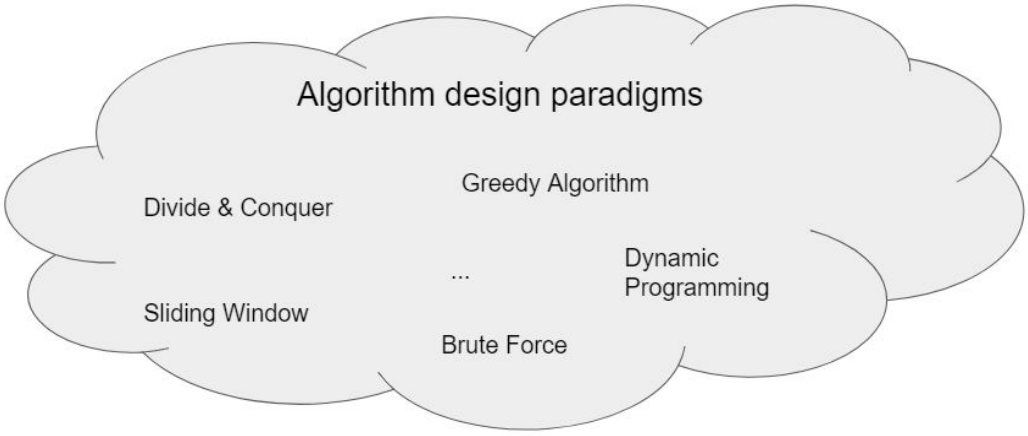
Going through all the problems may take 2-4h, depending if you are new to DP or just derusting. Good luck!

Introduction

We will start by establishing the bases, just so that we are all on the same page.

What "kind of thing" is DP?

DP is an *algorithm design paradigm*: a general strategy for solving problems with certain characteristics.



Why the name "Dynamic Programming"?

The paradigm is called "Dynamic Programming" for historic reasons, but the name is nonsense. The "programming" in "dynamic programming" does not even come from "coding" (I'm not kidding). A better name would be "Reduce & Remember".

What characteristics must a problem have in order for DP to be applicable?

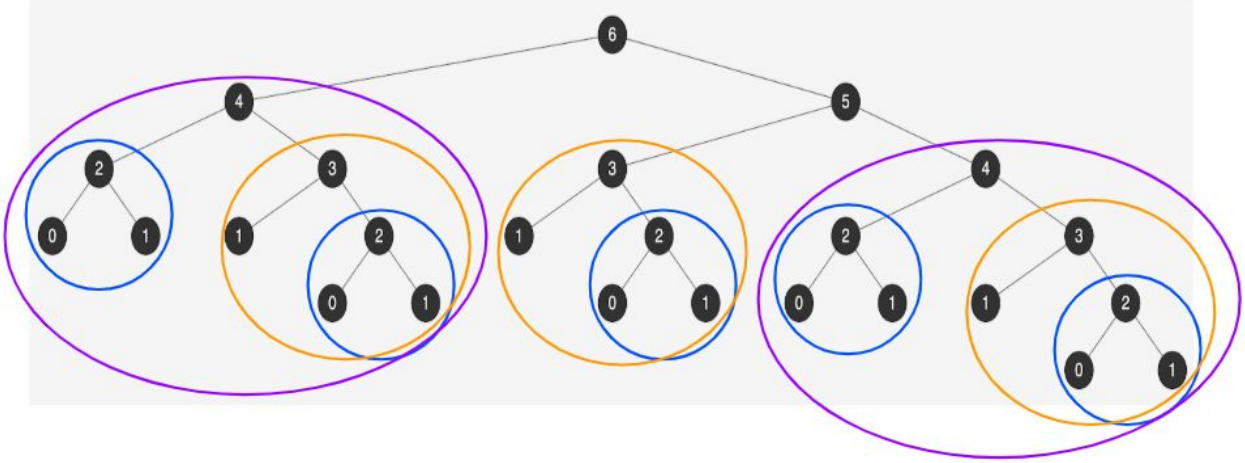
DP is for problems that satisfy two properties:
1. Optimal substructure: The problem can be broken down into similar but smaller subproblems, and the solutions to the subproblems help us find the solution to the original problem. In turn, these subproblems can be broken into smaller subproblems, and so on.
2. Overlapping subproblems: multiple subproblems rely on the same subproblem.

The simplest form of problem where DP is applicable is computing a math formula defined in terms of itself on smaller numbers. The quintessential example is Fibonacci:

F(n) = F(n-1) + F(n-2)
It has optimal substructure because F(n-1) and F(n-2) help us compute F(n). It has overlapping subproblems because both F(n) and F(n-1) rely on F(n-2).
F(6) = F(5) + F(4)
F(5) = F(4) + F(3)

The challenge of overlapping subproblems

If the subproblems overlap, if we expand the formula for the n-th term, its size starts growing exponentially:



The number of unique subproblems grows linearly, but the number of repetitions grows exponentially.

What constitutes a DP algorithm?

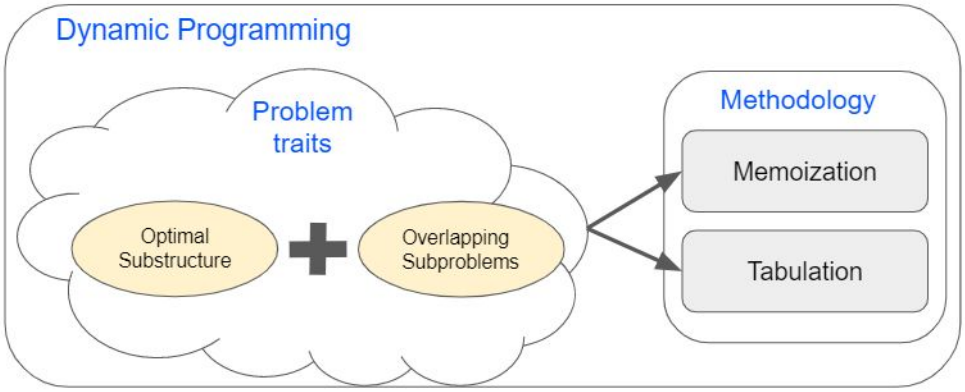
All DP algorithms have two common features, which correspond to the two properties of the problems mentioned above.
1. The logic of a DP algorithm is based on the optimal substructure of the problem. Say the algorithm gets some input X. To find the solution for X, we decompose X into smaller inputs, find the solutions to these smaller inputs, and then use those solutions to find the solution to X. In turn, the solution to these smaller inputs are found in the same way, repeatedly, until the inputs are so small that we can compute the solution directly.
2. A DP algorithm needs a way to handle overlapping subproblems. In a DP algorithm, a subproblem is never computed more than once. The idea is that the same subproblem always gives the same solution, so computing it more than once is actually unnecessary. Instead, we store / remember / "cache" the solution after we compute it the first time.

What is the difference between DP and divide-and-conquer?

Both DP and divide-and-conquer are algorithm design paradigms, and both reduce the original problem to smaller subproblems. The main difference is that in divide-and-conquer subproblems don't overlap, so there is no need to store their solutions.

What is "memoization" and "tabulation"?

Memoization and tabulation are the two main methodologies for storing the solutions to subproblems in a DP algorithm. You are free to choose between memoization and tabulation based on personal preference, because we can use either for any problem with the required traits (unless an interviewer says you must use one specifically). We will discuss them in detail.



What is top-down and bottom-up DP?

DP methodologies have been given different names over time. We stick to memoization and tabulation, but be aware that
Memoization DP, top-down DP, and recursive DP are synonyms.
Tabulation DP, bottom-up DP, and iterative DP are synonyms.

How do you know if DP is the appropriate paradigm for a problem?

Unfortunately, there are no hard rules for this. After doing many DP problems, like the ones in this guide, you will start to notice familiar patterns. You can think if the problem "sounds like" it can be broken into subproblems. Try it, and if it doesn't work, consider other paradigms. You can also review workshop 6.

In contrast, here are some hints that DP is NOT the appropriate paradigm: if the input is a graph which can have cycles. TODO

Template 1: Problem Solving / Finding Recurrences

We said that the logic of a DP problem is based on the optimal substructure of a problem: "The problem can be broken down into analogous but smaller subproblems, and the solutions to the subproblems help us find the solution to the original problem."
This relationship between larger and smaller subproblems is captured by what we call a "recurrence". A recurrence is a formula that calculates the solution to the problem and is defined in terms of itself on smaller inputs. Recurrences are also called "recurrence relations" or "recurrence equations". The Fibonacci formula F(n) = F(n-1) + F(n-2), together with the base cases F(0) = 1, F(1) = 1, is an example of recurrence, but, in fact, there is always a recurrence for every problem where we can use DP. The "problem-solving" part of DP is figuring out the recurrence that solves the problem.

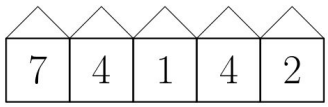
The template I propose for DP is a 2-part process:
1. Problem-solving part: find a recurrence that calculates the solution.
2. Implementation: turn the recurrence into efficient code.

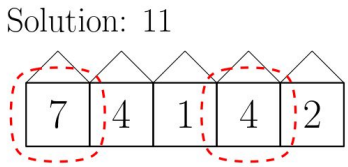
In the implementation part you can choose between memoization and tabulation. In both cases, we need to find a recurrence first. In my experience, part 1 is harder because we can memorize the steps for part 2.

In this section of the tutorial, we explain the main idea of how to find a recurrence for a problem.

Example of typical problem where we can apply DP

Problem name: House Robber
Link: https://leetcode.com/problems/house-robber/
Description: There is a row of houses. Each house has an amount of money. You are a thief and your goal is to steal as much money as possible without robbing from adjacent houses.
Input: an array houses of length n where houses[i] is the amount of money in the i-th house.
Example:



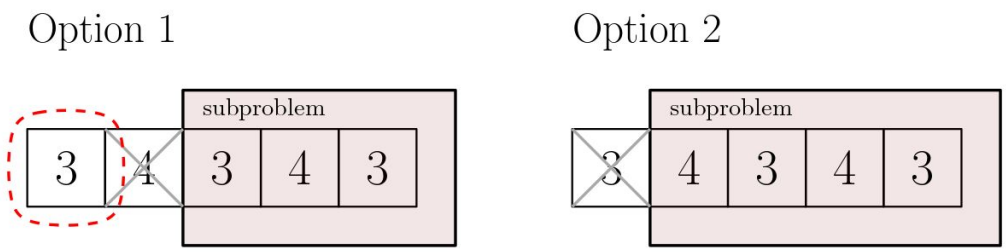


Note that there are 2^n subsets of houses; too many to check them all with brute force. Even if we consider only subsets of houses without adjacent houses, that is still an exponential number. We need to be smarter than that.

Typical DP thought process

This may be the most important section, as it covers the kind of thought process we should follow when approaching a DP problem.
Every recurrence starts with an idea of how we can relate the original instance to smaller instances. Informally, we think about ways to "handle" the first element in the input so that we can "remove" it. Then, the remainder of the input will be a little smaller. To "handle" the first element in the input we consider all the possible options of what we can do with it. Then, we try to express the situation after that option happens in terms of smaller subproblems.

Here is the idea for the house robber problem. For small cases, we can reason directly:
- If there is a single house, rob it.
- If there are two houses, rob the richest of the two, since we cannot rob both.
But after thinking about the smallest cases, we want to think about a general case with arbitrary size.
- If there are more than two houses, think about all the possible options for what can happen with house 1. There are two options: either we rob it or we don't. As is typical in DP problems, just by looking at the first house (or the first few houses) we don't know which is the best option, but what we know for sure at this point is that one of the two must happen. This is all we need.
The idea is simple: we will consider the value that we can get with each of the two options, and get the best one.
Thus, we need to express how much money we can steal in each case, in terms of smaller subproblems.
If we rob the first house, then we must skip the second one. This leaves n-2 houses: a smaller number, so we can handle it recursively.
If we skip the first house, then we can rob or skip the second one. This leaves n-1 houses.



Do not worry about how we will know the solutions to the subproblems in the first place. Assume you have a crystal ball that tells you the answer to any subproblems we want. We will take care of this in the implementation, using either memoization or tabulation.

To come up with an idea, it can help to toy with an example input. Interviewers usually provide an example input that you can use.

Typical recurrence for a DP problem

The idea above can be captured with a recurrence:

Rob(i): max value we can get from the suffix houses[i..n-1] (we use x..y to indicate a range)
Base cases:
Rob(n-1) = houses[n-1]
Rob(n-2) = max(houses[n-1], houses[n-2])
General case (0 ≤ i ≤ n-3):
Rob(i) = max(houses[i] + Rob(i+2), Rob(i+1))
Goal: Rob(0) (max value we can get from houses[0..n-1], that is, all the houses)

Let's break down the parts of the recurrence above.
- It has a description in words of what the recurrence calculates. Start with this to be clear about what we are calculating, the interviewer will appreciate it. The name "Rob" can be anything, like a short names related to the problem. "i" is the index or input. A recurrence can have more than one index, as we will see. To keep things simple, we say Rob(i) instead of something like Rob(houses[i..n-1]). Later, when we discuss implementation, we will use these indices to map subproblems to stored solutions.
- It has two base cases. Base cases correspond to subproblems where we can compute the value directly (not in terms of other subproblems). Every recurrence must have one (or more) base cases. They take the smallest possible inputs.
- It has a general case. It is helpful to state the boundaries of the general case (the 0 ≤ i ≤ n-3 part). The general case consists of two or more expressions involving subproblems, and a way to "aggregate" the expressions to get the solution for the current input (in this case, we aggregated them with "max").
- It has a goal, which is the solution to the original problem in terms of the recurrence. This is the value we actually care about. It shows how the recurrence solves the original problem.

Are recurrences necessary?

The point of recurrences is to transform a "fuzzy" idea into a format that is precise, concrete, and clear. It contains all the key things that we will need in our implementation. We are separating the "problem-solving" part from the implementation details, which helps us isolate and focus on the main challenge. However, it is also possible to go directly from the idea to implementation. Do what works best for you. Even if you think recurrences are not for you, check the rest of the section, because it goes over the things that you will need to take into account anyway.

Forward vs backward recurrences

To use DP, we always need to relate bigger problems to smaller subproblems. However, there are two ways to do that, and both are common. Compare the recurrence above with the following one. Both solve the house robber problem.

Rob(i): max value we can get stealing the prefix houses[0..i]
Base cases:
Rob(0) = houses[0]
Rob(1) = max(houses[0], houses[1])
General case ($2 \leq i \leq n-1$):
Rob(i) = max(houses[i] + Rob(i-2), Rob(i-1))
Goal: Rob(n-1) (max value we can get from all the houses)

The first recurrence we saw is what I call a "forward" recurrence, while this one is a "backward" recurrence. At a practical level, the difference between a backward and a forward recurrences is that everything is flipped: the goal and the base cases are in opposite ends, and the indices of the subproblems in the general case get larger (for a forward recurrence) instead of smaller (for a backward recurrence).

At a conceptual level, backward and forward recurrences reflect different approaches.
In a forward recurrence, we think chronologically: we think about where we start and what choices we have from there (e.g., whether to rob or skip the first house). In a backward recurrence, we start by thinking about the entire solution, and think about the last choice that led us there (e.g., whether to rob or skip the last house).

Should you use forward or backward recurrences?

This is entirely up to you. Both backward and forward recurrences are equally valid for solving problems. All problems can be solved with either. Forward recurrences can be more intuitive because of the chronological order. Workshop 6 (memoization) used mostly forward

recurrences. This workshop uses forward recurrences to be consistent, but we will show examples of both. You should probably choose one and stick with it.

Common patterns in recurrences

Every DP algorithm is based on a recurrence, and recurrences tend to look similar even for very different problems. Here, we go over the main types of recurrences, and then we will see examples of these in the practice problems. Knowing these patterns makes DP easier.

One of the things we need to decide is what are the subproblems. Based, on the input type, the subproblems will be one of the following:

- Number: smaller numbers
- Array: prefix, suffixes or subarrays (but not subsequences). Prefixes will usually come up in backward recurrences, and suffixes in forward recurrences.
- Matrix / multi-dimensional array: prefix/suffix of the rows, prefix/suffix of the columns, or prefix/suffix of each dimension.

For example, a string has O(n) prefixes/suffixes and O(n^2) substrings. Sometimes it's not obvious if the subproblems should be one or the other. It depends on the problem.

The general case of a recurrence consists of two or more expressions, some of which involve subproblems. We always try to find a set of options or "choices" that we know for sure that one of them must happen. Each expression corresponds to one option.

Usually, the number of options in the general case is a small constant number (2 in the simplest case), but it could also be a range. The range could depend on some input parameter or, in the extreme, there could be an option for every smaller subproblem. As we will see, in the implementation, this type of ranges become an inner for loop.

Note: if only one subproblem is needed, this most likely means that we don't have overlapping subproblems and thus we don't need DP. For example, consider the "palindromic string" problem: given a string s of length n, find if s is palindromic. We can express the solution as a recurrence:

Pal(i, j): True if s[i..j] is palindrome else False
Base cases:
Pal(i, i) = True for all i from 0 to n-1.
Pal(i, i+1) = True if s[i] == s[i+1] else False, for all i from 0 to n-2
General case (i < j):
Pal(i, j) = True if s[i] == s[j] and Pal(i+1, j-1), else False
Goal: Pal(0, n-1)

You can see that only one subproblem appears in the general case and we don't have overlapping subproblems.

Finally, we need to aggregate the options.

The way to do this depends on the type of the problem. There are 3 main problem types:
- Optimization: the goal is to maximize or minimize some quantity (This is the most common type of DP problem, "House robber" is an example of this). The expressions represent the "value" of each option, and aggregating means choosing the best one. For this, we use the "max" or "min" operator (depending if it's a maximization or minimization problem).
- Counting: the goal is to count the number of ways to do something. The expressions represent the number of ways to continue for each option. Aggregating means adding the count for each option. We do with the sum operator.
- Satisfiability: the goal is to find if the input satisfies some property. The expressions represent if it is possible to satisfy the property while taking each option. Aggregating means checking if any of the options works. We do this by doing a logical OR of all the expressions. Sometimes we need to check that all the options work so we use a logical AND instead.

The trick of adding constraints

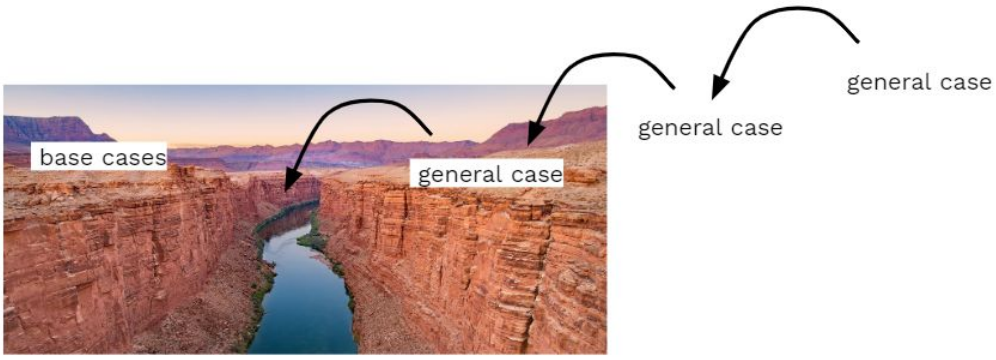
Usually, the recurrence computes exactly what the problem asks for, and the only thing that changes is the input size. However, sometimes, we write the recurrence to compute something slightly different to the original problem. This is necessary sometimes when it is hard to find a recurrence for the original problem, but adding some restriction makes it easier.

An extra constraint that is useful in many occasions is to force the first index to be included. We will see an example of how this is useful in the problem "Longest Increasing Subsequence". When the recurrence is not exactly the same as the original problem, it is specially important to define the recurrence in words as well as how to get the solution to the original problem from the recurrence (what we call the "goal").

The importance of index arithmetic

To write a recurrence (or implement a DP algorithm in general), we need to manipulate indices carefully. Consider the following questions:
- What is the range of the input? Does it go from 0 to n-1 or from 1 to n?
- What indices correspond to the base case(s)?
- What indices correspond to the general case?
- What index corresponds to the goal?

There's flexibility in where to put the boundary between the base case and the general case. However, we need to make sure that there is no "gap" between them. To check this, consider what is the index in the general case closest to the base cases, and make sure that the subproblems in the general case are covered by the base cases.



Template 2: Implementation

Memoization

It is straightforward to turn a recurrence into recursive code. However, that is very inefficient because of overlapping subproblems. Memoization is a way to "fix" the inefficient recursive code. We add a dictionary (usually a hash table) and we store the subproblems after they are computed for the first time. Whenever a subproblem needs another subproblem, before computing it recursively, we check if it is already in the dictionary. The keys in the dictionary correspond to the inputs to the recurrence.

Here is the "house robber" recurrence turned into (inefficient recursive code):

```
def rob(houses):
  n = len(houses)

  def robRec(i):
    if i == n-1: return houses[n-1]
    if i == n-2: return max(houses[n-1], houses[n-2])
    return max(houses[i] + robRec(i+2), robRec(i+1))

  return robRec(0)
```

Note that we make use of a "nested" function definition. This is not necessary but it is handy because, in Python, the nested function has visibility on the variables on the outer scope, so we don't need to pass houses as a parameter to each recursive call.
Here is the same code adding the memo dictionary

Also note how the base cases, the general case, and the goal in the recurrence appear in the code almost unchanged.

Here is the code with the added memoization dictionary.

```
def rob(houses):
  n = len(houses)
  memo = dict()

  def robRec(i):
    if i == n-1: return houses[n-1]
    if i == n-2: return max(houses[n-1], houses[n-2])
    if i in memo: return memo[i]
    memo[i]= max(houses[i] + robRec(i+2), robRec(i+1))
    return memo[i]

  return robRec(0)
```

The main structure of the code is the same, but we added 2 actions: storing the result in the dictionary, and checking if the result is in the dictionary. Note that the base cases are not added to the dictionary, they are just checked directly.

Instead of having an intermediate variable "res" for the result, we can store the result to memo[i] directly and return memo[i]. This way, we can't forget to store the result at the end:

```
  memo[i] = max(houses[i] + robRec(i+2), robRec(i+1))
  return memo[i]
```

There is not much more to memoization. In summary, turning a recurrence into efficient code using memoization is a two step process: recurrence -> recursive code -> recursive code + dict.

Tabulation

To turn a recurrence into code, the idea of tabulation is to create a table where each cell corresponds to a subproblem, and fill it iteratively (with a for loop), in order, from the base cases to the goal. We always do this in 4 steps:
1. Initialize the DP table of the right size (and number of dimensions).
2. Fill the base cases directly in the table.
3. Fill the rest of the cells using the general case.
4. Get the result from the table (usually the last filled cell).

Here is house robber again, this time using tabulation:

```
def rob(houses):
  n = len(houses)
  T = n*[0] #Python way of creating an array of n 0's
```

```
  T[n-1] = houses[n-1]
  if n == 1: return houses[0] #without this check, the next line could go out of bounds
  T[n-2] = max(houses[n-1], houses[n-2])
  for i in range(n-3, -1, -1): #same as C/Java for (i=n-3; i != -1; i = i-1)
    T[i] = max(houses[i] + T[i+2], T[i+1])
  return T[0]
```

Since we are using a forward recurrence, the base cases are at the end, so we iterate through the table from the end (n-1) to the start (0). To iterate backwards In Python, we use the "range" function with 3 arguments: the start index (included), the end index (excluded), and the step. Since the end index is excluded, to end at 0, we need to put -1.

Implementation of recurrences with multiple inputs

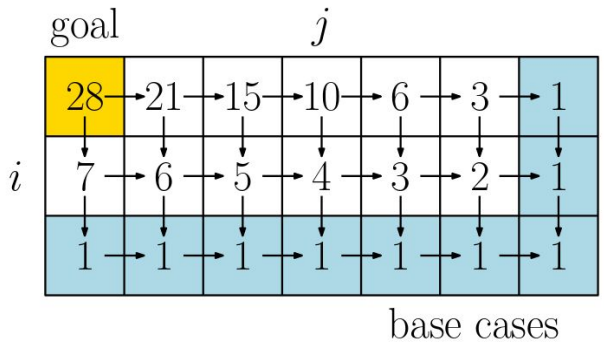
Sometimes, recurrences have more than one input. Technically, it could be more than 2 dimensions, but 1D and 2D are by far the most common, but the logic is the same. Here, we discuss how that affects the implementation, using the following example.

Problem name: Count paths in grid
Link: https://leetcode.com/problems/unique-paths/
Description: Count the number of different paths in a grid from the top-left cell to the bottom-right cell, moving only down or right.
Input: The number of rows, $r \geq 1$, and the number of columns $c \geq 1$.
Example:
Input: r = 3, c = 2
Solution: there are 3 ways to reach the bottom-right corner: RRD, RDR, DRR.

Approach: this is a counting problem, and from the starting position we have two options that we can take. This suggests that we should use DP to count the number of continuations for each option recursively, and then add them to get the total number.

Count(i, j): number of ways to reach the bottom-right corner (cell (r-1, c-1)) from cell (i, j).
Base cases:
Count(r-1, j) = 1 for all j < c (if you are in the last row, you can only go right)
Count(i, c-1) = 1 for all i < c (if you are in the last column, you can only go down)
General case ($0 \leq i \leq r-2$, $0 \leq j \leq c-2$):
Count(i, j) = Count(i+1, j) + Count(i, j+1)
Goal: Count(0, 0)

Here is a visual representation of the dependencies between the subproblems for the example with input r=3, c=7.



Memoization with 2D recurrence

For memoization, everything is the same, but the memo dictionary needs to take multiple values as keys. In Python, we can use tuples as dictionary keys. We can insert and check tuples as usual:

```
memo = dict()
memo[(i, j)] = 1
if (i, j) in memo: return memo[(i, j)]
```

Using a tuple is the most clear solution in Python, but it may not be available in other languages. We can also use a nested dictionary:

```
memo = dict()
if not i in memo: memo[i] = dict()
memo[i][j] = 1
if i in memo and j in memo[i]: return memo[i][j]
```

Here is the memoization implementation for "Count paths in a grid":

```
def countPathsInGrid(r, c):
    memo = dict()

    def countRec(i, j):
        if i == r-1 or j == c-1: return 1
        if (i, j) in memo: return memo[(i, j)]
        memo[(i,j)]= countRec(i+1, j) + countRec(i, j+1)
        return memo[(i,j)]

    return countRec(0, 0)
```

Tabulation with 2D recurrence

For tabulation, we follow the same 4-step outline as 1D, but we need to think about more details:

1. Initialize DP Table. How many dimensions should the table have? Which input corresponds to each dimension in the table? What's the range of each dimension?
2. Set the cells corresponding to the base cases directly. Which cells correspond to the base cases? They could be just a cell, or an entire row / column / diagonal.
3. Set the rest of cells using the general case. This requires a nested loop for each dimension. The most important part is to iterate through the table in an order such that cells are computed before they are read.
4. Return the result. Which cell contains the result?

Here is the tabulation implementation for the "Count paths in a grid". In essence, it is filling the table from the last picture, row by row, starting from the last one, and starting each row from the right.

```
def countPathsInGrid(r, c):
    T = [c * [0] for i in range(r)]
    for i in range(r): T[i][c-1] = 1
    for j in range(c): T[r-1][j] = 1
    for i in range(r-2, -1, -1):
        for j in range(c-2, -1, -1):
            T[i][j] = T[i+1][j] + T[i][j+1]
    return T[0][0]
```

Note: In Python, T = r * [c * [0]] does NOT do what we want: if initialized this way, T is an array r pointers to the same row, instead of r independent rows!

DP Analysis

After implementing an algorithm, the interviewer will ask or expect us to analyze it.

Both memoization and tabulation have the same runtime complexity in terms of big-O, but tabulation has a smaller constant factor. This is because arrays are faster to access than hash tables, plus there is no overhead from calling functions.

The analysis follows a simple formula:
Time: O((number of subproblems) x (time per subproblem))
Space: O(size of DP table)
In general, the size of the DP table is the same as the number of subproblems.
The time per subproblem depends on how complex the general case of the recurrence is, without counting recursive work. It will generally be constant time (O(1)) unless the general case contains a range of expressions, in which case it can be up to O(n).

There is a common technique called "space optimization" where, instead of storing every computed subproblem, we store only the ones that we will need in the future. For example, in

house robber, we only need to remember the value of the last two subproblems, so we only need O(1) space instead of O(n). For "counting paths in a grid", we only need to remember the values of the previous row, so we only need O(r) space instead of O(r*c). This optimization is only compatible with tabulation. It will be discussed in "advanced topics" section.

Should you use memoization or tabulation?

Both start with a recurrence, and you can always use either to turn it into efficient code. Both follow the structure of the recurrence (base cases, general case, goal), so the main difference is that one is recursive and the other iterative. The recursive calls in memoization become table reads in tabulation.

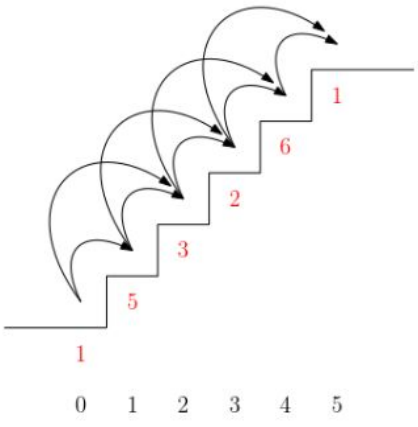
You should choose the one that comes easier to you (I don't know if interviewers may ask you to use one specifically. If you want to be prepared for that eventuality, learn both).

The runtime analysis should not be a major factor in the decision, unless you want to use the space optimization, in which case you should use tabulation.

Practice problems

Min-sum Stairs

Link: https://leetcode.com/problems/min-cost-climbing-stairs/
Description: You are climbing a staircase. Each step has a cost you pay to step on it.
You can climb 1 or 2 steps at a time. What's the minimum you have to pay to reach the last step?
Input: a list steps of length n≥1 with positive costs.
Example:



Approach:

Before we start solving the problem, we should ask clarifying questions about any part of the statement that is vague. Do we start on the first step or before it? Do we have to get the last step or after it? For this problem, assume that we start before the first step, and must get to the step after the last.

This problem is similar to "house robber". It is an optimization (minimization) problem, and there is an exponential number of ways to get to the top. These are clues to use DP.

At the beginning, there are two options, go up one step, or go up two. We don't know which one is optimal, but the important thing is that we know that one of the two options must happen. We can write a recurrence where we consider the two options, and take the best one with min.

Cost(i): minimum cost to pay to get to the top from step i. (this is a forward recurrence)
Base cases:
Cost(n-1) = steps[n-1] (the last step)
Cost(n-2) = steps[n-2] (second-to-last step, we can jump directly to the top)
General case ($0 \le i \le n-3$):
Cost(i) = steps[i] + min(Cost(i+1), Cost(i+2))
Goal: min(Cost(0), Cost(1)) (from the start we can jump to step 0 or step 1)

We always have some flexibility in defining the base cases. For example, we could have defined them as the point where we are already at the top: Cost(n) = 0, Cost(n+1) = 0. In that case, the general case would have to reach up to index n-1. The important part is that there is no gap between them.

Here is how we implement the forward recurrence with memoization:

```
def minCostStairs(steps):
    n = len(steps)
    memo = dict()

    def stairsRec(i):
        if i == n-1: return steps[n-1]
        if i == n-2: return steps[n-2]
        if i in memo: return memo[i]
        memo[i] = steps[i] + min(stairsRec(i+1), stairsRec(i+2))
        return memo[i]

    return min(stairsRec(0), stairsRec(1))
```

Here is how we implement the forward recurrence with tabulation. Recall the 4 steps: initialize the table, add the base cases directly, fill the rest with the general case, and return the solution based on the goal.

```
def minCostStairs(steps):
    n = len(steps)
    if n == 1: return 0 #early check to prevent out-of-bounds
    T = n*[0]
    T[n-1] = steps[n-1]
    T[n-2] = steps[n-2]
    for i in range(n-3, -1, -1):
        T[i] = steps[i] + min(T[i+1], T[i+2])
    return min(T[0], T[1])
```

We can also write a backward recurrence. We don't include the implementation to avoid repetition.

Cost(i): minimum cost to get to step i from the bottom. (backward recurrence)
Base cases:
Cost(0) = steps[0]
Cost(1) = steps[1]
General case ($2 \le i \le n-1$):
Cost(i) = steps[i] + min(Cost(i-1), Cost(i-2))
Goal: min(Cost(n-2), Cost(n-1)) (we can jump to the top from step n-1 or n-2)

Count stairs

Link: https://leetcode.com/problems/climbing-stairs/
Description: You are climbing a staircase with n steps. You can climb 1 or 2 steps at a time. In how many different ways can you get to the top?
Input: the number of stairs, $n \ge 1$.
Examples:
1 -> 1, 2 -> 1 (one by one or both at once), 3 -> 3

Approach: this is a counting problem, a hint that we should use DP. Since it is a counting problem, we aggregate the options with sum. In fact, this problem is almost exactly the same as the Fibonacci sequence. Here is a forward recurrence:

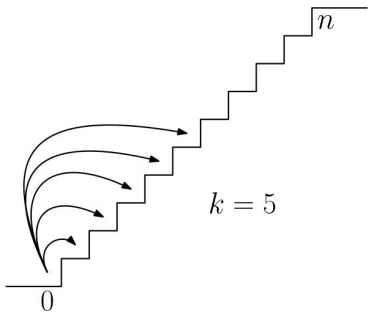
Count(i): number of ways to get to the top from step i.
Base cases:
Count(n-1) = 1 (the only option is to take the last step)
Count(n-2) = 2
General case ($0 \le i \le n-3$):
Count(i) = Count(i+1) + Count(i+2)
Goal: Count(0) (before the first step)

The memoization or tabulation codes are very similar to the previous problem. Here is tabulation:

```
def countStairs(n):
    if n == 1: return 1 #early check to prevent out-of-bounds
    T = n*[0]
    T[n-1] = 1
    T[n-2] = 2
    for i in range(n-3, -1, -1):
        T[i] = T[i+1] + T[i+2]
    return T[0]
```

Count stairs with hops

Link: not on leetcode
Description: the same as count stairs, but we can jump between 1 and k steps at a time.
Input: the number of stairs, n≥1 and the max hop, k≥1.
Example:
n = 3, k = 3
There are 4 ways to get to the top.
1 by 1, go up 1 and hop 2, hop 2 first and go up 1, or hop all 3 at once.



Approach: this is a generalization of the previous problem, so the logic is very similar. Often, if we solve a basic version of a problem, an interviewer will propose a generalization to make it harder. Here is the generalized forward recurrence:

Count(i): number of ways to get to the top from step i.
Base cases:
Count(n) = 1 (we are at the top)
Count(i) = 0 if i > n (we are past the top)
General case (0 ≤ i ≤ n-1):
Count(i) = Count(i+1) + Count(i+2) + ... + Count(i+k)
Goal: Count(0) (the starting point is before the first step)

Note that there is a range in the general case. In code, this becomes an inner loop. Here is a memoization version:

```
def countStairsWithHops(n, k):
    memo = dict()

    def countRec(i):
        if i == n: return 1
        if i > n: return 0
        if i in memo: return memo[i]
        memo[i] = 0
        for hop in range(1, k+1):
            memo[i] += countRec(i + hop)
        return memo[i]

    return countRec(0)
```

You can try to convert it into tabulation. Note that the table should have size n, since we go from index 0 to index n (both included). We can check for the base cases for i > n directly inside the general case, instead of storing them in the table.

We could also start with a backward recurrence:

Count(i): number of ways to get to step i from the base.
Base cases:
Count(0) = 1 (there is 1 way to be at the starting point)
Count(i) = 0 if i < 0 (it is impossible to be before the starting point)
General case (1 ≤ i ≤ n):
Count(i) = Count(i-1) + Count(i-2) + ... + Count(i-k)
Goal: Count(n)

Min-sum stairs with hops

Link: not on leetcode
Description: same as min-sum stairs, but we can jump between 1 and k steps at a time.
Input: a list steps of length n≥1 and the max hop, k≥1.

Approach: this is the last variation of stairs problems. This is an optimization problem, but now we have k options. Again, we choose the best one by using the min operator, but now we have to apply it to the entire range.

Cost(i): minimum cost to pay to get to the top from step i.

Base cases:
Cost(n) = 0 (we are at the top)
Cost(i) = INF if i > n (we don't want to go past the top. We can achieve this by using INF, because an option with cost INF will never be chosen by min)
General case (0 ≤ i ≤ n):
Cost(i) = steps[i] + min(Cost(i+1), ..., Cost(i+k))
Goal: min(Cost(0), ..., Cost(k-1)) (from the start we can jump to any of the first k steps)

The implementation is mostly the same as the previous problems. The inner loop in the general case can be implemented like this:

```
memo[i] = costRec(i+1)
for hop in range(i+2, i+k+1):
    memo[i] = min(memo[i], costRec(i+hop))
memo[i] += steps[i]
```

Analysis:
All the problems we have seen about stairs have O(n) subproblems. The time per subproblem is O(1) when we can only go up 1 or 2 steps, since that is a constant number of options. When we can go up by up to k steps at a time, the time per subproblem is O(k), since we need to iterate through k different options to get the best one. Thus, the runtime is O(nk).

The space for all the variations is O(n). We haven't covered it yet, but it is good to know that we can use the space optimization discussed in "Advanced topics". The idea is that we only need to remember the last 2 subproblem solutions or the last k subproblem solutions, depending on the variation. Thus, we can improve the space usage to O(1) or to O(k), respectively.

Word Break

Link: https://leetcode.com/problems/word-break/
Description: we are given a text and a dictionary containing a list of words. Can we break the text into a sequence of words from the dictionary?
Input: a string text of length n≥1, and a list of words wordDict.
Examples:
Input: text = "abababa", wordDict = ["ab", "aba"]
Output: True, because we can break it as "ab ab aba". Note that if we start using "aba" instead of "ab", there is no way to continue.
Input: text = "abababa", wordDict = ["ab", "ba"]
Output: False. We can see that there is no way because the words in dict have 1 "a" and 1 "b", but text has 4 "a" and 3 "b".
Input: text = "aaabaaabaaabaaa", wordDict = ["a", "aa", "b"]
Output: True. In fact, there are 3*3*3*3 = 81 ways, because each block of 3 a's can be separated in 3 ways.

Approach:
This is a "satisfiability" problem: we want to know if the input satisfies a property. We can try to "match" the text with words from the dictionary in order from left to right. However, if more than one word match a prefix of text, it is not clear which one to choose. This is a clue that we should try DP: we can try all the words that match a prefix of text. For each one, we will have left some suffix of text, i.e., a smaller subproblem. If any of these subproblems can be broken into words, then the entire string can be broken into words.

Recurrence:
Match(i): True if the suffix text[i..n-1] can be separated into words (forward recurrence).
Base cases:
Match(n) = True (since the string ends at index n-1, the suffix starting at index n is the empty string)
General case ($0 \le i \le n-1$):
Match(i) = True if there is a word w in wordDict such that w is a prefix of s[i..n-1] and Match(i+length(w)) is True
Goal: Match(0)

Memoization:

```
def wordBreak(text, wordDict):
    n = len(text)
    memo = dict()

    def wordBreakRec(i):
        if i == n: return True
        if i in memo: return memo[i]
        memo[i] = False #initialization before loop
        for w in wordDict:
            isPrefix = i+len(w) <= n and w == text[i:i+len(w)]
            if isPrefix and wordBreakRec(i+len(w)):
                memo[i] = True
                break
        return memo[i]

    return wordBreakRec(0)
```

Note: the inner loop implements a "logical OR" over all the words that are a prefix of text[i:n-1].

Analysis: there are O(n) subproblems. For each subproblem, we are looping over all the words in wordDict. The time per subproblem is O(k), where k is the sum of the lengths of the words in wordDict. Thus, the runtime is O(nk). For each subproblem, we only store True or False, so the space is O(n).

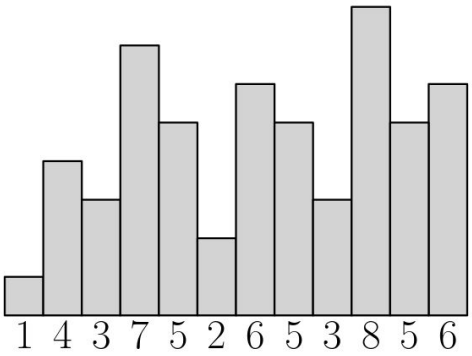
There is a data structure that we can use to improve the runtime of the algorithm: a trie. Tries maintain a set of words in a tree data structure. Tries are beyond the scope of this workshop, but in case you want to dig deeper into this, the final result is that with a trie, we can improve the runtime from O(nk) to O(nm), where m is the maximum length of any word in wordDict. If wordDict is an actual dictionary, it would contain thousands of words but each word would be itself quite short. In that case, this optimization would make a big difference.

Tabulation:

```
def wordBreak(text, wordDict):
    n = len(text)
    T = (n+1)*[False] #n+1 because the subproblems range from 0 to n, both included
    T[n] = True
    for i in range(n-1, -1, -1):
        for w in wordDict:
            isPrefix = i+len(w) <= n and w == text[i:i+len(w)]
            if isPrefix and T[i+len(w)]:
                T[i] = True
                break
    return T[0]
```

Longest Increasing Subsequence

Link: https://leetcode.com/problems/word-break/
Description: Given a list of integers, find the longest increasing subsequence (the elements of a subsequence do not need to be contiguous).
Input: an array v of length n≥1.
Example:

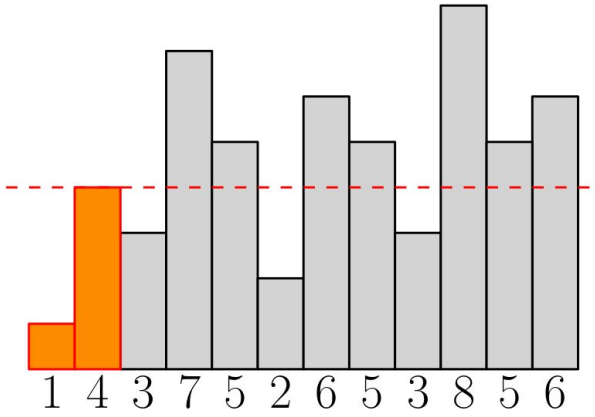


Output: 5. There are multiple solutions, one is 1, 3, 5, 6, 8. Another is 1, 2, 3, 5, 6.

Approach:

This is a maximization problem. We can think of constructing an increasing subsequence one element at a time. For each element, we will face a decision: add it to the subsequence or not. As you can imagine, sometimes it will not be obvious in hindsight if it is better to add an element or not. This is our clue that we should use DP: try both options, and "choose" the best one with "max".

The first decision we have to make is about the first element. In the option where we don't include it, we just reduce the problem to a subproblem of size n-1. In the case where we include it, we need to find the LIS of the remaining elements but filtering out the elements ≤ the last one. Thus, in the recurrence we need to keep track of two things: the length of the prefix and the max value. This is the first recurrence that we will see with two inputs.



If we choose 1 and 4, we need to filter out all numbers ≤ 4

Recurrence:
LIS(i, k): length of the LIS of the suffix v[i..n-1] with only values >k.
Base case:
LIS(n, k) = 0 for any k (v ends at index n-1, so the base case corresponds to the empty suffix)
General case (0 ≤ i ≤ n-1):
LIS(i, k) = max(LIS(i+1, k), 1+LIS(i+1, v[i])) if v[i] > k, else LIS(i+1, k)
(the "else" case is when the current element is below the filter, so our only option is to skip it)
Goal: LIS(0, -INF) (-INF because we start without any filter).

This is an example where the original problem and the recurrence are not exactly the same. We added an extra constraint.

Memoization:

```
def LIS(v):
    n = len(v)
    if n == 0: return 0
    memo = dict()
```

```
    def LISRec(i, k):
        if i == n: return 0
        if (i,k) in memo: return memo[(i,k)]
        if v[i] > k:
            memo[(i,k)] = max(LISRec(i+1, k), 1+LISRec(i+1, v[i]))
        else:
            memo[(i,k)] = LISRec(i+1, k)
        return memo[(i,k)]

    return LISRec(0, min(v)-1) #there is no -INF in Python, so we use a value smaller than any value in v.
```

Note how we use tuples as keys in a Python dictionary. This is very useful for memoization problems with recurrences with multiple inputs.

We just saw an example where the recurrence was not exactly the same as the original problem. It had an extra constraint (a filter). An extra constraint that is useful many times is to force the first index to be included. For example, instead of looking for the LIS of v[i..n-1], we look for the LIS of v[i..n-1] that includes v[i]. In this case, this allows us to have a recurrence for LIS with a single input instead of two:

Recurrence:
LIS(i): length of the LIS of the suffix v[i..n-1] starting at v[i].
Base case:
LIS(n-1) = 1
General case ($0 \leq i \leq n-2$):
LIS(i) = 1 + max(LIS(j)) over all $j > i$ such that $v[j] > v[i]$
Goal: max(LIS(i)) over all i from 0 to n-1

In this new recurrence, in the general case, we consider all the possible continuations (index j) for the LIS. In addition, since the LIS could start at any index, and not necessarily at index 0, the goal is the maximum over all the indices where the LIS could begin.

Memoization:

```
def LIS(v):
    n = len(v)
    if n == 0: return 0
    memo = dict()

    def LISRec(i):
        if i == n-1: return 1
```

```
        if i in memo: return memo[i]
        memo[i] = 1
        for j in range(i+1, n):
            if v[j] > v[i]:
                memo[i] = max(memo[i], 1 + LISRec(j))
        return memo[i]

    res = LISRec(0)
    for i in range(1, n):
        res = max(res, LISRec(i))
    return res
```

Tabulation:

```
def LIS(v):
    n = len(v)
    if n == 0: return 0
    T = n*[0]
    T[n-1] = 1
    for i in range(n-2, -1, -1):
        T[i] = 1
        for j in range(i+1, n):
            if v[j] > v[i]:
                T[i] = max(T[i], 1+T[j])
    return max(T)
```

Analysis: we will compare the two recurrences for this problem. The two recurrences have the same runtime, but one uses less space.

Recurrence	# subproblems	time per subproblem	total time	total space
LIS(i, k)	O(n^2)	O(1)	O(n^2)	O(n^2)
LIS(i)	O(n)	O(n)	O(n^2)	O(n)

For the first recurrence, the number of subproblems is O(n^2) because the second index only ever takes n+1 distinct values. However, the values of the second input can be very large relative to n (imagine input [1, 2, 10000]), so, for tabulation, we cannot use an array. The easiest solution use a dictionary instead of a table like we do with memoization. If we want to use a stick to using a table, another option is to use indices in the table instead of values, that is, redefine LIS(i, j) to be "the length of LIS of v[i..n-1] with values > v[j]". Then, the table clearly has size O(n^2). We will discuss multi-dimensional tables more in detail later.

Count Paths In Grid With Obstacles

Link: https://leetcode.com/problems/unique-paths-ii/
Description: Count the number of ways to go from the top-left cell of a grid to the bottom-right cell. You can only move right or down. Each cell is either empty or there is an obstacle, meaning that you cannot go through it.
Input: a matrix grid with r≥1 rows and c≥1columns, where each cell is either 0 (empty) or 1 (obstacle).
Examples:
`00000`
`00010`
We have two options at the beginning, but actually there is a single path: if we go down early, we cannot reach the bottom-right corner.

`00001`
`10000`
We only have one option at the beginning, but it is possible to reach the goal. In total we have to go down once, and there are 3 points where we can do that, so there are 3 different ways.

Approach: this is a generalization of "count paths in a grid". We simply need to discard paths that run into an obstacle. To make things easy, instead of the base cases being at the last row and column, we can put them beyond the valid cells and set them to 0.

Count(i, j): number of ways to reach the bottom-right corner (cell (r-1, c-1)) from cell (i, j).
Base cases:
Count(r-1, c-1) = 0 if grid[r-1][c-1] == 1, else 1
Count(r, j) = 0 for all j (cells past the last row)
Count(i, c) = 0 for all i (cells past the last column)
General case (0 ≤ i ≤ r-1 and 0 ≤ j ≤ c-1, except i=r-1 and j=c-1)):
Count(i, j) = 0 if grid[i][j] == 1, else Count(i+1, j) + Count(i, j+1)
Goal: Count(0, 0)

Here is a memoization implementation. Note that for the cells with obstacles, we don't remember the "0" in the memo dictionary. We can return directly.

```
def countPathsWithObstacles(grid):
    r, c = len(grid), len(grid[0])
    memo = dict()

    def countRec(i, j):
        if i == r-1 and j == c-1:
            if grid[r-1][c-1] == 0: return 1
```

```
            else: return 0
        if i == r or j == c: return 0
        if grid[i][j] == 1: return 0
        if (i, j) in memo: return memo[(i, j)]
        memo[(i, j)] = countRec(i+1, j) + countRec(i, j+1)
        return memo[(i, j)]

    return countRec(0, 0)
```

And here is a tabulation implementation:

```
def countPathsWithObstacles(grid):
    r, c = len(grid), len(grid[0])
    T = [(c+1) * [0] for i in range(r+1)] #one extra row/column for the off-grid base cases
    if grid[0][0] == 1 or grid[r-1][c-1] == 1: return 0 #we can stop  early
    T[r-1][c-1] = 1 #the other base cases are already set to 0 by default
    for i in range(r-1, -1, -1):
        for j in range(c-1, -1, -1):
            if i == r-1 and j == c-1: continue #skip the base case
            if grid[i][j] == 1: T[i][j] = 0
            else: T[i][j] = T[i+1][j] + T[i][j+1]
    return T[0][0]
```

Min-cost path in Grid

Link: https://leetcode.com/problems/minimum-path-sum/
Description: Find the cheapest way to go from the top-left cell of a grid to the bottom-right cell. You can only move right or down, and each cell has a cost.
Input: a matrix grid with $r \geq 1$ rows and $c \geq 1$columns, where each cell has a numeric cost (positive or negative).

Approach: this is a similar problem as the previous one, but it is an optimization problem instead of counting. This means that we will aggregate subproblems with "min". For this problem, we show a backward recurrence.

Recurrence:
Cost(i, j): minimum cost to reach cell (i, j) from cell (0, 0)
Base cases:
Cost(0, 0) = grid[0][0]
Cost(i, j) = INF if i == -1 or j == -1 (beyond the valid cells)
General case ($0 \leq i \leq r-1$ and $0 \leq j \leq c-1$, except i==0 and j==0):
Cost(i, j) = grid[i][j] + min(Cost(i-1, j), Cost(i, j-1))
Goal: Cost(r-1, c-1)

```
def minCostSumPath(grid):
    memo = dict()
    r, c = len(grid), len(grid[0])

    def cost(i, j):
        if i == -1 or j == -1: return 999999 #only OK if we know it's bigger than any valid path
        if i == 0 and j == 0: return grid[0][0]
        if (i, j) in memo: return memo[(i, j)]
        memo[(i, j)] = grid[i][j] + min(cost(i, j-1), cost(i-1, j))
        return memo[(i, j)]

    return cost(r-1, c-1)
```

Longest Common Subsequence

Link: https://leetcode.com/problems/longest-common-subsequence/
Description: Given two strings, find the length of the longest string that is a subsequence of both (the elements of a subsequence do not need to be contiguous).
Input: strings s1, of length n1≥1, and s2, of length n2≥1.
Examples:
s1: PATHRISE, s2: TOBEWISE
Output: 4, as "TISE" is a subsequence of both s1 and s2:
TISE
T****ISE
Note that the matched letters don't need to be at the same index. Otherwise, the problem would be much simpler.

s1: aabbaa, s2: ababab
The length of the LCS is 4, and there are many common subsequences of that length, like "aabb", or "abaa".

Approach:
This is an optimization problem. In order to apply DP, we should try to find a set of options that, even if we don't know which one is best in hindsight, we can be sure that one of them must happen. Then, we can try them all and get the best with "max". In order to find this set of encompassing options, it helps to focus on the first element of the input and think about all the things that could happen to it. Let's consider what happens to the first char of s1. If s2 does not contain that char, we need to discard it, so s1 becomes one char shorter. If s2 starts with the same char, we can add that char to the LCS, and both s1 and s2 become one char shorter. Finally, if the first char of s1 appears in s2 but not at the beginning, we have two options: we can discard it anyway, or we can "match" it with the first occurrence in s2, which means we are discarding all the chars in s2 before it. Note that it would not make sense to match the first char

of s1 with a char of s2 that is not the first occurrence. That would only discard additional chars, which clearly does not help.

In the second example, we can start by matching the first a in both strings. Then, we have left s1: abbaa, s2: babab. s1 starts with an "a". We can discard it, or we can discard the first "b" in s2 and match it with the following "a" in s2.

Recurrence:
LIS(i, j): LIS of s1[i..n1-1] and s2[j..n2-1].
Base case:
LIS(n1, j) = 0 for any j (empty suffix of s1)
LIS(i, n2) = 0 for any j (empty suffix of s2)
General case ($0 \le i \le n1-1$, $0 \le j \le n2-1$):
LIS(i, j) =
if s1[i] not in s2[j..n2-1]: LIS(i+1, j)
else if s1[i] == s2[j]: 1 + LIS(i+1, j+1)
else max(LIS(i+1, j), 1 + LIS(i+1, k)), where k is the smallest index in [j..n2-1] with s1[i] == s2[k]

We can turn it into memoization code:

```
def LCS(s1, s2):
    n1, n2 = len(s1), len(s2)
    memo = dict()

    def memoLCS(i, j):
        if i == n1: return 0
        if j == n2: return 0
        if (i, j) in memo: return memo[(i, j)]
        pos = s2.find(s1[i], j, n2) #first pos of s1[i] in s2[j..n2-1]
        if pos == -1: memo[(i,j)] = memoLCS(i+1, j)
        elif pos == j: memo[(i,j)] = 1+memoLCS(i+1, j+1)
        else: memo[(i,j)] = max(memoLCS(i+1, j), 1+memoLCS(i+1, pos+1))
        return memo[(i,j)]

    return memoLCS(0, 0)
```

Analysis: there are O(n1*n2) subproblems. The most expensive operation for each subproblem is the call to find(). This just checks char by char, so it takes linear time. Thus, the total runtime is O(number of subproblems * time per subproblem) = O(n1*n2*n2).

It is possible to use data structures to make the "find()" step faster and improve the runtime. For example, for each char in s2, we can store all its positions in a BST. Then, given an index j, we can find the smallest index >j containing the char in O(log n2) time.

However, we can also improve the runtime by using a different recurrence. The definition and the bases cases are the same, but the idea is that we don't need to bother finding the next occurrence of s1[i] in s2[j..n2-1] in the general case. Recall that our goal is to find a set of options that leave smaller subproblems and we know for sure that at least one of them must happen. So, if s1[i] != s2[j], what we know for sure is that we will need to discard s1[i], discard s2[j], or discard both. Discarding both is the same as discarding one first and then the other, so this option is already covered by the first two.

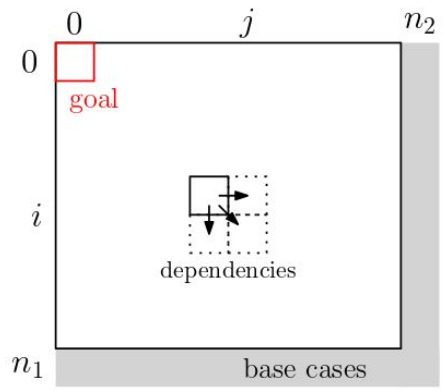
Recurrence:
LIS(i, j): LIS of s1[i..n1-1] and s2[j..n2-1].
Base case:
LIS(n1, j) = 0 for any j (empty suffix of s1)
LIS(i, n2) = 0 for any j (empty suffix of s2)
General case ($0 \le i \le n1-1$, $0 \le j \le n2-1$):
LIS(i, j) =
if s1[i] == s2[j]: 1 + LIS(i+1, j+1)
else: max(LIS(i+1, j), LIS(i, j+1))

Here is a memoization implementation for the second recurrence:

```
def LCS(s1, s2):
    n1, n2 = len(s1), len(s2)
    memo = dict()

    def memoLCS(i, j):
        if i == n1: return 0
        if j == n2: return 0
        if (i, j) in memo: return memo[(i, j)]
        if s1[i] == s2[j]: memo[(i,j)] = 1 + memoLCS(i+1, j+1)
        else: memo[(i,j)] = max(memoLCS(i+1, j), memoLCS(i, j+1))
        return memo[(i,j)]

    return memoLCS(0, 0)
```

The number of subproblems is the same, but the time per subproblem is now constant, so the runtime is O(n1*n2).

In order to implement the recurrence using tabulation, we need to think about the layout of the 2D table:



		t	o	b	e	w	i	s	e	
p	4	3	3	3	3	3	2	1	0
a	4	3	3	3	3	3	2	1	0
t	4	3	3	3	3	3	2	1	0
h	3	3	3	3	3	3	2	1	0
r	3	3	3	3	3	3	2	1	0
i	3	3	3	3	3	3	2	1	0
s	2	2	2	2	2	2	2	1	0
e	1	1	1	1	1	1	1	1	0
	0	0	0	0	0	0	0	0	0

The first picture is the schematics, and the second picture is a concrete example. The blue cells are the cells (i, j) where s1[i] and s2[j] match.

Interview advice: in a setting where we have access to a whiteboard, draw the table layout before starting to code. It helps identify the dimensions the table should have, the base cases that need to be filled upfront, the indices of the goal, and the order in which to fill the table. Drawing the table for an actual example can be very time consuming if it is a 2D table, so I would choose a very small example or not waste time on it.

Here is a tabulation implementation that fills the table above:

```
def LCS(s1, s2):
    n1, n2 = len(s1), len(s2)
    T = [(n2+1) * [0] for i in range(n1+1)]
    #base cases set to 0 in initialization already
    for i in range(n1-1, -1, -1):
        for j in range(n2-1, -1, -1):
            if s1[i] == s2[j]: T[i][j] = 1 + T[i+1][j+1]
            else: T[i][j] = max(T[i+1][j], T[i][j+1])
    return T[0][0]
```

Note that we can fill the table by rows or by columns. If we interchange the order of the two for-loops, the result is the same.

Minimum Insertions to make Palindrome

Link: https://leetcode.com/problems/minimum-insertion-steps-to-make-a-string-palindrome/
Description: Given a string, find the minimum number of characters that need to be added so that the string becomes a palindrome (reads the same forward and backward).
Input: strings s, of length n≥1.
Examples:

s: aabb
Output: 2, as we can make s a palindrome by adding "aa" at the end or "bb" at the beginning.

s: abcabc
Output: 3, we can add 2 c's and an a:

```
 abca b c (original string)
cabcacbac (palindrome with insertions)
```

s: programming
Output: 7

```
prog ra mmi  ng    (original string)
prognraimmiarngorp (palindrome with insertions)
```

Approach: this is a minimization problem. Let us try to think of all the options for the first char. If the first and last chars match, we can match them and get rid of them at no cost. Otherwise, we need to make one insertion to provide a match for at least one of them. Maybe we need to insert a match for both, but that is the same as inserting a match for one and then one for the other, so we do not need to consider this case separately.

Note: Using suffixes as subproblems doesn't work well in this problem because, from the logic above, subproblems get smaller on both ends. Thus, subproblems should be substrings. A substring can be indicated with two indices, the start and the end. Thus, we have O(n^2) subproblems.

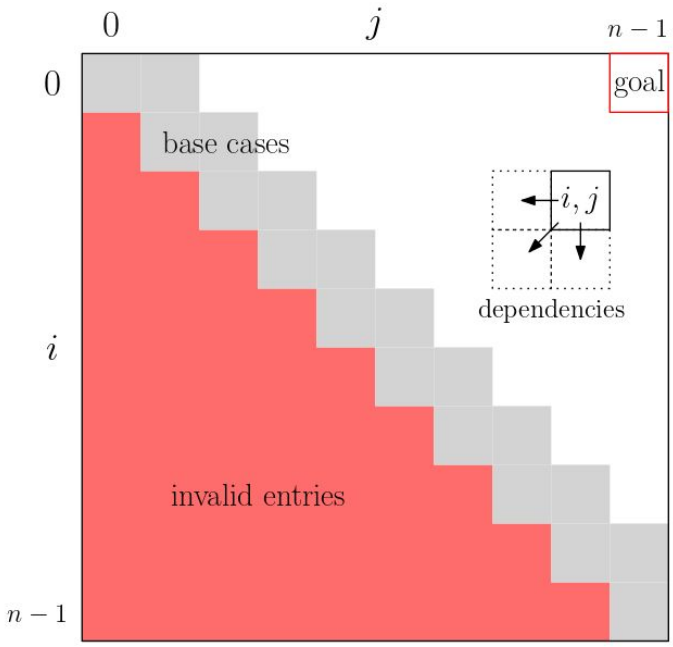
Recurrence:
INS(i, j): number of chars that need to be inserted in s[i..j] to make it a palindrome.
Base cases (substrings of length 1 and 2):
INS(i, i) = 0 (substrings of length 1, they are all palindromic)
INS(i, i+1) = 0 if s[i]==s[i+1], else 1 (substrings of length 2)
General case (substrings of length≥3):
INS(i, j) = INS(i+1, j-1) if s[i] == s[j], else min(1+INS(i+1, j), 1+INS(i, j-1))
Goal: INS(0, n-1) (the entire s)

Memoization implementation:

```
def minInsertionsPal(s):
    memo = dict()

    def INS(i, j):
        if j == i: return 0
        if j == i+1:
            if s[i] == s[j]: return 0
            return 1
```

```
        if (i,j) in memo: return memo[(i,j)]
        if s[i] == s[j]: memo[(i,j)] = INS(i+1, j-1)
        else: memo[(i,j)] = min(1 + INS(i+1, j), 1 + INS(i, j-1))
        return memo[(i,j)]

    return INS(0, len(s)-1)
```

For tabulation, we need to think about the table layout:



Entries with i > j are invalid because the start index should be smaller or equal than the end index. Base cases correspond to substrings of length 1 or 2, which are along the main diagonal and the diagonal above it. The goal is at the top-right corner because it has the smallest possible starting index and the largest possible ending index. Based on the order of the dependencies, we need to fill the rows from bottom to top, and the columns from left to right.

```
def minInsertionsPal(s):
    n = len(s)
    T = [[0] * n for i in range(n)]
    for i in range(0, n): T[i][i] = 0
    for i in range(0, n-1):
        if s[i] == s[i+1]: T[i][i+1] = 0
        else: T[i][i+1] = 1
    for i in range(n-3, -1, -1): #rows from bottom to top
        for j in range(i+2, n, 1): #columns from left to right, starting at 3rd diag
            if s[i] == s[j]: T[i][j] = T[i+1][j-1]
```

```
        else: T[i][j] = min(1 + T[i+1][j], 1 + T[i][j-1])
    return T[0][n-1]
```

Number of Roll Sequences With Target Sum

Link: https://leetcode.com/problems/number-of-dice-rolls-with-target-sum/
Description: You roll a die with f faces n times. Out of the f^n possible sequences of values that you can get, how many add up to a given target? For this problem, do not worry about integer overflow.
Input: 3 positive numbers f, n, target, with target ≥ n.
Examples:
f = 6, n = 2, target = 7 (we roll a die with 6 faces twice, and our target sum is 7)
Output: 6. Of all the possible sequences (first roll, second roll), only the ones along the second diagonal below add up to 7:

```
1 1, 1 2, 1 3, 1 4, 1 5, 1 6,
2 1, 2 2, 2 3, 2 4, 2 5, 2 6,
3 1, 3 2, 3 3, 3 4, 3 5, 3 6,
4 1, 4 2, 4 3, 4 4, 4 5, 4 6,
5 1, 5 2, 5 3, 5 4, 5 5, 5 6,
6 1, 6 2, 6 3, 6 4, 6 5, 6 6.
```

Approach: this is a counting problem. Let's think of all the options for the first roll, and what that means for the remaining rolls. The first roll will be some number between 1 and f. If the first roll comes out as some value x, in order for the total sum to equal "target", the sum of the remaining rolls must add up to target-x. Thus, if we count how many sequences with one fewer rolls add up to target-x, we will know how many ways we can add up to target if we roll x at the start. To get the total count, we aggregate by adding over all the possible values of x. This suggests that our subproblems should depend on the number of rolls left and the "remaining" target.

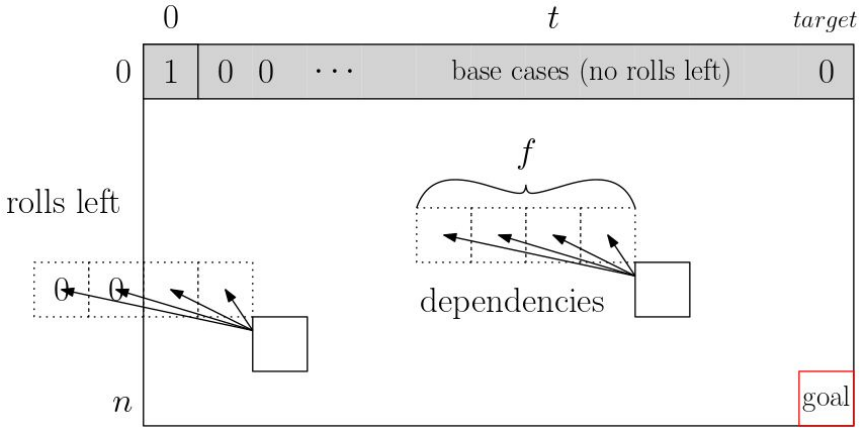
Recurrence:
COUNT(i, t): number of sequences of i rolls that add up to target t.
Base cases:
COUNT(0, 0) = 1 (there is a single sequence of 0 rolls, and its sum is 0)
COUNT(0, t) = 0 for all t > 0 (with 0 rolls, we cannot add to any value other than 0)
COUNT(i, t) = 0 if t < 0 for all i (there is no way to hit a negative target)
General case (i ≥ 1, t ≥ 0):
COUNT(i, t) = sum of all COUNT(i-1, t-x) where x goes from 1 to f
Goal: COUNT(n, target)

For this, I used a backward recurrence (just because I had already drawn the table and didn't want to have to redraw it...sorry!), but you can turn it into a forward recurrence as an exercise. To make a forward recurrence, you would redefine COUNT(i, t) as "number of ways to add up to target target-t with rolls i to n".

Here is a memoization implementation:

```
def numRollSeqsWithTargetSum(f, n, target):
    memo = dict()

    def COUNT(i, t):
        if i == 0 and t == 0: return 1
        if i == 0 and t > 0: return 0
        if t < 0: return 0
        if (i, t) in memo: return memo[(i, t)]
        memo[(i, t)] = 0
        for x in range(1, f+1):
            memo[(i, t)] += COUNT(i-1, t-x)
        return memo[(i, t)]

    return COUNT(n, target)
```

Here is the table layout for tabulation:



Here is a tabulation implementation that fills the table by rows, from top to bottom, and each row from left to right:

```
def numRollSeqsWithTargetSum(f, n, target):
    T = [[0] * (target+1) for i in range(n+1)]
    T[0][0] = 1
    for t in range(1, target+1): T[0][t] = 0
    for i in range(1, n+1):
        for t in range(0, target+1):
            T[i][t] = 0
```

```
        for x in range(1, f+1):
            if t-x < 0: break #this check corresponds to 3rd base case
            T[i][t] += T[i-1][t-x]
    return T[n][target]
```

Minimum max-sum m-partition

Link: https://leetcode.com/problems/split-array-largest-sum/
Description: You are given an array. Over all possible ways to partition the array into m non-empty subarrays, you have to find the one that minimizes the largest sum of any of the m subarrays, and return the sum of the subarray with the largest sum.

Input: an array v of length n and a positive number m ≤ n.
Example:
v = [5 3 7 5 3 2 8], m = 2
Output: 18. We need to find the splitting point that balances the two subarrays as best as possible. We can do it as [5 3 7 | 5 3 2 8]. The largest subarray is the second one, with sum 18.

v = [5 3 7 5 3 2 8], m = 3
Output: 13. Now we have to find two splitting points that minimize the largest sum of the 3 subarrays. We can do it as [5 3 | 7 5 | 3 2 8]. The largest subarray is the last one, with weight 13.

Approach: this is known as a "min-max" problem (minimize the maximum in a set of values), a common type of optimization problem that is often solved with DP. The symmetric type, "max-min" problems, are also common in DP. The mix of min and max can be confusing, so make sure you understand the problem.

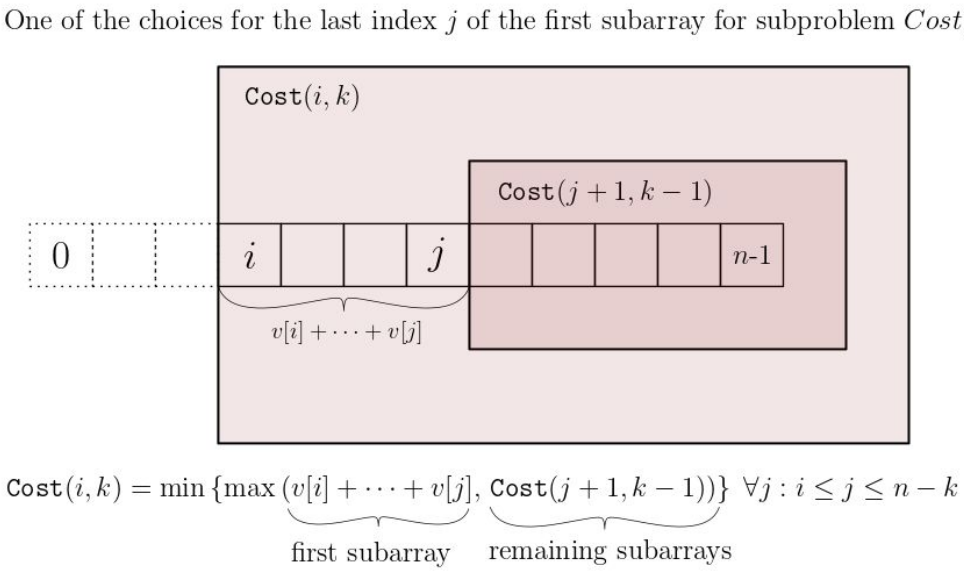
This is an optimization problem where we have to make decisions about where to put the splitting points, but it is not clear in hindsight where we should put them. This is a hint that we should try DP. Next, we need to find some set of options that we know for sure one of them must happen. Well, we don't know where the first subarray should end, but we know it must end somewhere.

Choices for where the first subarray can end in the example:
5 | 3 7 5 3 2 8 (sum of first: 5)
5 3 | 7 5 3 2 8 (sum of first: 8)
5 3 7 | 5 3 2 8 (sum of first: 15)
5 3 7 5 | 3 2 8 (sum of first: 20)
5 3 7 5 3 | 2 8 (sum of first: 23)
5 3 7 5 3 2 | 8 (sum of first: 25)
5 3 7 5 3 2 8 | (sum of first: 33)

Thus, we can consider all possible ending points of the first subarray. For each endpoint, say, j, we will know the sum of the first subarray (v[0]+...+v[j]), and we will have to split the rest into one fewer subarrays. The "cost" of that choice will be the maximum between the sum of the first subarray, and the best solution for the rest of the array (v[j+1..n-1]) with one fewer split. This suggests that subproblems should have two inputs: a suffix of v, and a number of subarrays to split it into. After we considered all the options for where to end the first subarray, we can "choose" the one with the minimum cost using min().

Recurrence:
Cost(i, k): minimum maximum sum across all k-partitions of v[i..n-1].
Base cases:
Cost(i, 1) = v[i]+...+v[n-1] for all i < n (single subarray, no splits)
General case (0 ≤ i ≤ n-1, 2 ≤ k ≤ m):
Cost(i, k) = min { max(v[i]+...+v[j], Cost(j+1, k-1)) } over all j with i ≤ j ≤ n-k
Goal: Cost(0, m)

In the general case, the range of j ends at n-k to make sure that there are at least k-1 elements remaining for the final k-1 subarrays. Here is a picture illustrating the general case:



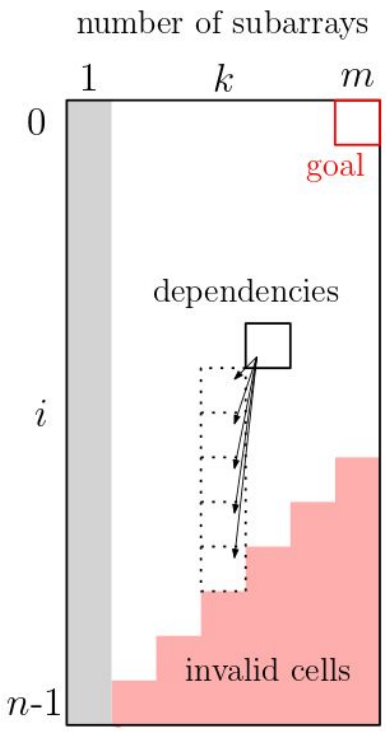
Here is a memoization implementation:

```
def minMaxSumPartition(v, m):
    n = len(v)
    memo = dict()
    accum = n * [0] #accumulated sum of prefixes of v
    accum[0] = v[0]
    for i in range(1, n):
```

```
    accum[i] = accum[i-1] + v[i]
  INF = accum[n-1]+1 #any value larger than the sum of v suffices

  #return v[i]+...+v[j] in O(1) time (after O(n) preprocessing)
  def subarraySum(i, j):
    if i == 0: return accum[j]
    return accum[j]-accum[i-1]

  def cost(i, k):
    if k == 1: return subarraySum(i, n-1)
    if (i,k) in memo: return memo[(i,k)]
    res = INF
    for j in range(i, n-k+1):
      res = min(res, max(subarraySum(i, j), cost(j+1, k-1)))
    memo[(i,k)] = res
    return res

  return cost(0, m)
```

Besides the main recursive function, we have an auxiliary "subarraySum" function that allows us to get the sum of subarrays efficiently. In order to do this, we initialize the array "accum" with the sums of all the prefixes, a common trick.

Here is the table layout for tabulation, along with the table filled for the example:



		k		
$v[i]$	i	1	2	3
5	0	33	18	13
3	1	28	15	10
7	2	25	13	10
5	3	18	10	8
3	4	13	8	8
2	5	10	8	0
8	6	8	0	0

The gray cells are the base cases (single subarray, no splits to be made). The cells are invalid when k is larger than the number of remaining elements in v. They are invalid because all the subarrays must be non-empty.

The dependencies are in the column to the left (one fewer cut) and extend all the way down to the bottom (to the invalid cells). Based on this, we need to fill the columns from left to right.

```
def minMaxSumPartition(v, m):
    n = len(v)
    accum = n * [0] #accumulated sum of prefixes of v
    accum[0] = v[0]
    for i in range(1, n):
        accum[i] = accum[i-1] + v[i]
    INF = accum[n-1]+1 #any value larger than the sum of v suffices

    def subarraySum(i, j):
        if i == 0: return accum[j]
        return accum[j]-accum[i-1]

    T = [(m+1) * [0] for i in range(n)]
    for i in range(n):
        T[i][1] = subarraySum(i, n-1)
```

```
    for k in range(2, m+1):
        for i in range(n-k+1):
            T[i][k] = INF
            for j in range(i, n-k+1):
                T[i][k] = min(T[i][k], max(subarraySum(i, j), T[j+1][k-1]))
    return T[0][m]
```

Advanced Topics

Solution Reconstruction

So far, we focused on how to find the value of the solution. What if we need the solution itself? This does not apply to counting problems, but it does to optimization problems and satisfiability problems.
For example, it is not the same to say that the LCS has length 4, than to say that the LCS is "AABB". Likewise, it is not to say that the text can be broken into words in the dictionary, than to say how to break it. You should ask the interviewer if the solution is necessary or only its value/cost. If the interviewer wants to make it extra hard, maybe she'll ask for every optimal solution.

If we need the solution and not only its value, start by finding the value first anyway! We do everything the same up to this point. After the fact, we will use the subproblems already computed to "guide" us in reconstructing the solution.

Recall that, when using DP for an optimization problem, we have to choose between a set of options (e.g., rob the last house or skip it), but, in hindsight, we do not know which one is the best. If we knew from the beginning which one is the best option, constructing the solution would be trivial, right?

The solutions to the subproblems give us exactly that! We can use the recursive function from memoization or the table from tabulation, it doesn't matter. We can use them like a crystal ball to tell us the final value that we would get by picking each option. In other words: instead of using max to choose the best option, since now the table contains the value of the best option, we can compare the options to the max value in the table to know which one is the best one.

In terms of the analysis, computing the value should still be the bottleneck, so adding the reconstruction part does not change the runtime or space analysis.

Todo: add reconstruction examples.

Space Optimization

So far, we have filled the entire table of subproblems for every problem. However, we can only remember the entries that will be needed again. To know which entries we need to remember, we need to think in terms of the dependencies between subproblems.

1D Examples:
- For Fibonacci and house robber, we only need to remember the last two values.
- For counting stairs in hops, we only need to remember the last k values.
- In Longest Increasing Subsequence (2nd recurrence we saw) we need to remember every smaller subproblem, so we can't optimize the space.

For 2D problems that depend only on the previous row (which is very common), instead of declaring an entire table, we declare only two rows of the table, called lastRow and newRow. The code then looks like this:

Fill the base cases in lastRow.
Fill the next row in newRow using the general case.
Copy newRow to lastRow.
Repeat from step 2 until you get to the last row, which contains your solution.

This reduces the space complexity from O(r*c) to O(r).

Note: this optimization is incompatible with reconstructing the solution, since to reconstruct the solution we use the entire DP table to guide our choices.