An introduction to Dynamic Programming aka DP

Definition

Technique for solving a complex problem by breaking it down into a collection of simpler (could be because of reduced parameters) subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Source: Wikipedia)

For example, let's say your task is to search a word in a dictionary. A classical and intuitive approach says, find a point x in dictionary (ideally a mid-point) and break down the problem of searching into whole dictionary into two parts:

- The one before x
- The one after x

And keep doing this recursively until we find the word. It's another thing that we can optimise this process by eliminating one of the parts as being the solution (since our solution lies in at most one part).

This technique, where the subproblems (breaking into two smaller problems) are disjoint (they don't overlap), is called divide and conquer. Dynamic programming is used where these subproblems overlap.

Step by step understanding

Recursion

As you might have observed, recursion is one of the most important components in expressing your requirements/ solution in terms of smaller problems. As a simple example, consider f(n, r) = number of ways to choose a subset of size r elements, disregarding their order, from a set of n elements.

You're trying to calculate f(n, r), which can be expressed as f(n, r) = f(n-1, r-1) + f(n-1, r). An advantage of expressing your problem is that the total number of different values (i.e. f(n, r) for all n, r) you've to calculate, is limited by n*r.

Base condition

But, how do we calculate? Note that every recursion has a base condition, here it says that f(n, 0) = 1 and f(0, r) = 0. How did we arrive here? Note that f(n, r) is not defined when n<0 or r<0 or

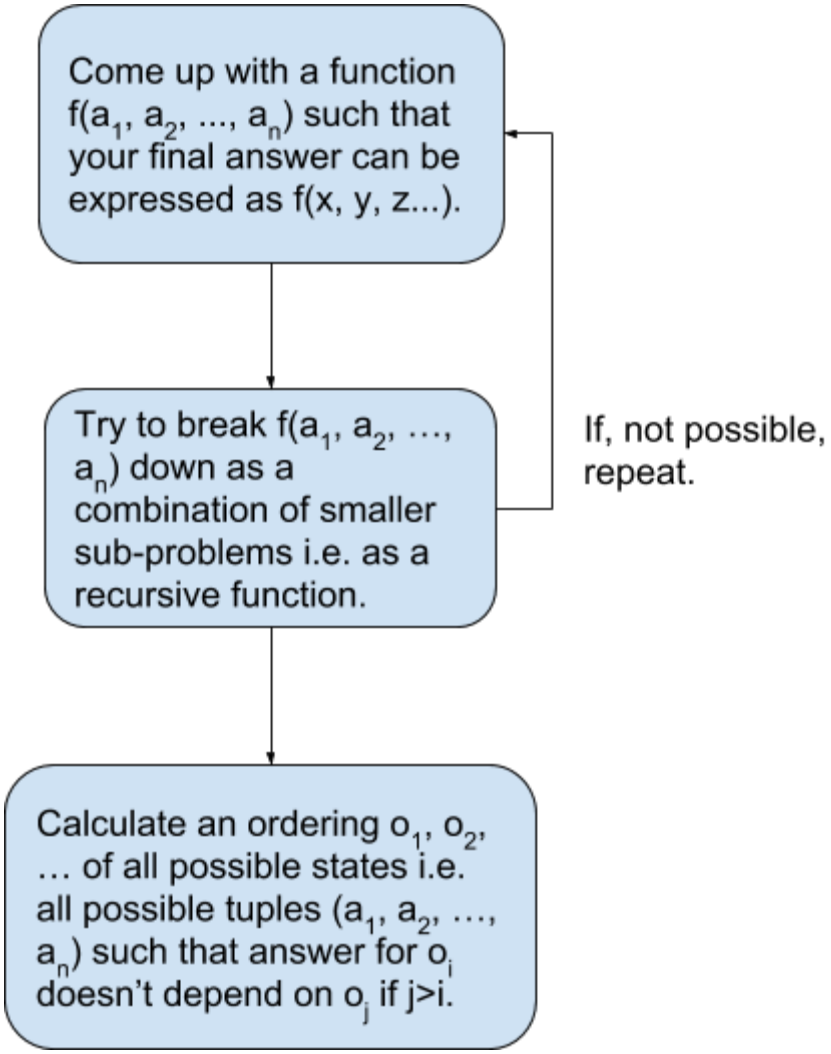
n<r. So, we search for a lattice point f(n, r) where getting the value of f(n-1, r-1) or f(n-1, r) is not possible i.e. n-1<0 or r-1<0 i.e. when n=0 or r=0.

Implementation

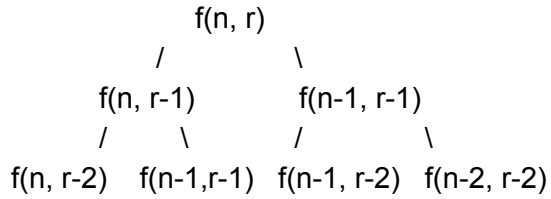
Now, we've to figure out an ordering of all lattice points (i, j) such that when calculating f(i, j), f(i, j-1) and f(i-1, j-1) has been calculated (notice our recursion). You can intuitively observe that if we traverse first in increasing j and then in increasing i i.e. in order (0, 0), (0, 1) … (0, r), (1, 0), (1, 1).... (1, r) … (n, r); this condition will be satisfied.

Algorithm to come up with a DP solution

So, in general terms our algorithm to solve a dynamic programming problem is:



However, we can simplify the last step using programmatic notion of recursion. We can leave this cognitive load of figuring out the ordering to the machine by making an optimisation. We get a little smart and store the values we've already calculated on the go, such that when they're required again, we need not compute them from base again. This technique is called 'memoization'. So, without memoization our computation cycle looks like:



and so on. Note that this huge tree is of exponential size. However, if you observe there are at most n*r distinct nodes. So, if we would start storing our learnings, we'll visit each node in this tree at most once i.e. a total of n*r times.

Complexity Analysis

A "general" rule of thumb for complexity calculation (take with a pinch of salt; not always applicable): If O(k) is the complexity of combining subproblems for a given state, and there are a distinct O(n) states, then total complexity of the solution is O(n*k). An intuitive way to arrive at this result is if you think you're spending O(k) time per state in going to its neighbours. However, you should be extra careful when some states have lesser number of transaction in terms In big-oh complexity.
Tip: How to find the total number of states given a recursive function? All the changing independent parameters of a recursion are part of the state. The tuples of all possible values of these parameters are all possible states.

Study Problems

Problem:

Let's start with a simple one. You're given an array A_1, A_2, …, A_n and you need to choose a contiguous subarray such that its sum is maximum.

DP states
First step is to come up with a state. A usual strategy is to think in random directions here, with practice and intuition, this step gets easier over time. Start with the simplest possible DP state in terms of dimensions.

A general template goes something like f(i, j, k…) denotes the answer if [some conditions on i, j,

k… while relate to the properties of the answer]. For example, in this case various such statements could be:

- f(i, j) is the answer if our answer subarray lies within range [i, j]
- f(i) is the answer if our subarray starts at i
- f(i) is the answer is our subarray end at i

However, not every time would your answer can be expressed as f(i, j, k…). You might need to do some computation over different values of f, to reach the answer.

Recursion

This is generally an easier task once you've come with a DP state. Let's stop here for sometime and try to come up with the recursion for the above defined three states in previous section.

- f(i, j) is the answer if our answer subarray lies within range [i, j].
    - So, our subarray can start at i, in which case f(i, j) = A_i + f(i+1, j). Wait, before proceeding further can you see the flaw here?
        - By our definition f(i, j) tells the maximum subarray sum if subarray lies between i and j. So, when we're adding A_i to f(i+1, j), the flaw is that value returned by f(i+1, j) might be for a subarray which begins at i+2 (let's say). Which means the chosen subset is not contiguous anymore. And this is what we mean by optimality substructure of DP's recursion. Are we considering every possible case by breaking our current problem into smaller problems or not.
    - It's totally possible you come up with an optimal substructure, but the complexity of this solution will always be at least $O(N^2)$ since number of states is lower bounded by this.
- Let's try another state. We say f(i) is the answer is our subarray ends at i. Your aim is to express f(i) in terms of some function $f(j_1)$, $f(j_2)$, … where $j_k < i$, for all k.
    - Our subarray ends at i. We need to concern ourselves with some properties of this subarray such as "where does it begin?". Say the optimal subarray(i.e. maximum sum subarray which ends at i) begins at j
        - where j < i. Then, we can say f(i) = f(i-1) + A_i. Why is it optimal?
        - where j = i. Then, we can say f(i) = A_i
    - Now, the only thing we've to do is take a maximum of these two cases to define our recursion.

Implementation and complexity

- This is the easiest part of all, if you implement our DP recursively using memoization.
- Remember, we also need to define a base case.
- For example, here is the pseudo for above problem:

```
int dp[N];
// dp[i] represents the maximum possible sum of a subarray that ends at position i
// For memoization, say we initialize it to -1 to identify if we've already calculated a DP state or not
// It's essential you initialize it with values which can never be your DP state's actual value

int rec(i, A):
    // base case
    // we're using one indexing
    if i < 1: return 0

    // memoization
    if dp[i] != -1:
        return dp[i]

    return dp[i] = max( rec(i - 1, A) + A[i], A[i])

main()
    scan(N, A)

    answer = 0
    for i = 1 to N:
        answer = max(answer, rec(i, A))
    print answer
```

Problem:

You have N distinct dice of each A sides numbered 1 to A. How many ways can you roll them all together to get the sum of top faces as B. Two ways are considered different if one of the dice has different number on top face in these two ways.

DP states

Let's see what possible states our DP can have.

- f(i) denotes number of ways to roll first i dices to get a sum of B.
    - If we want to relate f(i) with f(i-1) let's say, we have to make a decision on what face value i^{th} dice will have. If we fixate on j, we'll have to form a sum B-j. How do we express this in our DP state? Not possible.
- f(j) denotes number of ways to roll all dice to get a sum of j.

    - Let's related f(j) with f(j-1) say. We're saying we'll give a face value of 1 to one of the dice. Now, we're left with N-1 dice. It's impossible to express this information via our DP state.
- f(i, j) denotes number of ways to roll first i dice to get a sum of j.
    - As you can observe, this a mix of both above DP states.
        - Let's say we try to relate f(i, j) with f(i-1,....). We have a choice to provide any of the A face value to i^{th} dice. If we say i^{th} dice will have face value k, there are further f(i-1, j-k) ways to distribute face values to first i dice.

          Hence, $f(i, j) = \Sigma_{k = 1 \text{ to } A} f(i - 1, j - k)$
          Can you figure out the base cases?

Another way to reach a similar conclusion is to pick out a single entity in the problem (a dice in our case) and assign a possible value to it. Observe that dice left and sum left changed. Then try out all possible values and combine the results. It will reduce the problem to the same recurrence.

Implementation and complexity

```
// dp(i, j) represents number of ways to assign face values to i dices
// such that their sum is b.
int dp[N][B]
def f(diceLeft, sumLeft):
        // base case
        if diceLeft = 0 && sumLeft = 0: return 1

        else If diceLeft <= 0 || sumLeft < 0: return 0

        // memoization
        if dp[diceLeft][sumLeft] != -1:
                return dp[diceLeft][sumLeft]

        int ways = 0
        for i = 1 to A:
                ways += f(diceLeft - 1, sumLeft - i)

        return dp[diceLeft][sumLeft] = ways
```

Complexity?

Remember the trick mentioned before.
Number of states = N * B
Complexity of combining a single sub problem = O(A)
Total time complexity = O(N*B*A)
Total memory complexity = O(N*B)

Additional observations

1) Each state [diceLeft, sumLeft] depends only on diceLeft - 1. What if we store all the possible values in order of diceLeft from 0 to N? In this case we only need to store [diceLeft - 1, x] for computing all possible [diceLeft, y]. This can reduce the memory complexity to O(B)
2) Since a state depends on a contiguous block of states in the previous row, we can use a rolling window approach to compute answers faster. Ie f(N, B) depends on f(N-1, B-i) for i in 1 to A, and f(N, B-1) depends on f(N-1, B-1-i) for i in 1 to A. The difference in the 2 are just 2 values. We can rewrite the recurrence as:
   f(N, B) = f(N, B-1) + f(N-1, B-1) - f(N-1, B-1-A)
   This will reduce the time complexity to O(N*B)

As an exercise try to implement these optimizations handling the edge cases.

A modification to the problem:

You have N distinct dice having A sides each, numbered 1 to A. How many ways can you roll them all together to get the sum of top faces as B such that all dices have distinct numbers on their top faces.

If you use the DP state f(i, j) = number of ways to form a sum j using face value of first i dice and try to assign a value k to the i^{th} dice, you cannot use it for calculating f(i - 1, j - k). This is the missing piece in our DP state. Is there an efficient way to inculcate this information in our DP state? Note that since each number can occur only once, we only need the information whether a number has already been used or not. We can use a bitmask to represent this information!

- What is bitmask? An array of binary numbers represented as an integer in the programming language. An int32 is a binary array of size 32, int64 an array of size 64 and so on. Some common operations on bitmasks are
  - Check if i^{th} bit is set in mask m
    - (m & (1<<i) != 0)
  - Set i^{th} bit from right in mask m.
    - m = m | (1<<i)

So, we define our DP state to be f(i, j, msk) which denotes number of ways to make a sum j by giving face values to first i dice, where msk denotes what are the possible face values(if k^{th} bit is set, we can't use k).

A very simple implementation (without using memoization):

```
def f(i, j, msk):
        if i < 0 or j < 0: return 0
        ret = 0
        for bit = 1 to A:
                if msk & (1<< bit) == 0:
                        ret += f(i - 1, j - bit, msk | (1 << bit))
        return ret
```

Practice problems:

You should try to follow the approach described above. Don't worry too much about the complexity, but first come up with any DP solution and then you can try to optimise it further.

1. Given a number n, count the number of ways in which you can draw n chords in a circle with 2n distinct points such that no two chords intersect. Two ways are different if there exists a chord which is present in one way and not in the other.
2. You are given two positive integers n and m. For all permutations of [1, 2, …, n], we create a BST by inserting the values in a BST from left to right. How many of these have height of m.
3. There is a horizontal rod of length n. There are m points on this rod at a distance of a_1, a_2, …, a_m from left. You have to cut rod at all these points which can be performed in any order. Cost of making a cut at a point is the length of sub-rod in which it lies. Your aim is to minimise this cost.

Spoilers: Basic solution ideas

1. If we draw a chord between a pair of points, can you observe that we make two partitions A and B, where a chord from a point in A to a point in B cannot be drawn. Iterate over the sizes of these partitions to combine subproblems.
2. Does the actual values being inserted in a BST matter? Can we just deal with number of elements? Define f(n, m) as the number of permutations of m elements which when inserted into BSTs they generate BSTs of height exactly m. Iterate over the value of the root and try to combine subproblems.
3. Let's say first cut you make is at jth point, can you observe you're breaking your problem into two disjoint parts; ordering of cuts in set of points [0 j-1] would be irrelevant of ordering of set [j+1, m].
    a. Define your DP as f(i, j) denoting minimum cost to make cuts if rod starts at i^{th} point and ends at j^{th} point.
    b. Now, iterate over the first cut you're going to make to combine sub-problems.

Problems available online

Some more problems which you can code and submit on online judges. Links:

1. Link
2. Link
3. Link

Feedback/Questions?

Submit them via this form and we will try and address them in a livestream that we will host in early August (exact TBD). We will notify you of the livestream session via email.