Prerequisites:

- Basic Syntax
- Asymptotic Analysis (Time Complexities)

Overview

Dynamic programming is a wide category of algorithms and techniques. It is commonly defined as "a structure or algorithm which uses the answer to smaller subproblems to solve a larger problem".

Mindset

Dynamic programming is a broad category of problems, similar to geometry in mathematics and stoichiometry in chemistry. There is no single template for dynamic programming problems, whether easy or hard. Nonetheless, when solving dynamic programming problems, there are key concepts to keep in mind. However, simply knowing these strategies is not always enough to solve more difficult problems.

Initiation

https://wcipeg.com/problem/ccc00s4
Read and attempt the problem for 15 minutes, or until it is solved.

Solution:
Dynamic programming problems often involve a decision-making process which can be applied in real life. Consider the following idea:

Define `clubs[32]` as an array of club lengths.
In this problem, one can store an array *`dis`*, where *`dis`*`[i]` stores the minimum number of strokes to reach a certain distance. Initially, `dis[0]` is equal to 0. The remaining cells of the array can be completed by trying all clubs in `clubs[]`. Denote one such cell as *`dis`*`[i]`. One can calculate *`dis`*`[i]` by finding the minimum value of `dis[i-clubs[j]]+1` for all values of j.

Why does this work? Think of the algorithm as a logical process, one that

you could attempt in real life. It functions by attempting to use each club, at each distance. This type of program can be defined as an “exhaustive search”, as it tries all possible situations to find the optimal one. Many other algorithms are defined as “exhaustive search”, including recursive backtracking and shortest-path problems.

Generalization

The above solution may appear very specific and isolated to a single problem. However, dynamic programming problems, however difficult, can be solved with a similar general method.

1. **Identify the state -** Determine what data your program uses in order to define and solve subproblems. In the above problem, the *state* is the distance.
2. **Determine the start and end states -** Each DP problem starts with some data and ends with some data. The above problem starts at distance 0 and ends at distance D(provided in the input).
3. **Choose a possible transition -** The *transition* is the method used to convert information from smaller states into larger states. This is often the most challenging part of a dynamic programming problem. In the aforementioned "Golf" problem, this transition is attempting to use each of the clubs.
4. **Optimize your algorithm -** Not all dynamic programming problems require this step, and those that do are often quite difficult. This topic is often attributed as category separate from regular dynamic programming and will be the main topic of a future lesson. However, optimization is a useful tool to understand. Optimization is the use of an algorithm, data structure, observation or restriction to allow a program to run more efficiently. This is often the next step if your program receives a "Time Limit Exceeded" verdict.

Implementation tips (READ AFTER SOLVING EXAMPLE)

- Start with your state and by defining your array. Be careful not to go out of bounds. It is also recommended to add a small number to your array bounds, e.g. make an array of size 405 for n <= 400, in case of off-by-one errors. Example: making an array of size 5280 in the above problem may lead to runtime error or wrong answer, as looping to (0-index) 5280 will go out of bounds.
- Make sure to initialize your array if it is necessary and that you are initializing it to the correct value. Set the array to 0 if you are calculating the sum, 2^{30} if you are calculating the minimum, 1 if you are calculating the product, etc.
- When you have determined a suitable transition, but are unsure of how to implement it, simulate it by hand.

- If optimization is necessary, try to see if there are any states that are never visited, or if a transition can be improved by precalculating information.

Optional Practice

https://wcipeg.com/problem/cchange
https://wcipeg.com/problem/dwitefeb06p2
These problems will be repeated in the training section at the bottom of the document. Do these problems before moving on, unless you are confident in your dynamic programming ability (especially implementation).

Recursive transitions

Dynamic programming can use a variety of complex and interesting transitions. Although the following example is still quite simple, it demonstrates an important technique in implementing dynamic programming: using recursion to simplify transitions.

https://wcipeg.com/problem/ccc08s5
Read and attempt the problem for 15 minutes or until it is solved.

Solution:

In convention with the dynamic programming process, recognize that the state will be stored as [A][B][C][D], or the quantities of each type of particle. This has a complexity of O(N^4), well within the memory and time constraints. The start and end state is clear: the end state is one which is unable to form a reaction, and the start state is the one given in the input.

The transition relies on a key (and somewhat trivial) principle of game theory: it is possible to win from a state if and only if it is possible to move to a state where it is impossible to win. Take a moment to think about this idea if you are uncertain of why it is true.

Using that idea, it is simple to create a transition: for any state, try the 5 combinations listed in the input, if any of them are losing, set the current state to winning, otherwise, set the current state to losing. However, the implementation for this may seem complicated. Fortunately, it can be made clearer using recursion.

```
boolean [32][32][32][32] DP, visited
function f(a, b, c, d):
  if visited[a][b][c][d]:
    return dp[a][b][c][d]
  visited[a][b][c][d] = true
  for i in combinations:
    if possible(a, b, c, d, i) and
       not f(a - i.a, b - i.b, c - i.c,d - i.d):
         dp[a][b][c][d] = true
  return dp[a][b][c][d]
```

This function solves the above problem in a very effective manner. Additionally, it is faster than the iterative (loop-wise) method, since it does not visit as many states. Being able to implement dynamic programming using recursion is an important skill.

Optional Practice

https://wcipeg.com/problem/ioi9611
https://wcipeg.com/problem/ccc16s4
These problems will be repeated in the training section at the bottom of the document. Solving these problems now is highly recommended, especially the first one.

Practice problems (There are a lot :O)

It is extremely important to practice dynamic programming regularly. Try to solve as many of the problems as possible. Problems are sorted in roughly ascending difficulty.

Note: Try to finish all problems before the space before next class.

https://wcipeg.com/problem/dwitefeb06p2
https://wcipeg.com/problem/cchange
https://wcipeg.com/problem/ccc08s5
https://wcipeg.com/problem/ccc07s5
https://wcipeg.com/problem/ioi9611
https://wcipeg.com/problem/coci065p5

https://dmoj.ca/problem/year2017p6
https://dmoj.ca/problem/ccc16s4
https://dmoj.ca/problem/dmopc14c2p5
https://wcipeg.com/problem/ioi0914
https://wcipeg.com/problem/ccc16s4
https://wcipeg.com/problem/mockccc15s4
https://dmoj.ca/problem/ncco2d1p1
https://dmoj.ca/problem/tle16c8p3