1 Introduction

Dynamic programming was introduced by Richard Bellman in the 1950s as a technique for performing computational mathematical optimization. In the problem he considered, he considered *states*, in each of which certain *decisions* could be taken leading to different states. A *policy* is a mapping of states to decisions, and policies can be ascribed a cost (value).

To optimize policies, Bellman relied on the Principle of Optimality:[1]

An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.

It is a description of a recursion – a policy is optimal if, regardless of the past, its current decision leads to an optimal policy in the future. We shall think about dynamic programming similarly. We ascribe states (subproblems) and certain decisions (transitions) and solve the problem recursively, obtaining an (optimal) solution to each subproblem. An solution to each subproblem will, by the Principle of Optimality, lead us to a solution to the whole problem.

In computer science, we mostly consider dynamic programming to be a paradigm for constructing algorithms – not just in optimization, but also sometimes as part of data structure design or in combinatorics. Curiously, the same Bellman had his impact on the field of *reinforcement learning* (as part of machine learning) through **Bellman's equation**, which describes.

The name *dynamic programming* has nothing to do with the technique itself. In his autobiography, Bellman explains that the choice mostly had to do with his research environment. Funding for mathematical research at the time was difficult to come by – especially as Bellman's work was financed by the US military. Thus he came up with using *programming* in the name, as it sounded better than *planning*. *Dynamic* was prepended since, as he conjectured, the term cannot be used in a negative manner.

2 Technique

As a running example, we will use the *maximum independent subsequence sum problem*. You are given a sequence a of n numbers. You can pick any *independent* subsequence, meaning no two elements may be adjacent in the array. The goal is to maximise the sum of the chosen subsequence.

```
[$10]   $1   $2   [$9]   $15   [$9]   =>   10+9+9 = 28
```

We will construct the dynamic programming solution starting from the brute force solution. The naïve approach is to consider all possible subsequences (considered as subsets of indices $\{0, 1, \dots, n-1\} = [n]$), but predicating only on independent subsequences. The independence predicate $\mathcal{I}$ is given by:

$$\mathcal{I}(S) \equiv \forall i.\, i \notin S \lor i+1 \notin S$$

The problem is asking us to compute the optimal subsequence S^*:

$$\mathcal{S}^* = \underset{\forall S \subseteq [n].\, \mathcal{I}(S)}{\operatorname{argmax}} \sum_{i \in S} a_i$$

We can see that this can be solved in $\mathcal{O}(2^n)$ by considering all possible S. We can do much better!

2.1 This sentence has 27 characters. Finding transitions

A key trick in dynamic programming is describing a problem in terms of itself (or, possibly, a more general version). We seek to find a recursion (policy in the Principle of Optimality) that describes the optimal solution to the whole problem in terms of subproblems.

In this case, a common trick with (sub)sequences is to consider whether the **first** (or last) element is a part of the optimal solution. In case the first element a_0 is part of the solution, we know that a_1 cannot be to satisfy independence. Otherwise, a_1 can be part of the solution. Intuitively, on our example, we use the 10 and solve the rest of the problem skipping the 1, or just skip the 10:

$$\operatorname{solve}([10,1,2,9,15,9]) = \begin{cases} 10 \oplus \operatorname{solve}([2,9,15,9]) \\ \operatorname{solve}([1,2,9,15,9]) \end{cases}$$

[1]Richard Bellman: The Theory of Dynamic Programming – I don't recommend learning DP algorithms from this :)

Let's denote $[a:b] = \{a, \ldots, b\}$ ($[n] = [0:n]$). More formally, we can write two possible cases (selecting the one maximising $\sum_{i \in S} a_i$):

$$\begin{cases} 0 \in S \implies \{0\} \cup \boxed{\operatorname*{argmax}_{\forall S \subseteq [2:n].\mathcal{I}(S)} \sum_{i \in S} a_i} \\ 0 \notin S \implies \boxed{\operatorname*{argmax}_{\forall S \subseteq [1:n].\mathcal{I}(S)} \sum_{i \in S} a_i} \end{cases}$$

Note that this recursion eventually *terminates* (base cases included) – it is well-founded, as we always get to a "smaller" subproblem.

2.2 Abstracting states

Actually, the *subproblems* in the two $\boxed{\text{cases}}$ we are seeing are **similar**. They are both of the form

$$\mathcal{S}^*(p) = \operatorname*{argmax}_{\forall S \subseteq [p:n].\mathcal{I}(S)} \sum_{i \in S} a_i$$

for some index p. In fact, these *subproblems* correspond to the optimal subsequences for a suffix of a starting at index $p+1$. The solution to the entire problem is $\mathcal{S}^*(0)$. Then the two recursive cases – our dynamic programming algorithm's *transitions* – are:

$$\mathcal{S}^*(p) = \max_S \begin{cases} \{p\} \cup \mathcal{S}^*(p+2) \\ \mathcal{S}^*(p+1) \end{cases} \quad \text{by } \sum_{i \in S} a_i$$

Lest we forget the base case – at some point we run out of elements in the array:

$$\mathcal{S}^*(n) = \mathcal{S}^*(n+1) = \varnothing$$

What he have done is we **abstracted** the space of subproblems (states), forgetting the **concrete** set of indices we are allowed to use. Instead, we noticed it is sufficient to remember only what suffix we are still allowed to use. Thus, we reduced the problem to considering $\mathcal{O}(n)$ subproblems, with 2 cases each.

2.3 Abstracting solutions

However, there is still excess concrete information in our algorithm. To optimize the total sum of the solution, we need not keep track of the set of indices we want to include as our solution, but simply the sum (value) of the partial solution. I'll define:

$$s^*(p) = \sum_{i \in \mathcal{S}^*(p)} a_i$$

and bask in the beautiful simplicity of the final algorithm:

$$s^*(p) = \max\{a_p + s^*(p+2), s^*(p+1)\}$$

with the base cases following similarly. Instead of keeping track of the partial solutions, we abstract them to only keep track of some result/value.

2.4 Recovering solutions; Policies

Our abstractions within the dynamic program all correspond to the original problem.

States	Set of (general) subproblems
Results	Solutions to state subproblems
Transitions	Decisions/actions in state subproblems

In fact, our DP is generally *well-behaved*, and in many cases it is sufficient to recover the original full solutions. Suppose I have computed all the values of $s^*(p)$ as defined above. Since we're solving the entire problem, I shall look at the value $s^*(0)$. Then one of the following two cases holds:

$$S^* = \begin{cases} s^*(0) = a_0 + s^*(2), & a_0 \in S^*, \text{ proceed from } s^*(2) \\ s^*(0) = s^*(1), & a_0 \notin S^*, \text{ proceed from } s^*(1) \end{cases}$$

The same principle applies recursively, leading to recovering the optimal solution S^* in full.

Nondeterministic choices (where both cases apply) lead to the space of all optimal solutions, which are not necessarily unique (e.g. consider $[1, 1, 1, 1]$).