Chapter 8

Dynamic Programming

Repeated subproblems refers to situations where the same subproblem state can result from different partial solutions. For example, in the knapsack problem, the subproblem state is defined only by the remaining capacity in the pack and the set of items left to consider — exactly which items have already been chosen for the pack doesn't matter, just their total weight. Two different subsets of items which have the same total weight would result in repeated subproblems.

(Cases where the same subproblem state arises because the same partial solution is arrived at by making the same choices in a different order are addressed in section 7.1.)

With repeated subproblems, work can be saved by *memoization* — one a friend has solved a subproblem, save the result so that if the subproblem arises again, the result can be looked up instead of recomputing it. If there are enough repeated subproblems, the running time can be reduced to polynomial or pseudopolynomial time.

Dynamic programming is an algorithmic paradigm which applies memoization to a recursive backtracking formulation. In both dynamic programming and recursive backtracking, the solution is constructed by making a series of decisions; it each step, the alternatives for the current decision are considered and friends are asked to solve the subproblems resulting from each choice. The difference between dynamic programming and recursive backtracking is in how the subproblems are parameterized and enumerated — recursive backtracking uses a depth-first search of the *solution space*, enumerating all possible series of decisions, while dynamic programming iterates through the *subproblem states*, enumerating all possible subproblem states.

8.1 Does Dynamic Programming Work?

Any series-of-choices formulation, whether is it greedy, recursive backtracking, or dynamic programming, requires that the problem must have the *optimal substructure* property — an optimal (or legal) solution can be built from optimal (or legal) solutions of subproblems.

While dynamic programming can in theory be applied to any series-of-choices formulation, it is only advantageous if the number of subproblem states is significantly smaller than the full solution space (i.e. there are many repeated subproblems) and, due to the space required for memoization, is only practical if the number of subproblem states is polynomial or pseudopolynomial.

8.2 16 (Really 15) Steps to a Dynamic Programming Algorithm

Recursive backtracking and dynamic programming are really just two ways to implement a recursively-defined series-of-choices algorithm. Use the recursive backtracking development process (section 6.3), replacing the Algorithm step with a modified Implementation step:

12. *Algorithm.* Skip this.

15. *Implementation.*

(a) *Memoization.*
Identify how to parameterize subproblem state for efficient lookup, typically in an array.

(b) *Order of computation.*
Loop(s) can be used to fill in the array; identify whether the loop(s) go from small large index values or vice versa.

(c) *Dynamic programming.*
Combine the memoization and order of computation with the base and main cases and the top level.

8.3 Example

Refer to section 6.4 where a recursive backtracking solution for the knapsack problem is developed.

Implementation.

Memoization. Identify how to parameterize subproblem state for efficient lookup, typically in an array.

> Subproblems are defined by the set S' of items left to consider and the remaining capacity W' of the pack.
>
> The order items are considered in doesn't matter, so S' can be represented with an array of all of the items in S and an index k — S' would be the elements $S[k..n-1]$.
>
> If the pack's capacity W and the weights w_i are all integers, W' will be integer and can be used directly as an array index.
>
> We will use `V[k][Wp]` to store the highest value possible from the items `S[k..n]` and pack capacity `Wp`. The initial subproblem is `V[0][W]`.

Order of computation. Loop(s) can be used to fill in the array; identify whether the loop(s) go from small large index values or vice versa.

> The base cases are when there no items are left to consider and when there's no more room in the pack — $k = n - 1$ and small Wp.

Dynamic programming. Combine the memoization and order of computation with the base and main cases and the top level.

> `knapsack(S,W)` — find the highest-value subset of items in S whose total weight does not exceed the capacity W of the knapsack
>
> Input: set S of n items, each with a value $v_i > 0$ and weight $w_i > 0$; knapsack capacity $W > 0$
>
> Output: a subset of the items
>
> Legal solution: a subset of items with total weight $\leq W$
>
> Optimization goal: maximize total value of items taken

```
// initialize the base cases - k=n
for w = 0 to W do
  V[n-1][w] = 0

// fill in the rest of the array
for k = n-2 downto 0
  for w = 1..W
    if w_k <= w  // item k fits in the pack - consider both take and don't take
      V[k][w] = max(V[k+1][w-w_k]+v_k,V[k+1][w])
    else         // item k doesn't fit in the pack - consider don't take only
      V[k][w] = V[k+1][w]

return the set of items for V[0][W]
```

Time and space. Assess the running time and space requirements of the algorithm, providing sufficient implementation details (data structures, etc) to justify those answers.

> The array has nW slots and computing each value is O(1), so the total running time is O(nW). The space requirements are also O(nW).

This is *pseudopolynomial* — W doesn't depend on n, but it also isn't a constant factor that can just be eliminated. However, O(nW) it is still a vast improvement over O(2^n) in most cases, though space requirements become an issue if W is large.

8.4 Reconstructing the Solution

Only the value of the solution is stored, not the particular decisions that gave rise to that solution. The series of decisions can be reconstructed by working backwards from the initial subproblem — in each case, the solution for a subproblem is a combination of a particular choice and the solution to the subproblem resulting from that choice. Look at the solution value for each of those alternatives and compare it to the value for the current subproblem to determine which alternative was chosen, then continue with that subproblem.

For example, consider the knapsack problem (section 8.3). The main case is

```
if w_k <= w     // item k fits in the pack - consider both take and don't take
  V[k][w] = max(V[k+1][w-w_k]+v_k,V[k+1][w])
else            // item k doesn't fit in the pack - consider don't take only
  V[k][w] = V[k+1][w]
```

Start with the initial subproblem `V[0][W]`. If $w_0 \leq W$, either `V[0][W] = V[1][W-`w_0`]` (if item 0 is part of the solution) or `V[0][W] = V[1][W]` (if not). Compare `V[0][W]`, `V[1][W-`w_0`]`, and `V[1][W]`, add item 0 to the solution or not accordingly, and repeat the process with the next subproblem. (If $w_0 > W$, `V[0][W] = V[1][W]` — item 0 is not part of the solution.) Continue until a base case is reached.

The running time is $O(bh)$ — the branching factor b is how many alternatives have to be considered for each choice, and the length of the longest solution path h is the number of decisions made. While not $O(1)$, it is much more efficient than filling in the array in the first place so the running time is generally not an issue. An alternative would be to store the alternative chosen along with the solution value in the array, decreasing the running time to $O(h)$ but increasing the space required.