Chapter 9

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 8.4.)

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. There is also a shift in thinking about the output from the subproblem task — in recursive backtracking, the output may be the whole solution or just the solution to the subproblem; in dynamic programming, the output is just the solution to the subproblem.

9.1 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 where more than one alternative needs to be considered for each choice, 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.

9.2 Elements of Dynamic Programming Algorithms

9.2.1 Subproblems

For backtracking algorithms, the subproblem output can be defined to be a complete solution (including both the solution-in-progress and the solution for just the subproblem) or the solution for just the subproblem. For dynamic programming, the subproblem output is the solution for just the subproblem. It also does not include which alternative is chosen for each decision as that can be reconstructed later (section 9.2.3).

9.2.2 Memoization

The core of dynamic programming is *memoization*, the caching of subproblem solutions in order to avoid redundant computation — if the same subproblem is encountered again the previously-computed result can be returned instead of recomputed. This requires being able to look up the solution for a particular subproblem, with arrays providing $\Theta(1)$ lookup times, so a key aspect of memorization is determining how to parameterize the subproblem so that a given subproblem can be quickly mapped to a slot in an array.

9.2.3 Algorithm

The core of a dynamic programming algorithm is to solve the initial subproblem by filling in the memoization array. While dynamic programming algorithms can be implemented recursively, they typically are not. Instead, loop(s) are used to fill in the array, saving on the overhead of subroutine calls and having to check for each subproblem whether there is stored solution available.

For each loop, it is necessary to determine the order of computation — whether the loop goes from small index values to large or vice versa. This is based on where the base cases are stored in the array — fill those in first, then order the loops to fill in the rest of the array to go from the base cases to larger subproblems.

The memoization array stores only the value of the solution and not the specific alternative for each choice that led to that solution. The choices made 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.

9.2.4 Termination

The fill-in-the-array loops are counting loops and the array is finite, so all of the slots will eventually be filled and the loops will end. Reconstructing the solution involves one repetition per step in the solution path, which is also finite.

9.2.5 Time and Space

The number of slots in an array is the product of the dimensions of the array — the length for a 1D array, the number of rows times the number of columns for a 2D array, etc.

The total space required for the array is the number of slots in the array times the space required per slot (typically $\Theta(1)$).

The running time is dominated by the time to fill in the array, which is the number of slots in the array (the number of unique subproblems) times the work required for each slot. The work required per slot depends on the formulation of the problem but is at least $\Omega(b)$ where b (the branching factor) is the number of alternatives for the choice.

If it is not $\Theta(1)$ to determine the array slot for a given subproblem, that needs to be factored in as well.

Reconstructing the solution 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 $O(1)$, this is always much more efficient than filling in the array in the first place so its 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 $\mathrm{O}(h)$ but increasing the space required.

9.3 How to Design Dynamic Programming Algorithms

Recursive backtracking and dynamic programming are really just two ways to implement a recursively-defined series-of-choices algorithm, and so the initial steps of identifying the avenues of attack and formulating the recursive definition are the same.

The full process for dynamic programming algorithms is as follows:

Establish the problem. Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*
  State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, identify and distinguish between legal solutions and optimal ones.
- *Examples.*
  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack. Determine the algorithmic approach(es) to try.

- *Targets.*
  Backtracking algorithms are fundamentally exponential; with dynamic programming we hope to reduce the problem to polynomial or pseudopolynomial, but that is not always possible.
- *Paradigms and patterns.*
  Consider the patterns defined in section 8.1.
- *The series of choices.*
  Identify the series of choices (section 6.4.1). Take into account efficiency considerations when choosing between alternatives.

Define the algorithm. Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*
  The size of the problem is the number of choices left to make. The smallest problem size is 0 (a complete solution).
- *Generalize / define subproblems.*
  Identify several elements:
  - *Partial solution.*
    What constitutes a solution-so-far? This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).
  - *Alternatives.*
    What are the legal alternatives for the next choice?
  - *Subproblem.*
    Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified. Take into account efficiency considerations when defining the input and output.

    For dynamic programming, the subproblem's output is just the solution for the subproblem (not a complete solution).

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

- *Base case(s).*
  A complete solution has been found — which means that the solution to this subproblem is an empty solution. Identify which slot(s) of the memoization array will store the base case solutions, and what value(s) will be stored for each.

- *Main case.*
  Address how to solve a typical large problem instance (one that is not a base case). Express this in terms of an assignment statement — identify which slot of the memoization array corresponds to the current subproblem and give an expression for how to compute its value based on other slots of the array.

- *Top level.*
  The top level puts the context around the recursion.

  - *Initial subproblem.*
    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem — and identify what slot of the memoization array will hold that subproblem's solution.
  - *Setup.*
    Whatever must happen before the initial subproblem is solved.
  - *Wrapup.*
    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

- *Algorithm.*
  Assemble the algorithm from the previous steps and state it.

  Bring together the base case(s), main case, and top level with any handling needed for special cases — there shouldn't be new elements here, though this will be the first time that the loops to fill in the memoization array are written out. Identify the initial subproblem but it is not necessary to write out the process for reconstructing the solution itself as this is a standard process.

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.* Show that the recursion — and thus the algorithm — always terminates.

- *Correctness.* Show that the algorithm is correct.

  - *Establish the base case(s).*
    Explain why the solution is correct for each base case.
  - *Show the main case.*
    Explain why the desired solution will be found (all possible legal alternatives for the next choice are either considered or are safe to skip) and showing that the correct partial solutions and subproblems are handed to the friends.
  - *Final answer.*
    Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem. For the initial subproblem, establish that the correct input is provided – the (empty) solution-in-progress is legal and the rest of the problem is actually the whole problem.

Determine efficiency. Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.
- *Time and space.*
  Assess the running time and space requirements of the algorithm given the implementation identified.
- *Room for improvement.*
  The primary element in both time and space is the number of slots in the memoization array — can that be reduced? A second important element in the running time is the branching factor — can that be reduced? Also consider any elements that aren't $\Theta(1)$, such as the space required for each subproblem solution, the time it takes to determine which array slot corresponds to a subproblem, or the work it takes to generate the subproblems.