A Guide to Dynamic Programming

Jason D. Hartline

First version: February 17, 2017;
This version: April 9, 2017.

These notes describe how to construct dynamic programs and how to ensure that they are correct. Special focus is given for how to avoid common errors.

Dynamic programming is one of the main approaches for developing novel polynomial time algorithms. It is an algorithmic methodology that breaks an optimization problem into subproblems. Crucially, these subproblems (a) have a succinct specifications and (b) there is no manipulation of global state that could change the solution to the subproblems. Succinct specifications are important because the dynamic program calls for solving all subproblems. Succinctness of their specification implies that there are not that many subproblems to solve. A key step in specify a dynamic program, is expressing the solution to a subproblem in terms of the solution to smaller subproblems.

The specification of the solution to one subproblem in terms of the solution to smaller subproblems is an example of recursion. It is useful to understand the computation that is occurring in dynamic programming by the structure of this recursion. For example, there is a directed graph defined with vertices corresponding to subproblems and directed edges from a subproblem to its subproblems. Because each subproblem's subproblems are smaller than it, this directed graph is acyclic. There is generally overlap in the subproblems, a subproblem is generally one of the subproblems of more than one subproblem. In other words, in this graph of subproblems a vertex generally has multiple incoming edges. (Otherwise the graph would be a tree and could be solved with divide and conquer.) For this reason, dynamic programming problems should not be solved by simple recursion because simple recursion will inadvertently recompute the solution to each subproblem every time it is needed. To avoid this recomputation of the solution to subproblems the solutions to each subproblem are stored in what is called a *memoization table*. For example, one way to then solve the problem is to use recursion where first the memoization table is consulted for the answer to a subproblem, and only if the answer is not already in the table is the answer calculated (recursively; and this answer is then stored in the table for future use). These notes describe instead a simpler iterative method for filling in the memoization table and from which the answer to the original optimization problem can be easily determined.

In dynamic programming, it is customary to separate (a) the problem of computing the value of the optimal solution from (b) the problem of computing the solution itself. Section 1, below, describes how the value of the optimal solution can be computed from the values of optimal solutions to subproblems. These optimal values are stored in the memoization table. To compute the optimal solution, it then suffices to traverse the memoization table again, using the values of the optimal solution of subproblems guide the decision of which solution to pick. In Section 3, this algorithm is described. The basic algorithm for traversing the memoization table tends not to vary much across different applications of dynamic programs.

1 A Framework for Dynamic Programming

The *integral knapsack problem* will be a running example. In the integral knapsack problem there is a knapsack with integral capacity C and n objects $\{1, \ldots, n\}$. Each object i has a integral size s_i and a value v_i. The objective of the problem is to select a feasible subset $S \subset \{1, \ldots, n\}$, i.e., with $\sum_{i \in S} s_i \leq C$, of the n objects that has the highest total value $\sum_{i \in S} v_i$.

Part I: Identify the Subproblems

The first step of dynamic programming is to identify a collection of subproblems by describing them in English. For the integral knapsack problem, a collection of subproblems is:

$\mathrm{OPT}(i, D) =$

> The maximum total value of a subset of objects $\{1, \ldots, i\}$ that fits in a knapsack with capacity D.

Notice that in this definition of the subproblems all of the parameters, in this case i and D, are completely defined in English in the statement of the subproblem. Specifically i is the maximum index of the objects considered in the subproblem and D is the capacity of the knapsack for the subproblem. Checking that each parameter of the subproblem is clearly defined in the English statement of the subproblem helps to ensure that the dynamic program is correct.

The dynamic program must find the solution to every subproblem, thus the number of subproblems is a lower bound on the runtime of the dynamic program. Notice that the ranges of the parameters of the subproblems given above are small. Specifically, the range for i is $\{0, 1, \ldots, n\}$, a total of $n + 1$ possibilities, and the range for D is $\{0, 1, \ldots, C\}$, a total of $C + 1$ possibilities. In total there are $O(nC)$ subproblems given all possible combinations of i and D. Here notice that $i = 0$ corresponds to the subproblem of finding the optimal knapsack from the empty set of objects. Checking that each parameter only takes a small number of values helps to ensure that the runtime of the dynamic program is good.

Rubric I. a. *The parameters of the subproblem are explicitly mentioned and given context in the English description of the subproblem.*

Rubric I. b. *The subproblem is unambiguously described. It is clear, from the input to the problem and the English description to the subproblem, exactly what the subproblem is.*

Example Answers:

1. This description of the subproblems does not define parameter D:

   $\text{OPT}(i, D) =$ "The maximum total value of a subset of objects $\{1, \ldots, i\}$ that fits in the knapsack."

2. This description of the subproblem is ambiguous. Specifically, it is not clear which i of the n objects it refers to.

   $\text{OPT}(i) =$ "The maximum total value of a subset of i objects that fits in the knapsack."

3. This description of the subproblem, though it does not lead to a correct dynamic program, has its parameters defined and is unambiguous. Therefore, it is correct with respect to this rubric.

   $\text{OPT}(i) =$ "The maximum total value of a subset of objects $\{1, \ldots, i\}$ that fits in the knapsack."

4. There are multiple correct descriptions of subproblems. This description of the subproblem is different from the one above, but it is also correct.

   $\text{OPT}(i, D) =$ "The maximum total value of a subset of the remaining objects that fit in the remaining capacity of the knapsack, after putting a subset of objects $\{1, \ldots, i\}$ with total size D in the knapsack."

Part II: Express Solution to Subproblem as a Recurrence

The second step of dynamic programming is identifying a recurrence that solves any subproblem (that is not a base case) in terms of the solution to "smaller" subproblems. For example, in the integral knapsack problem, smaller subproblems than $\text{OPT}(i, D)$ are given by $\text{OPT}(i', D')$ with $i' \leq i$ and $D' \leq D$ except for $(i', D') = (i, D)$. It is important that the recurrence works towards smaller subproblems because the original problem will be solved by working upwards from the smallest subproblems, starting from the base cases, towards larger problems and eventually the original problem.

One way to think about expressing the solution to a subproblem in terms of its subproblems, i.e., via a recurrence, is to identify a decision that could be evaluated with the answer to the subproblems. In the case of the integral knapsack problem and the subproblem $\mathrm{OPT}(i, D)$, a decision that can be evaluated is whether object i is in the optimal knapsack with capacity D or not. Notice that if i is in the optimal knapsack with capacity D then the remaining set of objects must be the optimal subset of objects $\{1, \ldots, i-1\}$ that fits in a knapsack with capacity $D - s_i$ (assuming i fits in the capacity D, i.e., $s_i \leq D$). The total value of this knapsack is $v_i + \mathrm{OPT}(i-1, D-s_i)$. On the other hand, if i is not in the knapsack then the total value of the knapsack is $\mathrm{OPT}(i-1, D)$. Notice that determining which outcome of the decision is better can be easily assessed from the solutions to smaller subproblems (smaller, in this case, because the subset of objects is a smaller subset). As these are the only two possibilities for object i, we can express the value of the subproblem as follows:

If $s_i \leq D$:
    $\mathrm{OPT}(i, D) = \max\{v_i + \mathrm{OPT}(i-1, D-s_i), \mathrm{OPT}(i-1, D)\}$
else ($s_i > D$):
    $\mathrm{OPT}(i, D) = \mathrm{OPT}(i-1, D)$.

Justification: The optimal knapsack with objects $\{1, \ldots, i\}$ and capacity D either includes or does not include object i. The value of the optimal knapsack in these cases is $v_i + \mathrm{OPT}(i-1, D-s_i)$ or $\mathrm{OPT}(i-1, D)$, respectively. If object i is too big for the remaining capacity, then only the latter case is possible.

Correctness analysis of a dynamic program is by induction on the recurrence (and base cases). The *inductive hypothesis* is that previous subproblems have been correctly solved. The *inductive step* is arguing that the recurrence gives a correct solution to a subproblem from correct solutions to its subproblems. A dynamic program is correct if its base cases are correct and the recurrence correctly solves all remaining cases. The justification for the recurrence is its proof of correctness.

Note that object values and sizes as well as the remaining capacity of the knapsack are critical in determining the optimal value of the knapsack. A simple sanity check which all recurrences must satisfy is that these critical quantities are referenced in the recurrence. In the example, the value v_i and size s_i of object i and the remaining capacity D are referenced.

Rubric II. a. *The recurrence specifies the solution to every subproblem (except the base cases defined in IV) in terms of* smaller *subproblems.*

Rubric II. b. *The recurrence is correct given the description of the subproblem in English. The informal argument of the recurrence's correctness is correct.*

Example Answers:

1. This recurrence is not correct when $s_i > D$:

$$\mathrm{OPT}(i, D) = \max\{v_i + \mathrm{OPT}(i-1, D-s_i), \mathrm{OPT}(i-1, D)\}.$$

2. This recurrence, which is based on the incorrect definition of the subproblem of Example Answer 3 in Part I, is incorrect as it does not ensure that the selected objects fit in the knapsack (both the object sizes and the remaining capacity of the knapsack are ignored).

   $$\mathrm{OPT}(i) = \max(v_i + \mathrm{OPT}(i-1), \mathrm{OPT}(i-1)).$$

3. This recurrence, which is based on the incorrect definition of the subproblem of Example Answer 3 in Part I, is incorrect as it maintains global state that is not part of the parameters of the subproblems.

   $$\begin{aligned}&V = OPT(i-1)\\&C = C - s_i\\&V' = OPT(i-1)\\&OPT(i) = \max(V, V').\end{aligned}$$

Part III: Solve Original Problem from Subproblem Solutions

The third step of dynamic programming is to show how the original problem can be easily solved from the subproblems. Quite often the original problem is one of the subproblems itself. Other Times there is a simple calculation based on one or more subproblems that gives the solution to the original problem. For example, the optimal value of the original integral knapsack problems is one of the subproblems.

$$\boxed{\text{Optimal Integral Knapsack} = \mathrm{OPT}(n, C).}$$

When the solution to the original problem is one of the subproblems, a simple check to make sure that the solution is correct is to substitute the specific parameters of this subproblem into the description of the subproblem in English to make sure that the solution is correct. In the case above, for example, $\mathrm{OPT}(n, C)$ is "the total value of a subset of objects $\{1, \ldots, n\}$ that fits in a knapsack with capacity C" which is identical to the original problem.

Rubric III. *The solution to the original problem is correct given the description of the subproblem in English.*

Example Answers:

1. This solution to the original problem does not specify the correct parameters.

   $$\text{Optimal Integral Knapsack} = \mathrm{OPT}(1, C).$$

2. This solution to the original problem does not parameterize the subproblems correctly.

Optimal Integral Knapsack = OPT(C).

3. This solution to the original problem, though it is based on subproblems that do not lead to a correct dynamic program, is correct with respect to the incorrect subproblem. Therefore, it is correct with respect to this rubric. For the subproblem specified in Example Answer 3 in Part I, the following solution to the original problem is "correct":

   Optimal Integral Knapsack = OPT(n).

Part IV: Identify and Set the Base Cases

The fourth step of dynamic programming is identifying and setting the base cases. Typically the base cases are all the subproblems that can be solved without even looking at the input. In the case of the integral knapsack problem the base cases are the subproblems where there are no objects (specifically, $i = 0$). In these subproblems the optimal knapsack also has no objects and has no total value.

$$\boxed{\text{OPT}(0, D) = 0 \text{ (for all } D \in \{0, \ldots, C\}).}$$

Rubric IV. *The base cases are correct and complete. Correctness requires that the value set for the base cases is the correct value given the definition of the subproblem. Completeness requires that all of the base cases that are necessary for the recurrence are set.*

Example Answers:

1. This base case is correct but incomplete.

   $\text{OPT}(0, 0) = 0.$

2. This base case is complete but incorrect.

   $\text{OPT}(0, D) = \infty$ for all $D \in \{0, \ldots, C\}$.

3. This is the correct and complete base case for the subproblem specified in Example Answer 3 in Part I. Though this subproblem specification does not lead to a correct dynamic program, this base case satisfies this element of the rubric.

   $\text{OPT}(0) = 0.$

Part V: Iterative Dynamic Program

The fifth step of dynamic programming is combining the recurrence and the base case – which specify a recursive computation of the solution to the optimization problem – into an iterative dynamic program. This iterative implementation combines the previous parts:

1. Initialize the memoization table with the base case (from Part IV).
2. Iterate over the elements in the table filling them in with the formula from the recurrence (from Part II).
3. Output the value of the original program (from Part III).

The only part of this task that is not explicitly specified previously is the order in which elements of the table are to be filled in. Specifically, at the time an entry is filled in, all the entries corresponding to all subproblem that this entry depends on must also be filled in. To see the proper order to fill in the table, it is helpful visualize the space of subproblem and identify the direction of the dependencies between the subproblems. Intuitively the dependencies are in the direction of the size of the subproblem where small subproblems should be solved before the large subproblems.

For the integral knapsack problem the subproblems are given by a two-dimensional array indexed by i (rows) and D (columns). The base case is given by $i = 0$ (for all D), i.e., the top row. The recurrence says that $\text{OPT}(i, D)$ for $i \geq 1$ (in row i) can be calculated from two subproblems from row $i - 1$. Thus, the iterative dynamic program must fill in the memoization table from top rows (small i) to bottom rows (large i).

```
Integral Knapsack Dynamic Program:

  0. Input: $\{v_1, \dots, v_n\}$, $\{s_1, \dots, s_n\}$, $C$.
  1. Initialize $\text{OPT}[0, D] = 0$ for $D \in \{0 \dots, C\}$.
  2. For $i$ from 1 to $n$:
       For $D \in \{0, \dots, C\}$:
         Set $\text{OPT}[i, D] =$
           $\begin{cases} \max(v_i + \text{OPT}[i-1, D-s_i], \text{OPT}[i-1, D]) & \text{if } s_i \leq D \\ \text{OPT}[i-1, D] & \text{otherwise.} \end{cases}$
  3. Output: $\text{OPT}[n, C]$.
```

Notice that in the outer loop (over i), it is crucial that i is ordered from smallest to largest. On the other hand, in the inner loop (over D), the order is unimportant.

Rubric V. a. *The iterative dynamic program correctly initializes the base case (from IV), evaluates the recurrence and stores its results in a memoization table (from II), and outputs the value of the original problem (from III).*

Rubric V. b. *The iterative dynamic program fills in the memoization table in a correct order (from smaller subproblems to larger).*

Part VI: Runtime Analysis

The runtime of a dynamic program is the sum of the runtimes for initialization (preprocessing), iteratively filling in the memoization table from the recurrence, and calculating the value for the original problem from the memoization table (postprocessing). The runtimes of pre- and postprocessing are often dominated by the runtime of the iteration. One common exception, however, is when the preprocessing step includes sorting the input, with runtime $O(n \log n)$, while the iterative part of the dynamic program has runtime $O(n)$. For such a dynamic program the runtime is $O(n \log n)$.

The runtime from iteratively filling the dynamic programming table can be calculated by multiplying the number of subproblems, i.e., the size of the dynamic programming table, by the runtime it takes to compute any subproblem, i.e., any entry in the table, from its subproblems (assuming its subproblems are already solved).

For the integral knapsack problem, with n objects and knapsack capacity C, the size of the table is $O(nC)$ and the runtime for solving any subproblem from its subproblems is constant, i.e., $O(1)$.

The total runtime is $O(nC)$. Breakdown:

- number of subproblems: $O(nC)$.
- runtime per subproblem: $O(1)$.
- preprocessing: $O(C)$.
- postprocessing: $O(1)$.

Rubric VI. a. *The runtime analysis correctly identifies the size of the memoization table.*

Rubric VI. b. *The runtime analysis correctly identifies the runtime per subproblem.*

Rubric VI. c. *The runtime analysis is correct (the product of the size of the memoization table and the runtime per subproblem) plus any pre- and post-processing (e.g., sorting).*

Common mistakes include not including the preprocessing runtime when it exceeds the runtime of the iterative step and improperly calculating the number of subproblems or runtime required to solve each subproblem.

Part VII: Implementation

Typically dynamic programs can be very easily implemented, and implementing it is a good way to check that it is correct. The integral knapsack dynamic program can be implemented in Python as follows.

```
def integral_knapsack(v,s,C):
    assert len(v)==len(s),"v and s must be same length lists"
    n = len(v)

    OPT = {}
    for D in range(C+1):
        OPT[-1,D] = 0

    for i in range(n):
        for D in range(C+1):
            OPT[i,D] = max(v[i] + OPT[i-1,D-s[i]],OPT[i-1,D])\
                if s[i] <= D else OPT[i-1,D]

    return OPT[n-1,C]

print [integral_knapsack([1,3,4,3,2,10],[5,1,1,3,2,0],C)
          for C in range(6)]

# output: [10, 14, 17, 17, 19, 20]
```

Notes:

- In Python indices for arrays start at 0, hence relative to the discussion above the indices for elements are shifted down by one, i.e., the range for i in the memoization table is $\{-1, \ldots, n-1\}$ with the base case being $i = -1$.
- In Python the builtin `range(n)` gives the list `[0,...,n-1]`.
- The memoization table `OPT` is implemented as a dictionary which maps tuples `(i,D)` to the memoized value of the subproblem $\text{OPT}(i, D)$. In Python, `OPT[i,D]` is the same as `OPT[(i,D)]`.

Rubric VII. a. *The implementation is correct (with respect to the iterative dynamic program specified in V).*

Rubric VII. b. *The implementation has been demonstrated on a reasonable set of test cases. If there are errors in the dynamic program, these errors are evident in the test cases.*

2 Dynamic Programming: Summary of Rubric

Rubric I. a. *The parameters of the subproblem are explicitly mentioned and given context in the English description of the subproblem.*

Rubric I. b. *The subproblem is unambiguously described. It is clear, from the input to the problem and the English description to the subproblem, exactly what the subproblem is.*

Rubric II. a. *The recurrence specifies the solution to every subproblem (except the base cases defined in IV) in terms of* smaller *subproblems.*

Rubric II. b. *The recurrence is correct given the description of the subproblem in English. The informal argument of the recurrence's correctness is correct.*

Rubric III. *The solution to the original problem is correct given the description of the subproblem in English.*

Rubric IV. *The base cases are correct and complete. Correctness requires that the value set for the base cases is the correct value given the definition of the subproblem. Completeness requires that all of the base cases that are necessary for the recurrence are set.*

Rubric V. a. *The iterative dynamic program correctly initializes the base case (from IV), evaluates the recurrence and stores its results in a memoization table (from II), and outputs the value of the original problem (from III).*

Rubric V. b. *The iterative dynamic program fills in the memoization table in a correct order (from smaller subproblems to larger).*

Rubric VI. a. *The runtime analysis correctly identifies the size of the memoization table.*

Rubric VI. b. *The runtime analysis correctly identifies the runtime per subproblem.*

Rubric VI. c. *The runtime analysis is correct (the product of the size of the memoization table and the runtime per subproblem) plus any pre- and post-processing (e.g., sorting).*

Rubric VII. a. *The implementation is correct (with respect to the iterative dynamic program specified in V).*

Rubric VII. b. *The implementation has been demonstrated on a reasonable set of test cases. If there are errors in the dynamic program, these errors are evident in the test cases.*

3 Computing the Optimal Solution

We have focused, thus far, on computing the value of the optimal solution by dynamic programming. The optimal solution itself can be computed from traversing the memoization table and using the stored optimal values of subproblems to build up the solution from optimal decisions. The following algorithm, given the memoization table OPT$[\cdot,\cdot]$, identifies the objects in the optimal integral knapsack.

```
Integral Knapsack Solution Algorithm:

  0. Input: $C$, $\{s_1, \ldots, s_n\}$, memoization table OPT$[\cdot,\cdot]$.

  1. $D = C; K = \emptyset$.

  2. For $i = 1$ to $n$:

         If OPT$[i, D] >$ OPT$[i-1, D]$:
         (a) $K = K \cup \{i\}$.
         (b) $D = D - s_i$.

  3. Output: $K$.
```