CS/ECE 374 A (Spring 2024)
Tips on Dynamic Programming

Dynamic programming (DP) is a very useful algorithm design technique that can be applied to numerous problems. However, some students find the technique challenging to apply. The best way to learn is to (i) attend the lectures and re-watch the lecture recordings (in addition to reading the relevant chapter in Jeff's book), and (ii) practice doing more problems (e.g., see the labs and past HWs). Here are some supplementary notes and review that you may find helpful (assuming that you are already familiar with the lecture material and examples from class and the labs).

Follow the steps!

As you know, a typical DP solution involves the following steps:

1. First define subproblems.
2. Then derive a recursive formula to solve your subproblems (including base cases), with justification.
3. Specify a valid evaluation order.
4. Give pseudocode to evaluate your recursive formula bottom-up (using tables, and loops instead of recursion).
5. Analyze the run time.
6. And if we explicitly ask for it, give pseudocode to output an optimal solution rather than just the optimal value.

Steps 4 and 6 are supposed to be straightforward and can be done automatically, without thinking (see below), assuming that you have done steps 1–3 properly. So, we sometimes allow you to omit step 4 and/or 6—this will be specified in the question statement. But if we do ask you to include them in a homework/exam, we are really giving you opportunity for some easy points—don't miss these easy points, and make sure you know how to do them.

Do not skip step 1 (unless we give away the precise definition of subproblems in the question). If you are reading some outside sources (e.g., wikipedia) on specific dynamic programming algorithms, beware that they sometimes jump directly to pseudocode (step 4). Don't do that! The goal of this course is to teach you to *design* algorithms on your own when faced with a new problem, not just follow or implement an existing algorithm, and steps 1–2 are actually the crucial steps for coming up with a new DP algorithm.

Designing algorithms requires experience and creativity, and a certain amount of trial-and-error[1] For DP, the creative part actually lies in step 1. If the subproblems are set up properly in step 1, your formula in step 2 will be determined from your step 1—you should be able to "derive" the formula. (Step 1 has to be done first: you can't randomly come up with a recursive formula

[1]This is not our fault! There is no systematic procedure for designing algorithms that we can teach you in general. Many "meta-problems" about algorithms (or, equivalently, Turing machines) are provably *undecidable*, as we will find out near the end of the course...

without stating clearly what it is supposed to compute in the first place. A different definition of subproblems in step 1 will lead to an entirely different formula in step 2.) If you do not succeed in deriving a recursive formula in step 2, try again and go back to step 1 with a different class of subproblems (e.g., by adding another parameter etc.).

Types of problems

DP is typically applied to *optimization problems*, i.e., problems of the form

> find (some solution) which **minimizes/maximizes** (some objective function)
> **subject to** (some constraints)

E.g., in the longest increasing subsequence (LIS) problem (Problem 1 from Lab 7b), a solution is a subsequence of the input sequence, the objective function is its length, and the constraint is that the subsequence is increasing.

Some standard terminologies that you should be aware of: A *feasible solution* refers to any solution that satisfies the stated constraints. An *optimal solution* refers to a feasible solution with the minimum/maximum objective function value. The *optimal value* refers to the value of an optimal solution (we sometimes use *cost* instead of *value* when dealing with minimization problems).

DP is occasionally also applied to decision problems, or problems about finding a feasible solution without any objective function, e.g., the subset sum problem (3/7's lecture) or the shuffle problem (Past HW6).

Before tackling a new problem with DP, it is useful to identify some known examples that the problem is similar to, and use those examples as a guide (that's why we have covered many different types of examples in class and in the labs!). Here are some common types of problems that DP is often applied to:

(A) Finding a subsequence of an input sequence. E.g., LIS and other problems from Lab 7b, or HW6.1. Note that HW6.2 also fits this category: after sorting the points by x-coordinates, the input/solution can be viewed as a sequence/subsequence.

(B) Dividing an input sequence into multiple *contiguous* subsequences or "blocks". E.g., the palindrome partitioning problem from 3/5's lecture. Note that this type of problem can be re-interpreted to fit type (A), since the block boundaries themselves can be viewed as a subsequence.

(C) Problems involving two (or more) sequences as input. E.g., the LCS problem from 2/29's lecture, or the shuffle problem from Past HW6.

(D) Splitting an input sequence into two (or more) (non-contiguous) subsequences.

(E) Problems about numbers similar to subset sum or knapsack (3/7's lecture).

(F) Problems involving a tree as input. E.g., the independent set problem for trees from 3/7's lecture, or Old.7.2 from Past HW7.

(G) Problems where the solution (not the input) forms a tree-like structure (note that this is different from (F)). E.g., the optimal binary search tree problem from Jeff's book, the problem from Lab 8b (where a solution is a parenthesized expression, which corresponds to a tree), or Old.7.1.

1 Definition of subproblems

Typically, we introduce some input parameters (the number of parameters corresponds to the dimension of the table), and define a class of subproblems like the following:

> Given parameters $i, j, \ldots$, **define** $F(i, j, \ldots)$ to be (the optimal value) **over** (some restricted subset of the input) **subject to** (some modification of the original constraints, dependent on the parameters $i, j, \ldots$, possibly with some extra constraint).

The definition should be mathematically precisely written (and it should not be a recursive formula, which will come later).

E.g., in the second solution to LIS from Lab 7b, we use a one-parameter class of subproblems:

> Given parameter $i \in \{1, \ldots, n\}$, **define** $LIS(i)$ to be the length of the longest subsequence **over** the suffix $A[i], \ldots, A[n]$ **subject to** the constraint that the subsequence is increasing **and the extra constraint** that the subsequence begins at $A[i]$.

On the other hand, in the first solution from Lab 7b, we use a two-parameter class of subproblems:

> Given parameters $i, j \in \{1, \ldots, n\}$ with $i < j$, **define** $LIS(i, j)$ to be the length of the longest subsequence **over** the suffix $A[j], \ldots, A[n]$ **subject to** the constraint that the subsequence is increasing **and the extra constraint** that all of its elements are larger than $A[i]$.

(For decision problems without any objective function, the output of a subproblem should instead be a Boolean, i.e., we define $F(i, j, \ldots)$ to be true iff there exists (a solution satisfying some modification of the original constraints, and possibly some extra constraint).)

Remember to indicate how the answer to the original problem can be derived from the answers to these subproblems (sometimes it is just the answer to one "final" subproblem, sometimes it is the max/min/or of the answers of some subset of these subproblems).

You need a sufficient number of parameters in order for a recursive formula to be possible. On the other hand, having too many parameters (i.e., too many subproblems) might increase the run time and space.

Important Note 1 *When defining your subproblems, the output of the function (the answer) should always be an optimal value rather than an optimal solution itself.*

In our experience, attempts to have the function output the solution directly often lead to incorrect recursive formulas, incorrect analysis, or slower run time. Besides, recovering the solution can always be done later, in step 6, which as mentioned is straightforward.

Some general tips (which may be a little vague in places, but hopefully still useful):

- For problems about finding subsequences (type (A)), a standard trick is to restrict to a *prefix* or *suffix* of the input (as in the LIS example above, or in HW6.1). Such a prefix or suffix can be specified by one index parameter. Some prefer prefixes (which requires working "backward"), and others prefer suffixes (which requires working "forward")—whichever way suits your thinking better.

- If the problem has additional parameters and extra constraints involving those parameters, these parameters may need to be incorporated into the subproblem definition. E.g., HW6.2 has an additional size restriction (the parameter k), and so we end up with a two-parameter class of subproblems there.

- If the problem involves two (or more) sequences (type (C)) like LCS, we may need two (or more) index parameters for different prefixes (or suffixes) of those sequences.

- For problems like subset sum or knapsack (type (E)), we may need a parameter for the target integer (leading to so-called "pseudo-polynomial" algorithms only).

- For problems where the input is a tree (type (F)), a standard trick is to use a tree node v as a parameter and restrict the input to the subtree rooted at v.

- For problems of type (G), we often need to restrict to a contiguous block of the input rather than a prefix or suffix, and so use 2 parameters for the start and end indices.

- Sometimes we may need a flag as an extra parameter. E.g., in HW6.1, we have a parameter with a constant number of possibilities $\{\texttt{//}, \texttt{\\\\}, \texttt{/\\}, \texttt{\\/}\}$. In the first solution to Problem Old.7.2, we have a parameter with a Yes/No flag (or equivalently two functions, one for Yes and one for No).

- How many parameters do we need, and why do we need them? Let's consider a problem about subsequences (type (A)), and let's say we are using an approach based on suffixes. One potentially helpful perspective is to think of parameters as the things we need to "remember" when building up a solution as we examine the input from left to right. Thus, subproblems loosely corresponds to "states", if we draw analogy with DFAs. E.g., for the longest convex subsequence problem from Lab 7b, we need to "remember" the past 2 elements, in order to check convexity when the next element is chosen (in contrast to LIS where we only need to remember 1 past element to check the increasing property). In HW6.1, we need to remember two characters in $\{\texttt{/}, \texttt{\\}\}$ to know if the next character causes the pattern `/\/` or `\/\` to occur.

- After learning graph algorithms (the next topic in the course), you may go back to DP with another useful perspective: many DP applications (with the main exceptions of type (F) or (G) problems) actually correspond to finding paths in some directed acyclic graph. Subproblems then correspond to vertices in the constructed graph.

2 Recursive formula

Generally, the recursive formula is obtained by considering all possibilities for the "next" choice in the solution:

- We first pretend that we are already given what is the right choice ("a little birdie told me"), and then try to find a formula relating the answer of the given subproblem in terms of the answer of other subproblems—usually we just need to figure out how the parameters change in the subproblem (e.g., some indices may decrement or increment), and also how the output value changes (e.g., add or subtract something to the output value).

- But we don't know the right choice beforehand. So, we just take the min/max of the formula over all feasible choices. (In decision problems like subset sum, it will be "or" ($\vee$) instead of min/max.)

Because we are taking min/max over all feasible choices at each iteration, correctness is usually not difficult to justify (unlike greedy). And as we are taking min/max only over the "next" choice rather than the entire solution, we potentially avoid exponential time.

Sometimes, we take min/max over two or a constant number of choices at each iteration. E.g., in the first solution to LIS in Lab 7b, there are two choices: not using $A[j]$, or using $A[j]$. This leads to a max of two terms: $\mathit{LIS}(i,j) = \max\{\mathit{LIS}(i,j+1),\ \mathit{LIS}(j,j+1)+1\}$. This assumes $A[j] > A[i]$. In the other case when $A[i] \geq A[j]$, the second choice is no longer feasible, and so the formula degenerates to the max of just one term (i.e., we just have the first term); see the lab solutions.

Sometimes, we take min/max over a large number of choices. E.g., in the second solution to LIS in Lab 7b, the choices correspond to possible indices j for the second element $A[j]$ of the subsequence: $\mathit{LIS}(i) = \max\limits_{j\in\{i+1,\ldots,n\}:\ A[j]>A[i]} \mathit{LIS}(j)+1$.

Your formula should be written in correct math syntax. Some quick words about math notation: max of two terms is obviously written as

$$\max\big\{\textsf{(something)},\ \textsf{(something)}\big\},$$

but max of multiple terms over a running variable j is written as

$$\max_{j\in\textsf{(some range)}:\ \textsf{(some condition)}} \textsf{(something)}.$$

This is analogous to summing $\sum$ or integrating $\int$ over a running variable. Min is similar. For "or" over a running variable, use $\bigvee$.

Alternatively, the latter expression may be written as max over a set:

$$\max\big\{\textsf{(something)} :\ j \in \textsf{(some range)},\ \textsf{(some condition)}\big\}.$$

Either way is acceptable, whichever you prefer. (In set notation, some people use the vertical bar | instead of colon.)

By standard convention, max over an empty set is $-\infty$, min over an empty set is ∞, and $\bigvee$ over an empty set is false; this will be convenient to know.

In math, a formula involving multiple cases is usually written like the following:

$$F(\ldots) = \begin{cases} \textsf{(something)} & \text{if } \textsf{(case 1)} \\ \textsf{(something)} & \text{if } \textsf{(case 2)} \\ \quad\vdots & \end{cases}$$

(You may find LaTeX convenient for typesetting such expressions. You can typeset it differently so long as it is easy to read, and mathematically precise.)

Important Note 2 *The right-hand side of your recursive formula should solely depend on the input parameters of your subproblem (and the original input), and should never refer to any part of a subproblem's optimal solution, other than its optimal value.*[2]

[2] Of course, when justifying or deriving the formula, we may do a case analysis over what the optimal solution may look like. But the final formula itself should not refer to the innards of a solution.

Bad Example 1: for the longest convex subsequence problem in Lab 7b, if you define $LCS(i)$ as the length of the longest convex subsequence of the suffix $A[i], \ldots, A[n]$ under the extra constraint that the first element chosen is $A[i]$, the following recursive formula is fundamentally wrong:

$$LCS(i) \;=\; \max\{LCS(j)+1:\; j>i,\; A[i] + (\text{the 2nd element of } LCS(j)) > 2A[j]\} \qquad \text{WRONG!}$$

First, $LCS(j)$ is a value, not a subsequence. Second, when you need to refer to the 2nd element of the subsequence corresponding to $LCS(j)$, you really should be considering some modified optimization problem with an extra constraint on the 2nd element.[3] If you find yourself in this situation, you should go back to step 1 and try adding an extra parameter to the definition of subproblems.

Bad Example 2: for the first solution to the LIS problem in Lab 7b, the following is not a valid recursive formula:

$$LIS(i,j) = \begin{cases} LIS(i,j+1) & \text{if } A[i] \text{ is not part of the LIS} \\ LIS(j,j+1)+1 & \text{if } A[j] > A[i] \text{ and } A[i] \text{ is part of the LIS} \end{cases} \qquad \text{WRONG!}$$

The recursive formula itself should not refer to the optimal solution (we don't know the LIS, so how would we know which case we are in?!). The actual formula should be $LIS(i,j) = \max\{LIS(i,j+1),\; LIS(j,j+1)+1\}$ if $A[j] > A[i]$, and $LIS(i,j) = LIS(i,j+1)$ else. (The division into cases, whether $A[i]$ is part of the LIS or not, should go in the justification, but in not the final formula.)

Bad Example 3: for the second solution to the LIS problem in Lab 7b, the following is wrong:

$$LIS(i) \;=\; LIS(j)+1 \quad \text{where } j \text{ is the smallest index in } \{i+1,\ldots,n\} \text{ such that } A[j] > A[i] \qquad \text{WRONG!}$$

Here, the recursive formula is mathematically well-defined, but it is not really DP, but is actually greedy! We are not taking max over several choices, but are committing to one particular (greedy) choice of j. Greedy algorithms in general may not always find an optimal solution, and if they happen to be correct (by chance), nontrivial correctness proofs are required. (In this example, the above greedy algorithm is wrong, as there are counterexamples.)

More general tips:

- For problems about finding subsequences (type (A)), if you are using suffixes indexed by i (working "forward"), formulas typically have the form $F(i,\ldots) = \max\left\{F(i+1,\ldots)+(\mathsf{something}),\ \ldots\right\}$ or $F(i,\ldots) = \max_{j\in\{i+1,\ldots,n\}:\ \ldots} (F(j,\ldots)+(\mathsf{something}))$, or with min instead of max; the choices are often about the first or second element in the optimal solution for $F(i,\ldots)$. If you are using prefixes (working "backward"), $F(i,\ldots) = \max\left\{F(i-1,\ldots)+(\mathsf{something}),\ \ldots\right\}$ or $F(i,\ldots) = \max_{j\in\{1,\ldots,i-1\}:\ \ldots} (F(j,\ldots)+(\mathsf{something}))$; the choices are often about the last or second-to-last element in the optimal solution for $F(i,\ldots)$.
- For problems with tree-like solutions (type (G)), we often see formulas of the form $F(i,j,\ldots) = \max_{k\in\{i+1,\ldots,j-1\}} (F(i,k,\ldots)+F(k,j,\ldots)+(\mathsf{something}))$ (or variants with k replaced by $k\pm 1$).

[3]Note the difference between (i) checking whether the optimal solution to an optimization problem satisfies a certain condition, vs. (ii) solving an optimization problem under the extra constraint that the solution must satisfy this condition. Adding an extra constraint changes the nature of the optimization problem. (To draw an analogy, checking whether the top CS374 student is an Aries is different from finding the top CS374 student among those who are Aries.)

- For those who find the perspective of directed acyclic graphs helpful: as mentioned earlier, subproblems correspond to vertices, and the formula for a subproblem usually is obtained by taking min/max over all out-neighbors (if we are moving "forward") or all in-neighbors (if we are moving "backward") of the corresponding vertex in the constructed graph.

Remember to include base cases. Even though we usually write the base cases before the main recursive formula, a general advice is to actually derive the main recursive formula first and fill in base cases afterwards. The base cases are all the boundary or edge cases not covered by your recursive formula. Watch out for out-of-bound indices etc. Sometimes, sentinel values (e.g., setting $A[0] = \pm\infty$ or $A[n+1] = \pm\infty$, or setting default values regarding out-of-bound indices) might simplify the treatment of some of the base cases.

3 Evaluation order

A valid evaluation order is typically easy to spot.

- It is usually just "increasing order of i" or "decreasing order of i" for one of the parameters i. Sometimes, it may be something like "increasing order of i, and in case of ties, increasing order of j".
- Generally, you examine the right-hand side of your recursive formula, and see which of the parameters always decreases, or always increases.
- For type (A) problems, if you are working with prefixes, it tends to be increasing order of the index associated with the prefixes. If you are working with suffixes, it tends to be decreasing order of that index.
- For problems with trees as input (type (F)), evaluating according to a *postorder* traversal of the tree would often work.
- For type (G) problems with a recursive formula of the form $F(i,j,\ldots) = \max_{k\in\{i+1,\ldots,j-1\}}(F(i,k,\ldots) + F(k,j,\ldots) + (\mathsf{something}))$, "increasing order of $j-i$" would work, since the subproblem size roughly corresponds to the difference $j-i$ (although alternatively we can also do "increasing order of j, and in case of ties, decreasing order of i").

4 Pseudocode

This part should be straightforward, after you have done step 1–3.

- It is mostly cutting-and-pasting your recursive formula, with some parentheses replaced by square brackets (i.e., recursive function calls replaced by table lookups), and adding some for-loops based on your evaluation order.
- Remember to include the computation of the base cases, and the final answer at the end.
- When the formula involves min/max over a large number of terms, we sometimes explicitly write out an inner loop to compute the min/max for each table entry. (This may not be truly necessary when writing pseudocode rather than actual code, but it may make run time analysis clearer later.)

(There is an alternative way to implement DP that does not use loops, but instead uses recursion together with a table to avoid duplicate recursive calls. You don't need to know this alternative approach for homework/exams.)

(In more advanced DP algorithms, some extra preprocessing steps or the use of data structures may help speed up run time.)

5 Run time analysis

This part is usually easy, especially if you have written out your pseudocode, since you should already know how to analyze pseudocode that involves only for-loops (as you know, we pay attention to how the loops are nested).

Even if step 4 is omitted, an upper bound on the run time can be determined straightforwardly by just multiplying (i) the total number of subproblems (i.e., the total size of the table) with (ii) the run time to evaluate the formula for one subproblem (i.e., the time needed to compute each table entry). The number in (i) is the product of the number of possible values of each parameter. The cost of (ii) is $O(1)$ if the formula is the min/max over a constant number of terms and each term can be computed in constant time, but it may be larger e.g. if we are taking the min/max over a large number of terms.

(Sometimes, a tighter bound on the run time can be obtained by analyzing a summation.)

If we ask for analysis of space, this should be easy: it is usually just the number in (i) (i.e., the size of the table).

6 Pseudocode for outputting the solution

This part is also straightforward. E.g., see the palindrome partitioning problem from 3/5's lecture, or Old.6.1(b) from Past HW6.

- Usually it is helpful to modify the pseudocode from step 4 to compute an additional table called the *predecessor table*, which stores an index to the term that yields the min/max for each entry of the DP table. (This may help lower the run time of step 6, especially if the recursive formula involves min/max over a large number of terms. For formulas where we take min/max over a constant number of terms, this extra predecessor table may not be essential.)

- To output the solution, we just use backtracking. (The reason backtracking does not create exponential blow-up here is that we know at each iteration which choice leads to the optimal solution, by looking up the predecessor table.) We recommend you imitate the pseudocode in the abovementioned examples from class/past HWs and write out the pseudocode *using recursion* for this part (although alternatively one could also write it without recursion using a loop).

- The run time for this part is usually linear in the sum of the dimensions of the table, and is thus subsumed by the run time of the main pseudocode in step 4.