CSE 3101 Design and Analysis of Algorithms
Meta Steps for Unit 5

Jeff Edmonds

This contains the **most important** concepts in this unit. You will not be able to pass this course without knowing and understanding these. The steps provided **must** be followed on all assignments and tests in this course. Do **not** believe that because you know the material, you can answer the questions in your own way. Though this material is necessary, it does not contain everything that you need. You must read the book, go to class, review the slides, and ask lots of question.

Chapters 17,18,&19: Dynamic Programming Algorithms

When providing a dynamic programming algorithm and its proof of correctness, the following are all the paragraphs that should be included, their headings, and what they should contain. (See HTA pg 268 for the full description of the steps)

1) Specifications: This is likely part of the question and hence does not need to be written. However, before writing anything consider: What is the set of instances, for each instance what is its set of valid solutions, and for each solution what is its cost/value.

Algorithm using Trusted Bird and Friend: I have my instance I. The little bird knows an optimal solution $optS$ to it.

2) Question for Bird: I ask the little bird a little question about it. Choose something like one of the following:

- Is the last object from the instance in the solution?
- What is the last object in the solution?
- The solution forms a tree. What is the object at the root?

3) Possible Answers from Bird: Define the list of possible answers that she may give. (Enumerate them with $k \in [K]$). We want the number K of these answers to be small.

4) Trust Her: Assume that the little gives me the answer indexed by k. Trust her. What does this tell you about the solution $optS$ to your instance. More importantly, what does it tell the parts of the solution that she did not tell you?

5) Constructing Subinstances: The bird gave me some of the solution $optS$ I am looking for. I want my recursive friend to give me the rest of it. However, I can only ask him a smaller instance to my same computational problem. What instance $subI$ should I give my friend so that he will give me what I want. Again, we don't micro manage him, but trust him. He gives me an optimal solution $optSubSol$ for his instance $subI$ and its cost.

6) Constructing a Solution for My Instance: I produce an optimal solution $optSol[k]$ for my instance I from the bird's answer k and the friend's solution $optSubSol$. If my little bird happens to be trust worthy, then this is an optimal solution. But even if not, this will be the best solution for my instance from amongst those consistent with the k^{th} bird answer.

7) Costs of Solution: Similarly, I compute the cost $optCost[k]$ of our solution $optSol[k]$.

Recursive Back Tracing Algorithm:

8) Best of the Best: I can trust the friend because he is a recursive version of myself. Not actually having a little bird, I try all her answers and take best of best.

9) Base Cases: An instance that is so small that there are no smaller instances which can be given to a friend is considered to be a base case. The base case instances and their solutions need to be considered.

Dynamic Programming Algorithm:

10) The Set of Subinstances: I imagine running the above recursive backtracking algorithm on my instance I. Determine the complete set S of subinstances $subI$ ever given to me, my friends, their friends. Note that this set S needs to 1) contain my instance I; 2) be closed under this "sub"-operator; 3) all (or at least most) of these subinstances should be needed.

11) Construct a Table Indexed by Subinstances: I index these subinstances with i (and maybe j) so that $subI[i,j]$ denotes a particular subinstance. Which subinstance $subI[i,j]$ denotes needs to be carefully described. I build a table indexed by these subinstances so that $optS[i,j]$ stores an optimal solution for instance $subI[i,j]$, $optCost[i,j]$ the cost of this solution, and $birdAdvice[i,j]$ the advice given by the bird on this subinstance. (Actually we don't store the solution because it is too big.)

12) The Order in which to Fill the Table: The order in which the friends must solve their subinstances must be determined. It must be an an order so that nobody has to wait, i.e. from smaller to larger instances.

13) Loop Over Subinstances: A key line in the dynamic programming algorithm is the iteration over these instances in this order. Let $subI[i,j]$ denote the one currently being worked on. Be clear what this instance is. The task now is to find an optimal solution for it and to store it and its cost in the table at $optS[i,j]$ and $optCost[i,j]$. This is done in the exact same way that it is above. The only difference is how the friends communicate.

14) Question for Bird: I have my instance $subI[i,j]$ and the little bird knows an optimal solution $optS[i,j]$ to it. I ask her a little question about it. A key line in the dynamic programming algorithm is the iteration over these bird answers indexed by $k \in [K]$. Be clear what the current answer is.

15) Constructing Subinstances: I have my instance $subI[i,j]$ and the little bird has given my his k^{th} answer. Trusting the bird, I consider what her answer tells me about the parts of the solution that she did not tell me. I design a subinstance $subI[i',j']$ to give my friend so that his solution will give me the parts of the solution that the little did not give me.

16) Communication Between Friends: The key difference between recursive backtracking and dynamic programming is how the friends communicate. When you as the recursive backtracking friend on instance I wants help from a friend on instance $subI$, you recurse and wait until he computes and returns a solution. In contrast, when you as the dynamic programming friend on instance $subI[i,j]$ want help from a friend on instance $subI[i',j']$, you assume that because his instance is smaller, he has already found an optimal solution for his instance and has stored it and its cost in the table at $optS[i',j']$ and $optCost[i',j']$. All you need to do is look it up. Similarly, when you have finally completed solving your instance you will store the solution and its cost in the table at $optS[i,j]$ and $optCost[i,j]$.

17) Constructing a Solution for My Instance: I produce a solution $optSol_{\langle\langle i,j\rangle,k\rangle}$ for my instance $subI[i,j]$ from the bird's answer k and the friend's solution $optSubSol$. This will be the best solution for my instance $subI[i,j]$ from amongst those consistent with the k^{th} bird answer.

18) Costs of Solution: Similarly, I compute the cost $optCost_{\langle\langle i,j\rangle,k\rangle}$ of my solution $optSol_{\langle\langle i,j\rangle,k\rangle}$.

19) Best of the Best: Once I have for each bird's answer k, the best solution $optSol_{\langle\langle i,j\rangle,k\rangle}$ for my instance $optS[i,j]$ from amongst those consistent with the k^{th} bird's answer, I take best of best. This will be an overall best solution. I store it and its cost in the table at $optS[i,j]$ and $optCost[i,j]$.

20) Solution Too Big: Actually, the solution $optS[i,j]$ is too big and hence it takes too much time and space storing and copying this from friend to friend. Hence, we comment out every line of code involving solutions. Instead, we store the cost of this solution in $optCost[i,j]$ and the advice given by the bird on this subinstance in $birdAdvice[i,j]$.

21) Base Cases: A recursive back tracking algorithm says, "If the instance I that I am personally given is so small that there are no smaller instances which can be given to a friend, then I must solve it myself and return its answer". **Do not** do this in a dynamic programming algorithm. Note that even if the end user never calls the algorithm on such a base case instance, a recursive program needs to handle these because it calls itself on these smaller and smaller instances, stopping at the base cases. In contrast, dynamic programming algorithms do not recurse. Hence, my instance is a base case only if the end user gives it. Despite this a dynamic programming algorithms must always solve a collection of base cases. Recall that S is defined to be the complete set of subinstances $subI$ ever given to me, my friends, their friends .... The table is indexed by these subinstances and each needs to be solved. In fact, the base cases, being the smallest of these are solved first. Start the algorithm by storing for each base case instance $subI[i,j] \in S$, its (commented out) optimal solution $optS[i,j]$, cost $optCost[i,j]$ of this solution, and the bird's advice $birdAdvice[i,j]$ into the table.

22) Code: Your code **must** have the following structure.

algorithm $DynamicProgrammingAlg\,(I)$

$\langle \boldsymbol{pre-cond} \rangle$**:** I is my instance.

$\langle \boldsymbol{post-cond} \rangle$**:** $optSol$ is an optimal solution for I and $optCost$ is it's cost.

begin

% Table: $subI[i,j]$ denotes the subinstance indexed by $\langle i,j \rangle$ (Describe).
$optSol[i,j]$ would store an optimal solution for it, but it is too big. Hence, we store only the bird's advice $birdAdvice[i,j]$ given for the subinstance and the cost $optCost[i,j]$ of an optimal solution.

$table[range_i, range_j]$ $optCost, birdAdvice$

% Base Cases: Describe the base cases and their solutions.
loop over base cases
% $optSol[basecases]$ = its solution
$optCost[basecases]$ = its cost
$birdAdvice[basecases]$ =?
end loop

% General Cases: Loop over subinstances in the table.
for $i \in [range_i]$
for $j \in [range_j]$
% Solve instance $subI[i,j]$ and fill in table entry $\langle i,j \rangle$.
% Try each possible bird answer.
for $k \in [K]$
% The bird and Friend Alg: see above
% $optSol_{\langle\langle i,j\rangle,k\rangle}$ = Describe how to construct the solution to our instance $subI[i,j]$ from the bird's advice k and the solution $optSol[friend]$ to our friends instance $subI[friend]$. This will be the best solution for our instance from amongst those consistent with the k^{th} bird answer.
$optCost_{\langle\langle i,j\rangle,k\rangle}$ = Describe how to construct the cost of this solution from the bird's advice k and the cost $optCost[friend]$ of our friends solution.
end for
% Having the best, $optSol_{\langle\langle i,j\rangle,k\rangle}$, for each bird's answer k, we keep the best of these best.
k_{min} = "a k that minimizes $optCost_{\langle\langle i,j\rangle,k\rangle}$"
% $optSol[i,j] = optSol_{\langle\langle i,j\rangle,k_{min}\rangle}$
$optCost[i,j] = optCost_{\langle\langle i,j\rangle,k_{min}\rangle}$
$birdAdvice[i,j] = k_{min}$
end for
$optSol = AlgWithAdvice\,(I, birdAdvice)$

return $\langle optSol, optCost[\text{initial instance}]\rangle$
end algorithm

23) Constructing the Solution: We do all of this work, but because we commented out the lines of code having to do with solutions, we do not in the end have the optimal solution for our initial instance I. However, all this work was not in vain. We now have the advice the little bird would give for each every subinstance in S. We then can run the recursive back tracking algorithm to obtain the optimal solution for I. This computation is now very fast because we do not need to try all the bird's answers. Jeff does not require you to include the code for this during your exam.

24) Running Time: Clearly state the number of subinstances in the table. Clearly state the number of bird answers per subinstance. The running time is the product of these.