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.

======

New method.

1. Given an instance, go through the game of life.
   What does Pooh Bear remember about his past so that he can continue?
   How do you index the states that he might be in?
   What actions might he take from where he is? (The bird tells him which to take.)
   Describe the nodes, the source node s, the sink node t, the edges, and the edge weights of the resulting graph.
   Give special thought to the edges into t.
   Explain why.
   Hint: Be sure that there is a 1-1 mapping between the valid solutions of the problem and the st-paths of your graph. (You do not need to discuss this.)
   Hint: Be sure to have as few edges as possible.

2. In order to make the code more esthetic, we turn the everything around.
   Give the bird-friend algorithm.

   - What is the set of subinstances solved? What is the input/output of each?
   - What question do I ask the bird?
   - What answers does she give?
   - For each such answer k, what subinstance do you give your friend and what does he give you?
   - And how do you form your answer $optSol_{\langle i,k\rangle}$?

3. Which subinstances are base cases?

4. Complete the following lines of the code:

```
    loop over the subinstances ... (give order)
        loop over the bird answers ...
            % $optSol_{\langle i,k\rangle}$ =
              $optCost_{\langle i,k\rangle}$ =
```

5. What is the running time of this algorithm?

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 $DynamicProgrammingAlg\,(I)$
$\langle$pre−cond$\rangle$: $I$ is my instance.
$\langle$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.

===

Reduction of any Dynamic Programming Algorithm to *LeveledGraph* (G, s, t): I have said at various times that most dynamic programming algorithms can be *reduced* to optimum s-t path in a leveled graph.

***LeveledGraph* (G, s, t):** We covered the problem *LeveledGraph* (G, s, t). Its input is $I_{LG} = \langle G, s, t\rangle$ where G is a weighted leveled graph and s and t are nodes. A solution S_{LG} is a path from s to t through G. The cost of a solution $cost_{LG}(S_{LG})$ is the sum of the weights of the edges in the given path. If we climb the mountain, the oracle $Oracle_{LG}$ will give us an optimal solution $Oracle_{LG}(I_{LG}) = S_{LG}$, for which we can compute its cost. This optimal could be that of minimum or of maximum cost depending on what is needed.

Algorithm for Problem $P(I_P)$ Consider some problem $P(I_P)$. Its input is I_P and its solution is S_P with cost $cost_P(S_P)$. Our goal is construct an algorithm Alg_P for problem P. Clearly it must take I_P as input. It maps this given instance I_P to instance I_{LG} using our program $I_{LG} = InstanceMap(I_P)$. Our algorithm then gives I_{LG} to $Oracle_{LG}(I_{LG})$ who gives us S_{LG}. Our algorithm then maps solution S_{LG} to solution S_P using our program $S_P = SolutionMap(S_{LG})$.

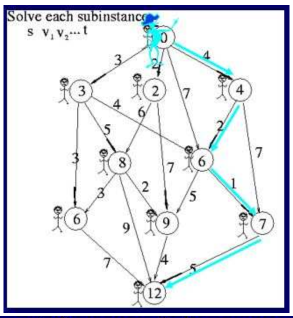
Correct: All you need to do to prove that your algorithm Alg_P works is to show that the solution map is a bijection $SolutionMap^{-1}(S_P) = S_{LG}$ and show that this bijection keeps the solutions ordered with respect to cost, namely $cost_P(S_P) \geq cost_P(S'_P)$ if an only if $cost_{LG}(S_{LG}) \geq cost_{LG}(S'_{LG})$. From this, we can conclude that Alg_P returns the optimal solution if and only if the oracle does.

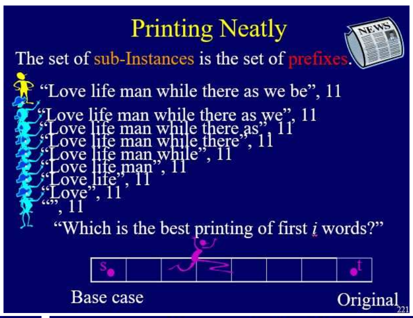
Hey recursion is like a reduction to the same problem - expect with smaller instances.

Hey we discussed how a dynamic programming algorithm with a non-linear global cost can suffered from the fact that the solution bijection did not keeps the solutions ordered with respect to cost.

Graph G: Algorithm Alg_P take its input I_P and maps it using $InstanceMap(I_P)$ to $I_{LG} = \langle G, s, t\rangle$. The oracle give us the optimal s-t path through this graph G. Lets try to understand this graph G in three different ways.

1. The nodes of G are the cells of the dynamic programming table. See the five examples below.
2. Take the tree of stack frames produced by a recursive back tracking algorithm and to merge some of its nodes. This gives you the graph G.
3. Look at life as following a path between the possible states that one might be in, i.e. a DFA automaton.





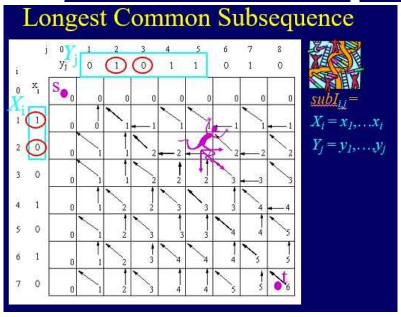
Printing Neatly

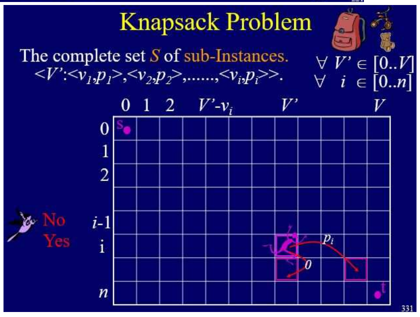
The set of sub-Instances is the set of prefixes.

"Love life man while there as we be", 11
"Love life man while there as we", 11
"Love life man while there as", 11
"Love life man while there", 11
"Love life man while", 11
"Love life man", 11
"Love life", 11
"Love", 11
"", 11

"Which is the best printing of first *i* words?"

s
t
Base case
Original





Nodes of G are Subinstances:

Recursive Backtracking: Recursion given an original instance I_P recurses on smaller subinstances which in turn recurses on smaller subinstances and so on. This is implemented by a tree of stack frames. In this tree, there is a node for every subinstance that is ever solved when starting at our original instance I_P. Recursive backtracking sometimes solves a given subinstance many many times. When this happens, dynamic programming reduces the total computation time by solving each subinstance only once. Imagine what happens to the recursion tree when you merge a set of its nodes into one. It become a DAG, i.e. a directed acyclic graph, i.e. a leveled graph, i.e. an instance to $LeveledGraph(G, s, t)$. Here node t, being our destination, will be our original instance I_P, i.e. the root of the tree of stack frames. We might have to add a new node s to this collapsed graph G to be for starting point and put an edge from it to every base case, i.e. to every node with no children. The oracle give us the optimal s-t path that winds back up though this newly formed graph G.

Subinstances/Nodes: Instead of constructing Alg_P for problem P, suppose instead we formed a dynamic programming algorithm for P. In this case, we imagine running the recursive backtracking algorithm and collecting this set of subinstances. Then we "index" a table with these subinstances. This is done by specifying each subinstance $subI[i,j]$ with one or two integers $\langle i,j\rangle$ and then indexing the table by these.

Edges: When construction this dynamic programming algorithm for P, focusing your attention on solving one of its subinstances $subI[i,j]$, imagine sitting the corresponding node of this graph. You ask your little bird to tell you some information about the solution. The recursive back tracking stack frame $subI[i,j]$ iteratively tries all the possible answers $k \in [K]$ that she might tell you. For each k tried, the stack frame recurses. Each time it recurses, the tree of stack frames has an edge from $subI[i,j]$ to the subinstance $subI[i',j']$ that you ask your friend. Hence, the bird's answer k effectively tells you which of the outgoing edges you should take. Following this edges gets you to your friend's subinstance $subI[i',j']$. In this way, the bird step by step provides you with the oracle's the optimal t-s path that starts at the root t and follows the bird's path down to the base case nodes and on to node s.

Connection to the *LeveledGraph* Algorithm: Recall that when solving *LeveledGraph* using dynamic programming, there is a subinstance for every node $subI[i,j]$ asking for the best path from s to this node $subI[i,j]$. The bird then tells you the last edge to take in this path, i.e. the edge from some $k = subI[i',j']$ to our last node $subI[i,j]$.

G as the Graph of Life:

Loop Invariants: Remember Jeff's religion about loop invariants. They are NOT actions but static pictures of what is true at this instance in time. Then the code/action within the loop takes you from one such assertion/invariant to another.

DFA Automata: If you took EECS2001 from Jeff, he had religion there too. Each state in an automata (DFA) also specifies one such point in time. Jeff said to label this state with every thing the automata knows to be true at this flash in time (everything Pooh bear writes on his finite sized black board.) If the number of states grows with the size of the input, then it is not a "finite" automata and if you have a little bird than it might not be a "deterministic" one, but the idea of a state will be the same. The edges between these states in this automata represent action that take it from one such state to another.

Path of Life: What is life but a path through this automata graph, i.e. a sequence of snapshot states and/or the sequence of actions/happening that move your from one state to another. Some but not all of these states will happen to you. What "cause/decides" which edge to follow next is the whole debate of free will vs determinism vs divine intervention. But but in an optimal world, for each state that you might be in, the little bird's advice tells you the last action k you took to get to this state.

Read the Input I_P and Produce the Solution S_P: As you follow a path from start state s to the destination state t, your over all task will be to read the input I_P and produce the solution S_P. As such, each edge $\langle state[i,j], state[i',j']\rangle$ our graph G needs to be associated/labeled with following information.

States: Clearly the edge is identified with the states $\langle state[i,j]$ and $state[i',j']$ that this edge goes between.

Action: What action needed to take place for this change of state to occur.

Part of Input I_P: Which part of the input is being read during this action.

Part of Solution S_P: There needs to be a bijection $S_P = SolutionMap(S_{LG})$ between the solution to these two problems. Here S_{LG} is a path through your the leveled graph G. Each such path needs to be associated with a solution S_P (what ever that looks like) to the problem P. This solution S_P needs to be read off the edges of this path S_{LG}. Hence, each edge must be labeled with a small part of this solution S_P.

Incremental Cost: Each edge needs to be labeled with a weight. The cost/value of a solution/path S_{LG} will be the sum of the weights along this path. So the weight of the edge will be the incremental cost/value of the sub-solution for $state[i, j]$ and that for $state[i', j']$.

Forgetting: The running time of your algorithm will depend on the number of edges (and nodes) of the graph G. Each node/state is labeled with every thing you know when in that state. Knowing a "lot" means that there are lots of different possibilities of what you may have known. Hence, there needs to be lots of states. So the goal is to forget any information that you don't need to continue on. Suppose you a front line worker so know every detail of everything that has ever happened. You need to tell your boss just enough information so that she can make an informed decision about what to do next. Each state $state[i, j]$ is labeled with this information. Specifically, $\langle i, j\rangle$ informs the boss of what he needs to know.

Cutting Task Into Two Independent Tastk: Xxxxxxxx

Examples:

Knapsack: I am walking down an isle of objects $\langle obj_1, \ldots, obj_n\rangle$ and my life goal is put an optimal subset of the objects into my knapsack. When I am in state $state[i, j]$, I have already walked past the objects $\langle obj_1, \ldots, obj_i\rangle$ and have put a (hopefully optimal) subset of them into my knapsack with the total volume filled so far being at most j. Which such subset I have, depends on the path of actions I took to get here. As I walk past object obj_{i+1}, I take one of two following possible actions. With action $k = yes$, I put obj_{i+1} into my knapsack increasing the volume j filed so far by the volume of this object. The weight of this action/edge is the value of this object. With action $k = no$, I do not put it in. The weight of this action/edge zero. Our goal is to get from state $s = state[0, 0]$ to $t = state[n, V]$ with the maximum weighted path.

Longest Common Subsequence: My life goal is to build a longest common subsequence of the two strings $\langle x_1, \ldots, x_n\rangle$ and $\langle y_1, \ldots, y_m\rangle$. When I am in state $state[i, j]$, I have already moved my first string's courser past $\langle x_1, \ldots, x_i\rangle$, my second past $\langle y_1, \ldots, y_j\rangle$, and I hold a (hopefully longest) common subsequence LCS of what I have moved past. Which such LCS I hold depends on the path of actions I took to get here. When $x_{i+1} \neq y_{j+1}$, I cannot change my LCS, but my action either moves my first courser past x_{i+1} or my second past y_{j+1}. These actions are denoted $k =\downarrow$ and $\rightarrow$ because this can be viewed as the direction I am traveling through this matrix of states. The weight of this action/edge is zero. When $x_{i+1} = y_{j+1}$, I extend my LCS with this new character and move both coursers past it. This action is denoted $k =\searrow$. The weight of this action/edge is one. Our goal is to get from state $s = state[0, 0]$ to $t = state[n, m]$ with the maximum weighted path.

Printing Neatly: My life goal is to print a sequence of words neatly. When I am in state $state[i]$, I have already printed the first i words completing some number of lines. How I printed them depends on the path of actions I took to get here. My next action is to decide some number of words k and to print them on the next line. The weight of this action/edge is the cube of the number of blanks on the end of this new line. Our goal is to get from state $s = state[0]$ to $t = state[n]$ with the minimum weighted path.

Same: Note that subinstance $subI[i, j]$ and state $state[i, j]$ are effectively the same and that these define the SAME graph G. It is this graph that you are giving to the oracle.

Edge Length: Consider whether the average length of the edges in G be short or be long. Short edges mean that your path from birth to death will traverse lots and lots of short actions. However, we don't care how many edges are in the optimal path. Running time is not the work of traversing this path. but the work of searching for it. The search time depends on the total number of possible edges possible that we need to search amongst, whether or not taken.

Long edges are scarier because you don't really know where in the future they are leaping to. The further you go, the more places you can get to, and hence the more edges there are to search and hence the longer this searching takes.

The little bird tells you either the next edge or the last edge that you should take in an optimal path. Because the algorithm must try all the possible bird answers, you want the number to be small. All the edges go from the start node s all the way to the destination t, then you would need a separate edge for each of the exponential number of solutions to I_P. This would be too many.

There should be an edge in your graph G for every "atomic" action that takes you from one state to another. Non-atomic actions are not needed because they can be broken into a sequence of atomic actions.

Running Time: The running time of our leveled graph dynamic programming oracle is the number of subinstances times the number of edges $deg(u)$ coming into node u. To be more accurate the time is $\sum_u deg(u)$. This equals the number of edges in the graph. Hence, like depth first and breadth first search, this the running time is linear in the number of edges. For this reason, it does not matter how many nodes are the graph I_{LG} that you give the oracle $Oracle_{LG}(I_{LG})$ who is solving $LeveledGraph_{LG}(G, s, t)$ for you. Give her as many nodes as you like. However, truly try to give her as few edges as possible. **Carefully** consider every edge you are adding to the graph G and make sure that it truly makes a difference to the result.

Can't Reduce: Parsing is a dynamic programming algorithm that cant easily be reduces in this way because it recurses on the left and the right sub-tree.