600.325/425 — Declarative Methods
Assignment 5: Dynamic Programming

Spring 2017
Prof. Jason Eisner

Due date: Friday, May 5, 11:59 pm
(Note: Remaining late days may only be used through May 11)

The questions in this assignment concern dynamic programs. Some questions are closely related to problems you did in previous assignments or from class.

Instructions

Academic integrity: As always, the work you hand in should be your own, in accordance with the regulations at `http://cs.jhu.edu/integrity-code`.

325 vs. 425: Problems marked "**425**" are only required for students taking the 400-level version of the course. Students in 325 are encouraged to try and solve them as well, and will get extra credit for doing so.

Handing in your work: You will submit **only** a written part on Gradescope. **Please do not put your name anywhere.** There is no programming part.

Engaging Further with KNAPSACK

1. [**20/25 points**] Let's review dynamic programming using the KNAPSACK problem.

   As explained on the class slides, you are considering N objects with non-negative integer weights $w_1, w_2, \ldots w_N$ and corresponding values $v_1, v_2, \ldots v_N$. So the subset $\{3, 4, 8\}$ would weigh $w_3 + w_4 + w_8$ and would have value $v_3 + v_4 + v_8$. Your knapsack can hold at most W in weight. So you would like to find the maximum-value subset with weight $\leq W$.

   For $0 \leq n \leq N$ and any weight w, let $C[n, w]$ denote the maximum-value subset of $\{1, 2, \ldots n\}$ that has weight $\leq w$. So you would ultimately like to find $C[N, W]$.

   Let the value of $C[n, w]$ be called $c[n, w]$. In other words, $C[n, w]$ is a *set* (the maximum-value subset) and $c[n, w]$ is a *number* (that subset's actual value). Clear?

   (a) [**1 point**] Write a formula for $c[n, w]$ in terms of other values of c, such as $c[n-1, w-w_n]$. Try to do it on your own, but use your class notes if necessary (we did this in class). You are encouraged to use Dyna notation, but ordinary mathematical notation is also okay. Make sure to get the base cases right—computing $c[n, w]$ using this formula should not recurse forever!

   (b) [**3 points**] Write procedural pseudocode for a *backward-chaining* function that computes $c[n, w]$. This program should be very short and should look a lot like your formula in part 1a. Make sure to memoize the result, though.

   (If instead of pseudocode, you would prefer to write real code in a concise procedural language like Python or C, that is okay too. You are not required to compile it, unless you'd like to test it.)

(c) **[3 points]** Write procedural pseudocode that uses *forward chaining* (no memoization) to compute $c[n,w]$ for all integers $0 \le n \le N$ and all integers $0 \le w \le W$. So when you are all done, you will have $c[N,W]$ (among other things!). Make sure that you order your computation so that when you use a value like $c[n-1, w-w_n]$, it will already have been computed—you should never use an uninitialized value.

(d) **[3 points]** What is the worst-case runtime of computing $c[N,W]$ using your forward-chaining program? Justify your answer (which should be in big-O notation, in terms of N and W).

(e) **[3 points]** What is the worst-case runtime of computing $c[N,W]$ using your backward-chaining program? Justify your answer (which should be in big-O notation, in terms of N and W).

(f) **[2 points]** Would you get any kind of speedup or slowdown if all the integer weights $w_1, \ldots w_N$ happened to be divisible by 10? Explain.

(g) **[5 points]** Now for the most useful part. Write pseudocode to print out $C[n,w]$ (which is the actual subset of items you should take, rather than just the value of that subset).

We didn't do this in class. But note that any code that computes the maximum value $c[n,w]$ must *implicitly* be identifying some subset of items with that value. Your job is to enhance the code so that it identifies that subset *explicitly*.[1] In other words, try modifying the forward-chaining or backward-chaining pseudocode that you wrote in question 1b or 1c (your choice) so that it keeps track of the subset somehow. You may have to write additional pseudocode to actually extract the subset.

(h) **[425]** **[5 points]** Your answer to question 1f should have established that backward chaining will sometimes be faster than forward chaining, depending on the weights. Modify your forward chaining pseudocode from question 1b to speed it up, by ensuring that it *always* computes *only* entries of $c[n,w]$ that backward chaining would compute.

Hint: Let $\mathit{interesting}[n,w]$ be `true` or `false` according to whether $c[n,w]$ is actually needed to compute $c[N,W]$. Just as you already have equations that define c for larger $[n,w]$ pairs in terms of smaller $[n,w]$ pairs, you can also write equations that define *interesting* for smaller $[n,w]$ pairs in terms of larger $[n,w]$ pairs. If you now forward-chain these equations to compute everything you can, you will end up with pseudocode that *first* identifies the $[n,w]$ pairs such that $\mathit{interesting}[n,w]$ is `true`, and *then* determines their actual values $c[n,w]$. The first phase will progress from larger to smaller $[n,w]$ pairs (like question 1c), while the second phase will progress from smaller to larger $[n,w]$ pairs (like question 1b, but now limited to the interesting pairs). This technique is called the "magic sets" transformation.

Increasing Subsequences

2. **[18 points]** In assignment 3, you wrote Prolog code to find the longest increasing subsequence of a given list. (*Strictly* increasing—i.e., no duplicates were allowed in the subsequence.)

   Suppose the input is `[3,5,2,6,7,4,9,1,8,0]`. We pointed out in assignment 3 that it would be a bad idea to generate all subsequences, such as `[3,5,6,4,9]`, and then keep only the ones that were increasing. There would be 2^n subsequences to generate and test.

   Instead, you did the `<` checks as you went along. You never even built `[3,5,6,4,9]`—because you wouldn't have been willing to stick `6` at the front of `[4,9]` at an earlier step.

[1] Similarly, in SAT, you often want to go beyond just *proving* that a formula is satisfiable. You want to give a *constructive proof* by actually constructing a particular satisfying assignment—which serves as a *witness* that the formula is satisfiable. In the present problem, you want to construct a *witness* $C[n,w]$ that demonstrates that the maximum value is (at least) $c[n,w]$.

So your Prolog code generated only the *increasing* subsequences only, and then picked the longest.

(a) [**2 points**] Though better, why was that still inefficient? Give an example of a length-10 list where even generating all *increasing* subsequences would be slow.

(b) [**3 points**] Someone suggests the following "greedy" recursive solution: "To build the longest increasing subsequence of `[3,5,2,6,7,4,9,1,8,0]`, first build the longest increasing subsequence of `[5,2,6,7,4,9,1,8,0]`, then glue 3 on the front if you can."

```
function LIS(list)
  if list.empty() :
    return []
  else
    subproblem = list.rest()
    ▷ find just the best solution to subproblem—not all solutions as in the Prolog version!
    subsolution = LIS(subproblem)
    if subsolution.empty() or list.first() < subsolution.first() :
      return cons(list.first(), subsolution)
    else
      return subsolution
```

Equivalently, here's a Dyna version:

```
lis([]) = [].
lis([X|Xs]) := lis(Xs).                                   % default
lis([X]) := [X].                                          % override default
lis([X|Xs]) := [X | lis(Xs)] if X < Y & [Y|Ys]==lis(Xs).  % override default
```

Try this function *by hand* on the list `[3,5,2,6,7,4,9,1,8,0]`. What is the big-O runtime? What answer does the function get? What is the correct answer?

(c) [**1 point**] Now you'll correct the above solutions, using dynamic programming.

As preparation, remember the in-class problem of maximum-weight independent set in a tree. (For simplicity, let's restrict to just a simplified *binary*-tree version.) The algorithm had to solve 4 recursive subproblems rather than 2. It wasn't enough just to get the max-weight independent set of the left subtree and also of the right subtree! Rather, in each of the two subtrees, we had to get the max-weight independent set *and* the max-weight *unrooted* independent set. Knowing that a partial solution was unrooted allowed us to determine that we could legally combine it with other partial solutions in certain ways.

In other words, we decided to solve a slightly harder problem than the original. In effect, we wrote MIS(tree, must_be_rooted). Then we called both MIS(tree, false) and MIS(tree, true), at different times. So MIS had a slightly more general problem to solve, but was able to solve it *by relying on recursive copies of itself that could also solve more general problems*!

(This trick is known as "strengthening the inductive hypothesis." You've done proofs by induction—basically a proof that calls itself recursively. Sometimes the only way to write one is to decide to prove a stronger theorem than you were assigned, because otherwise the recursive call—the "inductive hypothesis"—won't establish all the results that you need in order to solve the original problem.)

Now go back to the LIS problem. What do you have to know about an increasing subsequence of the subproblem `[5,2,6,7,4,9,1,8,0]` in order to know whether you are allowed to glue 3 onto the front?

(d) **[7 points]** Given your answer to 2c, improve the pseudocode or Dyna code (your choice) from 2b so that it will get the *correct* answer. Like MIS, your LIS will have an extra argument that is used in those recursive calls. And like MIS, it will have to call itself more than once.

(e) i. **[1 point]** How should you call your new two-argument LIS function if you want the longest increasing subsequence of `[3,5,2,6,7,4,9,1,8,0]`?

ii. **[1 point]** The fact that your LIS is multiply recursive suggests that it might benefit from dynamic programming (i.e., reuse of subsolutions). Give an example of a subproblem $\text{LIS}(x, y)$ that must be solved at least twice during the call you just proposed.

iii. **[1 point]** What specific technique would avoid the duplicate computation?

iv. **[2 points]** What happens to your runtime if you *don't* avoid the duplicate computation?

v. **[Extra Credit]** Prove your answer to the previous question.

Fast Binary Search Trees

3. **[15/23 points]** In assignment 3, you wrote Prolog/ECLiPSe code to generate balanced binary search trees.

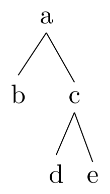
Balanced trees might not always be what we want in practice, though. If some of the keys in the tree will be searched for much more often than the rest, then it will be more efficient in the long run to store those keys closer to the root, even if this means pushing other keys further down.

Suppose that we have a fixed set of keys $k_1, k_2, ..., k_n$ that we want to store in a search tree, and we know (or can guess) exactly how many times we will want to search for each key (denoted $t_1, t_2, ..., t_n$). We want to construct a search tree so that the *total* number of nodes visited is minimized.

Because of the property that a subtree of an optimal search tree is itself an optimal search tree (for the keys in the subtree), constructing optimal search trees can be solved efficiently by dynamic programming.

Example: The following tree is *not* a solution because it's not a legal search tree. When searching for a key in a subtree, we'd like to recurse to the left or right subtree according to whether the key is less or greater than the subtree's root. So we need a guarantee that the left (or right) subtree stores only keys that are less (or greater) than the root.

This means that in a legal search tree, the infix order of the nodes will be sorted, i.e., `[a,b,c,d,e]` rather than `[b,a,d,c,e]` as in the illegal search tree below:



However, suppose this were a legal search tree, and the keys `[a,b,c,d,e]` have cost $50, 10, 20, 1, 3$ respectively, its cost would be

$$\begin{aligned}\text{cost} &= \sum_x (\text{time to find } x) \cdot (\text{how often we look for } x) \\ &= \sum_x (\text{depth of } x) \cdot (\text{frequency of } x) \\ &= 1 \cdot 50 + 2 \cdot 10 + 2 \cdot 20 + 3 \cdot 1 + 3 \cdot 3 = 122\end{aligned}$$

(a) **[10 points]** Come up with a dynamic programming algorithm to solve this problem. Be clear with your solution! Specifically:

i. Devise a function. What does it represent?
ii. How are you calling smaller subproblem(s)?
iii. What is the goal for a problem with n keys?

You should write your algorithm as one or more recursive formulas, as in question 1a. The solution should be fairly short. You are encouraged to use Dyna notation, but ordinary mathematical notation is also okay.

(b) [**5 points**] What is the worst-case runtime of computing the optimal search tree for n keys. Justify your answer!

(c) [**425**] [**5 points**] Write pseudocode for your algorithm.

(d) [**425**] [**3 points**] Your pseudocode should be saving computation along the way to avoid duplication. How does the program's big-O runtime compare with one which does not avoid the duplicate computation?

(e) [**Extra Credit**] What is the worst-case runtime of the unmemoized version of the algorithm?