Recitation 16

Dynamic Programming

Dynamic Programming generalizes Divide and Conquer type recurrences when subproblem dependencies form a directed acyclic graph instead of a tree. Dynamic Programming often applies to optimization problems, where you are maximizing or minimizing a single scalar value, or counting problems, where you have to count all possibilities. To solve a problem using dynamic programming, we follow the following steps as part of a recursive problem solving framework.

Dynamic Programming Steps (SR. BST)

1. Define **Subproblems** subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** Subproblems $x(i) = f(x(j), \ldots)$ for one or more $j < i$
   - State topological order to argue relations are acyclic and form a DAG
3. Identify **Base** Cases
   - State solutions for all reachable independent subproblems
4. Compute **Solution** from Subproblems
   - Compute subproblems via top-down memoized recursion or bottom-up
   - State how to compute solution from subproblems (possibly via parent pointers)
5. Analyze Running **Time**
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = W$ for all $x \in X$, then $|X| \times W$

Once subproblems are chosen and a DAG of dependencies is found, there are two primary methods for solving the problem, which are functionally equivalent but are implemented differently.

- A **top down** approach evaluates the recursion starting from roots (vertices incident to no incoming edges). At the end of each recursive call the calculated solution to a subproblem is recorded into a memo, while at the start of each recursive call, the memo is checked to see if that subproblem has already been solved.

- A **bottom up** approach calculates each subproblem according to a topological sort order of the DAG of subproblem dependencies, also recording each subproblem solution in a memo so it can be used to solve later subproblems. Usually subproblems are constructed so that a topological sort order is obvious, especially when subproblems only depend on subproblems having smaller parameters, so performing a DFS to find this ordering is usually unnecessary.

Top down is a recursive view, while Bottom up unrolls the recursion. Both implementations are valid and often used. Memoization is used in both implementations to remember computation from previous subproblems. While it is typical to memoize all evaluated subproblems, it is often possible to remember (memoize) fewer subproblems, especially when subproblems occur in 'rounds'.

Often we don't just want the value that is optimized, but we would also like to return a path of subproblems that resulted in the optimized value. To reconstruct the answer, we need to maintain auxiliary information in addition to the value we are optimizing. Along with the value we are optimizing, we can maintain parent pointers to the subproblem or subproblems upon which a solution to the current subproblem depends. This is analogous to maintaining parent pointers in shortest path problems.

Exercise: Text Justification

Text Justification is the problem of fitting a sequence of n space separated words into a column of lines with constant width s, to minimize the amount of white-space between words. Each word can be represented by its width $w_i < s$. A good way to minimize white space in a line is to minimize **badness** of a line. Assuming a line contains words from w_i to w_j, the badness of the line is defined as $b(i,j) = (s - (w_i + \ldots + w_j))^3$ if $s > (w_i + \ldots + w_j)$, and $b(i,j) = \infty$ otherwise. A good text justification would then partition words into lines to minimize the sum total of badness over all lines containing words. The cubic power heavily penalizes large white space in a line. Microsoft Word uses a greedy algorithm to justify text that puts as many words into a line as it can before moving to the next line. This algorithm can lead to some really bad lines. LaTeX on the other hand formats text to minimize this measure of white space using a dynamic program. Describe an $O(n^2)$ algorithm to fit n words into a column of width s that minimizes the sum of badness over all lines.

Solution: First, we use dynamic programming to compute all badnesses $b(i,j)$ in $O(n^2)$ time!

1. **Subproblems**
   - $x(i,j)$: sum of word lengths w_i to w_j
2. **Relate**
   - $x(i,j) = \sum_k w_k$ takes $O(j-i)$ time to compute, slow!
   - $x(i,j) = x(i,j-1) + w_j$ takes $O(1)$ time to compute, faster!
   - Subproblems $x(i,j)$ only depend on strictly smaller $j - i$, so acyclic

3. **Base**
   - $x(i, i) = w_i$ for all $0 \leq i < n$, just one word
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Compute each $b(i, j) = (s - x(i, j))^3$ in $O(1)$ time
5. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

Now after precomputing $b(i, j)$ for all pairs i and j, we can return any $b(i, j)$ in $O(1)$ time. Then we use dynamic programming to solve the original problem.

1. **Subproblems**
   - Choose suffixes as subproblems
   - $\boxed{x(i)\text{: minimum badness sum of formatting the words from } w_i \text{ to } w_{n-1}}$
2. **Relate**
   - The first line must break at some word, so try all possibilities
   - $x(i) = \min\{b(i, j) + x(j + 1) \mid i \leq j < n\}$
   - Subproblems $x(i)$ only depend on strictly larger i, so acyclic
3. **Base**
   - $x(n) = 0$ badness of justifying zero words is zero
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(0)$
   - Store parent pointers to reconstruct line breaks
5. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(n)$
   - $O(n^2)$ running time

Exercise: Longest Increasing Subsequence

Given an array of n integers $a_0, \ldots, a_{n-1}$, we know how to sort them in a in $O(n \log n)$ time (or faster depending on the range of the integers). Suppose instead of reordering the array, we simply want to find a long subsequence of the integers (not necessarily contiguous) that is in increasing order within the array. Describe an $O(n^2)$ time algorithm to find the longest increasing subsequence of an array of integers.

Solution:

1. **Subproblems**
   - Choose prefixes as subproblems
   - $x(i)$: length of the longest subsequence from first $i+1$ integers that also includes a_i
2. **Relate**
   - To solve $x(i)$, the second to last integer in a longest subsequence ending in a_i must be a_j for some $j < i$ satisfying $a_j < a_i$.
   - $x(i) = \max\{x(j) + 1 \mid 0 \leq j < i \text{ and } a_j < a_i\}$
   - Subproblems $x(i)$ only depend on strictly smaller i, so acyclic
3. **Base**
   - $x(i) = 1$ if $a_j \geq a_i$ for all $0 \leq j < i$
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is maximum of $x(i)$ for $0 \leq i < n$
   - Store parent pointers to reconstruct a maximizing subsequence
5. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(n)$
   - $O(n^2)$ running time

Note that it is possible to use data structures you've already learned to improve this running time! In particular, as we compute the subproblems $x(i)$ for increasing i, we could store each $x(i)$ in a data structure that supports an operation to find largest $x(j)$ having $a_j < a_i$, in faster than linear time! In fact, this sounds like a one-sided range query! As a hint, insert memoized subproblems into an AVL tree sorted by (a_i, i) while augmenting each node with the index j in its subtree having maximum $x(j)$, and show how to support such queries in $O(\log n)$ time. Then, maximum calculation in the recursive call can also be evaluated in $O(\log n)$ time, leading to an $O(n \log n)$ time algorithm.

```python
# bottom up implementation
def lis(A):
    x = [1 for _ in A]          # memo
    parent = [None for _ in A]  # parent pointers
    for i in range(1, len(A)):  # solve dynamic program
        for j in range(i):
            if (A[j] < A[i]) and (x[i] < x[j] + 1):
                x[i] = x[j] + 1
                parent[i] = j
    last = 0                    # find largest subproblem
    for i in range(1, len(A)):
        if x[last] < x[i]:
            last = i
    sequence = []               # reconstruct backward sequence
    while last is not None:
        sequence.append(A[last])
        last = parent[last]
    return sequence[::-1]       # return reversed sequence
```