Dynamic Programming 2

Workshop in Competitive Programming – 234900
About Dynamic Programming (reminder)

• Dynamic Programming ≈ Recursion + Re-use

• When to use it?
  • An optimal solution can be constructed efficiently from optimal solutions of subproblems.
  • Some subproblems overlap.

• What does it do?
  • Compute the value of a recursive function $f$, assuming:
    • $f$ has no cyclic dependencies
    • Each argument has a finite number of possible value
    • The set of all possible parameter combinations is small enough to fit in memory
General Scheme (reminder)

• Steps for using DP:

1. Define the subproblem (the function meaning and parameters)
2. Find the recursive rule
3. Solve base cases
4. Define the target value (which value do you need to solve the problem)
5. Define the computation order (e.g. ascending order of n)
Classic DP problems

Knapsack (reminder)

• Input: items having weight and value, and a bag of a certain capacity
• Output: the max value that can fit in the bag

• Solution:
  • $f(item, capacity)$ provides the max possible value in the bag, where $item$ is the largest item id considered and $capacity$ is the remaining capacity.
  • Go over all items, at each step take the max between taking the item or ignoring it.
Classic DP problems

Coin exchange

• Input: target price, and a list of coin values
• Output: the min number of coins that add up to the target price
• Solution:
  • \( f(price) \) provides the number of coins needed to reach \( price \)
  • At each step take the min between all coin types.
    \[
    f(price) = 1 + \min_i \{ f(price - coinValue[i]) \}
    \]
Classic DP problems

Traveling Salesman

• Input: full weighted graph (distances between cities)
• Output: min weight of a Hamiltonian cycle (tour visiting all cities)
• Solution:
  • Set some starting node $s$.
  • $f(last, visited)$ provide the min weight of a path starting at $s$ and ending at $last$, where $visited$ are exactly the ids of the visited nodes.
    • use bitmask to represent the set of visited nodes in one integer.
  • At each step take the min between all possible (unvisited) next nodes.
Memoization

• This was the bottom-up approach, but there is also a top-down approach called memoization:

  Use normal recursion, but save values you already calculated.

• Memoization is intuitive, in fact some of you already used it without knowing.

  “To help with the time efficiency I used a hash table to hold the values already calculated”
  (anonymous student)

  “The calculation is made "with memory" such that if we already calculated the number of options with these bounds, we save it and can use it later on instead of calculating it again.”
  (another anonymous student)

Snippets from the text files submitted with Wooden Signs in HW2 this year
DP vs. memoization

• How do I know which one to use?

<table>
<thead>
<tr>
<th></th>
<th>Bottom-up DP</th>
<th>Top-down memoization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All possible sub-problems are computed exactly once</td>
<td>Required sub-problems are computed on-demand</td>
</tr>
<tr>
<td>Pros:</td>
<td>• If many sub-problems are revisited, this is faster (No overhead from recursive calls calling to calculate the same value)</td>
<td>• If many sub-problems are not required, this could be faster (no need to fill in the entire table)</td>
</tr>
<tr>
<td></td>
<td>• If each column only depends on the previous one, can save memory (we only need to save one column at a time, not the entire table)</td>
<td></td>
</tr>
</tbody>
</table>

• If you are not sure, try both.
Further Reading

• Algorithms 1 for proofs and more variations: http://webcourse.cs.technion.ac.il/234247

• Wikipedia tries to give intuitive explanations and usually contains code snippets.

• The Stanford ACM-ICPC Notebook contains optimized implementations for most graph algorithms: https://github.com/jaehyunp/stanfordacm
On a different note

• Good team work and time management are important for success in the competition (and in class).

• Try to identify the bottlenecks stopping you from solving more problems, and work on that.

• Your time is precious. If your partner is using the computer to solve a problem and you feel like you are not doing much, do something!
  • Try to find edge cases to fail your partner’s solution
  • Start working on another problem:
    Write code on paper! Write code that compiles, not pseudo-code. This will save time when you get the computer (and also ensure you have thought on all parts).