Cheat-Sheet for Dynamic Programming

Written by Marcelo Siero, based on UCSC's class on Algorithm Design taught by Prof. Dmitris Achlioptas, with ideas passed on by Greg Levin.

Dynamic Programming is approached as a DAG (Directed Acyclic Graph) of sub-problems. A good way to think about them is to look at solution of a last sub-problem whose answer is dependent on the sub-answers of all the previous sub-problems (i.e. think of strong induction with the recurrence being the induction step and the terminating condition being the base-case for an induction proof.)

In this cheat-sheet we provide a short-hand for the solutions to a wide-variety problems that are solved with the techniques of dynamic programming.

Template for Solution Short-Hand

All the DP examples are described using the following form:

1. Description of the subproblem to be solved.
2. Base case.
3a. Type of choice (multi-way or binary) 3b. Recurrence Relation, describing the subproblem.
4. How to invoke the DP problem. 5. How to do a traceback for the solution (memoized approach). 6. Runtime.

Table of Contents of DP Prototypical Programs Examined

The Following Types of Problems Are Considered:

Summary of Dynamic Programming Prototypical Programs

Stock Market Buy/Sell gains (an accumulation problem):

A[i] is the buy price per day.
A[j] is the sell price each day.
Problem is to find the best profit where j¿i.

So essentially the problem is to calculate the maximum profit at each sale day relative to all the possible previous buy days.

Brute Force Solution: $O(n^2)$

```
Create a 2D array P[i,j] of profits.
for i=1 to n:
    for j=i+1 to i:
        P[i,j] = A[j] - A[i]
```

Div and Conquer Approach to Problem:
This solution is accomplished by recursively splitting the problem in half with a sub-solution found either on the left, the right or across the two halves. This has a runtime of O(n)

Stock Market by Dynamic Programming: binary choice, O(n)

Accumulate profits in the form of: sell(high) - buy(low) on selected days to maximize profit:

1) OPT(k) is the best profit that can be made on that day (based on accumulation of profit of previous days).

2) OPT(1)=0, i.e. first buy day.

3b) OPT(k) = max{0, OPT(k-1) + A[k] - A[k - 1]}

3a) Binary choice:
Choice 1: A 0, means don't hold the stock, try buying that day (a low day).
(no profit made on the day you buy - thus OPT(k) = 0 for that day)
A 0, can be thought of implying that we would have sold on the previous day for getting that profit.

Choice 2: Keep the stock and accumulate more profit.
4) OPT(n)

5) To find solution, Keep track of largest value (the optimal sale day), backtrack to its previous 0, the optimal purchase day.

6) OPT(n)

Knapsack 1 (1 of each kind of item): (2 OPT vars, double decision), 2D memoize

Problem: robber puts any of n items in knapsack w. weight cap. W. items are indexed with i, weights w_i, value v_i. We wish to optimize the value of the heist.

WRONG SOLUTION FOR Knapsack 1:

3b) OPT(i) = $max\{$ OPT$(i-1)+v_i$, OPT$(i-1)\}$ It does not account for weight. Note that solutions of previous OPT[i] will change as i advances due to weight limits.

RIGHT SOLUTION FOR Knapsack 1:

1) OPT[i,c] is the optimal value in knapsack after making the best decision to steal items i or not, along with all previous such decisions for items less than i, their corresponding possible capacities.

2) Base case: OPT(0,W) = 0

3a) Decision (2-way): whether to grab item i (choice 1) or not grab it (choice 2)

3b) $$\mathrm{OPT}(i,c) = \max\begin{cases} \mathrm{OPT}(i-1,c) & \text{if } c \leq 0 \\ \mathrm{OPT}(i-1,c-w_i)+v_i,\ \mathrm{OPT}(i-1,c)\} & \text{otherwise} \end{cases}$$

We create a temp var w and record a memoized a 2D array OPT[i,w]; $c - w_i$ reduces the remaining weight capacity in knapsack, or thought another way it increases the weight in the knapsack and how close we are to W.

We can store values on an $n * W\ OPT[i,c]$ array. All valid leftover capacities get explored within this 2D array.

4) Call with OPT(n,W)

5) PTR keeps track of the best decisions sequence wrt i's taken, to create solution.

6) Runtime is $O(2nW)$ i.e. $O(nW)$.

Knapsack 2 w. infinite multi-way qty of each item, uses 2D storage

Problem: is the same as Knapsack 1, but it can take an infinite number of each kind of item.

1) OPT(i) is the optimal value after decision to steal items i or not.

2) Base case: OPT(0,W) = 0

3a) Decision (multi-way): how much of the item to grap, with quanta related to its weight.

3b) We create a temp var w and store best values of items for different knapsack capacities on a 2D array OPT[i,c]

$$\mathrm{OPT}(i,c) = \max_{0 \leq j \leq c/w_i} \{\ \mathrm{OPT}(i-1, c-j*w_i) + j*v_i\}$$

The $c - j * w_i$ term reduces the remaining weight capacity in knapsack, or thought another way it increases the weight in the knapsack and how close we are to W.
We must memoize values on an array mxW OPT[i,c] array. All valid W's get explored within this 2D array.

4) Call with OPT(n,W)

5) PTR can be kept to keep track of the best decisions in terms of qty taken of each item at each capacity, used to create the solution. A pretty monstrous program.

6) Runtime is $O(nW \min(W/w_i))$

Optimization notes: It would probably be best to cache the W dimension, and PTR array in a hash to avoid the enormous use of space, for unused indices. Also using W as lcd(all w_i) would reduce the size of the space, and the runtime instead of trying all integer values of the weight.

Weighted Interval Scheduling: (binary choice)

Problem: Select a subset $S \subseteq \{1, ..., n\}$ of mutually compatible intervals, to maximize sum of values of the selected intervals, $\sum_{i \in S} v_i$, where v_i are the weights of these intervals.

TRICK: Trick here is to start with the end, potential optimal choice and work backwards in terms of compatibility, to earlier good choices. Latter choices are dependent on earlier choices. Working backwards means looking back at previous compatible interval. We use function p(j) for closest previous compatible interval. This is true for most all Dynamic Programming Problems.

1) OPT(j) provides the max sum of weights for an optimal set of the first j compatible intervals.

2) Base Case: OPT(0) = 0

3a) We call O_j the optimal scheduling set and $p(j)$ is the largest index $i < j$ such that interval i ends before j begins. The choices for OPT(j) are binary in form:
Choice 1: $j \in O_j$
Choice 2: $j \notin O_j$.

It belongs to O_j iff the 1st option is at least as good as the 2nd. In other words, request j belongs to an optimal solution on the set 1,...,j iff $v_j + OPT(p(j)) \geq OPT(j-1)$.
Thus recurrence OPT(j) expresses optimal value in terms of the optimal values to smaller subproblems as DP requires.

3b) $\text{OPT}(j) = max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1))$.
See proof in page 255 of text for inductive proof style of this.

4) Code gets invoked with OPT(n)

5) The solution is obtained by storing the choices made as to what j intervals are part of O_j into array PTR or exracting that out in a backtrace. This then will provide the O_j set that will maximize the results.

6) Runtime is O(n).

7) Here is the code with memoization:

```
M-Compute-Opt(j)
    If j = 0 then
        Return 0
    Else if M[j] is not empty then
        Return M[j]
    Else
        M[j] := max(vj+M-Compute-Opt(p(j)), M-Compute-Opt(j - 1))
    Return M[j]
Endif
```

Segmented Least Squares, Pretty-Printing:

Basic Observation: If last segment of optimal partition is $p_i, ..., p_n$, then the value of the optimal solution is OPT$(n) = e_{i,n} + C +$ OPT$(i-1)$. (i corresponds to the beginning of each optimal segment, where $e_{i,n}$ is the least square errors (cached) of a line segment from i to n. Cerr is an additive penalty incurred as more segments get created. Note that for different j's end values: multi-way min tests out all possible partitionings.

Template Desc: OPT(j) multi-way choice with internal variable $(1 \leq i \leq j)$ and 1D storage. Note: memoize/store OPT[j] gets bigger the values of OPT[j] would change.
1) OPT(j) is min total error score up to point j with best chosen partitionings at each point.

2) OPT(0) = 0

3a) Select best previous segment start point i ending in j that optimizes overall error.

3b) OPT$(j) = min_{1 \leq i \leq j}\{err(i,j) + Cerr +$ OPT$(i-1)\}$
Cerr is a constant provided if we wish to force a change in the number of segments that paritions the segments.
4) Invocation: OPT(n)

5) Choices of optimal values for i choices get stored in PTR to recoup the solution, scanned backwards. These make up the optimal end points of the segments.

6) Runtime $O(n^2)$ assuming that err(i,j) can be computed or pre-computed in $O(n^2)$ also.

Finding Longest Path/Segment in a DAG:

(Can be modified easily to be the shortest path)
Directed edges here all go from indexes u to v, where $u < v$.

OPT(i) is the longest path up to node v starting at node 1.

$$\text{OPT}(v) = max_{1 \leq u \leq v-1} \begin{cases} \{ \text{ OPT}(u) + 1 & if e(u,v) \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

Here OPT[v] memoizes the best value at v.
PTR[v] stores the previous u that maximum up to that point.

Diff, String Similarity, and Genomic Sequence Alignment Problems:

1) OPT(i,j) is a best score for converting one string to another up to char i in source and j in target. String x_i for source is n long, and string y_j is m long. Advancing in j is a deletion, advancing in i is an insertion, advancing in both is a match or conversion. A deletion or an insertion create a gap, which has a penalty cost of δ. A character substitution has a penalty of $\alpha_{x_i y_j}$ to convert from x_i to y_j. Edits are by deleting inserting, or substituting. There is a deletion cost c_d, an insertion cost c_e and a substitute cost: $C(A[i], A[j])$. In Molecular Biology this kind of alignment is referred to as sequence alignment. The Smith-Wasserman Algorithm is similar to this.
2) Initialize $A[i,0] := i * \delta$ for each i
Initialize $A[0,j] := j * \delta$ for each j

3a) OPT(i) requires a 3-way decision for delete, insert, substitute.

3b) min alignment costs satisfy the following recurrence for i 1 and j 1:

$$\text{OPT(i,j)} = \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases}$$

Moreover, (i,j) is in an optimal alignment M for this subproblem iff the minimum is achieved by the first of these values.

5) Call with: OPT(n,m)

6) Runtime is O(nm).

7) Invocation code can be done by calling this function:

```
Alignment(X,Y):
   Array A[0 ... m,0 ... n]
   Initialize A[i,0]= i \delta for each i
   Initialize A[0,j]= j  for each j
   For j = 1,...,n:
       For i = 1,...,m:
           Use the recurrence 3b. to compute A[i, j]
       Endfor
   Endfor
   Return A[m,n]
```