Advanced DP lecture notes

Artem Vasilyev

October 2nd, 2018
Hello Barcelona ICPC Bootcamp

1 Introduction

Dynamic programming (DP) is a vast field in competitive programming. Every contest usually contains at least one problem that is solved using DP. In this lecture we will discuss the advanced DP principles and tricks.

2 DP on profiles

DP on profiles is often used when you have a long, but narrow grid, for example, you have to put zeros and ones on a 10×1000 grid with some restrictions. In this case, a DP state will be the vector of 10 zeros and ones that were put in the last column. Transitions are made as follows: you iterate over all possible new columns (2^{10} possibilities), check if the transition is possible, then update DP values.

2.1 Broken profiles; hard states

DP on profiles is often explained the way I did earlier. But the running time and code complexity is lower, when your state is not a full profile (a state for the whole column), but a «broken» profile (picture needed), consisting of last n cells in column-major order.

You have to remember some information about the cells in this broken profile: colors of cells, decomposition into sets, or something else. It is usually easier (and more efficient) to add one cell at a time, than make a transition to whole new column at once.

2.2 Example: number of hamilton cycles on a grid

Consider the following problem: you have a grid with some cells removed. You have to find the number of hamilton cycles on this grid.

So what do you have to store in a DP state? Look at the broken profile; see what information we need. At any moment, we are only interested in the components which the cells in the profile belong to. The number of different possible decompositions into sets it called a *Bell number* and grows quite rapidly: $B(10) = 115975$. Now we can notice that not every set decomposition is reachable. For example, all sets (connected components) must have two vertices in the profile, and the ends of all connected components should form a valid *bracket sequence.* That means that for a profile of size $2n$ there are only about $\frac{4^n}{n^{1.5}}$ possible profiles, instead of a Bell number, which is a lot smaller.

2.3 Implementation details

There is a nice trick that allows for a very easy implementation. First, write a brute-force code going over all possible solutions by just adding one cell at a time (in column-major order) and recalculating what you need. Often in these kinds of problems you would want to write a brute-force solution anyway to check the correctness of your DP implementation.

After you have the brute-force solution, it's easy to convert it into the DP solution: just add memoization! Whenever you enter a recursive function call, hash everything you need in this state (current cell coordinates, some properties of profile, flags, etc) and check whether this hash is present in hash map. If it is (assuming no collisions), you already calculated this value and can exit. Since in this scheme you don't need to be able to restore state from it's identifier (i.e. ternary mask, or decomposition into sets), so you can just hash everything you will need later in bruteforce (everything in broken profile) and you are done. This can sometimes be slow, but often it's enough and you probably need a brute-force solution for hard problems anyway.

3 DP on subsets

DP on subsets is often employed when you are solving some kind of NP-complete problem: finding a Hamiltonian path/cycle, solving knapsack with large weights and so on.

3.1 «Broken sets»; dividing into subsets $\leq S$

Consider the following problem: you have n items with weights w_i; you have to divide them into subsets with sum $\leq S$.

The first idea of the solution would be as follows: mark all subsets with sum $\leq S$ as admissable, then do a dp: $min[mask] = \min\limits_{submask \subseteq mask} min[mask \setminus submask] + 1$, iterating over all admissable submasks. This works in $O(3^n)$, because there are 3^n pairs of $(mask, submask \subseteq mask)$.

This can be optimized to $O(2^n n)$. The idea is similar to DP with profiles vs. broken profiles: instead of adding the whole set at once, we will add one item at a time. Now $min[mask]$ will contain a pair: (minimum number of sets, size of the last set). We should minimize this pair lexicographically: a pair with less sets is better, if the number of sets is the same, we should minimize the size of the last set. This idea can be applied to many other problems.

Exercise: you have a weighted graph with $n + 1$ vertices, vertex 0 being the special one. Cover all vertices of this graph with cycles of given lengths $c_1, c_2, \ldots, c_k$ ($\sum(c_i - 1) = n$), each one starting in vertex 0 in $O(2^n n)$ time.

3.2 Fast subset convolution

Sometimes in problems (usually combinatorics) you have to use something like this:

```
for i = 0..2^k - 1:
  for j = 0..2^k - 1:
    C[i or j] += A[i] * B[j]
```

This takes $O(4^k)$ or $O(3^k)$, depending on problem. It is possible to calculate array C in $O(2^k k)$ time.

First, let's calculate array $As[i] = \sum\limits_{j\&i=j} a_j$, in other words, the sum over all submasks of i. It can be done in $O(2^k k)$ with the following algorithm (in-place):

[illegible]

```
for bit = 0..k - 1:
  for i = 0..2^k - 1:
    if (i & (1 << bit)) != 0:
      A[i + (1 << bit)] += A[i]
```

After that, $Cs[i]$ will be equal to $As[i] \cdot Bs[i]$. The inverse transform looks almost identical, except with `A[i + (1 << bit)] -= A[i]`.

4 Matrix exponentiation

Some problems have a little number of states (about 100), but ask you to make a lot of similar transitions (but only as a linear combination of previous states). An example would be a calculation of Fibonacci numbers or any linear recurrence of order $m \leq 100$. This time, we can represent a DP transition as a matrix-vector multiplication. This way, when we want to calculate $A^n v_0$, we use binary exponentiation in $O(m^3 \log n)$. Sometimes you'll have to optimize the constant factor in this solution.

Usually binary exponentiation is explained as follows: to calculate A^n, first, we compute $A^1, A^2, A^4, \ldots, A^{2^{\log n}}$, then multiply the matrices that we need looking at the binary representation of n. This can be optimized by the fact the we don't need A^n, but rather $A^n v_0$, so instead of $2 \log n$ matrix multiplications, we only need $\log n$ matrix multiplications ($O(m^3)$) and $\log n$ matrix-vector multiplications ($O(m^2)$), which is about two times faster.

Of course, a lot of notes about matrix multiplications apply here. For example to multiply two matrices A and B modulo P, first, transpose B, so you have row-row scalar products instead of row-column. Second, while calculating $\sum_k A[i][k] \cdot B^T[j][k]$, you can calculate modulo P^2 (if P fits into `int`) or even $\lfloor \frac{10^{18}}{P} \rfloor \cdot P$. This way we can check overflow with only one `if` and subtraction, so you dont use modulo operation m^3 times, but only m^2 times, which is a lot faster.

There are ways to calculate linear recurrences of order m in time $O(m^2 \log n)$ or even $O(m \log m \log n)$ using generating function and polynomial modulo operations, see editoral on CodeChef: `https://discuss.codechef.com/questions/65993/rng-editorial`.

4.1 State merging

Sometimes, even this is not enough. Due to nature of the problem, ther might be some equivalent states, which can be merged into one. For example, if our state is a permutation, then maybe our transitions only depend on the *cyclic structure* of the permutation, that is, only set of lengths of cycles matters in our context.

To find out which states can be merged, you can either manually find equivalent states (states, for which the answer is always the same), or use an equivalent of automaton minimization algorithm to find all distinct pairs of states, then merge all non-distinct ones.

4.2 Extrapolation

There are problems where the answer is a polynomial or a linear recurrence. For example, if you can use matrix exponentiation to find the result, and the size of matrix is m (which is the number of states), it is always a linear recurrence of order m. If it's a polynomial, calculate values for some small values, then use either Lagrange or Euler interpolation and calculate the needed value. If the answer to the problem can be found via linear recurrence, then you can restore the coefficients of linear recurrence using Berlekamp-Massey algorithm (you can read a bit about it here: `https://codeforces.com/blog/entry/61306`).

5 DP Optimizations

In this section, we'll consider the following DP problem: suppose we have function $cost(i, j)$ — the cost of segment $[i, j]$, which can be calculated in $O(1)$. Now we want to calculate $f(n, k)$ — minimum possible cost to divide all items in $[1, n]$ into k segments. Define $A_{i,j}$ as the optimal left point of the last segment in the optimal decomposition of $[1, i]$ into j segments (in case of ties, take the leftmost point).

5.1 Quadrangle inequality

Often, the *cost* function satisfies the *quadrangle inequality*: for all $a \le b \le c \le d$: $cost(a,d) + cost(b,c) \ge cost(a,c) + cost(b,d)$. If this is true, then $A_{i-1,j} \le A_{i,j}$.

5.2 Divide and Conquer optimization

Using the fact that $A_{i-1,j} \le A_{i,j}$ and that all $f(*,j)$ can be calculated from all $f(*,j-1)$, we can apply divide and conquer optimization.

Consider this recursive function `Calculate`$(j, I_{min}, I_{max}, T_{min}, T_{max})$ that calculates all $f(i,j)$ for all $i \in [I_{min}, I_{max}]$ and a given j using known $f(*,j-1)$.

```
function CALCULATE($j$, $I_{min}$, $I_{max}$, $T_{min}$, $T_{max}$)
    if $I_{min} > I_{max}$ then
        return
    $I_{mid} \leftarrow \frac{1}{2}(I_{min} + I_{max})$
    calculate $f_{I_{mid},j}$ naively, let $T_{opt}$ be the optimal $t \in [T_{min}, T_{max}]$
    CALCULATE($j$, $I_{min}$, $I_{mid} - 1$, $T_{min}$, $T_{opt}$)
    CALCULATE($j$, $I_{mid} + 1$, $I_{max}$, $T_{opt}$, $T_{max}$)
```

The initial call to this function will be `Calculate`$(j, 1, n, 0, n)$ for all j from 1 to k. The time speedup comes up from the fact that all naive calculations of $f_{I_{mid},j}$ on each level of recursion take $O(n)$ in total, because each recursive call splits the segment $[T_{min}, T_{max}]$ into 2 (almost) disjoint segments. The depth of recursion is $O(\log n)$, so the running time of each `Calculate` call is $O(n \log n)$. After calculating all k layers in $O(kn \log n)$ time, we get the answer.

5.3 Knuth optimization

The Knuth optimization uses the fact that $A_{i,j-1} \le A_{i,j} \le A_{i+1,j}$. Both these inequalities follow from quadrangle inequality. This allows us to prune the search space on each step, reducing the running time to $O(n^2)$. If you calculate $f_{i,j}$ in order of increasing j and decreasing i, then at the moment of calculating $f_{i,j}$, values of $A_{i,j-1}$ and $A_{i+1,j}$ are already known, so you can only check $t \in [A_{i,j-1}, A_{i+1,j}]$.

This optimization results in $O(n^2)$ running time (look at the diagonals of matrix $A_{i,j}$).

The same optimization can be applied to DP on subsegments: if we set $dp[i][j] = cost(i,j) + \min_k(dp[i][k] + dp[k+1][j])$ and set $A_{i,j}$ to be the optimal k in this definition, then the same speedup is true and results in $O(n^2)$ running time.

5.4 Convex Hull Trick

This optimization requires $cost(i,j)$ to have a specific form: $cost(i,j) = x_i + y_j + a_i \cdot b_j$, so for fixed i it's a linear function of b_j. This is often the case.

Suppose $cost(i,j) = \left(\sum_{t=i}^{j} a_t\right)^2 = (S_j - S_{i-1})^2 = S_j^2 - 2S_jS_{i-1} + S_i^2$. *Convex Hull Trick* is a data structure that allows you to do these two operations:

1. Add a line $ax + b$ into the current set.
2. For a given x_0 find a maximum value of $ax_0 + b$ among all lines.

Both these operations can be implemented in $O(\log n)$. If there's some additional information, for example, x_0 are increasing, or a coefficients are increasing, or both, we can do a speedup up to $O(1)$ for both queries. This way we get $O(nk \log n)$ or $O(nk)$ running time for the original DP problem.

5.5 «Aliens» trick

This trick was known before, but became a lot more widespread after IOI 2016 problem «Aliens». Let $g(k) = f(n,k)$ — the cost of the division into k segments. If the function $g(k)$ is *convex*: $g(k-1) - g(k) \geq g(k) - g(k+1)$, then we can apply the following trick.

Let the function $h(C)$ be the optimal value for division into segments with no restriction on the number of segments, but each segment costs C. In other words, $h(C) = \min_k g(k) + kC$. Often we can compute $h(C)$ efficiently, for example, in case of convex hull trick optimization. Also, define $opt(C)$ as the optimal number of segments in a decomposition that minimizes $h(C)$.

It's evident that if $C = 0$ we should take n segments, and if $C = +\infty$, then we should take 1 segment. If $g(k)$ is convex, then $opt(C)$ will be non-increasing and we can use binary search to find the minimum C where $opt(C) \leq k$. This optimization takes off the logarithmic factor off the solution, if $h(C)$ and $opt(C)$ can be calculated efficiently.