Dynamic programming optimizations

Maxim Akhmedov
Moscow State University, Yandex

January 27th, 2017

This text contains the brief description of several dynamic programming optimizations techniques that often appear on programming competitions.

1 Optimum monotonocity / binary search / two pointers

Problem: professor lives in an n floor building and has k transistors. He knows that there exists some floor i ($1 \le i \le n-1$) that if he throws a transistor from floor i or lower it won't be broken, and if he throws it from floor $i+1$ or higher, it will definitely be broken. In which smallest number of throws professor can determine that critical i?

DP: $D[n][k]$ is a minimum number of throws needed to professor to find out the critical i if he knows that floor l is still OK, floor r is not OK, $r - l = n$ and he has k transistors left. Then:

- $D[1][k] = 0$ since we already found the critical i;
- $D[n][k] = \min\limits_{1 \le j \le n-1} \max(D[j][k-1], D[n-j][k])$;

This is an $O(n^2 k)$ DP.

Optimization 1: $f(j) = D[j][k-1]$ is decreasing, $g(j) = D[n-j][k]$ is increasing, hence $\max(D[j][k-1], D[n-j][k])$ decreases till some moment (while $D[j][k-1] \ge D[n-j][k]$) and then increases (in this statement terms "increasing/decreasing" allow equality, i.e. they are not strict). Hence, we may find an optimum point $optj[n][k]$ as a root of a function $f(j) - g(j) = D[j][k-1] - D[n-j][k]$ using the binary search. This leads to an $O(nk \log n)$ solution.

Optimization 2: when we move from n to $n+1$, the function $f(j) = D[j][k-1]$ stays the same and $g(j) = D[n-j][k]$ is replaced with $g^*(j) = D[n+1-j][k]$. Note that $g^*(j) \ge g(j)$, hence $optj[n+1][k] \ge optj[n][k]$. In order to calculate $optj[n+1][k]$, assign it to $optj[n][k]$ and increase it until $f(optj[n+1][k])$ becomes smaller than $g(optj[n+1][k])$. This leads to an $O(nk)$ solution.

2 Convex hull trick (linear version)

Problem: You are given n numbers $x_1 < x_2 < \ldots < x_n$ and a constant C. Choose some subsequence of them $y_1, \ldots, y_k$ such that $y_1 = x_1$, $y_k = x_k$ and the value $\sum\limits_{i=1}^{k-1} (y_{i+1} - y_i)^2 + Ck$ is as small as possible.

DP: $D[i]$ is the smallest possible $\sum\limits_{i=1}^{j-1} (y_{j+1} - y_j)^2 + Cj$ if $y_j = x_i$ for some j. Then:

- $D[1] = -C$;
- $D[i] = \min\limits_{1 \le j \le i-1} (D[j] + (x_i - x_j)^2 + C)$;

This is an $O(n^2)$ solution.

Optimization 1:

$$\begin{aligned} D[i] = \min_{1 \le j \le i-1} (D[j] + (x_i - x_j)^2 + C) = \\ x_i^2 + C + \min_{1 \le j \le i-1} (D[j] + x_j^2 - 2x_i x_j) = \\ x_i^2 + C + \min_{1 \le j \le i-1} (x_j, D[j] + x_j^2) \cdot (-x_i, 1) \\ x_i^2 + C + \max_{1 \le j \le i-1} (x_j, D[j] + x_j^2) \cdot (x_i, -1) \end{aligned}$$

Let $\vec{P_j} = (x_j, D[j] + x_j^2)$. Keep the lower hull of $\vec{P_j}$. The j such that $\vec{P_j} \cdot \vec{v_i} \to \max$ is always some point of a convex hull of $\{P_j\}$; namely, the lower hull of those points because the y-component of a vector $\vec{v_i}$ in our case is negative.

Lower hull may be kept in the stack. New points are added to the right of the old ones (since x_j increases), so the stack may be recalculated in amortized $O(1)$ (similar to the Andrew monotone chain algorithm).

Optimum j may be find via the binary search over the convex hull since $(\vec{P_j}, \vec{v_i})$ increases up to some moment and then decreasing over all j belonging to the lower hull.

The complexity is $O(n \log n)$.

Optimization 2: note that vector $\vec{v_i}$ also moves to the right (its x-component increases). It means that the pointer on the optimum point on lower hull also moves only to the right. Keep the optimum pointer $opt[i]$ and try to move it to the right while it is profitable when moving from i to $i + 1$.

The complexity is $O(n)$.

3 Divide and Conquer optimization

Problem: You are given n integers $x_1, x_2, \ldots, x_n$. Divide them into k consecutive groups such that $\sum\limits_{i=1}^{k} w_i \log w_i \to \min$ where w_i is the sum in the k group.

DP: $DP[i][j]$ is the minimum penalty for dividing first j numbers into i groups. Then:

- $DP[0][0] = 0$;
- $DP[i][j] = \min_{0 \le z \le j-1}(DP[i-1][z] + (S[j] - S[z]) \log(S[j] - S[z]))$ where $S_j = x_1 + x_2 + \ldots + x_j$;

This is an $O(n^2 k)$ soluition.

Optimization: notice the important property of optimal point monotonicity. Denote as $optz[i][j]$ the value of z that is the optimum for the expression above.

Lemma: $optz[i][j] \le optz[i][j+1]$.

Lemma proof: use induction and Karamata's inequality.

Let's calculate the i-th layer of DP using the following recursive procedure:

- void $calc(i, l, r)$
- Pre-requisite: $optz[i][l-1]$ and $optz[i][r+1]$ are already calculated (let $optz[i][0] = 1$ and $optz[i][n+1] = n$);
- If $l > r$, return;
- Let $m = \lfloor (l+r)/2 \rfloor$, calculate $optz[i][m]$ by iterating with z between $optz[i][l]$ and $optz[i][r]$;
- Make a recursive call of $calc(i, l, m-1)$ and $calc(i, m+1, r)$.

In total, each level of recursion works in $O(n)$ and there are $\log n$ recursion levels. Hence, everything works in $O(nk \log n)$.

4 Knuth optimization

Problem: You are given values $x_1, x_2, \ldots, x_n$. Organize them into a binary tree (without reordering) so that the sum of the values multiplied by their depths in the tree is as small as possible.

DP: $D[l][r]$ is the cost of the best tree that may be built over the elements from l-th to r-th.

- $D[l][l-1] = x_l$;
- $D[l][r] = \min\limits_{l \le i \le r} (D[l][i-1] + D[i+1][r] + (x_l + x_{l+1} + \ldots + x_r)) = \min\limits_{l \le i \le r} (D[l][i-1] + D[i+1][r] + (S[r] - S[l-1]))$ where $S[r] = x_1 + x_2 + \ldots + x_r$.

This is an $O(n^3)$ DP.

Optimization: Consider $opti[l][r]$ to be the optimum value of i for the formula above.

Lemma: $opti[l][r-1] \le opti[l][r] \le opti[l+1][r]$.

Lemma proof: prove it by yourself. Prove the $opti[l][r-1] \le opti[l][r]$ by contradiction, consider the right path inside the optimum binary search tree. and find the contradiction.

Now, calculate DP in order of increasing $r - l$. Iterate with i only in range $[opti[l][r-1], opti[l+1][r]]$. Thus, the running time for a fixed $r - l = d$ will be proportional to $opti[d+1][2] - opti[d][1] + opti[d+2][3] - opti[d+1][2] + \ldots + opti[n][n-d+1] - opti[n-1][n-d] = opti[n][n-d+1] - opti[d][1] = O(n)$. So, the overall running time is $O(n^2)$.

5 Lagrange optimization

Problem: IOI2016 Aliens [http://ioinformatics.org/locations/ioi16/contest/day2/aliens.pdf]

DP and optimization: Refer to the analysis of the contest [http://ioinformatics.org/locations/ioi16/contest/IOI2016_analysis.pdf]