Last update: December 7, 2023 Original

Knuth's Optimization

Knuth's optimization, also known as the Knuth-Yao Speedup, is a special case of dynamic programming on ranges, that can optimize the time complexity of solutions by a linear factor, from $O(n^3)$ for standard range DP to $O(n^2)$.

Conditions

The Speedup is applied for transitions of the form

$$dp(i,j) = \min_{i \le k < j} [dp(i,k) + dp(k+1,j) + C(i,j)].$$

Similar to divide and conquer DP, let $opt(i,j)$ be the value of k that minimizes the expression in the transition (opt is referred to as the "optimal splitting point" further in this article). The optimization requires that the following holds:

$$opt(i,j-1) \le opt(i,j) \le opt(i+1,j).$$

We can show that it is true when the cost function C satisfies the following conditions for $a \le b \le c \le d$:

1. $C(b,c) \le C(a,d)$;
2. $C(a,c) + C(b,d) \le C(a,d) + C(b,c)$ (the quadrangle inequality [QI]).

This result is proved further below.

Algorithm

Let's process the dp states in such a way that we calculate $dp(i,j-1)$ and $dp(i+1,j)$ before $dp(i,j)$, and in doing so we also calculate $opt(i,j-1)$ and $opt(i+1,j)$. Then for calculating $opt(i,j)$, instead of testing values of k from i to $j-1$, we only need to test from $opt(i,j-1)$ to $opt(i+1,j)$. To process (i,j) pairs in this order it is sufficient to use nested for loops in which i goes from the maximum value to the minimum one and j goes from $i+1$ to the maximum value.

Generic implementation

Though implementation varies, here's a fairly generic example. The structure of the code is almost identical to that of Range DP.

```
int solve() {
    int N;
    ... // read N and input
    int dp[N][N], opt[N][N];

    auto C = [&](int i, int j) {
        ... // Implement cost function C.
    };

    for (int i = 0; i < N; i++) {
        opt[i][i] = i;
        ... // Initialize dp[i][i] according to the problem
    }

    for (int i = N-2; i >= 0; i--) {
        for (int j = i+1; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j-1]; k <= min(j-1, opt[i+1][j]); k++) {
                if (mn >= dp[i][k] + dp[k+1][j] + cost) {
                    opt[i][j] = k;
                    mn = dp[i][k] + dp[k+1][j] + cost;
```

```
                }
            }
            dp[i][j] = mn;
        }
    }

    return dp[0][N-1];
}
```

Complexity

A complexity of the algorithm can be estimated as the following sum:

$$\sum_{i=1}^{N}\sum_{j=i+1}^{N}[opt(i+1,j)-opt(i,j-1)] = \sum_{i=1}^{N}\sum_{j=i}^{N-1}[opt(i+1,j+1)-opt(i,j)].$$

As you see, most of the terms in this expression cancel each other out, except for positive terms with $j = N$ and negative terms with $i = 1$. Thus, the whole sum can be estimated as

$$\sum_{k=1}^{N}[opt(k,N)-opt(1,k)] = O(n^2),$$

rather than $O(n^3)$ as it would be if we were using a regular range DP.

On practice

The most common application of Knuth's optimization is in Range DP, with the given transition. The only difficulty is in proving that the cost function satisfies the given conditions. The simplest case is when the cost function $C(i,j)$ is simply the sum of the elements of the subarray $S[i,i+1,\ldots,j]$ for some array (depending on the question). However, they can be more complicated at times.

Note that more than the conditions on the dp transition and the cost function, the key to this optimization is the inequality on the optimum splitting point. In some problems, such as the optimal binary search tree problem (which is, incidentally, the original problem for which this optimization was developed), the transitions and cost functions will be less obvious, however, one can still prove that $opt(i,j-1) \le opt(i,j) \le opt(i+1,j)$, and thus, use this optimization.

Proof of correctness

To prove the correctness of this algorithm in terms of $C(i,j)$ conditions, it suffices to prove that

$$opt(i,j-1) \le opt(i,j) \le opt(i+1,j)$$

assuming the given conditions are satisfied.

Lemma

$dp(i,j)$ also satisfies the quadrangle inequality, given the conditions of the problem are satisfied.

Proof ›

Now, consider the following setup. We have 2 indices $i \le p \le q < j$. Set $dp_k = C(i,j) + dp(i,k) + dp(k+1,j)$.

Suppose we show that

$$dp_p(i,j-1) \ge dp_q(i,j-1) \implies dp_p(i,j) \ge dp_q(i,j).$$

Setting $q = opt(i, j-1)$, by definition, $dp_p(i, j-1) \geq dp_q(i, j-1)$. Therefore, applying the inequality to all $i \leq p \leq q$, we can infer that $opt(i, j)$ is at least as much as $opt(i, j-1)$, proving the first half of the inequality.

Now, using the QI on some indices $p+1 \leq q+1 \leq j-1 \leq j$, we get

$$\begin{aligned}
& dp(p+1, j-1) + dp(q+1, j) \leq dp(q+1, j-1) + dp(p+1, j) \\
\implies & (dp(i, p) + dp(p+1, j-1) + C(i, j-1)) + (dp(i, q) + dp(q+1, j) + C(i, j)) \\
\leq & (dp(i, q) + dp(q+1, j-1) + C(i, j-1)) + (dp(i, p) + dp(p+1, j) + C(i, j)) \\
\implies & dp_p(i, j-1) + dp_q(i, j) \leq dp_p(i, j) + dp_q(i, j-1) \\
\implies & dp_p(i, j-1) - dp_q(i, j-1) \leq dp_p(i, j) - dp_q(i, j)
\end{aligned}$$

Finally,

$$\begin{aligned}
& dp_p(i, j-1) \geq dp_q(i, j-1) \\
\implies & 0 \leq dp_p(i, j-1) - dp_q(i, j-1) \leq dp_p(i, j) - dp_q(i, j) \\
\implies & dp_p(i, j) \geq dp_q(i, j)
\end{aligned}$$

This proves the first part of the inequality, i.e., $opt(i, j-1) \leq opt(i, j)$. The second part $opt(i, j) \leq opt(i+1, j)$ can be shown with the same idea, starting with the inequality $dp(i, p) + dp(i+1, q) \leq dp(i+1, p) + dp(i, q)$.

This completes the proof.

Practice Problems

- UVA - Cutting Sticks
- UVA - Prefix Codes
- SPOJ - Breaking String
- UVA - Optimal Binary Search Tree

References

- Geeksforgeeks Article
- Doc on DP Speedups
- Efficient Dynamic Programming Using Quadrangle Inequalities

Contributors:
adamant-pwn (74.36%) omkarbajaj073 (25.13%) mhayter (0.51%)