Divide and Conquer DP

Table of Contents

Divide and Conquer is a dynamic programming optimization.

Preconditions

Some dynamic programming problems have a recurrence of this form:

$$dp(i, j) = \min_{k \leq j} \{ dp(i-1, k) + C(k, j) \}$$

where $C(k, j)$ is some cost function.

Say $1 \leq i \leq n$ and $1 \leq j \leq m$, and evaluating C takes $O(1)$ time. Straightforward evaluation of the above recurrence is $O(nm^2)$. There are $n \times m$ states, and m transitions for each state.

Let $opt(i, j)$ be the value of k that minimizes the above expression. If $opt(i, j) \leq opt(i, j+1)$ for all i, j, then we can apply divide-and-conquer DP. This known as the *monotonicity condition*. The optimal "splitting point" for a fixed i increases as j increases.

This lets us solve for all states more efficiently. Say we compute $opt(i, j)$ for some fixed i and j. Then for any $j' < j$ we know that $opt(i, j') \leq opt(i, j)$. This means when computing $opt(i, j')$, we don't have to consider as many splitting points!

To minimize the runtime, we apply the idea behind divide and conquer. First, compute $opt(i, n/2)$. Then, compute $opt(i, n/4)$, knowing that it is less than or equal to $opt(i, n/2)$ and $opt(i, 3n/4)$ knowing that it is greater than or equal to $opt(i, n/2)$. By recursively keeping track of the lower and upper bounds on opt, we reach a $O(mn \log n)$ runtime. Each possible value of $opt(i, j)$ only appears in $\log n$ different nodes.

Note that it doesn't matter how "balanced" $opt(i, j)$ is. Across a fixed level, each value of k is used at most twice, and there are at most $\log n$ levels.

Generic implementation

Even though implementation varies based on problem, here's a fairly generic template. The function `compute` computes one row i of states `dp_cur`, given the previous row $i - 1$ of states `dp_before`. It has to be called with `compute(0, n-1, 0, n-1)`.

```cpp
int n;
long long C(int i, int j);
vector<long long> dp_before(n), dp_cur(n);

// compute dp_cur[l], ... dp_cur[r] (inclusive
void compute(int l, int r, int optl, int optr)
{
    if (l > r)
        return;
    int mid = (l + r) >> 1;
    pair<long long, int> best = {INF, -1};

    for (int k = optl; k <= min(mid, optr); k+
        best = min(best, {dp_before[k - 1] + C
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}
```

Things to look out for

The greatest difficulty with Divide and Conquer DP problems is proving the monotonicity of opt. Many Divide and Conquer DP problems can also be solved with the Convex Hull trick or vice-versa. It is useful to know and understand both!

Practice Problems

- Codeforces - Ciel and Gondolas (Be careful with I/O!)
- SPOJ - LARMY
- Codechef - CHEFAOR
- Hackerrank - Guardians of the Lunatics
- ACM ICPC World Finals 2017 - Money

References

- Quora Answer by Michael Levin
- Video Tutorial by "Sothe" the Algorithm Wolf