Dynamic Programming

Dynamic programming is a powerful technique that you should absolutely add into your arsenal. The basic idea is induction: once you have an answer to previous "smaller" problems, you can use them to solve subsequent problems. Let us see a few concrete examples.

Interval Scheduling Problem

We have seen the (unweighted) problem in the last class. This time the difference is that there is a weight $w : J \to \mathbb{R}^+$ associated with the jobs. We want a subset $J' \subseteq J$ of non-intersecting jobs so that the total weight $\sum_{j \in J'} w(j)$ is maximized.

Let us order the jobs by their increasing deadlines $d(j)$. For each job j, we also define a *predecessor* $p(j)$, which is the latest job whose interval does not overlap with j. (In case there is no such job, just let $p(j) = \emptyset$.

Look at the last job n. In the unknown optimal solution, there can be only two possibilites: either I take it or I don't. If I take it, then I know for sure that jobs $p(n) + 1, \cdots n - 1$ cannot be part of the optimal solution. If I don't, then essentially I just have to choose a subset of jobs from the set $\{1, \cdots, n - 1\}$. This observation tells me that

$$OPT = \max\{q(p(n)) + w(n), q(n - 1)\}.$$

where $q(t)$ means the optimal solution using only jobs in $\{1, \cdots, t\}$. The important thing here is that we have now two smaller problems: $\{1, \cdots, n - 1\}$ and $\{1, \cdots, p(n)\}$. So this implies that I can solve the entire problem by recursion.

But here the recursion will be wasteful—and the idea of dynamic programming kicks in. We will solve the problem in a "bottom-up" manner. (We assume that $q(0) = 0$ and $q(\emptyset) = 0$.)

For $i = 1$ to n
    $q(i) = \max\{q(p(i)) + w(i), q(i - 1)\}$

It is obvious that $q(n)$ gives the final solution.

Knapsack Problem

Imagine that we are given a set of items I, each with a profit $p(i)$ and a size $s(i)$. We are also given a budget B. We want to take a subset $I' \subseteq I$ of items so that the total profit $\sum_{i \in I'} p(i)$ is maximized while respecting the budget $\sum_{i \in I'} s(i) \leq B$.

It may be very tempting to attack the problem by greedy: sorting the items by their densities $p(i)/s(i)$ and take them one by one as long as the budget B is not exceeded. This strategy should fit our intuition well: high-profit-small-sized items should be prioritised. Unfortunately, this idea does not work.

Instead we will attack by dynamic programming. Order the jobs arbirarily and let us ask a (seemingly naive) question: what would be the best solution among jobs $\{1, \cdots, n\}$ using *exactly* the budget $B' \leq B$? Let us do as the last time: look at the last item n. Either I take it or not. If I take it, I should have a solution among $\{1, \cdots, n-1\}$ using exactly the budget $B' - s(n)$; if I don't, I should have a solution among $\{1, \cdots n-1\}$ using exactly the budget B' (observe that again we have two smaller problems).

So let us define a table T. $T(x, y)$ means the most profitable solution using exactly the budget of y among the jobs $\{1, \cdots x\}$. The above discussion implies that $T(n, B') = \max\{T(n-1, B' - s(n)) + p(n), T(n-1, B')\}$.

Again we don't want recursion. We will fill in the table row by row. The first row is trivial. For row $x > 1$, we know that

$$T(x, y) = \{T(x-1, y - s(n)) + p(x), T(x-1, y)\}.$$

After we fill in the whole table, the optimal solution is then the best solution in the last row $T(n, y)$ among $1 \leq y \leq B$. The running time of this algorithm is $O(nB)$. Such a running time is called *pseudo-polynomial* time.

Shortest Path with Negative Distance

Given a directed graph $G = (V, E)$ with distance $d : E \to \mathbb{R}$, we want a shortest path from $s \in V$ to $t \in V$. If all distances are non-negative, you have already learned Dijkstra's algorithm (which is itself a greedy). But it does not work when the distances can be negative.

Of course this problem is ill-posed if there is a directed negative cycle. So let us assume that this does not happen.

We now explain the Bellman-Ford algorithm. Again we want to build a table T. Here $T(u, \tau)$ means the shortest distance from vertex u to t using *at most* τ hops (if there is no such path, we assume that $T(u, \tau) = \infty$).

Imagine that we have filled in columns $1, \cdots, \tau - 1$. How should we fill in $T(u, \tau)$? If there is a path from u to t, it has to use one of its outgoing edges (u, v). This suggests that $T(u, \tau) = \min_{(u,v) \in E} d(u, v) + \tau(v, \tau - 1)$.

So if we build the table T up to column $\tau - 1$, we should be able to fill in the τ-th column. As we assume that there is no negative cycle, the optimal solution uses at most $n - 1$ hops. Therefore, after we fill in the column $n - 1$, $T(s, n - 1)$ gives the optimal solution.

The running time of the algorithm is proportional to the size of the table and the time we need to fill in one cell. In the worst case, each cell needs $O(n)$ time. So this suggests that the total running time is $O(n^3)$. But by being a little bit more careful, you should be able to show that the running time is a better $O(nm)$.