Lecture 17: Dynamic programming

We continue the discussion of dynamic programming. We discussed the following problems: weighted interval scheduling, subset sum, 0-1 Knapsack problem.

Weighted interval scheduling is covered in the textbook (section 6.1). You should read and understand it.

The subset sum problem is covered in section 6.4 of the textbook. You should read and understand it. Note that the subset sum problem asks for the maximum weight that is not over some given bound $W$, one can obtain by choosing a subset of items.

In the 0-1 Knapsack problem, we have $n$ items, each weight $w_i$ (which is an integer) pound and worth $v_i$ dollars. The goal is find a subset of items as valuable as possible but no more than $W$ pounds (here $W$ is a given integer). For each item, we must either take or not take and thus is called 0-1 Knapsack.

Suppose the optimal solution contains $k$ items, $b_1, b_2, \ldots, b_k$. Suppose we remove $b_k$, then we have the following property: the remaining items, $b_1, b_2, \ldots, b_{k-1}$ must the most valuable items under the constraints that total weight is no more than $W - w_k$. Otherwise, we have a contradiction that $b_1, b_2, \ldots, b_k$ are the most valuable items with no more than $W$ total weight.

Now we define $M[i, w]$ as the highest value (in dollar) when we choose items from 1 to $i$, and total weight is no more than $w$. Clearly, the solution to our Knapsack problem is simply $M[n, W]$. It is easy to see that $M[0, w] = 0$, and $M[i, 0] = 0$ for all $i/w$. When $i \geq 1$ and $w > 0$, we have: $M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w])$ if $w_i < w$, and otherwise $M[i, w] = M[i-1, w]$. This is because we either take item $i$ or not. If we take item $i$, then the remaining capacity of Knapsack is reduced by the weight of item $i$. If not, the capacity remains unchanged. In either case, we should pack as valuable as we can using the remaining items.

1: for $w = 0$ to $W$ do
2: \hspace{1cm} $M[0, w] = 0$
3: end for
4: for $i = 1$ to $n$ do
5: \hspace{1cm} $M[i, 0] = 0$
6: for $w = 1$ to $W$ do
7: \hspace{2cm} if $w_i \leq w$ then
8: \hspace{3cm} $M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w])$
9: \hspace{2cm} else
10: \hspace{3cm} $M[i, w] = M[i-1, w]$
11: \hspace{2cm} end if
12: end for
13: end for
14: return $M[n, W]$.

Running time: we have $O(nW)$ cells in $M$ array, each taking $O(1)$ to compute. So the total time is $O(nW)$. Is this a polynomial-time algorithm? Not quite but it works well when $W$ is relatively small.