Lecture 2: Knapsack Dynamic Programs

Objectives of this lecture

- Familiarize with dynamic programming: states, transitions, and base cases.
- Know the formulation of knapsack, subset sum, and its many variations.
- Understand how DP transitions based on knapsack can be optimized.
- Be aware of connections between states and vertices, via connections between knapsack and shortest paths.

1 Philosophy

Knapsack problems seek to maximize the value of some subset of objects subject to the sum of their attributes not exceeding some given limit. When this sum is integers in some limited range, it's a natural dynamic program state.

A dynamic program consist of:

1. States.
2. Transition function.
3. Base case.

The most important of these is the DP state. There are several ways to come up with states:

- 'Seen this structure before' (aka. came to this sequence of lectures).
- Design a brute force search, then reduce the number of inputs to the search function until they are few enough to be remembered.

Once you have states/transition/base case down, it's useful to:

1. Justify correctness of the transitions, specifically how they cover how answers to the states are pieced together from answers to other states.
2. Analyze the running time of computing all the states using the transitions.
3. Describe how to read the answer from the states once you compute them: for 'optimum value / number of values', this is often just one state, for the actual solution one often needs to trace the states backwards using the transitions again.

Knapsack DPs often have simple transition functions consisting of single for loops. So for many variants of knapsack, first-principled optimizations lead to significant runtime gains.

2 Example Problems

Problem 1: subset sum with replaceable items

Is it possible to make Y dollars using coins of value $x_1 \dots x_n$ (each $x_i \geq 0$), each of which can only be used an arbitrary number of times.

$$DP[y] = \begin{cases} 1 & \text{if it's possible to make value } y \\ 0 & \text{if it's not possible to make value } y \end{cases}$$

The base case here is $DP[0] = 1$, since state 0 is reachable.

The transition function is that to make value y, we must have used some coin with value x_i. Then it must be possible to make value $y - x_i$ as well. So we get

$$DP[y] = \vee_{i:x_i \leq y} DP[y - x_i]$$

where $\vee$ denotes OR.

To analyze the performance of this algorithm, we will use asymptotic complexity. Recall that big-O notation, in its simplest form, allows us to ignore constants, and only track the leading term of the complexity. Here we only need to track states between 0 to Y, so $O(Y)$ states, for a total running time of $O(nY)$.

Things get trickier when each coin can only be used once:

Problem 2: subset sum with irreplaceable items

Decide if it's possible to make Y dollars using coins of value $x_1 \dots x_n$ (each $x_i \geq 0$), each of which can be used at most once.

we define a 2-dimensional state:

$$DP[i][y] = \begin{cases} 1 & \text{if it's possible to make value } y \text{ using coins } 1 \dots i\text{, each at most once} \\ 0 & \text{otherwise} \end{cases}$$

Once we get this state, the base case and transition kind of writes themselves. For base case, we have $DP[0,0] = 1$, while the transition becomes just checking whether item i can be used:

$$DP[i][y] = DP[i-1][y] \vee DP[i-1][y - x_i]$$

where $\vee$ denotes OR. Note that the second case should only be considered if $y \geq x_i$.

The running time of this is still $O(Yn)$, but the memory usage becomes $O(Yn)$ as well. To get that down, the conceptually easier way is to observe that at any given point in time, we only need $DP[i][*]$ and $DP[i-1][*]$, so we can use a rolling table where only two rows of the DP table are kept at any given point of time.

That still adds a fair amount of extra code. The even simpler way to realize this is to run things backward, using the fact that $x_i \geq 0$. That is, we perform the update

$$DP[y] = DP[y] \vee DP[y - x_i]$$

in *decreasing* order of y. What happens is that the suffix of the array (y and after) becomes $DP[i][y]$, while the prefix stays the same as $DP[i-1][y]$.

Problem 3: subset sum, variable amounts

Is it possible to make Y dollars using coins of value $x_1 \ldots x_n$ (each $x_i \geq 0$), where coin i can be used between 0 and n_i times?

The simplest way to solve this is to turn coin i into n_i separate coins, each of which can be used at most once. That leads to a runtime of

$$O\left(\sum_i n_i \cdot Y\right).$$

With binary representations of numbers, we can do better by creating copies that correspond to powers of 2. Say we have 7 coins with value x, we can divide them into

$$x, 2x, 4x$$

so that any value between 0 and $7x$ can be made using a subset of these coins. There is some slight trickiness to this for general values of n_i, e.g. $10 = 2 + 8$ does not allow one to create 1. There the method is to find the largest sum of powers of 2 that's at most n_i, create powers of 2 up to there, and then create one coin corresponding to the rest of the sum.

To get the runtime back to $O(nY)$, consider the 'backward filling' routine above: instead of checking whether $y - x_i$ is 1, we need to check whether any of

$$y - x_i, y - 2x_i, \ldots y - n_i \cdot x_i$$

are 1s. **Note this already involves the implicit backward table filling idea from Problem 2 above.** Naively, this still incurs an extra factor of n_i, but now think about what happens if we check y, $y - x_i$, and etc in that order. That is, we only work on the indices with a particular remainder modulo x_i. Then as we move 'down' the y by x_i, the only 'new' entry that we need to consider is

$$y - x_i - n_i \cdot x_i.$$

In other words, the set of indices that we consider is gradually moving downward. All we need to do is to track the *smallest* index that's within the range, and 1 in the current DP table. This leads to a routine that performs $O(1)$ per transition, for a total running time of $O(nY)$.

Some questions / comments from last time:

1. WHY IS THIS PROBLEM DYNAMIC PROGRAMMING?
2. groups for projects, collaboration policy.

the variant below was not covered in class, but is in both recitation 1 and homework 1 (question 2).

Knapsack also have optimization versions, where the items have both weight and value, and the goal is to maximize the values of the items taken.

Problem 4: Value maximization knapsack with irreplaceable items

I have n items, each with size $x_i \geq 0$, and value v_i. Find the maximum value of a subset with size at most Y.

This is knapsack without replacement. Here instead of storing whether it's possible to make weight y, we store the maximum value of a set of items with weight y.

Formally, we let $DP[i][j]$ to denote the maximum value of a subset of $x_1 \ldots x_i$ whose total weight is y. This leads to the transition

$$DP[i][j] = \max\left\{DP[i-1][y], DP[i-1][y - x_i] + v_i.\right\}$$

Note that the answer needs to take the max over all values at most Y as well. That is, we return

$$\max_{0 \leq y \leq Y} DP[n][y].$$

The materials below (further speeding up balanced subset sum, connections with graphs) were not covered, and will not be in any assessment.

This routine above for arbitrary number of copies can also be leveraged to give faster algorithm for the second problem (coins that can only be used once) when Y is close to the total sum of the x_is, e.g. checking whether it's possible to divide up a set of coins can be divided into two even valued halves. Specifically, a complexity of

$$O\left(\left(\sum_i x_i\right)^{1.5}\right)$$

is possible by just calling a 'right' mix of the two algorithms above: Let $S = \sum_i x_i$, and note that the number of x_i s.t. $x_i > \sqrt{S}$ is at most

$$S/S^{1/2} = S^{1/2},$$

so combined with the at most $S^{1/2}$ different sizes from $1 \ldots S^{1/2}$, we get that there are at most $2S^{1/2}$ distinct item sizes.

With fast convolution routines like FFT, a runtime of $O(n \log^2 n)$ is even possible, but I don't recommend implementing that version.

Knapsack can also be combined with other problems by augmenting the nodes with extra states.

Problem 5: knapsack in a car

Get from point s to point t in a road network with n vertices and m edges, and each edge having non-negative integer time / toll values, in the fastest time while paying a total budget of at most Y.

To make progress on this problem, observe that shortest path algortihms themselves are dynamic programming based. The state is the vertex that one is at, but the transition function is evaluated 'on the fly':

1. Bellman-Ford repeatedly evalutes it for n rounds.
2. Dijkstra's algorithm picks the next one to extend from on the fly, in increasing order of distance.

In either of these cases though, we can just augmenting the state with the amount of toll that has been paid so far:

$$DP[u][y] = \text{min time needed to get to vertex } u \text{ after paying } y \text{ units of toll}$$

With Dijkstra's algorithm, this leads to an extra factor of Y on the running time, for a total of $O(mY \log n)$.

The reverse direction is also possible: in cases where ONE of the values is small, we can actually get faster knapsack algorithms using shortest path.

Problem 6: knapsack using shortest paths

Check if one can make Y dollars using an arbitrary number of coins of value $x_1 \ldots x_n$ (each $x_i \geq 0$) in $O(x_1^2 + n)$ time.

Here we want to reduce the number of states from Y to x_1. To do this, observe that if $DP[y]$ is true, then so is $DP[y + x_1]$. That is, similar to the situation earlier with multiple copies of x_i, the states with a particular remainder mod x_1 belong to the same class.

Furthermore, because we have an arbitrary number of coins with size x_1s, we can instead create the state

$$DP_{MIN}[z] = \min\left\{y : y \mod x_1 \equiv z \text{ AND } DP[y] = 1\right\}$$

That is, for each group of ys (separated by their residue mod x_1), we store the *smallest* value that is reachable.

This leads to a shortest path problem: if we add x_i to state z, the total size is now $DP[z] + x_i$, and the state that we go to is now

$$(z + x_i) \mod x_1.$$

So the transition function becomes

$$DP_{MIN}[z] \leftarrow \min\{DP_{MIN}[z], DP_{MIN}[z - x_i \mod x_1] + x_i\}$$

taken over all n coins i.

Using dense graph shortest path, this gives $O(x_1^2)$ time.