DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

The goal of this class is to give another layer to the topics taught in algorithms and data structures, specifically dynamic programming, binary search trees, and divide-and-conquer.

We start with dynamic programming because the subproblem based view taken by it extends to most subsequent problems.

For this week, we will focus on the knapsack problem, which is a ‘portfolio optimization’ problem of maximizing some subset of weights subject to some constraints on budget.

The simplest such problem is the making change problem:

**Problem 1.1.** Is it possible to make $Y$ dollars using coins of value $x_1 \ldots x_n$ (each $x_i \geq 0$), each of which can only be used an arbitrary number of times.

The tool by which we will approach these is dynamic programming. A dynamic program contains:

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

The most important of these is states. There are several ways to come up with states, the easiest of which is ‘I have seen this structure before’. The most standard knapsack state that’s often taught is:

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

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

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

where $\lor$ 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 1.2.** Is it possible to make \( Y \) dollars using coins of value \( x_1 \ldots x_n \) (each \( x_i \geq 0 \)), each of which can be used at most once.

Before jumping to the solution, it’s useful to pause a bit to think about what’s wrong with just using \( DP[i] \) as above.

The issue is that a coin can be used more than once by the transitions.

To resolve this, we define a 2-dimensional state:

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

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

\[
DP[i][y] = DP[i - 1][y] \lor DP[i - 1][y - x_i]
\]

where \( \lor \) 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 \( DP[i, \ast] \) and \( DP[i - 1, \ast] \), 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

\[
DP[y] = DP[y] \lor DP[y - x_i]
\]

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 \( DP[i - 1][y] \).

Note that in the case where the entries are 0-1, this method can be accelerated further using Bitset. This is because our representation of integers actually store 64 bits at once. There is a nice package that encapsulates such functionalities in C++, \url{https://en.cppreference.com/w/cpp/utility/bitset}. Using that representation, the code for doing one step of transition to all entries of the \( DP \) vector becomes:

\[
DP = (DP \gg x_i).
\]

Note however that this speed up is only able to cover the situation when states are yes/no: it’s not able to extend to optimization versions that require storing values in each of the DP states. In general in this class we will mention, but avoid relying on, such ‘non-robust’ speedups.

———Materials covered on Wednesday
**Problem 1.3.** 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 with the implicit backward table filling idea from Problem 1.2. 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 otherwords, 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 $O(nY)$.

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 $x_i$s, 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 FFT, a runtime of \( O(n \log^2 n) \) is even possible, but I don’t recommend implementing that version. Unfortunatley we also couldn’t find high quality data for this problem, so instead you can refer to [https://codeforces.com/contest/755/problem/F](https://codeforces.com/contest/755/problem/F), which has one more layer wrapped around outside, but also editorial + solution codes posted.

Knapsack also have optimization versions, e.g. the original ‘lootbox’ problem.

**Problem 1.4.** 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 \( DP[i][y] \) is possible, we store the maximum value that can be stored in \( DP[i][y] \).

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

Note that the answer needs to take the max over all values at most \( Y \) as well. That is, we return
\[
\max_{0 \leq y \leq Y} DP[n][y].
\]

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

**Problem 1.5.** 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 algorithms 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 evaluates 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:

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

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 1.6.** 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 \( x_1 \)s, we can instead create the state

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

That is, for each group of \( y \)'s (separted 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.

Finally, a rather specialized, but algorithmically intricate, parameterization of knapsack is when the \( x_i \)'s can be negative.

**Problem 1.7.** Check if \( n \) items with values \( x_i \) s.t. \( -X \leq x_i \leq X \) (\( x_i \) can be possibly negative) has a non-empty subset that sums to 0.

The standard number of states is \( \sum |x_i| \leq O(nX) \), as the max number that could be obtained is the sum of the positives, and the minimum is the sum of the negatives. This would lead to a running time of \( O(n^2X) \).

This can actually be made better: to about \( O(n^{1.5}X) \). More on this, as well as variants of the other problems above, will be in the first problem set.