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.

In this lecture we discuss more direct ways of speeding up dynamic programming. Specifically, we want to reorder the states so that more redundant things simply get omitted.

Problem 11.1. Given \( n \) objects with integer width \( w[i] \) and height \( h[i] \), partition them into three groups in order to minimize

\[
\text{(max total width of a group)} \times \left( \sum_{\text{groups}} \text{max height among this group} \right)
\]

in time \( O(nW^2) \), where \( W = \sum_i w[i] \).

The standard approach is to do knapsack dynamic program on the partitions of widths that are possible, but then record the total height. So the dynamic program state is a 6-tuple consisting of the current width / max heights of each group. Which means the number of states is

\[
O \left( W^3 n^3 \right)
\]

Now we need to go through all \( n \) objects in order, so the total runtime is \( O(W^3 n^4) \): note that this is quite far away from the intended \( O(W^2 n) \).

We can speed this up in several steps:

1. First, observe that if we considered the first \( i \) objects, the total width of the third group can be deduced from the sum of these \( i \) objects, and the size of the first two groups. So we only have \( W^2 \) possible combinations of widths, for a total runtime of \( O(W^2 n^4) \).

2. Next, observe that the largest element must be in some group, so that group’s height is just the max height. By symmetry we may assume that one is in group 3, so we only need to track max heights of groups 1 and 2. This gets us down to \( O(n^2) \) height combinations of these two groups, for a total cost of \( O(W^2 n^3) \).

3. Next, note that the states only store what’s reachable. Instead, we can store, for each height of the second group, and combination of widths, the min possible height of the first group. That reduces states by another factor of \( n \), to \( O(W^2 n) \) states per prefix, and a total cost of \( O(W^2 n^2) \).
The last, and most ridiculous speedup, is to observe that if we sorted the objects in decreasing order of heights, we pay toward the total heights the first time we used an object in the group. Furthermore, in the dynamic program state, we actually have whether objects have been used in the groups: it’s just whether the width of that group is non-zero. So we can incorporate both heights into a single state.

This type of techniques are very adhoc: it’s usually some way of reordering the elements (sort by some value) that lets one either set up dynamic programs, or significantly reduce the number of DP states. It’s even possible to completely alter the states in ways that still optimize things.

**Problem 11.2.** Given \( n \) positive numbers \( A[1 \ldots n] \), partition them into the maximum number of contiguous intervals (leaving nothing) so that the sum of each interval is at most the sum of the next one, in \( O(n) \) time.

At a glance, this appears to be interval based dynamic programming, on the most recent segment. That is, we use \( DP[i][j] \) to store the highest tower with

\[
i + 1 \ldots j
\]

as the bottommost layer. Then the base case is \( DP[0][i] \) gets initialized to 1, and the transition is

\[
DP[i][j] = \max_{k<i, \sum A[k+1 \ldots i] \leq \sum A[k+1 \ldots i]} DP[k][i] + 1.
\]

This takes \( O(n^3) \) time at a glance: with a bit more work, it can be done in \( O(n^2 \log n) \) time using range minimum queries.

We want to do better, which means we need to first reduce the number of states. The key observation is that for each \( j \), we only care about the largest \( i \) where \( DP[i][j] \) is possible. That is, we can instead focus on making the sum of the last group as small as possible.

That is, formally, we state

**Lemma 11.3.** For any \( j \), and any \( i_1 < i_2 \) where both \( DP[i_1][j] \) and \( DP[i_2][j] \) are feasible, we have

\[
DP[i_1][j] \leq DP[i_2][j]
\]

**Proof.** The proof is a swapping based argument: let the optimum solution where the last one is at \( i_1 \) have segments that end at

\[
x_1 = i_1 > x_2 > \ldots x_{h_1}
\]

in reverse order, and the optimum solution where the last one is at \( i_2 \) be

\[
y_1 = i_2 > y_2 > \ldots y_{h_2}
\]
Note that since $h_1 > h_2$, there must be some $k$ such that $x_k > y_k$. Furthermore, $k > 1$ by the assumption of $i_1 < i_2$. This means that we have

$$x_{k-1} < y_{k-1}$$

Then we can simply swap $x_k \ldots x_{h_1}$ into the sequence of $y$: this is because for the next group of numbers, we have

$$\text{SUM}(A[x_k+1 \ldots x_{k-1}]) \leq \text{SUM}(A[x_k+1 \ldots y_{k-1}])$$

due to $y_{k-1} > x_{k-1}$. This gives a partition that completes with $i_2, j$ that’s at least as long as the one for $i_1, j$, a contradiction with the assumption about DP values. 

So instead, we just store one value at each $j$, the minimum possible length of the last segment of a valid partition that ends at it. The transition for some $i$ is then maximizing over the last point:

$$\max \{ j \mid \text{SUM}(A[j+1 \ldots i]) \geq \text{MINLAST_LAYER}[j] \}$$

If we treat the sums as partial sums, the condition becomes

$$S[i] - S[j] \geq \text{MINLAST_LAYER}[j],$$

or upon rearranging:

$$S[i] \geq S[j] + \text{MINLAST_LAYER}[j].$$

Because the $S[i]$s are monotonic, each $j$ has an activation point that we can binary search for. Inserting that $j$ in that location, and continuing to sweep, gives an algorithm that runs in $O(n \log n)$ time.

There is also an $O(n)$ time way of doing this that forgoes the binary search.

**Problem 11.4.** There are $n$ tasks each with two components, to be completed on different days. Each task has a due date $d[i]$, and can only be started $x$ days before it’s due (for a global value of $x$). Each day, you can either do nothing, or work at most $k$ of the components. Minimize the number of days you work.


First, note that to check whether it’s feasible to complete all tasks, we just at each day, complete as many components as possible, in increasing order of due date.

The issue comes with the goal of minimizing the number of days spent working.

Note that when we pick $k$ things to do, we can pick up to $x$ items’ first component, and up to $y = k - x$ items’ second components.

We can further make the following observations about how to pick these items:

1. If we pick $y$ items’ second components, we should prioritize the ones due first. To prove this formally, consider a swapping argument: suppose $d[i] < d[j]$, and there is an optimum solution where $j2$ is done before $i2$, swapping their ordering will still allow things to work.
2. Similarly, we can guess that if we pick $x$ items’ first components, we will pick the ones due first. This is a lot more complicated to prove: once again, say we have $d[i] < d[j]$, but $j1$ is completed before $i1$. There are two cases to consider:

(a) If $j2$ is completed simultaneously, or before $i1$, then we just completely swap $i1$ with $j1$, $i2$ with $j2$, and still get a valid solution.

(b) If $j2$ is completed after $i1$, then we can just swap $i1$ and $j1$.

Note that in the first case, we cannot just swap $j1$ and $i1$: it would result in $j1$ and $j2$ happening on the same day.

With these proven, the state of incompleted tasks are now two intervals in the tasks sorted by deadlines:

1. a suffix of tasks where no components have been done.

2. another interval before that consisting of the items with first component done, no progress on the second component.

and the transition at each step is to pick how many ‘new’ items first component to complete, after which we complete the second components of as many states as possible. So the number of transitions per state is $O(n)$, and we can bound the number of states by $O(n^3)$ by noticing that there is no point starting a job more than $n$ days before it’s due (with some extra work for ‘disjoint’ intervals of jobs). The total runtime is then $O(n^4)$.

**Problem 11.5.** Given a sequence of $n$ positive integers, erase at most $k$ of them, to maximize the value of the remaining numbers when read as a single integer formed by concatenating the numbers. in $O(nk)$ time.

This is problem B of [https://codeforces.com/gym/100714](https://codeforces.com/gym/100714).

First, observe that we want a number as long as possible: so for numbers of different lengths, we want to always remove shortest numbers. So the problem simplifies to removing up to $k$ numbers, each with length $l$.

Then consider enumerating the first digit of the final number, then the second, then the third, and etc. If we removed $x$ numbers, and are at digit $t$ of the final number, we’d be looking at location

$$t + x \cdot l.$$

So our DP state is whether digit $t$ of the final number, it’s possible that we removed $x$ numbers. Among all these valid digits, we pick the one with the minimum digit, and only advance on that one. This gives $O(nk)$ states, and a runtime of $O(nk)$. 
