Chapter 1

Lecture 4

1.1 Dynamic Programming

Problem: Weighted Interval Scheduling

Given: Intervals 1...n where i has start time s_i, finish time f_i and i has weight w_i
Goal: Compute largest weight set of disjoint intervals

We could try to solve it by a Greedy Algorithm but Sorting by s_i or w_i doesn't work. We can start by sorting f_i where $f_1 \leq f_2 \leq ... \leq f_n$.
We call the optimal solution OPT.
Simple Observation: Look at the last interval. Either OPT contains it or it does not.
From this, we can say thta OPT = the best of either the best solution containing n and the best solution not containing n.
We can now set $P(j)$ = the last interval $i < j$ such that i and j are disjoint. So $P(j)$ is the closest interval before j that does not overlap with j. Now we can make a function for OPT where $OPT(j)$ = value of the optimal solution using only the first j intervals. But we also want to get the actual intervals and not just the maximum value. Our function OPT only returns the max value. So we can define $O(j)$ = the solution set of intervals.

Now we can try to write $O(n)$ for the first equation on the page.

$$OPT(n) = max\{OPT(P(n)) + w_n, OPT(n_1)\}$$

Now we have a recurrence that can work with any interval j, not just the last interval n like in the formula.

Psuedocode

```
ComputeOPT(j):
    if j = 0 then 0
    else
        return recurrence;
O(j):
    if j = 0 return {}
    else
        if w[j] + OPT[P[j] >= OPT[j-1]
            return {j} U O[P[j]]
        else
            return O[j-1]
```

1.2 Proving Correctness

We can use induction because dynamic programming is basically an extension of induction into an algorithm.
Base case: $OPT(0)$
Inductive step: assume $OPT(0)...OPT(j-1)$ are correct
Our recurrence IS just our inductive step. So we just need to argue the correctness of the sentence. We can simply do this in an one sentence explanation of each part of the recurrence.

1.3 Running Time

Running time: $T(n) = T(n-2) + T(n-1)$
If that looks familiar, it's because it's the Fibonnaci sequence. And we know that this method of computing it takes exponential time, so our algorithm must be exponential time at least. But we can do better with a simple trick.

Definition: Memoization

Save the function results, indexing them by the input value so that no computation has to be repeated.

But we're not going to use memoization because there's an even simpler way to do it iteratively.

```
Set OPT[0] = 0
For j = 1 to n
    OPT[j] = max{w[j] + OPT[P[j]], OPT[j-1]}
return OPT[n]
```

And the runtime of this would be $O(n)$ as we are just going through the array once.

But what about the runtime of $P(j)$? To use brute force, we can trivially do it in $O(n^2)$.

But it's actually possible to do it in $O(nlogn)$. *Hint: we start by sorting the values*

Finally, to keep track of the actual set of intervals, we can keep a "back pointer" array to show which choice we chose in the recurrence on each step. Then, once we get the answer, we can backtrace using these pointers to get the set of intervals.
We can do this in once sentence on homeworks/tests by simply stating "we can trace back inthe array to find the optimal solution".

1.4 Dynamic Programming Principles

- Subproblems - creating and building them up
- Compute the solution to the subproblem from previous subproblems (aka the Recurrence)

Definition: Dynamic Programming

Dynamic: Cool sounding name that doesn't make it sound like a theory
Programming: Making a list of values (such as used in "TV Programming")

Be careful when explaining the array created! Don't say "best of j ", say "value of the best solution given only the first j intervals".