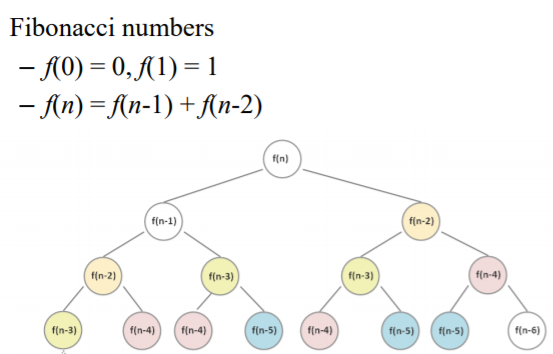
**Dynamic Programming**

*Lecture 17:*

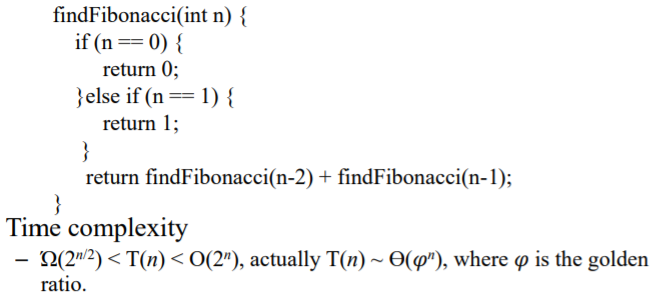
What is Dynamic Programming?

When you try to solve a problem, you first solve sub-problems, remember their solutions and then reuse them receptively.

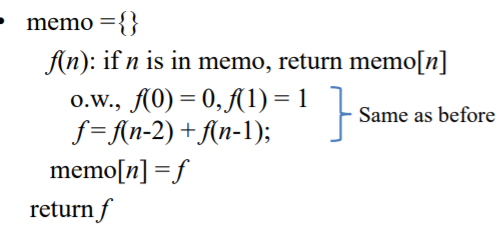
Example: Fibonacci numbers



The normal way to solve it is with a recursive function:



Now if we add Dynamic Programming to the mix, it gets much faster:



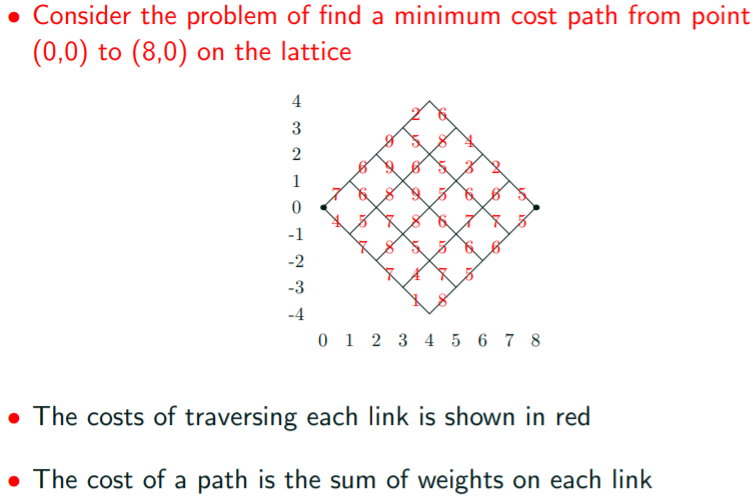
This way we only have to compute f(k) once!

Runtime = number of sub-problems \* time per sub-problem.

Therefore the time complexity is Big-Theta (n)

(Sace complexity = Big-Theta(n) again).

Another example: Toy Problem



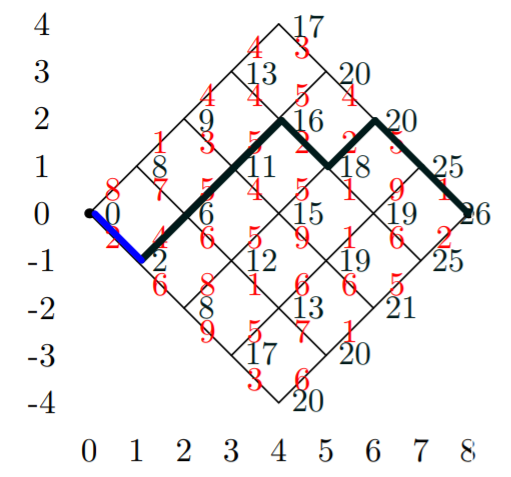
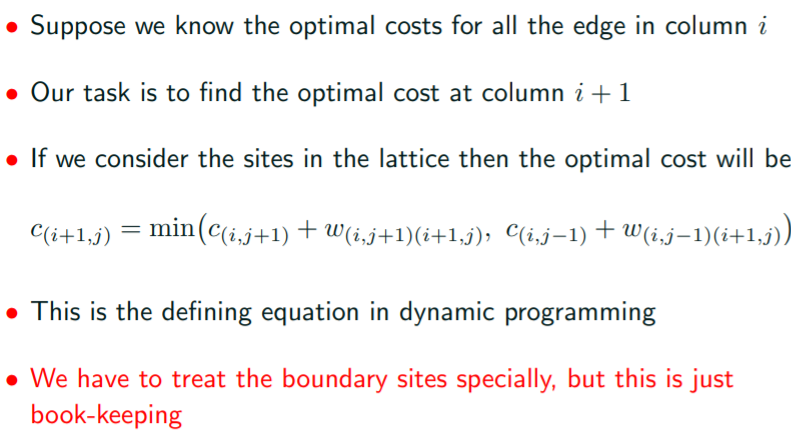
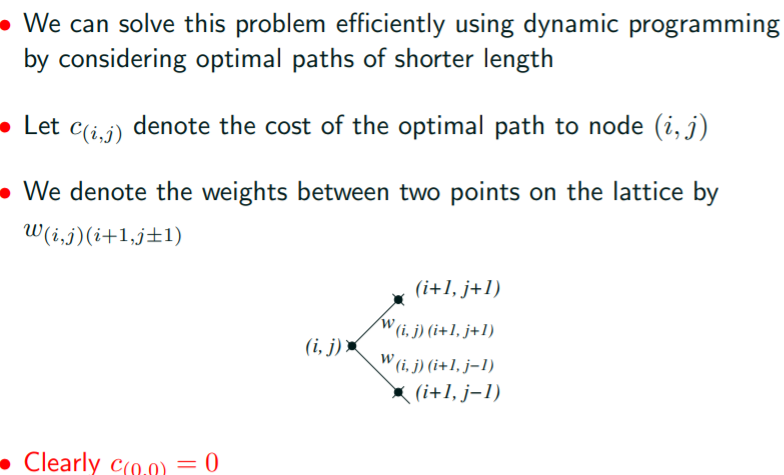
Brute force takes too much time. For this problem there are 70 paths. (n=8)

If n=100 there would be 1.01 \* 10^29 paths.

We can solve this faster with dynamic programming.

We can consider optimal pats of shorter length.

Building a solution: (easier with an example)



For a step-by-step on the above example look at slides.

Having found the optimal costs c(i, j) we can find the optimal path starting from (n,0)

At each step we have a choice of going up or down.

We chose the direction which satisfies the constraint.



**If both directions satisfy the constraint we have more than one optimal path.**

In our dynamic programming solution we had to compute the cost c(i, j) at each lattice point.

There were (n+1)^2 lattice? points.

It took constant time to compute each cost so the total time to perform the forward algorithm was Big-theta(n^2). The time complexity to perform the backward algorithm was Big-theta (n).

Also Dynamic programming works fine with DAGs.