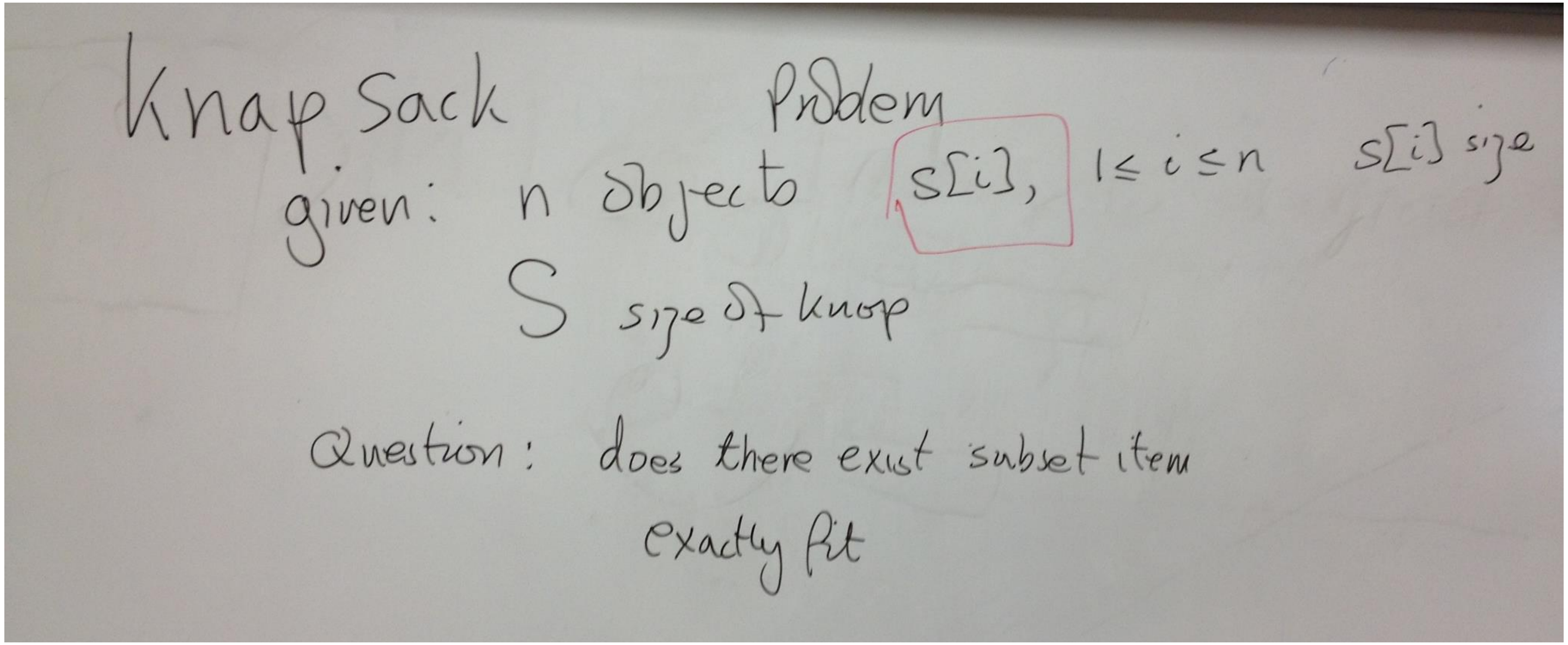
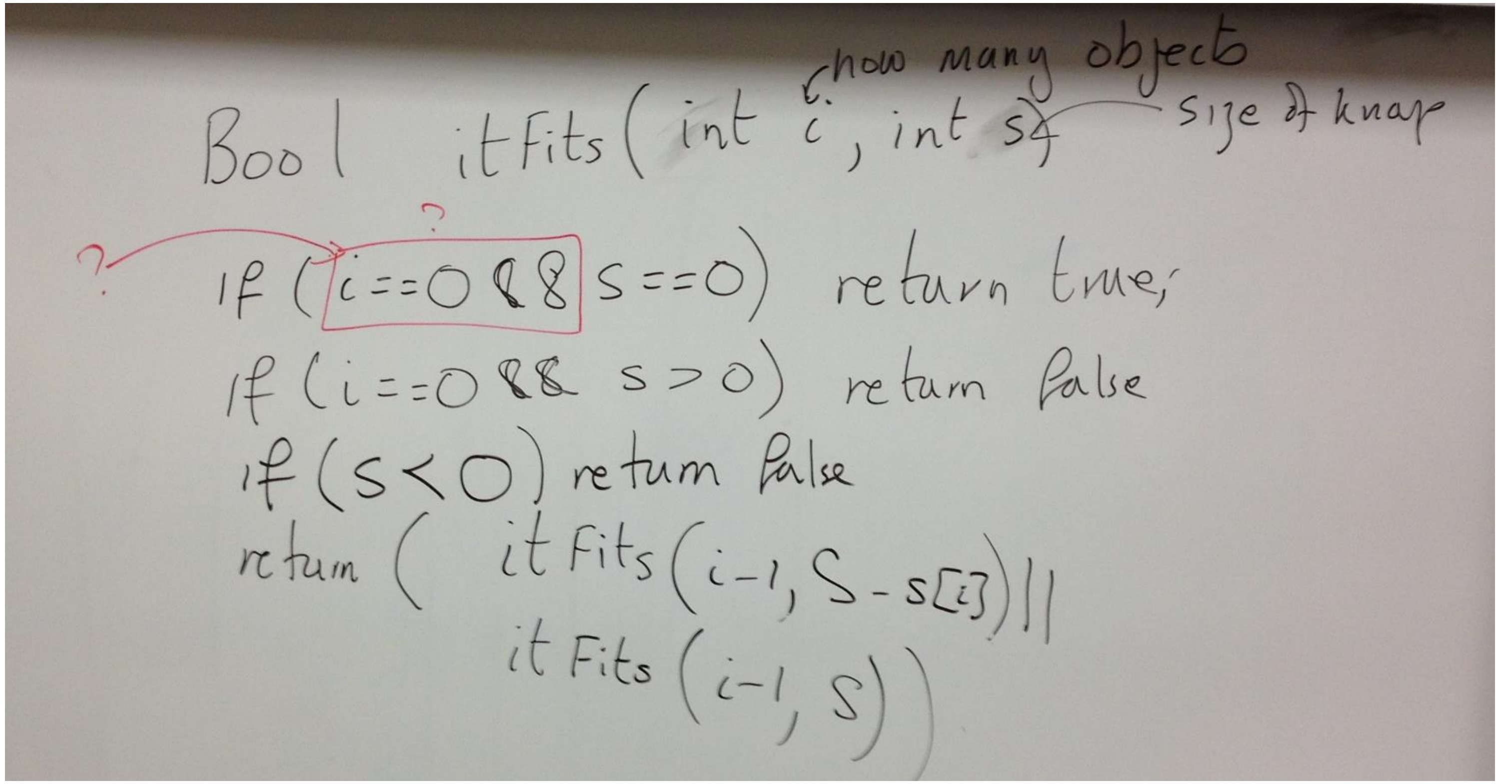
# CS 5050

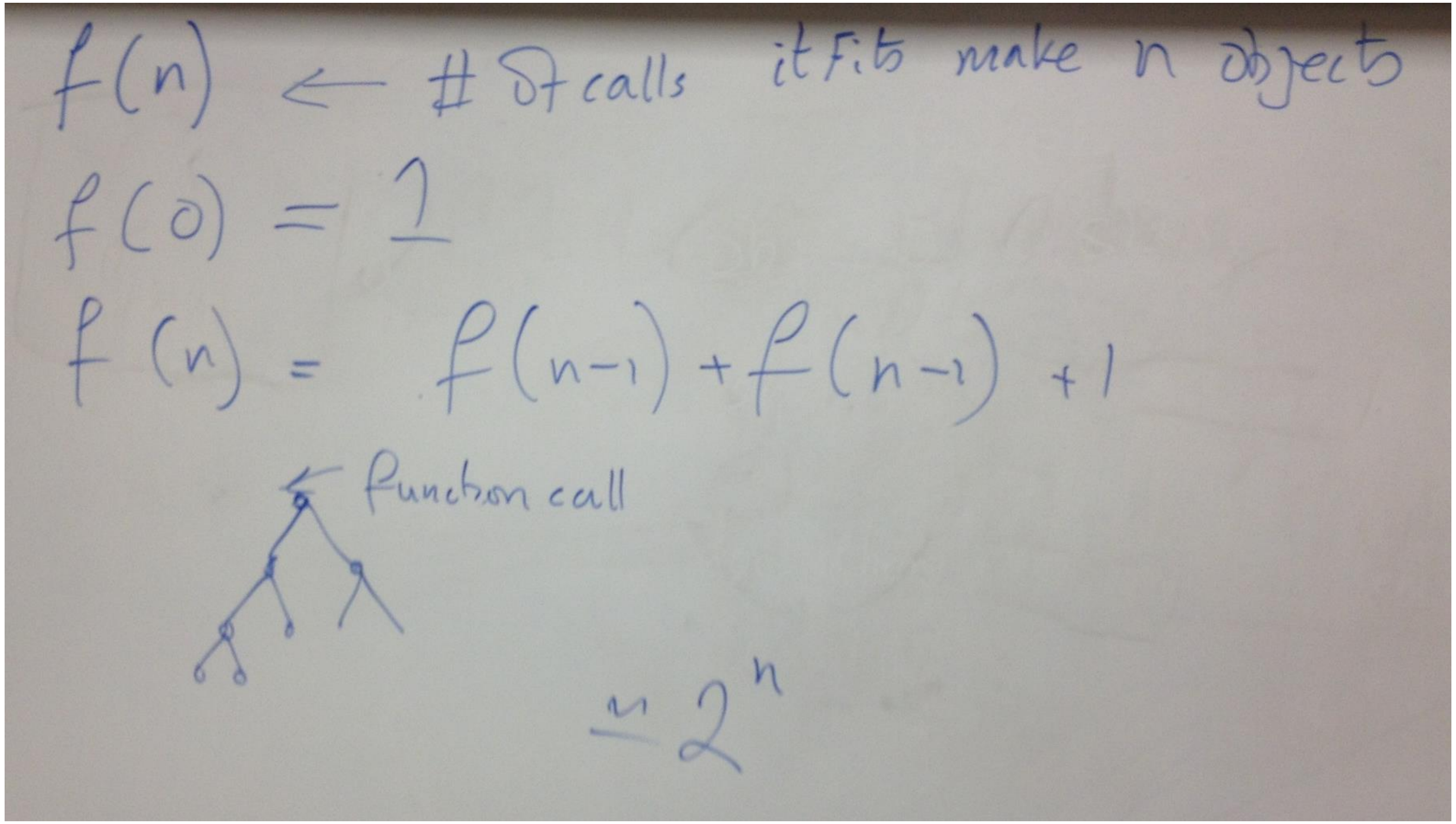
01 14 14




The simple Boolean knapsack problem. Note s[i] are integers




Solution generated by applying the meta-algorithm
Arguments describe the problem instances
Base cases are simple problems → simple solutions
There are two ways to make a smaller problem: either put the object in the knapsack or don’t
Return true if either sub problem returns true

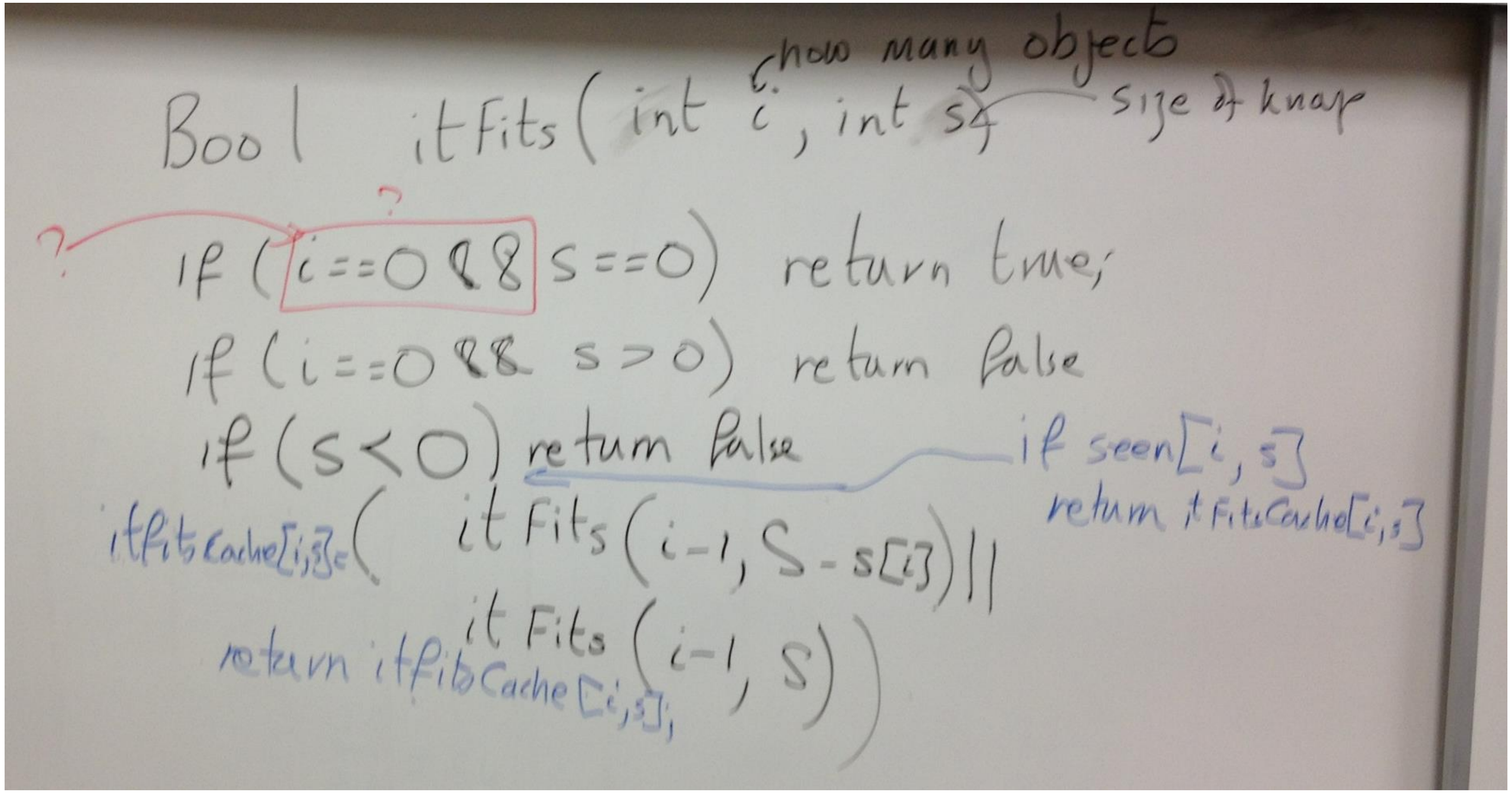



Estimate the number of function calls given n objects. Assume the worst case

Setup the function

Base case is when there are no objects

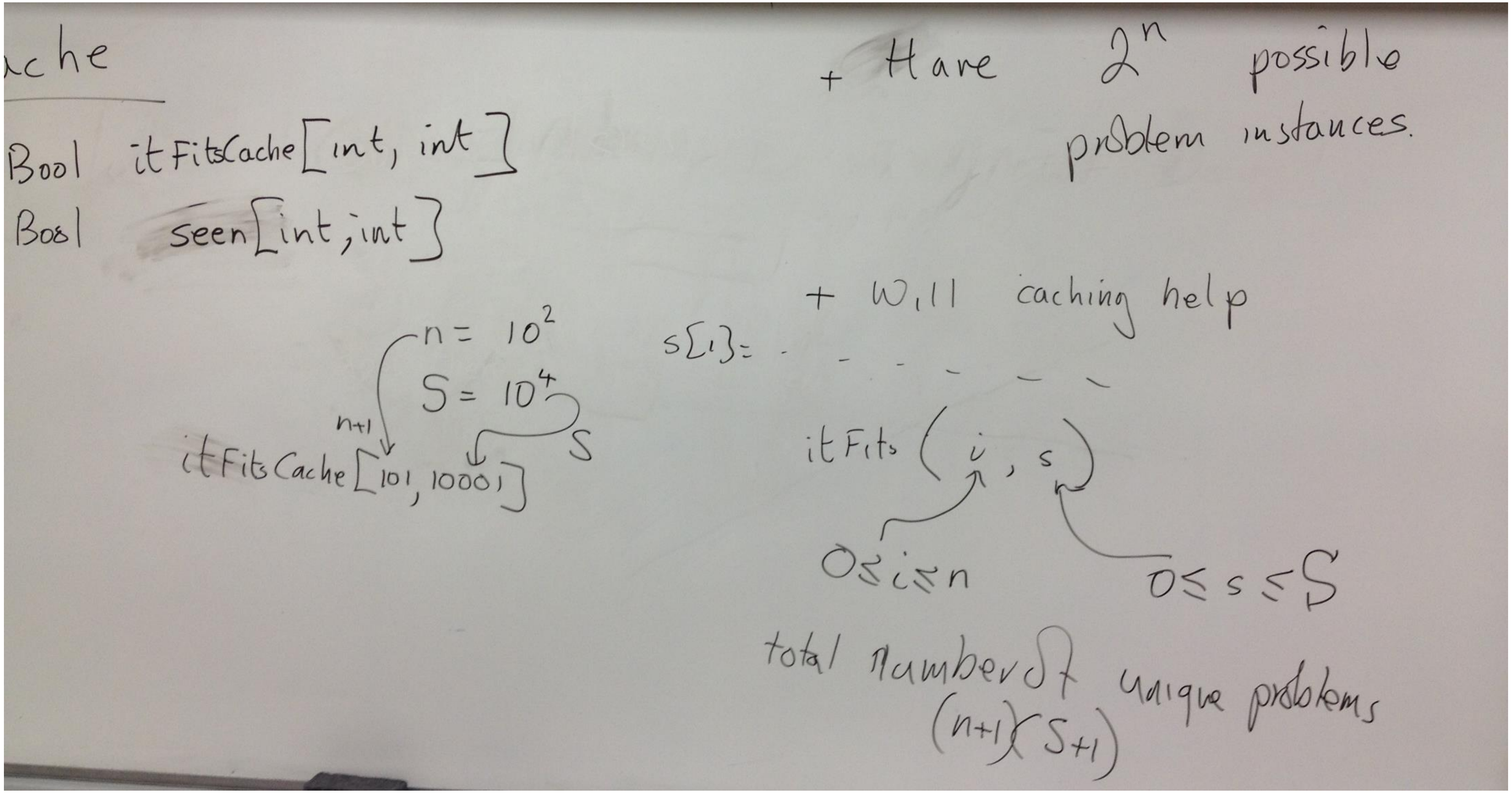
The number of calls needed to solve a problem of size n is the twice the amount of calls it takes to solve a problem of size n-1 (plus one for this call)



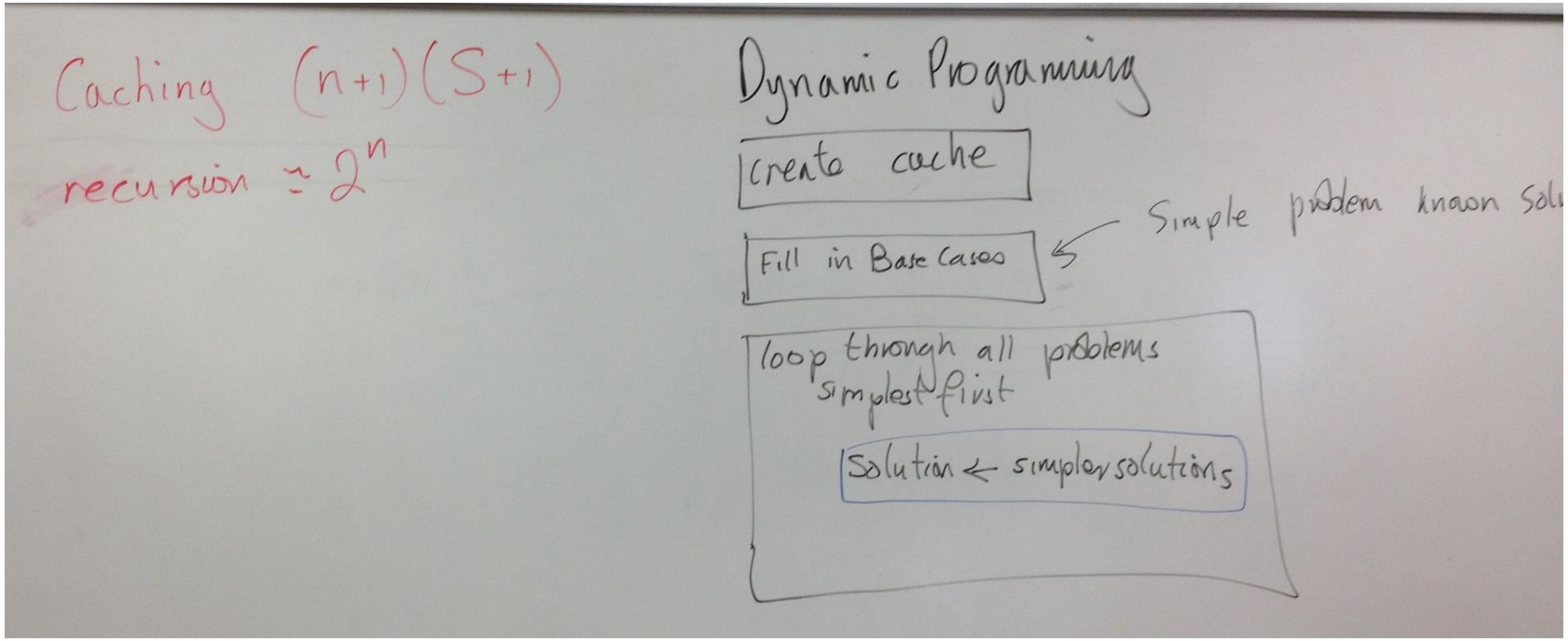

We can use caching to avoid making redundant calls

Cache is a data structure that stores solutions and is indexed by problem instances

so a 2D Bool array of dimensions n+1 by S+1 will work




Will caching actually help? Not so obvious as the Nim case.
Need to count how many unique function calls are possible.
Considering the limited range of the two function inputs, there are a maximum of (n+1)(S+1) unique calls




Caching will help! Since the caching solution avoids an exponential growth

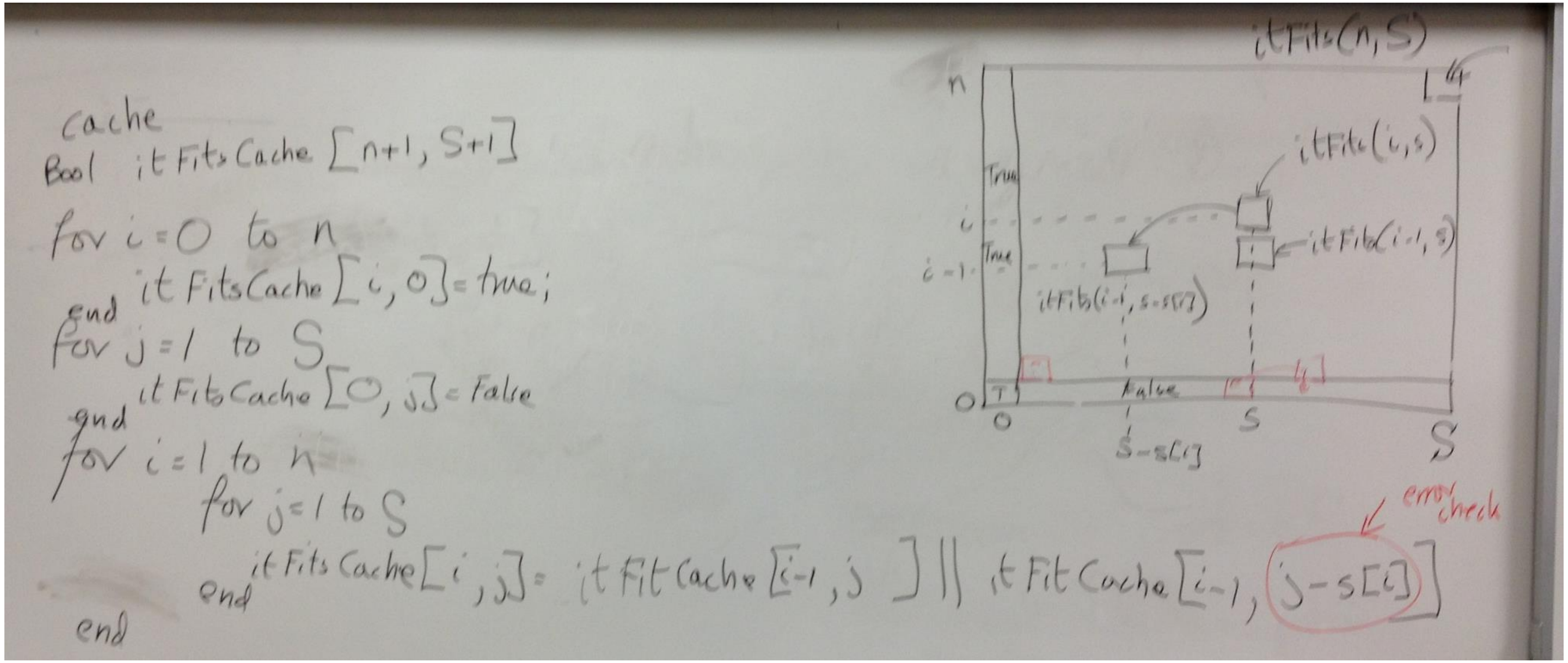
Dynamic Programming

Another meta-algorithm that takes a recursive solution and turns it into an iterative algorithm

Uses the same caching data structure

Eliminates the calling stack by computing solutions “bottom up” from simpler solutions

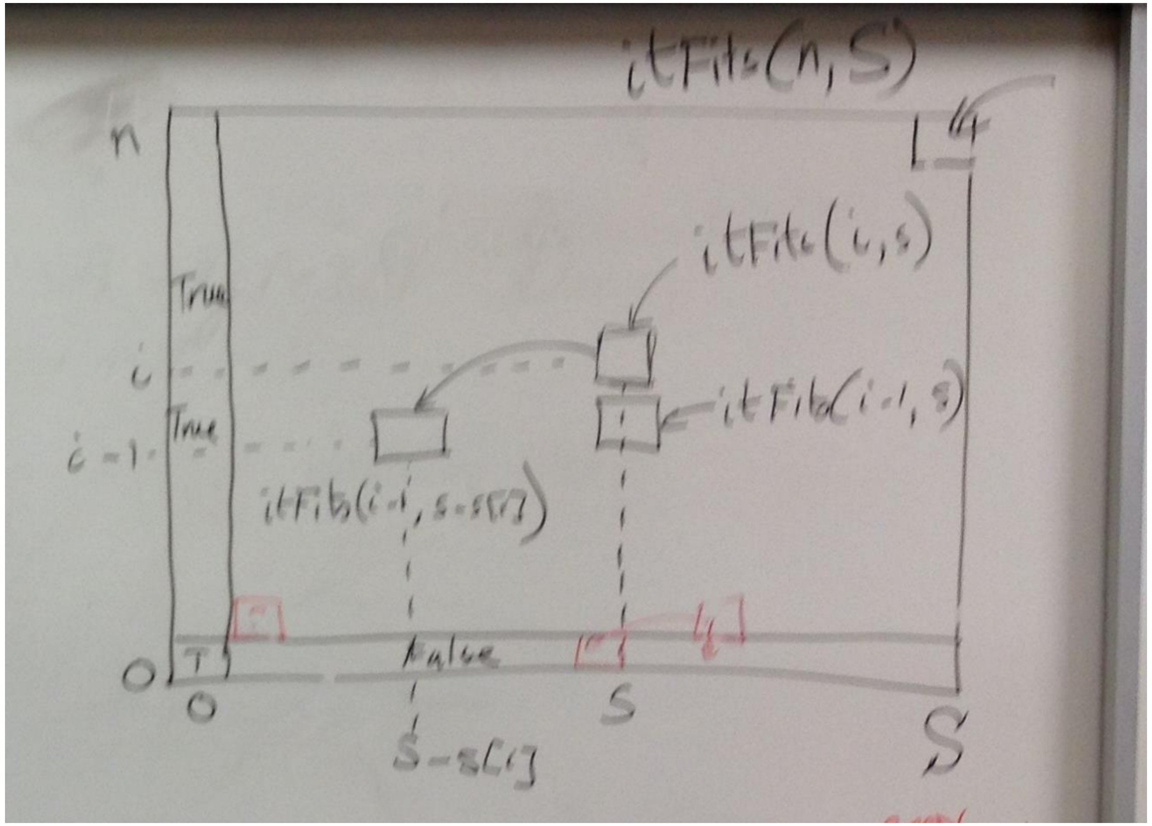
Just need to fill in the components of the DP schema from the caching and recursive algorithms



The dynamic programming algorithm for solving the Boolean Knapsack Problem
Left:

- Create cache
- Fill in simple true solutions from base case 1
- Fill in simple false solutions from base case 2
- Scan over the cache computing solution i,j from the two smaller solutions according to recursive algorithm
- Loops must touch all non-base case solutions and scan the solutions in an order such that the smaller solutions needed have already been computed

"Eager algorithm" because it computes all possible solutions




Close up of the solution cache
The left most trues are the case when we have filled the knapsack
The bottom row 1..S is when we cannot fill the remaining space
The middle shows how solution i,s is computed from two smaller solutions
i-1,s and i-1, s-s[i]
The solution that is returned is the upper right corner itFits(n,S)