

CS 5050

01 14 14

Knap Sack Problem
given: n objects $s[i]$, $1 \leq i \leq n$ $s[i]$ size
 S size of knop

Question: does there exist subset item
exactly fit

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

Bool itFits (int ^{how many objects} i, int ^{size of knap} s)

? if (i == 0 && s == 0) return true;

if (i == 0 && s > 0) return false;

if (s < 0) return false;

return (itFits(i-1, s - s[i]) ||
 itFits(i-1, s));

Solution generated by applying the meta-algorithm

Arguments describe the problem instances

Base cases are simple problems → simple solutions

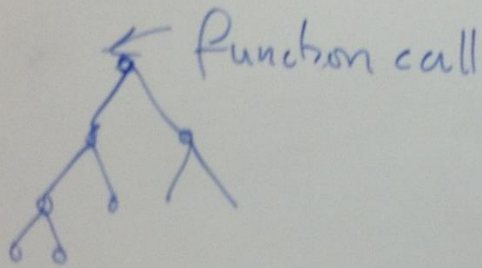
There are two ways to make a smaller problem: either put the object in the knapsack or don't

Return true if either sub problem returns true

$f(n) \leftarrow \# \text{ of calls it fits make } n \text{ objects}$

$$f(0) = 1$$

$$f(n) = f(n-1) + f(n-2) + 1$$



$$\frac{1}{2^n}$$

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)

Bool itFits(int ^{how many objects} i, int ^{size of knap} s)

? if (i == 0 && s == 0) return true;

if (i == 0 && s > 0) return false;

if (s < 0) return false;

ifFitsCache(i, s) {
 ifFits(i-1, s-s[i]) ||
 return ifFits(i-1, s);
 }

if seen[i, s] return ifFitsCache[i, s]

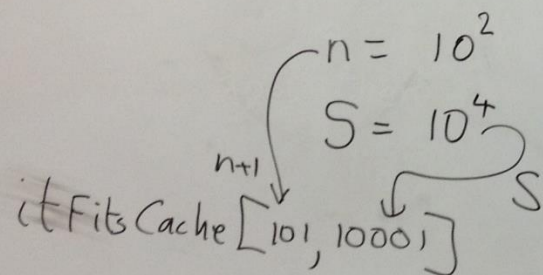
We can use caching to avoid making redundant calls

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

cache

Bool itFitsCache[int, int]

Bool seen[int, int]



+ Have 2^n possible problem instances.

+ Will caching help

$s[1] = \dots$

itFits(i, s)

$0 \leq i \leq n$

$0 \leq s \leq S$

total number of unique problems
 $(n+1)(S+1)$

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 $(n+1)(S+1)$

recursion $\approx 2^n$

Dynamic Programming

create cache

Fill in Base Cases

Simple problem known Sol

loop through all problems
simplest first

Solution \leftarrow simpler solutions

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

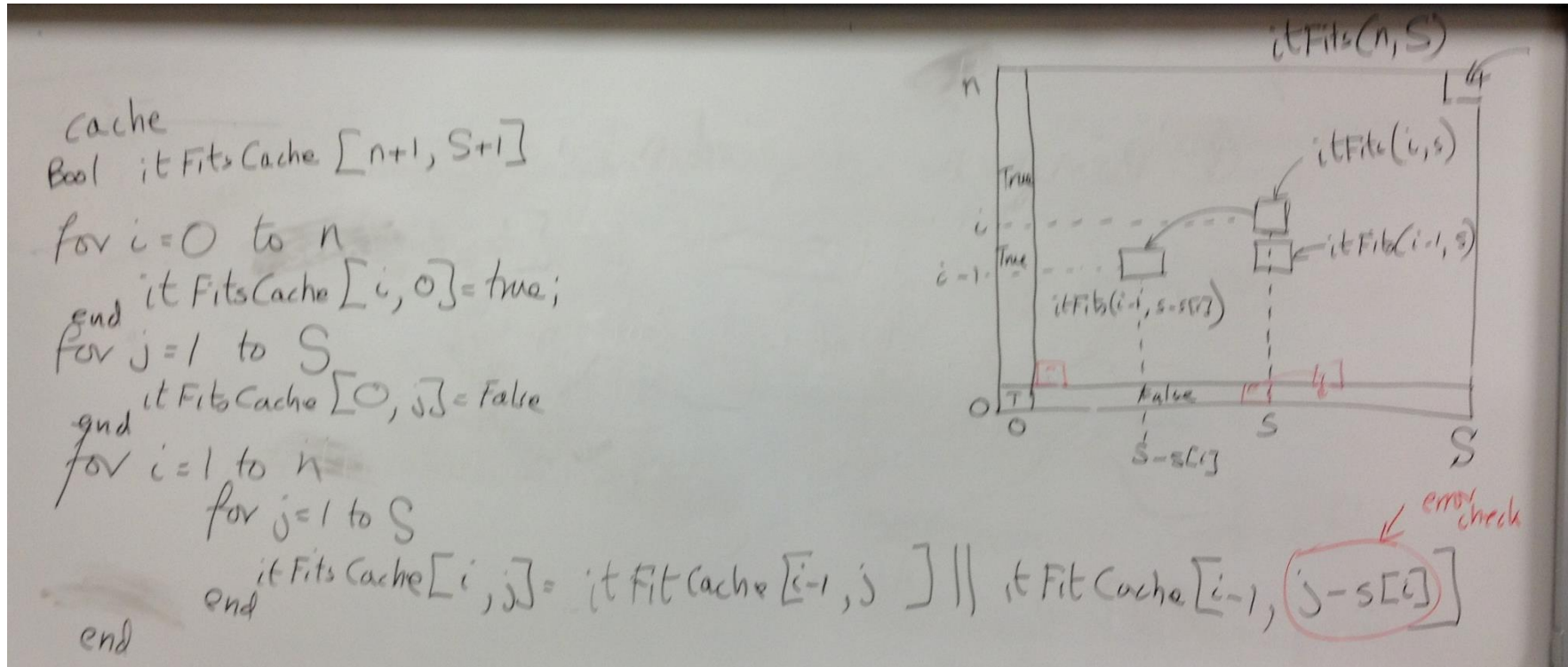
Dynamic Programming

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

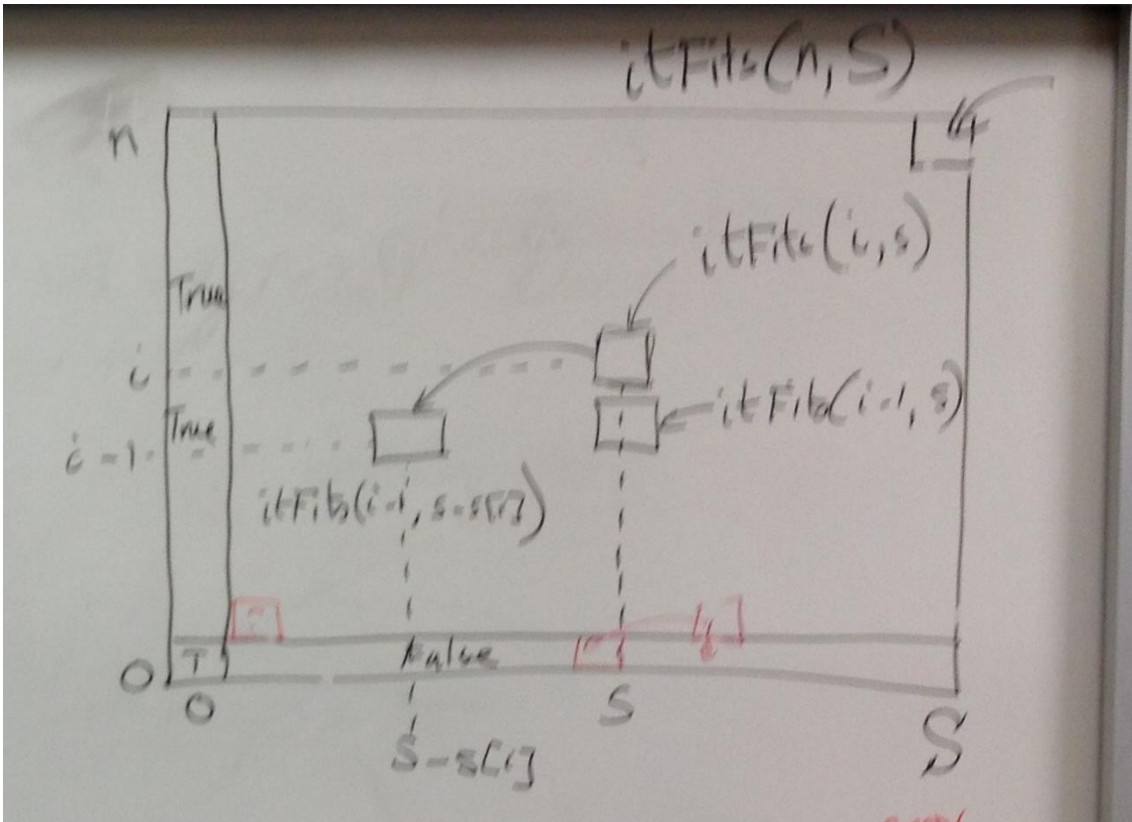
Fill in simple true solutions from base case 1

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 $itFits(n, S)$