Sarah MacAdam
Algorithms
Written 4
3/26/15

1

Problem Description
Find the number of unique ways n doors can be locked so that S doors are secure.
Inputs
int n: the number of doors
int S: the number of secure doors
Outputs
int y: the number of ways the doors can be locked
Assumptions
A door is secure when
1. It is locked and
2. It is either the first door in the row or the door to its left is locked as well
Strategy Overview
Use top down memorization. Consider the two possibilities for a row of doors of length n. The last door can either be unlocked or locked. We can make this a boolean value. Locked is True unlocked is False. Sub problems for n doors can be split into these two cases. For True, the subproblems are $(n-1, s-1, True)$ where s is the number of doors, and $(n-1, s, False)$. Subproblems for False are $(n-1, s, False)$ and $(n-1, s, True)$. Adding the solutions to all four subproblems will give the solution for n doors where s doors are secure.
Two 2D arrays will be used for memoization. Initialize everything to infinity. One will be the True array and the other will be the False array. The number of doors n will be the columns and the number of secure doors S will be the rows. The solution will be in the bottom right corner when the algorithm is finished.
Algorithm Description
The method to call is Sec(n, s, b) where b is the boolean value for the last door in the row.

Base Case: $n = 1$
if ($b = True$ && $s = 1$) //There's one door and it is the secure door

$True[1][1] = 1$
else if ($b = False$ && $s = 0$) //There's one door and it is not secure

$False[0][1] = 1$
else

$b[s][n] = 0$ //There are no other solutions for 1 door

Subroutines: //some subroutines will be omitted if they are an invalid index in the array
Sec(n, s, b)
if(b[s][n] = ∞) //the value has not been found yet, so we do the subroutines
Locked Cases, $b = True$

int a_0 =Sec($n - 1$, $s - 1$, $True$)

int a_1 =Sec($n - 1$, s, $False$)

Unlocked Cases, $b = False$

int a_2 =Sec($n - 1$, s, $False$)

int a_3 =Sec($n - 1$, s, $True$)

$a_0 + a_1 + a_2 + a_3 = a_4$

Memoize in the correct array:
if ($b = True$)

$True[s][n] = a_4$;
else
$False[s][n] = a_4$;

return a_4;

else // if b[s][n] has been found already, return the value stored there

return b[s][n];

2

Problem Description
Given a list of integers, a target sum, and an int K find the number of ways K or fewer integers can be added
to give the target sum
Inputs
list $A = \{a_1, a_2, ..., a_n\}$: of integers
int S: the target sum
int K: the maximum number of integers that can be used in the sum
Outputs
int y: the number of valid sums
Assumptions
An item in the list may only be used once.
Integers in A may be negative
S may be negative
Strategy Overview
There are two possibilities for each number in the list, it is either used or not used. Make this a boolean value b. b is True when the integer is used and False when it is not used. Using top down memoization, we will calculate all possible suproblems and their sums. Whenever a sum is equal to S, we will increment y.
Create two 2D arrays to store the sums, one will be called False and the other True

Algorithm Description

Start with the list item in the list a_n
Method: Sum(k, a_n, b) // $k = K - 1$, so we can keep track of how many integers have been used in the sum

Base Cases:

if Sum(k, a_n, b)

$k = 0$ //the max number of integers have been added
memoize a_n
return a_n
if Sum(k, a_n, b)

$n = 0$ //we've reached the last integer in the list
memoize a_n
return a_n

Subroutines:

Sum(k, a_n, b) //When a_n is used
Sum(k-1, a_{n-1}, True) $+a_n$
Sum(k-1,a_{n-1}, False) $+a_n$
Sum(k, a_n, b) //When a_n is not used
Sum(k, a_{n-1}, True)
Sum(k, a_{n-1}, False)
if any of the sums are equal to K, increment y

3

Problem Description
Find the maximum number of boxes that can be nested inside of each other. State the runtime.
Inputs
list B: the set of boxes
Outputs
List M: the set of boxes in the solution
Assumptions
A box can be rotated
Each box b_i has a height, width, and length

Strategy Overview
Sort the boxes by volume. Use a bottom up approach to determine the maximum number of boxes that can fit inside
of the next box.
Algorithm Description
//Sort the boxes by volume, least to greatest
//Create an array Y the size of the list
//Index each box by their sorted order

//Base Case: The first box, b_0 corresponds with index zero
//It has the smallest volume and no boxes fit inside of it
//Make $Y[0] = 0$

//Bottom Up: for every next box b_n
//See if b_{n-1} fits in b_n by sorting the length, width, and height fields for both boxes and comparing them in order
//If all of b_{n-1}'s fields are less than a unique field of b_n,
//we can conclude b_{n-1} fits inside of b_n
//Add 1 to the value at b_{n-1}'s corresponding index and insert it at b_n's index
//Decrement b_{n-1} to b_{n-2} and repeat
//Replace the value in b_n's index if the next solution is greater than the current value at the index
//Repeat until you reach b_0

//Do a final scan of array Y for the greatest number of nested boxes.

//Trace from this index to find the exact set of boxes that yield this solution
//Add the box that corresponds to this index to the list Y
//If the value at the current index is x, scan to the left to find the first index with the value $x - 1$
//Check that the box that corresponds with this index is compatible with the one added most recently to Y
//If it is, add the box to Y and scan left for $x - 2$ etc. until you reach 0.
//If a box is not compatible, continue scanning left until you find the one that is.

//Return Y as the solution

Runtime

The runtime of this algorithm is $\theta(n^2)$ where n is the number of boxes in the list.

4

Problem Description
Find the minimum number of days to make all n runs from list R. The runs must be made in the order they are listed. Your satisfaction at the end of each day is determined by the function *twd*. After finding the minimum number of days, find the itinerary that minimizes *twd*.

$$twd(t) = \begin{cases} 0t = 0 \\ -C11 \leq t \leq m \\ (t-m)^2 otherwise \end{cases}$$

where C is some constant
Develop an algorithm that minimizes the number of days to ski and then find the minimal solution for *twd*
given that minimum number of days.
Inputs
list $R = \{r_1, r_2, ..., r_n\}$: integers, the time is takes each run to get down and back up the mountain
int L: the number of minutes to ski in a day
int m: the number of minutes we want at the end of the day
Outputs
list Y: The itinerary that minimizes the number of days and then minimizes *twd*
Assumptions
1. A run must be completed in one day
2. The runs are done in order
3. Run r_n must be completed on or before day n.

Strategy Overview
Start with a greedy approach to determine the minimum number of days. Then use dynamic programming to minimize *twd*.

Algorithm Description
Start with a greedy algorithm to find the minimum number of days to complete all runs.
Do as many runs a day as possible without going over L minutes
Do this again until all of the runs have been designated to a day
Count the number of days filled, we will call this d
This is the number of days we will use for the dynamic programming
We will refer to this greedy solution to determine if a day is compatible with a run.
For example if run r_6 was put in day d_3 in the greedy solution, then r_6 is not compatible with any day before d_3.

Minimizing twd
Use the method Ski(r_n, d_n) where r_n is the run and d_n is the day. It will return the minimum sum of twd for all of the days required.

Make two 2D arrays, Itin[][] to hold the value of t, and TWD[][] to hold the solutions for $twd(t)$, both arrays are the number of days by the number of runs

Base Case: Ski(r_1, d_1) // r_1 must be in day 1
int t = $L - r_1$
Itin[0][0] = t;
TWD[0][0] = twd(t);

Ski(r_n, d_n)
Check if the day and run are compatible based on the greedy solution and assumption 3
If compatible
min(Ski(r_{n-1}, d_n), Ski(r_{n-1}, d_{n-1}))

else
Ski(r_{n-1}, d_{n-1})