Project 2: Dynamic Programing

COT 4400 Analysis of Algorithms

Group 25:
Jinhao Chen, Shariyf Mitchell, Isai Felix

April 1, 2016

Problem Break Down

The task in this project was to solve the *(b, n, k)-basket* problem. This problem is a generic combinational problem where we have a certain number of baskets (*n*), where each can hold up to *k* balls. We have to determine how many ways a given number of balls (*b*) can be distributed into each basket.

Initially when the algorithm is called, f(b,n,k) will produce the number of ways a ball can be distributed into n amount of baskets. The function f(b > nk, n , k), is a scenario where there are more balls than the space calculated, will not be able to compute a value which will return 0. The function f(b, n=0, k), is a scenario where there are no baskets for b balls, will not be able to compute a value which will return 0. Ultimately, the algorithm will get the result by calculating f(b,n-1,k) + f(b-1, n-1, k) + … + f(b-k, n-1, k).

A problem that arises is we end up computing the same sub result over and over again. For example, f(3,3,2) will become f(3,2,2) + f(2,2,2) + f(1,2,2). For f(3,2,2) the sub results produced are f(3,1,2) + f(2,1,2) + f(1,1,2), which results in f(3,2,1) = 0 (because the k = 2, and that is the max amount of balls that can be placed into a basket), f(2,1,2) = 1, and f(2,1,2) = 1. For f(2,2,2) the sub results produced are f(0,1,2) + f(1,1,2) + f(2,1,2) which results in f(0,1,2) = 1, f(1,1,2) = 1, f(2,1,2) = 1. For f(1,2,2) the sub results produced are f(0,2,2) + f(1,2,2), which results in f(0,2,2) = 1 and f(1,2,2) = 1. So, f(3,2,2) = 2, f(2,2,2) = 3, and f(1,2,2) = 2. The final result of the example is f(3,2,2) + f(2,2,2) + f(1,2,2) = 7. Notice how we computed the same sub-result more than once. To fix this problem we implemented a 2D array to represent an iterative solution. a[b][n] based on a[b][n-1] + a[b-1][n-1] + a[b-2][n-1] + …. + a[b-k][n-1]. The solution that the algorithm will produced will be placed in the a[b][n].

Recurrence Method

The following is the recurrence method utilized:
memoization (b,n) = sum of memoization from i=0 to k, from (b-i, n-1, k)

The following code segment shows how the recurrence method was implemented in the code:

```
for i=0 to k
        if b-i >= 0 and n >= 0 then
                a[b][n] += memoization(b-i, n-1, k, a);
end
```

Base Case

If b and n equal zero there are no balls and no baskets available then return 1. If n is less than zero there are balls but no baskets then return 0. In all cases, k is greater than or equal to 1.

```
if b==0 and n==0 then
        a[b][n] = 1;
        return a[b][n];
else if n==0 then
        return a[b][n] = 0;
else if b==1 then
        if  k==0 then
                return a[b][n] = 0;
        else then
                return a[b][n] = n;
```

Memoization Pseudocode

```
Algorithm: Memoization (b, n, k, a)
if  a[b][n] > 0
        return a[b][n];
else if b==0 and n==0 then
        a[b][n] = 1;
        return a[b][n];
else if n==0 then
        return a[b][n] = 0;
else if b==1 then
        if  k==0 then
                return a[b][n] = 0;
        else then
                return a[b][n] = n;
for i=0 to k
        if b-i >= 0 and n >= 0 then
                a[b][n] += memoization(b-i, n-1, k, a);
end
return a[b][n];
```

Iterative Pseudocode

```
Algorithm: Iterative
for i=0 up to b do    ---------------------------------------------------- b
        for j=0 up to n do    ---------------------------------------- n
                for l=0 up to k do    ---------------------------------- k
                        if i==0 and j==0 then ---------------------- O(1)
                                a[i][j] = 1;    ------------------------- O(1)
                        else  if i-l >= 0 and i>0 then    ----------O()
                                        a[i][j] += a[i-l][j-1]; -------O(1)
                        end
                end
        end
end
return a[b][n];    ------------------------------------------------------------ O(1)
```

Complexity of Iterative Algorithm

Time complexity: O(bnk)

As seen on the pseudocode above, the way we got the time complexity for the iterative algorithm was by combining the individual time complexities of each line of code. The outer FOR loop from i=0 to b has a time complexity of b. The FOR loop from j=0 to n has time complexity of n. The innermost FOR loop from l=0 to k has time complexity of k. All of the inside if else statements as well as the final return statement all had the overall time complexity of O(1), which is constant. Therefore, it was not be included in the total time complexity of the iterative algorithm.

Space complexity: O(bn)

As seen on the pseudocode above, the way we got the space complexity is by implementing a 2D array. The array has a space complexity of bn.