

Group 8: Vaddanak Seng, Micah Parcher, Darshankumar Patel

Algorithms Group Project 2 Report: Dynamic Programming

1. Describe in English how you can break down the larger problem into one or more smaller problem(s). This description should include how the solution to the larger problem is constructed from the subproblems.

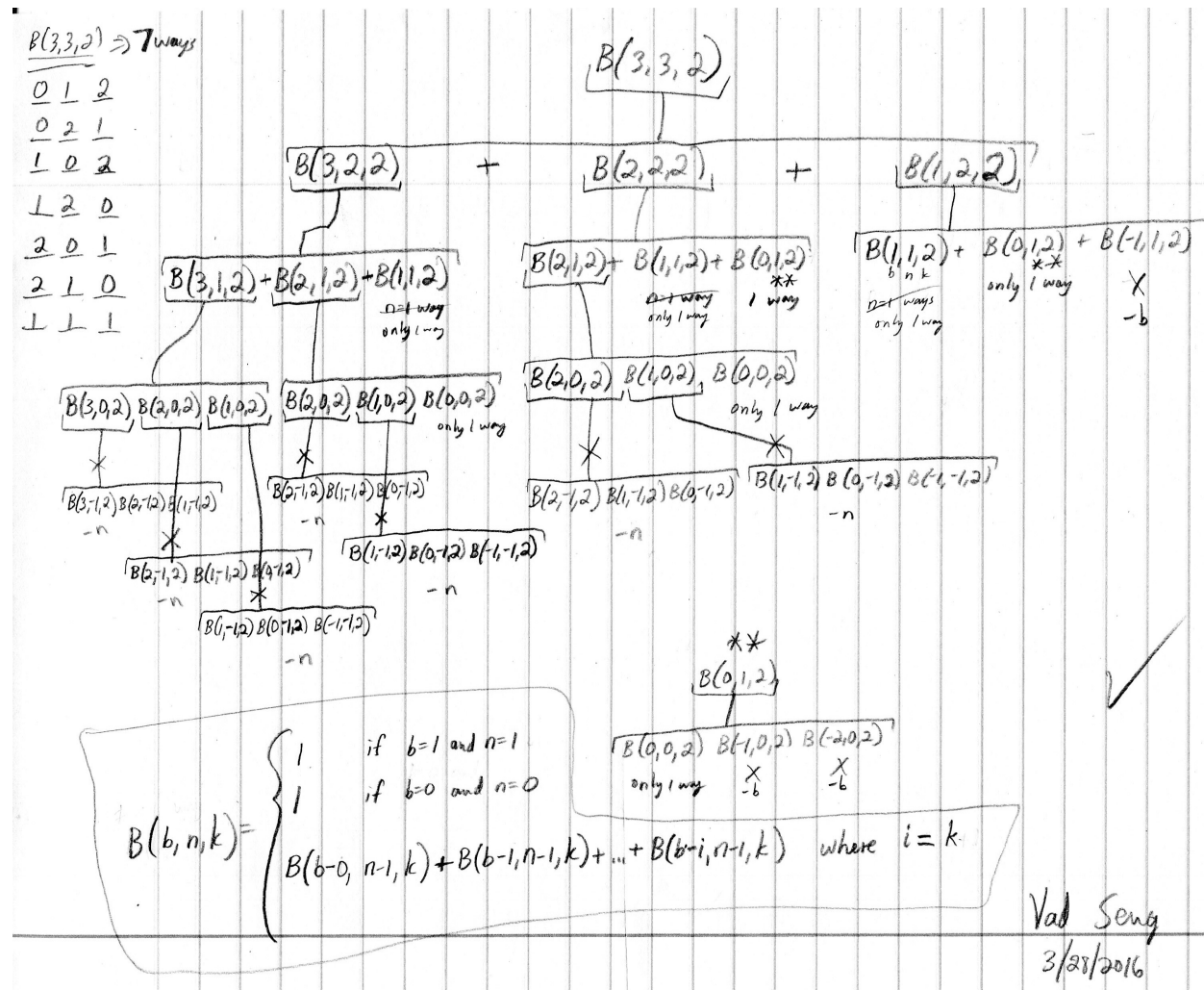
We are tasked to find the number of ways to arrange a given number of balls in a given number of baskets and each basket has a capacity. The larger problem is succinctly $\text{memo}(b, n, k)$, where b represents the number of balls, n is the number of baskets, and k is the maximum number of balls that can be put in a basket. The task is recursive in nature and each subproblem is optimally the same in nature as the larger problem. The larger problem is broken down into a number of subproblems equal to the number of ways to fill one basket at a time, which means for each basket, the number of tries equal to the total number of balls we can put in the basket. This is better illustrated using a diagram (Figure 1). We start with a problem $\text{memo}(3,3,2)$. The next line in the diagram shows how it is broken down into 3 smaller problems. It follows the recursive formula:

$$\text{memo}(b,n,k) = \begin{cases} 1 & \text{if } b==0 \text{ and } n==0 \\ \text{memo}(b-0,n-1,k) + \text{memo}(b-1,n-1,k) + \dots + \text{memo}(b-i,n-1,k) & \text{where } i=k \end{cases}$$

So, the first line represents the result after we distribute to the first basket. The first subproblem $\text{memo}(3,2,2)$ tells us that after we distribute zero ball to the first basket, there are 3 balls remaining and 2 baskets remaining. $\text{Memo}(2,2,2)$ shows that we distributed one ball to the first basket and so 2 balls remain and 2 baskets remain. $\text{Memo}(1,2,2)$ shows us that after we distribute 2 balls to the first basket, we have 1 ball left and 2 baskets remain. We stop here for the first row because we reached the basket capacity of two. Then, for each of the 3 subproblems, we divide each into their respective 3 subproblems, for a total of 9 subproblems in the next line of the diagram. This next line shows the result after we distribute the ball to the second basket. The previous line accounted for all the ways to fill the first basket. This next line will try all the different ways to fill the second basket. For example, $\text{memo}(2,1,2)$, $\text{memo}(1,1,2)$, and $\text{memo}(0,1,2)$ are subproblems for $\text{memo}(2,2,2)$, where we distributed 1 ball to the first basket. So, in this next line, we have 2 balls to distribute. $\text{Memo}(2,1,2)$ shows that we distributed zero ball to the second basket and so we have 2 balls left and 1 basket remaining. $\text{Memo}(1,1,2)$ shows that one ball was distributed to the second basket and so we have 1 ball remaining and 1 basket remaining. $\text{Memo}(0,1,2)$ is the result after we distributed 2 balls to the second basket and so zero ball remains and 1 basket left. The results of these 3 subproblems combined will give the answer to the larger problem in the line above. If we continue with the flow of the diagram and consider $\text{memo}(2,1,2)$ in the second line, we can subdivide this problem into 3 of its own subproblems, specifically $\text{memo}(2,0,2)$, $\text{memo}(1,0,2)$, and $\text{memo}(0,0,2)$. By the third line down the diagram flow, we have 2 balls left and we need to fill the third and last basket. Recall, we used 1 ball in the first line, 0 ball in the second line, and so we have 2 balls to

distribute to the last basket. Memo(2,0,2) is the result after we distribute 0 ball to the last basket; we have 2 balls left and zero basket left. This subproblem won't give us the right answer because the ball was not distributed properly since we have balls leftover. So, we would not count this subproblem as a solution to the previous big problem (ie it's parent problem). Memo(1,0,2) is similar because after we distribute 1 ball to the third basket, we have 1 ball left and no more basket to use. Memo(0,0,2) is the result after we distribute 2 balls in the third basket and we are left with zero ball. So, this represents a solution because we have arranged all the ball in all the baskets and did not exceed the capacity of each basket. The solution of this third line will be the solution for the parent, larger problem. Since the first two subproblems were not solutions and the third subproblem was, the solution returned to the parent problem is $0+0+1$. If we revisit the second line in the diagram, the memo(2,1,2) problem now has solution equal to 1. If we follow the flow in a similar way for the memo(1,1,2) and memo(0,1,2) problems, we will discover that they each also have solution of 1. These three problems in the second line are subproblems for the memo(2,2,2) in the first line. So, the solution $1+1+1$ is returned as the solution for problem memo(2,2,2). Memo(2,2,2) problem has solution 3. The other problems on line one are memo(3,2,2) and (1,2,2) and they have solutions of 2 and 2, respectively. These 3 problems on the first line are subproblems of the original problem memo(3,3,2), which now receives a solution of $2+3+2=7$ from the 3 subproblems in the first line below.

Figure 1: Modelling recursive algorithm using problem memo(3,3,2).



2. What recurrence can you use to model the problem using dynamic programming?

The recursive model that we used is:

$$\text{memo}(b,n,k) = \begin{cases} 1 & \text{if } b==0 \text{ and } n==0 \\ \text{memo}(b-0,n-1,k) + \text{memo}(b-1,n-1,k) + \dots + \text{memo}(b-i,n-1,k) & \text{where } i=k \end{cases}$$

This model is illustrated and also given in Figure 1. Because it is dynamic programming, we also stored previously seen recursive calls in a 2 dimensional array, ie Array(b,n). In figure 1, we can see that problem memo(2,1,2) is encountered twice in the recurrence. The result of the first

call will be stored in the array at Array(2,1). The second call will simply retrieve the result from the array and avoid the overhead of executing its own recursive calls.

Pseudocode shows how it was implemented:

```
for i = 0 to k:  
    if b-i >= 0 and n-1 >= 0:  
        arr [b, n] += memo(b - i, n - 1, k)  
end
```

3. What are the base cases of this recurrence?

The base case is when $b==0$ and $n==0$, it returns 1. The array look up could serve as a base case also since it ends a recursive call and returns a terminal value.

Following are the base cases for the recurrence in previous question:

```
if(arr[b][n] > 0)  
    return arr[b][n];  
if(b==0 && n==0)  
    return 1;
```

4. Describe a pseudocode algorithm that uses memoization to solve the problem.

The parameters used are arr, b, n, and k. arr is a two dimensional array with row length equal to the number of balls and column length equal to the number of baskets. b is the number of balls. n is the number of baskets to be filled. k is the capacity for each basket. So, we have b balls and we need to arrange all of them in n baskets such that no basket exceeds k balls.

```
arr = 2D Array; arr[0...b][0...n]  
b = number of balls  
n = number of baskets  
k = max capacity per basket
```

The main function allocates space for the 2 dimensional array and initializes each element to 0. It calls the memoized function which will return the number of ways that b balls can all be distributed in n baskets that has a maximum capacity of k ball per basket.

```
main ( ):  
arr = Array (b + 1, n + 1)  
initialize arr with zero's  
ways = memo (b, n, k)
```

The memoized function stores previously function calls in a 2-dimensional array data structure. The function first checks if this function was previously called by looking up in the array. If this function was called before, then its result was stored in the array. In such case, the recurrence ends at this statement and the correct value will be returned. So, this effectively serves as a base case which terminates further recursive calls. The next if statement is also the base case to stop further recursive calls. When there is no balls left ($b==0$) and there is no baskets left to fill ($n==0$), the function returns 1 to indicate that all balls have been distributed to all the baskets. The for-loop iterates k number of times, which is the capacity of a basket. For a given basket, we want to try each possibilities where a given basket gets 0, 1, 2, ... up to k balls. The if statement checks if each possibilities make sense. It ensures that we don't try the possibilities that include negative number of balls and negative number of baskets. The possibilities that are valid (ie positive number of balls and baskets) are passed as arguments to subproblems to the current problem by a recursive call for each possibilities. The solution of the current problem is returned.

```
memo (b, n, k, arr):
if arr[b, n] > 0:
    return arr [b, n]
if b==0 and n==0:
    return 1
for i = 0 to k:
    if b-i >= 0 and n-1 >= 0:
        arr [b, n] += memo(b - i, n - 1, k)
return arr[b, n]
```

5. Describe a pseudocode algorithm that solves the problem iteratively (using dynamic programming). Your algorithm should be optimal in terms of its time and space complexity.

The iterative algorithm uses a 2 dimensional array to store results. The function parameters include B , N , and K , where B is the number of balls, N is the number of baskets, and K is the maximum capacity per basket. The number of rows is $B+1$ and the number of columns is $N+1$. The $+1$ is for trying zero number of balls and up to B number of balls. The function allocates an $\text{Array}(B+1, N+1)$ and then initializes all the elements to 0. The element at location $\text{arr}[0][0]$ is set to 1.

We follow a model similar to the recursive model:

memo(b,n,k) = { 1 if $b==0$ and $n==0$
{ memo(b-0,n-1,k) + memo(b-1,n-1,k) + ... + memo(b-i,n-1,k) where $i==k$

But, we don't have a base case.

The logical flow is illustrated in Figure 2 for problem **iterative(3,3,2)**. The iterative algorithm employs nested for-loops. The outer for-loop will iterate over the number of baskets starting with $n=1$ and up to the total number of baskets. In figure 2, this corresponds to column 2. The inner for-loop will iterate over the number of balls for each basket. For each basket and ball intersection, we retrieve the previous elements in the array and add them up, which will be set in the current array intersection as the solution. This is done by iterating over the K values. So, the previous elements from which we retrieve the values are located in the previous column (basket) and the rows in the previous column that includes the current row, the row above it, and up to k rows above the current row. The if statement prevents looking up array cells that are out of bounds. The solution in the current array intersection is returned.

iterative(B, N, K):

arr = Array(B+1, N+1)

initialize arr with zero's

arr[0][0] = 1

for n=1 to N:

for b=0 to B:

for i=0 to K:

if b-i >=0 and n-1>=0:

if arr[b-i][n-1]>0:

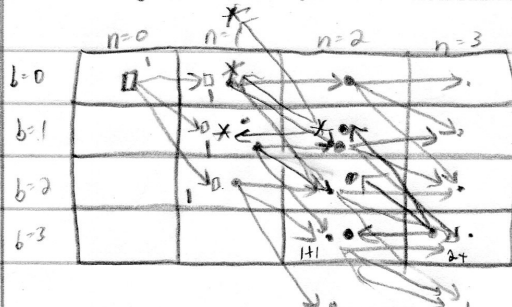
arr [b][n]+= arr[b-i][n-1]

return arr[b][n]

Figure 2: Modelling iterative dynamic programming

memo(B=3, N=3, k=2)

7 ways

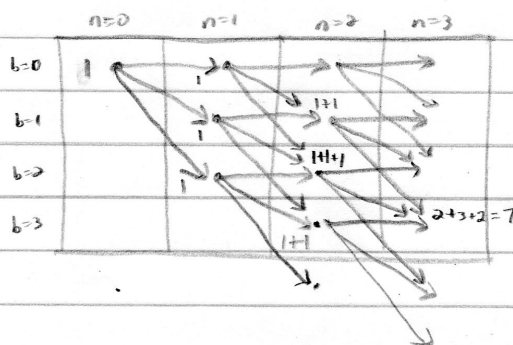


memo(B=3, N=3, k=2)

7 ways

arr = Array(B+1, N+1)

initialize arr with zeros



arr[0][0] = 1

for n=1 to N:

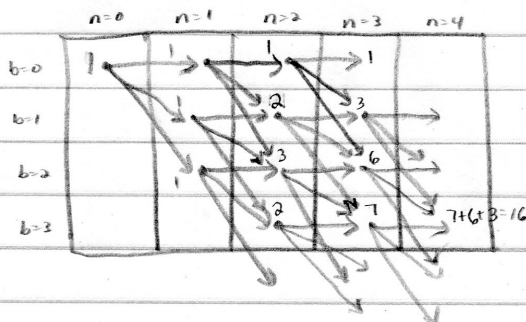
for b=0 to B:

for i=0 to k:

if b-i ≥ 0 and n-i ≥ 0:
if arr[b-i][n-i] > 0:
arr[b][n] +=
arr[b-i][n-i]

memo(B=3, N=4, k=2)

16 ways



return arr[b][n]

Vad Seng
3/30/2016

✓

6. Analyze the complexity of your iterative algorithm in terms of b, n, and k.

The complexity of the iterative algorithm is illustrated in Figure 3. Allocating memory for Array(B+1, N+1) takes constant time. Initializing the array with zeros will require visiting each array elements so it will take $\Theta(BN)$ time. Setting the first array element to 1 takes constant time. The first for-loop iterates N times. The second for-loop iterates B. The third for-loop iterates K times. The statements inside the third for-loop each takes constant time $\Theta(1)$. The return statement takes constant time. Each for-loop iterates the same number of iterations per iteration and does not depend on the outer for-loop so we can multiply to obtain the final complexity, which is $\Theta(BNK)$. The $\Theta(BN)$ and $\Theta(1)$ times were dominated by $\Theta(BNK)$.

Figure 3: Complexity of iterative algorithm

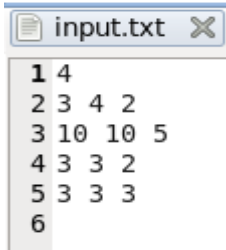
	Algorithm: iterative(B, N, K)	
$\Theta(1)$	arr = Array(B+1, N+1)	
$\Theta(BN)$	initialize arr with zeros	
$\Theta(1)$	arr[0][0] = 1	
	for n = 1 to N:	
	for b = 0 to B:	
$\Theta(BNK)$	for i = 0 to K:	
	if b-i >= 0 and n-i >= 0:	
	if arr[b-i][n-i] > 0:	
	arr[b][n] +=	
	arr[b-i][n-i]	
$\Theta(1)$	return arr[b][n]	
	$\Theta(BN) + \Theta(1) + \Theta(BNK) = \Theta(BNK)$	

7. What is the space complexity of your iterative algorithm?

Because it only concerns about how much storage we need, and in our case, B and N are both inputs since B is a number of balls and N is a number of baskets. K is only dealing with storage capacity which isn't needed when calculating space complexity. By implementing the 2D array as seen in the pseudocode above, we get the space complexity of: $\Theta(BN)$

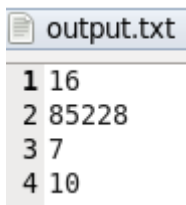
Results:

After implementing memoization, our program was able to solve the (b, n, k)-basket problem. The program reads inputs from the file **input.txt**. Following are the inputs from **input.txt** file, where the first line is a single positive integer indicating that there are 4 problem instances:



```
1 4
2 3 4 2
3 10 10 5
4 3 3 2
5 3 3 3
6
```

The output was written to **output.txt** file, one line for each problem solution:



```
1 16
2 85228
3 7
4 10
```

To compile and run the program on command line, make sure you are in the correct directory and enter the following command: **g++ memoize.cpp -o memoize**

To run the program, enter: **./memoize**

Here is the output for the memoized program that implemented the memoized pseudocode (it implemented the iterative pseudocode as well):

```

[darshan@login3 project2_ver3]$ g++ memoize.cpp -o memoize
[darshan@login3 project2_ver3]$ ./memoize
memoize(b=3,n=4,k=2,arr): 16
0 1 1 0 0
0 0 2 3 0
0 1 3 6 0
0 0 2 7 16
iterative(B=3,N=4,K=2,arr): 16
1 1 1 1 1
0 1 2 3 4
0 1 3 6 10
0 0 2 7 16
memoize(b=10,n=10,k=5,arr): 85228
0 1 1 1 1 1 1 1 1 0 0
0 0 2 3 4 5 6 7 8 0 0
0 1 3 6 10 15 21 28 36 0 0
0 1 4 10 20 35 56 84 120 0 0
0 1 5 15 35 70 126 210 330 0 0
0 1 6 21 56 126 252 462 792 1287 0
0 0 5 25 80 205 456 917 1708 2994 0
0 0 4 27 104 305 756 1667 3368 6354 0
0 0 3 27 125 420 1161 2807 6147 12465 0
0 0 2 25 140 540 1666 4417 10480 22825 0
0 0 1 21 146 651 2247 6538 16808 39303 85228
iterative(B=10,N=10,K=5,arr): 85228
1 1 1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 10
0 1 3 6 10 15 21 28 36 45 55
0 1 4 10 20 35 56 84 120 165 220
0 1 5 15 35 70 126 210 330 495 715
0 1 6 21 56 126 252 462 792 1287 2002
0 0 5 25 80 205 456 917 1708 2994 4995
0 0 4 27 104 305 756 1667 3368 6354 11340
0 0 3 27 125 420 1161 2807 6147 12465 23760
0 0 2 25 140 540 1666 4417 10480 22825 46420
0 0 1 21 146 651 2247 6538 16808 39303 85228
memoize(b=3,n=3,k=2,arr): 7
0 1 0 0
0 0 2 0
0 1 3 0
0 0 2 7
iterative(B=3,N=3,K=2,arr): 7
1 1 1 1
0 1 2 3
0 1 3 6
0 0 2 7
memoize(b=3,n=3,k=3,arr): 10
0 1 1 0
0 0 2 0
0 1 3 0
0 1 4 10
iterative(B=3,N=3,K=3,arr): 10
1 1 1 1
0 1 2 3
0 1 3 6
0 1 4 10
[darshan@login3 project2_ver3]$ █

```