DESIGN AND ANALYSIS OF ALGORITHMS

SEMESTER PROJECT

M. Nabeed Haider Muhammad Omer

22i-0871 22i-0921

Section: G Section: G

TABLE OF CONTENTS:

* Q1

1. Pseudocode
2. Analysis
3. Working
4. Refrences

* Q2

1. Recursive Definition + Optimal Substructure
2. Pseudocode
3. Analysis
4. Working

* Q3

1. Pseudocode
2. Analysis
3. Working

* Q4

1. Pseudocode
2. Analysis
3. Working

**Q1**

**Pseudocode**

Function count\_ways(n, prev, memo)

    if (n == 0)

        return 1

    if ((n, prev) in memo)

        return memo[(n, prev)]

    ways = 0

    for i =prev + 1 to n + 1

        ways += count\_ways(n - i, i, memo)

    memo[(n, prev)] = ways

    return ways

Function CountWaysStrictDecreasing(n)

    if (n <= 1)

        return 0

    memo = {}

    ways = 0

    for i = 1 to n

        ways += count\_ways(n - i, i, memo)

    return ways

# Driver program

Function main()

    n = 9

    ways = CountWaysStrictDecreasing(n)

    print("Number of ways to represent {n} blocks as a strictly decreasing sequence: {ways}")

ANALYSIS

Count\_ways runs in O(n^2) and it is called n times overall time complexity is

O(n^3)

WORKING

The given calculates the number of ways n blocks can be arranged as sum of sequence of strictly decreasing blocks which is the requirement of the problem statement. It uses memoization to store values to reduce the time complexity. We consider all possibilities by using n-I,n in count\_ways as arguments and add this to the possible number of ways for each I between 1 to n , to give us correct result at the end.

REFRENCES

https://www.geeksforgeeks.org/count-ways-express-number-sum-consecutive-numbers/

**Q2**

Recursive Defination

max\_cal(start, end, m, n, arr, sum1):

1. Create a new Obj instance called temp.

2. Initialize temp's attributes: temp.value = 0, temp.index1 = 0.

3. Initialize max\_sum as the minimum integer value.

4. Initialize sum as 0.

5. If the difference between end and start is 1:

a. Set temp.value to arr[start][end].

b. Set temp.index1 to start.

6. Otherwise:

a. Iterate over a range from start to end - 1 (inclusive) using variable a:

i. Initialize sum as 0.

ii. Iterate over a range from start to a (exclusive) using variable i:

- Subtract arr[i][a] from sum.

iii. Set i based on the condition: if m is not equal to n, set i = a; otherwise, set i = a + 1.

iv. Iterate over a range from i to end - 1 (inclusive):

- If m is not equal to n, add arr[a][i+1] to sum; otherwise, add arr[a][i] to sum.

v. If sum is greater than max\_sum:

- Update max\_sum to sum.

- Update temp's attributes: temp.value = sum, temp.index1 = a.

7. If m is not equal to n:

- Set temp's start and end attributes.

- Return temp.

8. Subtract max\_sum from sum1.

9. If m is equal to 1, do nothing.

10. Otherwise:

a. Initialize index\_break as temp's index1.

b. Initialize index\_break2 as index\_break + 1.

c. Initialize count as 2.

d. Create a dynamic array obj\_arr of Obj instances with size count.

e. Set obj\_arr[0] to the result of calling max\_cal recursively with appropriate arguments.

f. Set obj\_arr[1] to the result of calling max\_cal recursively with appropriate arguments.

g. If m is equal to 2, subtract the maximum value from obj\_arr[0] and obj\_arr[1] from sum1.

h. Otherwise:

i. Iterate over a range from 1 to m (exclusive) using variable i:

- Find the maximum value and its index in obj\_arr.

- Create a new dynamic array temp of Obj instances with size count.

- Iterate over obj\_arr, excluding the maximum value:

\* Add the Obj instances to temp.

- Update sum1 by subtracting the value of the maximum Obj instance from obj\_arr.

- Update temp's attributes by recursively calling max\_cal with appropriate arguments for the maximum Obj instance.

- Update temp's attributes by recursively calling max\_cal with appropriate arguments for the maximum Obj instance.

- Assign temp to obj\_arr.

11. Set temp's value to sum1.

12. Return temp.

OPTIMAL SUB STRUCTURE

When we partition or attack, we basically rail road is divided into pieces that are not connected and the DP concept is applied to the segments individualy, so therefore as the number of segments increase the time complexity decreases

Pseudocode

Struct Obj

int value

int index1

int start

int end

Function max(a, b)

return a if a >= b else b

Function max\_cal(start, end, m, n, arr, sum1)

temp = new Obj

temp.index1 = 0

temp.value = 0

max\_sum = -∞

sum = 0

if end - start == 1 then

temp.value = arr[start][end]

temp.index1 = start

else

for a = start to end - 1

for i = start to a - 1

sum -= arr[i][a]

if m != n then

i = a

else

i = a + 1

for i to end - 1

if m != n then

sum += arr[a][i + 1]

else

sum += arr[a][i]

if sum > max\_sum then

max\_sum = sum

temp.value = sum

temp.index1 = a

if m != n then

temp.start = start

temp.end = end

return temp

sum1 -= max\_sum

if m == 1 then

# Do something

else

index\_break = temp.index1

index\_break2 = temp.index1 + 1

count = 2

obj\_arr = new array of Obj[count]

obj\_arr[0] = max\_cal(start, index\_break, m, m - 1, arr, sum1)

obj\_arr[1] = max\_cal(index\_break2, end - 1, m, m - 1, arr, sum1)

if m == 2 then

sum1 -= max(obj\_arr[0].value, obj\_arr[1].value)

else

for i = 1 to m

max = obj\_arr[0].value

index = 0

for j = 1 to count

if obj\_arr[j].value > max then

max = obj\_arr[j].value

index = j

count++

temp = new array of Obj[count]

index2 = 0

for k = 0 to count - 1

if k != index then

temp[index2++] = obj\_arr[k]

else

sum1 -= obj\_arr[index].value

temp[index2++] = max\_cal(obj\_arr[index].start, obj\_arr[index].index1, m, m - 1, arr, sum1)

temp[index2++] = max\_cal(obj\_arr[index].index1 + 1, obj\_arr[index].end, m, m - 1, arr, sum1)

obj\_arr = new array of Obj[count]

for k = 0 to count - 1

obj\_arr[k] = temp[k]

temp.value = sum1

return temp

Function Calculate\_Products(arr, cols, rows, depos)

for i = 0 to rows - 1

for j = i + 1 to cols - 1

arr[i][j] = depos[i] \* depos[j]

for i = 0 to rows - 1

for j = 0 to cols - 1

print arr[i][j] + " "

print newline

Function Cal\_Sum(arr, rows, cols)

sum = 0

for i = 0 to rows - 1

for j = 0 to cols - 1

sum += arr[i][j]

return sum

Function main()

depos = [4, 5, 3, 2, 1, 4]

attacks = 1

vals = new array of arrays

for i = 0 to 5

vals[i] = new array of 6

for j = 0 to 5

vals[i][j] = 0

Calculate\_Products(vals, 6, 6, depos)

sum = Cal\_Sum(vals, 6, 6)

A = max\_cal(0, 6, 3, 3, vals, sum)

print A.value

ANALYSIS

Overall time complexity is O(mn^2)

WORKING

The program starts by defining a structure Obj to represent an object with attributes for storing information about a subsequence of blocks. Then, a max function is defined to return the maximum of two integers. The max\_cal function is the main part of the program, tasked with calculating the maximum sum of a sequence of blocks under certain constraints. It iterates over different subsequences within a given range, calculating their sums and updating the maximum sum accordingly. The function recursively divides the sequence into smaller subsequences if needed, based on parameters m and n. The calculated maximum sum is stored in the value attribute of the Obj structure. The Calculate\_Products function computes the products of pairs of block values, while Cal\_Sum calculates the total sum of elements in the 2D array representing block values. Finally, in the main function, the program initializes block values, computes products and sums, and then calls max\_cal to find the maximum sum of a strictly decreasing sequence of blocks within the specified constraints, which is then printed out. Overall, the program efficiently solves the problem of finding the maximum sum of a sequence of blocks using dynamic programming principles and recursion.

**Q3**

**Pseudocode**

Function Find\_Missing\_Segment():

arraySize = 5

arr[arraySize]

maxValue = arr[0]

Output "Enter array elements:"

For i from 0 to arraySize - 1:

Input arr[i]

maxValue = arr[0]

For i from 1 to arraySize - 1:

If arr[i] > maxValue:

maxValue = arr[i]

maxValue += 1

frequency[maxValue + 1] = {0}

For j from 0 to arraySize - 1:

frequency[arr[j]]++

missingPositiveNumber = -1

For k from 0 to maxValue + 1:

If frequency[k] == 0:

missingPositiveNumber = k

Break

If missingPositiveNumber != -1:

start = 1, end = 1

check = false

For l from 0 to arraySize - 1:

If arr[l] > missingPositiveNumber:

end = l + 1

Break

Else If arr[l] == maxValue - 1:

end = l + 1

check = true

Break

If check:

For m from end to arraySize - 1:

If arr[m] == maxValue - 1:

check = false

Break

If not check:

Output "Segments:"

Output start, end

Output end + 1, arraySize

Else:

Output -1

Else:

Output missingPositiveNumber

Find\_Missing\_Segment()

ANALYSIS

Overall time complexity is O(arraySize).

WORKING

The program begins by prompting the user to input five integers to populate an array. It then scans through this array to identify the maximum value present. Next, it initializes a frequency array to track the occurrences of each integer within the input array. By iterating through the frequency array, the program identifies the first instance of a zero-frequency element, indicating a missing positive number within the original array. Subsequently, the program locates the first segment of consecutive integers in the array that precedes the missing positive number. It verifies the validity of this segment and outputs its start and end indices if valid. If no missing positive number is found, the program outputs the identified missing number. Overall, the program efficiently identifies and outputs missing segments within the given array of integers.

**Q4**

**Pseudocode**

Function maxPower(rows, cols, levels):

memoizationTable = 2D array of size rows x cols, initialized to zeros

maxIndexes = 1D array of size cols, initialized to zeros

secondMaxIndexes = 1D array of size cols, initialized to zeros

for i from 0 to rows - 1:

memoizationTable[i][0] = levels[i][0]

max1 = -1, max2 = -1

for i from 0 to rows - 1:

if memoizationTable[i][0] > max1:

max2 = max1

secondMaxIndexes[0] = maxIndexes[0]

max1 = memoizationTable[i][0]

maxIndexes[0] = i

else if memoizationTable[i][0] > max2:

max2 = memoizationTable[i][0]

secondMaxIndexes[0] = i

for i from 1 to cols - 1:

for j from 0 to rows - 1:

temp = maxIndexes[i - 1] if temp != j else secondMaxIndexes[i - 1]

memoizationTable[j][i] = memoizationTable[temp][i - 1] + levels[j][i]

max1 = 0, max2 = 0

for k from 0 to rows - 1:

if memoizationTable[k][i] > max1:

max2 = max1

secondMaxIndexes[i] = maxIndexes[i]

max1 = memoizationTable[k][i]

maxIndexes[i] = k

else if memoizationTable[k][i] > max2:

max2 = memoizationTable[k][i]

secondMaxIndexes[i] = k

finalMax = -1

for i from 0 to rows - 1:

if finalMax < memoizationTable[i][cols - 1]:

finalMax = memoizationTable[i][cols - 1]

return finalMax

Function main():

numRows, numCols = input from user

powerLevels = 2D array of size numRows x numCols

for i from 0 to numRows - 1:

for j from 0 to numCols - 1:

powerLevels[i][j] = input from user

if numRows == 1:

max = maximum power in the row

Output "Max Power is: " + max

Exit

Output "Max Power is: " + maxPower(numRows, numCols, powerLevels)

ANALYSIS

Overall time complexity is O(cols \* rows^2).

WORKING

The provided C++ program aims to find the maximum power that can be achieved by selecting blocks from a grid of power levels, subject to certain rules. It first initializes a memoization table and two arrays to track the maximum and second maximum indexes for each column. Then, it iterates through each column of the grid, updating the memoization table based on the maximum and second maximum indexes from the previous column. This process efficiently calculates the maximum power achievable at each cell of the memoization table. After populating the memoization table, the program scans the last column to find the maximum power attained. Finally, it outputs this maximum power. Overall, the program employs dynamic programming principles to optimize the computation of the maximum power while traversing the grid column by column, resulting in an efficient solution.

Top of Form