Applied Algorithms

Written Assignment-2

Jaya Sandeep Ketha

October 2023

Q1. For (b), (c), (d), solve the following recursions with Big-O notation, assuming that $T(1) = constant$. Please use the techniques covered in the course.

(a) Analyze the asymptotic (big-O) running times for the modified Mergesort and Quicksort algorithms that you implemented. Hint: Write down the recurrences for your algorithms with k = 2, 3, 4 respectively first

Answer:

Merge Sort:
k = 2:
To solve the recurrence relation $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$ using the unrolling method, we'll iteratively expand the recurrence until we reach the base case. The unrolling method involves repeatedly substituting the recurrence into itself and expanding it until we can easily identify a pattern. Here's the unrolling process:

1. Start with the original recurrence: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

2. Expand it once:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n = 2 \cdot \left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 4 \cdot T\left(\frac{n}{4}\right) + 2n$$

3. Expand it again:

$$T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2n = 4 \cdot \left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n = 8 \cdot T\left(\frac{n}{8}\right) + 3n$$

4. Continue this process k times until we reach the base case $T(1)$:

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + kn$$

5. When $\frac{n}{2^k} = 1$, we reach the base case, which means $\frac{n}{2^k} = 1$, or $n = 2^k$. Solving for k, we get $k = \log_2(n)$.

6. Now, we can write the recurrence relation in terms of the base case $T(1)$ by substituting $k = \log_2(n)$:

$$T(n) = 2^{log_2(n)} \cdot T\frac{n}{2^{log_2(n)}} + n \cdot log_2(n)$$

7. Now we get

$$T(n) = n \cdot T(1) + n \cdot log_2(n)$$

Now, we can analyze the dominant term in the equation to determine the time complexity in Big O notation. The dominant term here is $n \cdot log_2(n)$. While $T(1)$ is a constant, it's the $n \cdot log_2(n)$ term that dominates the behavior of $T(n)$.

Therefore, the time complexity of $T(n)$ is $O(nlog_2(n))$.

So, the solution to the recurrence $T(n) = 2T(n/2) + n$ with $T(1)$ as a constant using Big O notation is $O(nlog_2(n))$.

k = 3:
To solve the recurrence relation $T(n) = 3 \cdot T\left(\frac{n}{3}\right) + n$ using the unrolling method, we'll iteratively expand the recurrence until we reach the base case. The unrolling method involves repeatedly substituting the recurrence into itself and expanding it until we can easily identify a pattern. Here's the unrolling process:

1. Start with the original recurrence: $T(n) = 3 \cdot T\left(\frac{n}{3}\right) + n$

2. Expand it once:

$$T(n) = 3 \cdot T\left(\frac{n}{3}\right) + n = 3 \cdot \left[3 \cdot T\left(\frac{n}{9}\right) + \frac{n}{3}\right] + n = 9 \cdot T\left(\frac{n}{9}\right) + 2n$$

3. Expand it again:

$$T(n) = 9 \cdot T\left(\frac{n}{9}\right) + 2n = 9 \cdot \left[3 \cdot T\left(\frac{n}{27}\right) + \frac{n}{9}\right] + 2n = 27 \cdot T\left(\frac{n}{27}\right) + 3n$$

4. Continue this process k times until we reach the base case $T(1)$:

$$T(n) = 3^k \cdot T\left(\frac{n}{3^k}\right) + kn$$

5. When $\frac{n}{3^k} = 1$, we reach the base case, which means $\frac{n}{3^k} = 1$, or $n = 3^k$. Solving for k, we get $k = \log_3(n)$.

6. Now, we can write the recurrence relation in terms of the base case $T(1)$ by substituting $k = \log_3(n)$:

$$T(n) = 3^{log_3(n)} \cdot T\frac{n}{3^{log_3(n)}} + n \cdot log_3(n)$$

7. Now we get

$$T(n) = n \cdot T(1) + n \cdot log_3(n)$$

Now, we can analyze the dominant term in the equation to determine the time complexity in Big O notation. The dominant term here is $n \cdot log_3(n)$. While $T(1)$ is a constant, it's the $n \cdot log_3(n)$ term that dominates the behavior of $T(n)$.

Therefore, the time complexity of $T(n)$ is $O(nlog_3(n))$.

So, the solution to the recurrence $T(n) = 3T(n/3) + n$ with $T(1)$ as a constant using Big O notation is $O(nlog_3(n))$.

k = 4:
To solve the recurrence relation $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + n$ using the unrolling method, we'll iteratively expand the recurrence until we reach the base case. The unrolling method involves repeatedly substituting the recurrence into itself and expanding it until we can easily identify a pattern. Here's the unrolling process:

1. Start with the original recurrence: $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + n$

2. Expand it once:

$$T(n) = 4 \cdot T\left(\frac{n}{4}\right) + n = 4 \cdot \left[4 \cdot T\left(\frac{n}{16}\right) + \frac{n}{4}\right] + n = 16 \cdot T\left(\frac{n}{16}\right) + 2n$$

3. Expand it again:

$$T(n) = 16 \cdot T\left(\frac{n}{16}\right) + 2n = 16 \cdot \left[4 \cdot T\left(\frac{n}{64}\right) + \frac{n}{16}\right] + 2n = 64 \cdot T\left(\frac{n}{64}\right) + 3n$$

4. Continue this process k times until we reach the base case $T(1)$:

$$T(n) = 4^k \cdot T\left(\frac{n}{4^k}\right) + kn$$

5. When $\frac{n}{4^k} = 1$, we reach the base case, which means $\frac{n}{4^k} = 1$, or $n = 4^k$. Solving for k, we get $k = \log_4(n)$.

6. Now, we can write the recurrence relation in terms of the base case $T(1)$ by substituting $k = \log_4(n)$:

$$T(n) = 4^{log_4(n)} \cdot T\frac{n}{4^{log_4(n)}} + n \cdot log_4(n)$$

7. Now we get

$$T(n) = n \cdot T(1) + n \cdot log_4(n)$$

Now, we can analyze the dominant term in the equation to determine the time complexity in Big O notation. The dominant term here is $n \cdot log_4(n)$. While $T(1)$ is a constant, it's the $n \cdot log_4(n)$ term that dominates the behavior of $T(n)$.

Therefore, the time complexity of $T(n)$ is $O(nlog_4(n))$.

So, the solution to the recurrence $T(n) = 3T(n/4) + n$ with $T(1)$ as a constant using Big O notation is $O(nlog_4(n))$.

Randomized Quick Sort:
k = 2:
We can use the substitution method to verify that O(n lg n) is an upper bound for the solution to the recurrence. We show that $T(n) \leq d \cdot n \cdot lgn$, where d is a suitable positive constant.
We have
Recurrence Relation for k = 2 as:

$$T(n) = 2T(n/2) + n$$

$$T(n) \leq d \cdot 2 \cdot \frac{n}{2} log \frac{n}{2} + n$$
$$\leq d \cdot n \cdot log \frac{n}{2} + n$$
$$\leq d \cdot n \cdot logn - d \cdot n \cdot log2 + n$$

as long as $d \geq -1/log(n/2)$ we can have T(n) = O(n log n).

k = 3:
We can use the substitution method to verify that O(n lg n) is an upper bound for the solution to the recurrence. We show that $T(n) \leq d \cdot n \cdot lgn$, where d is a suitable positive constant.
We have
Recurrence Relation for k = 3 as:

$$T(n) = 3T(n/3) + n$$

$$T(n) \leq d \cdot 3 \cdot \frac{n}{3} log \frac{n}{3} + n$$
$$\leq d \cdot n \cdot log \frac{n}{3} + n$$
$$\leq d \cdot n \cdot logn - d \cdot n \cdot log3 + n$$

as long as $d \geq -1/log(n/3)$ we can have T(n) = O(n log n).
k = 4:
We can use the substitution method to verify that O(n lg n) is an upper bound for the solution to the recurrence. We show that $T(n) \leq d \cdot n \cdot lgn$, where d is a suitable positive constant.

We have
Recurrence Relation for k = 4 as:

$$T(n) = 4T(n/4) + n$$

$$T(n) \leq d \cdot 4 \cdot \frac{n}{4} log \frac{n}{4} + n$$
$$\leq d \cdot n \cdot log \frac{n}{4} + n$$
$$\leq d \cdot n \cdot logn - d \cdot n \cdot log4 + n$$

as long as $d \geq -1/log(n/4)$ we can have T(n) = O(n log n).

(b) T(n) = T(n/3) + n
Answer:

To solve the recurrence relation $T(n) = T\left(\frac{n}{3}\right) + n$ using the unrolling method, we'll iteratively expand the recurrence until we reach the base case. The unrolling method involves repeatedly substituting the recurrence into itself and expanding it until we can easily identify a pattern. Here's the unrolling process:

1. Start with the original recurrence: $T(n) = T\left(\frac{n}{3}\right) + n$

2. Expand it once:
$$T(n) = T\left(\frac{n}{3}\right) + n = \left[T\left(\frac{n}{9}\right) + \frac{n}{3}\right] + n = T\left(\frac{n}{9}\right) + \frac{n}{3} + n$$

3. Expand it again:
$$T(n) = T\left(\frac{n}{9}\right) + \frac{n}{3} + n = \left[T\left(\frac{n}{27}\right) + \frac{n}{9}\right] + \frac{n}{3} + n = T\left(\frac{n}{27}\right) + \frac{n}{9} + \frac{n}{3} + n$$

4. Continue this process $k + 1$ times until we reach the base case $T(1)$:
$$T(n) = T\left(\frac{n}{3^k}\right) + \frac{n}{3^{k-1}} + \frac{n}{3^{k-2}} + \ldots + \frac{n}{3} + n$$

5. When $\frac{n}{3^k} = 1$, we reach the base case, which means $\frac{n}{3^k} = 1$, or $n = 3^k$. Solving for k, we get $k = \log_3(n)$.

6. Now, we can write the recurrence relation in terms of the base case $T(1)$ by substituting $k = \log_3(n)$:
$$T(n) = T(1) + \frac{n}{3^{log_3(n)-1}} + \frac{n}{3^{log_3(n)-2}} + \ldots + \frac{n}{3} + n$$

7. Now we get
$$T(n) = T(1) + 3 + 9 + \ldots + \frac{n}{3} + n$$

8. Using the formula for the sum of a geometric series:
$$\frac{3 \cdot (1 - 3^k)}{1 - 3}$$

10. Since $k = \log_3(n)$, we can simplify further:
$$\frac{3 \cdot (1 - 3^k)}{-2} = \frac{3 \cdot (1 - 3^{log_3(n)})}{-2} = \frac{3 \cdot (1 - n)}{-2}$$

11. Finally, we can write the solution for $T(n)$ in terms of the base case $T(1)$:
$$T(n) = T(1) + \frac{3n - 3}{2}$$

Now, we can find the solution for $T(n)$ based on the base case $T(1)$.

After unrolling the recurrence, we found that:

$$T(n) = T(1) + \frac{3n-3}{2}$$

Now, we can analyze the dominant term in the equation to determine the time complexity in Big O notation. The dominant term here is $3n/2$. While $T(1)$ is a constant and $\frac{-3}{2}$ is also constant, it's the $3n/2$ term that dominates the behavior of $T(n)$.

Therefore, the time complexity of $T(n)$ is $O(n)$.

So, the solution to the recurrence $T(n) = T(n/3) + n$ with $T(1)$ as a constant using Big O notation is $O(n)$.

(c) T(n) = T(n/2) + n^2
Answer:

To solve the recurrence relation $T(n) = T\left(\frac{n}{2}\right) + n^2$ using the unrolling method, we'll iteratively expand the recurrence until we reach the base case. The unrolling method involves repeatedly substituting the recurrence into itself and expanding it until we can easily identify a pattern. Here's the unrolling process:

1. Start with the original recurrence: $T(n) = T\left(\frac{n}{2}\right) + n^2$

2. Expand it once:

$$T(n) = T\left(\frac{n}{2}\right) + n^2 = \left[T\left(\frac{n}{4}\right) + (\frac{n}{2})^2\right] + n^2 = T\left(\frac{n}{4}\right) + (\frac{n}{2})^2 + n^2$$

3. Expand it again:

$$T(n) = T\left(\frac{n}{4}\right) + (\frac{n}{4})^2 + n^2 = \left[T\left(\frac{n}{8}\right) + (\frac{n}{4})^2\right] + (\frac{n}{2})^2 + n^2 = T\left(\frac{n}{8}\right) + (\frac{n}{4})^2 + (\frac{n}{2})^2 + n^2$$

4. Continue this process k times until we reach the base case $T(1)$:

$$T(n) = T\left(\frac{n}{2^k}\right) + n^2 + (\frac{n}{2})^2 + (\frac{n}{4})^2 + (\frac{n}{8})^2 + \dots$$

5. When $\frac{n}{2^k} = 1$, we reach the base case, which means $\frac{n}{2^k} = 1$, or $n = 2^k$. Solving for k, we get $k = \log_2(n)$.

$$T(n) = T\left(\frac{n}{2^{log_2(n)}}\right) + n^2 + (\frac{n}{2})^2 + (\frac{n}{4})^2 + (\frac{n}{8})^2 + \dots$$

$$T(n) = T(1) + +n^2 + (\frac{n}{2})^2 + (\frac{n}{4})^2 + (\frac{n}{8})^2 + \dots$$

The dominant term here is n^2. While $T(1)$ is a constant.

We can bound this from above as $T(n) \leq 2(n^2)$ and from below with $T(n) \geq n^2$, giving $T(n) = O(n^2)$.

Note: We do not need to use induction to prove this; since we can create an upper bound of $T(n) \leq c_1 f(n)$ and a lower bound of $T(n) \geq c_2 f(n)$, we can use the definition of O(n) to establish the time complexity of the relation.

Therefore, the time complexity of $T(n)$ is $O(n^2)$.

So, the solution to the recurrence $T(n) = T(n/2) + n^2$ with $T(1)$ as a constant using Big O notation is $O(n^2)$.

(d) T(n) = T(n/3) + 2

Answer:

To solve the recurrence relation $T(n) = T\left(\frac{n}{3}\right) + 2$, we can use the unrolling method. This method involves repeatedly substituting the recurrence into itself and expanding it until we reach the base case. Here's the process:

1. Start with the original recurrence: $T(n) = T\left(\frac{n}{3}\right) + 2$.
2. Expand it once:

$$T(n) = T\left(\frac{n}{3}\right) + 2 = \left[T\left(\frac{n}{9}\right) + 2\right] + 2 = T\left(\frac{n}{9}\right) + 4$$

3. Expand it again:

$$T(n) = T\left(\frac{n}{9}\right) + 4 = \left[T\left(\frac{n}{27}\right) + 2\right] + 4 = T\left(\frac{n}{27}\right) + 6$$

4. Continue this process k times until we reach the base case $T(1)$:

$$T(n) = T\left(\frac{n}{3^k}\right) + 2k$$

5. When $\frac{n}{3^k} = 1$, we reach the base case, which means $\frac{n}{3^k} = 1$, or $n = 3^k$. Solving for k, we get $k = \log_3(n)$.

6. Now, we can write the recurrence relation in terms of the base case $T(1)$:

$$T(n) = T(1) + 2\log_3(n)$$

So, the solution to the recurrence relation $T(n) = T\left(\frac{n}{3}\right) + 2$ is $T(n) = T(1) + 2\log_3(n)$.

In Big O notation, the time complexity of this recurrence is $O(\log_3(n))$.

Q2. Given two integers x and n, describe an algorithm that computes x^n in pseudocode and analyze its time complexity. Is there any way to go faster than O(n)?

Answer:

One way to solve x^n problem is to use the popular way recursive method.

```
# Recursion Method
# Time Complexity is O(n)

x, n —> Integer
Step 1: Take x and n input
Step 2: Calculate power(x, n) method
function power(x, n):
check base condition if n==0 return 1
  # If x^0 return 1
    if (n == 0):
        return 1
check base condition if n==1 return x
   # If we need to find of x^1
    if (n == 1):
        return x
recursively call power(x,n-1) and go to step 2;
    # For all other cases
    return x * power(x, n - 1)
Step 3: Print result
```

In the provided code, the power function calculates x^n using a recursive approach. Let's analyze its time complexity using a recurrence relation.

The recurrence relation for this code can be expressed as follows:

$$T(n) = T(n-1) + O(1)$$

Here:
- T(n) represents the time taken to calculate x^n when n is a positive integer.
- T(n - 1) represents the time taken to calculate $x^{(n-1)}$.
- O(1) represents the constant time taken for basic operations (comparisons and multiplications).

Now, let's analyze this recurrence relation by expanding it:

$$T(n) = T(n-1) + O(1)$$

$$T(n-1) = T(n-2) + O(1)$$

$$T(n-2) = T(n-3) + O(1)$$

...

$$T(2) = T(1) + O(1)$$

$$T(1) = T(0) + O(1)$$

Let's sum up all these equations:

$$T(n) = T(0) + O(1) + O(1) + ... + O(1)$$

Since we have O(1) at each step from 0 to n, and there are n steps in total, the total time complexity is:
T(n) = O(1) + O(1) + ... + O(1) (n times)
T(n) = O(n)

So, the time complexity of the provided code for calculating x^n using recursion is O(n). It means that the time taken by the code is directly proportional to the value of the exponent n.

Time Complexity: O(n) because pow(x,n) is called recursively for each number from 1 to n.

Yes, there is another way to go faster than O(n). Instead of recursive approach, we can use divide and conquer technique to improve the time complexity by calling pow(x, power/2).

```
# Divide and Conquer Method
# Time Complexity is O(log(n))

x, n --> Integer
Step 1: Take x and n input

Step 2: Calculate power(x, n) method

function power(x, n)

check for base condition if n==0 return 1
    if n == 0 then
        return 1
```

```
    else
        m <— power(x, n/2)

check if n is even
    if n%2 == 0 then

n—> even call pow(x, n/2) * power(x, n/2)
        return m * m        // if n is even

n—> odd – call pow(x, n/2) * power(x, n/2) * x
if n is odd
    else
        return m * m * x     // if n is odd
```

Step 3: Print result

The recurrence relation for exponential problem using divide and conquer is given as,

T(n) = T(n/2) + 1

Using iterative method, it is not difficult to show that the solution to this equation would be O(log2n)
Let us solve this by iterative approach

T (n) = T(n/2) + 1 ... (1)

Substitute n by n/2 in Equation (1) to find T(n/2)

T(n/2) = T(n/4) + 1 ... (2)

Substitute value of T(n/2) in Equation (1),

$T(n) = T(n/2^2) + 1 \ldots (3)$

Substitute n by n/2 in Equation (2) to find T(n/4),

T(n/4) = T(n/8) + 1

Substitute value of T(n/4) in Equation (3),

$T(n) = T(n/2^3) + 3$

After k iterations,

$T(n) = T(n/2^k) + k$

Binary tree created by binary search can have maximum height $log_2(n)$

So, $k = log_2(n)$

T(n) = T(2k/2k) + k

= T(1) + k

From base case of recurrence,

T(n) = 1 + k = 1 + $log_2(n)$

T(n) = O($log_2(n)$)

Time Complexity: O(log n)

Q3. Finds the median of two sorted arrays of equal length n in O(n) time. Describe your algorithm in pseudocode and analyze its time complexity. Bonus Question(10 Points) Can you do this faster?.

Answer:

```
# Simply count while merging.
# Time Complexity O(n)

arr1, arr2 —> Array 1 and Array 2
Step 1: Take arr1 and arr2 as input in function findMedianSortedArrays.
 function findMedianSortedArrays(arr1, arr2):

Step 2: Calculate size of input array and create an empy array to store elements after
merge.
    n = length of arr1
    merged = []  # Create an empty array to store the merged values

Step 3: Create two pointers to iterate over arr1 and arr2.
    i = 0  # Pointer for arr1
    j = 0  # Pointer for arr2

Step 4: Loop over arr1 and arr2 to merge elements in sorted order.
    while i < n and j < n:
        if arr1[i] <= arr2[j]:
            merged.append(arr1[i])
            i = i + 1
        else:
            merged.append(arr2[j])
            j = j + 1

Step 5: If elements are left in arr1 or arr2 merge append at end of merged array.
    # If there are remaining elements in arr1, append them to the merged array
    while i < n:
        merged.append(nums1[i])
        i = i + 1

    # If there are remaining elements in arr2, append them to the merged array
    while j < n:
        merged.append(nums2[j])
        j = j + 1

Step 6: Return median of this sorted array.
    # The merged array now contains all elements from both arrays
```

```
    # The median is the average of the two middle elements
    mid = n  # Middle index of merged array
    return (merged[mid - 1] + merged[mid]) / 2.0
```

For this particular algorithm, the time complexity analysis is straightforward:

Initializing variables and creating the merged array: O(1).

Merging the sorted arrays: This step involves iterating through both input arrays entirely, and in the worst case, it takes O(n) time, where n is the total number of elements in both arrays.

Copying any remaining elements: If there are remaining elements in either of the input arrays, they are copied to the merged array. This also takes O(n) time in the worst case.

Calculating the median: Finding the middle element(s) in the merged array can be done in O(1) time.

Overall, the time complexity is dominated by the merging and copying steps, both of which take O(n) time in the worst case. Therefore, the overall time complexity of this algorithm for finding the median of two sorted arrays is O(n).

Yes, there is a faster way to achieve median of two sorted arrays of equal length than this approach which uses Binary Search with time complexity O(log n).

```
# Binary search
# Time Complexity O(log n)

Step 1: Initialize count_less_than_or_equal_to_mid function to calculate the number
of elements less than or equal to mid in the given arrays.

function count_less_than_or_equal_to_mid(mid, arrays):
    count = 0
    for array in arrays:
        count += len([x for x in array if x <= mid])
    return count

Step 2: Initialize low = -10^9, high = 10^9 and pos = n
def find_kth_element(arrays, n):
    ans = 0.0
    low = -1e9
    high = 1e9
    pos = n

Step 3: Run a loop while(low <= high):
# Binary search to find the kth element
    while low <= high:

Step 4: Calculate mid = low + (high - low)>>1
        mid = low + (high - low) // 2

Step 5: Find total elements less or equal to mid in the given arrays
        count = count_less_than_or_equal_to_mid(mid, arrays)

Step 6: If the count is less or equal to pos
        if count <= pos:
```

```
Step 7: Update low = mid + 1
            low = mid + 1

Step 8: Else high = mid — 1
        else:
            high = mid - 1

Step 9: Store low in ans, i.e., ans = low.
            ans = low
            # Update pos and repeat the binary search to find the (n-1)th element
            pos = n - 1
            low = -1e9
            high = 1e9

Step 10: Again follow step3 with pos as  n - 1

Step 11: Return (sum + low * 1.0)/2
```

Q4. Paving a road. You have a road that is n meters long, and you also have stones that are 2, 3, 5 meters long respectively. Design a dynamic programming algorithm that counts the number of ways in which the road can be paved by these kinds of stones. Your algorithm should contain a recursive formula that uses memoization. Describing your algorithm in pseudocode is not necessary.

Answer:

```python
def paving_ways_count(n):
    # If length of way is 0, no way to construct it.
    if n == 0:
        return 0
    # Create a memoization table to store intermediate results
    memo_table = [-1] * (n + 1)

    # Recursive function to count the number of ways to pave the road
    def paving_ways_recursive_count(length):
        # Base case: If the length is 0, there is one way (no stones)
        if length == 0:
            return 1

        # If the length is negative, it cannot be paved
        if length < 0:
            return 0

        # If the result is already memoized, return it
        if memo_table[length] != -1:
            return memo_table[length]

        # Initialize the count of ways to pave the road to 0
        count = 0

        # Use stones of lengths 2, 3, and 5 to pave the road
        count += paving_ways_recursive_count(length - 2)
```

```
        count += paving_ways_recursive_count(length - 3)
        count += paving_ways_recursive_count(length - 5)

        # Memoize the result
        memo_table[length] = count
        return count

    # Start the recursion from the length of the road
    return paving_ways_recursive_count(n)

# Example usage:
n = 10  # Length of the road
count = paving_ways_count(n)
print(f"Number of ways to pave a {n}-meter road: {count}")
```

Note: Different combinations of stones are also considered as one unique way to lay path.

Here's an overview of how the algorithm works:

Initialization: Initialize a memoization table to store intermediate results. This table is used to avoid redundant calculations.

Recursive Function: Define a recursive function $paving_waysrecursive_count$ that takes the current length of the road as its parameter.

Base Cases: If the length of the road is 0, it means there are no stones to place, so there is only one way (no stones used).
If the length becomes negative during recursion, it's not possible to pave the road, so return 0.

Memoization: Before making further calculations, check if the result for the current road length has already been computed and stored in the memoization table. If yes, return the stored result to avoid redundant calculations.

Calculation: Initialize a count variable to keep track of the number of ways to pave the road.

Recursively calculate the number of ways to pave the road by considering three options:
Using a stone of length 2, which reduces the road length by 2 meters.
Using a stone of length 3, which reduces the road length by 3 meters.
Using a stone of length 5, which reduces the road length by 5 meters.
Sum up these counts to get the total number of ways.

Memoization (Again): Store the calculated count in the memoization table for the current road length to avoid recalculating it in the future.

Return Result: Finally, return the count of ways to pave the entire road of length n meters by invoking the recursive function with the initial road length.

Q5. Grid pathway problem. Given a grid of size n × m, find a dynamic programming algorithm that calculates the number of shortest pathways that starts from the bottom-left vertex and ends at the top-right vertex. Your algorithm should contain a recursive formula that uses memoization. Describing your algorithm in pseudo code is not necessary.

Answer:

```
def shortest_paths_counter(n, m):
    # Creating a memoization table to store intermediate results
    memo_table = [[-1 for _ in range(m)] for _ in range(n)]

    # Recursive function to calculate the number of shortest paths

    def recursivePathCount(my_row, my_column):
        # Base case: if we reach the top-right vertex, return 1
        if my_row == 0 and my_column == m - 1:
            return 1

        # If the result is already memoized, return it
        if memo_table[my_row][my_column] != -1:
            return memo_table[my_row][my_column]

        # Initialize the count of paths to 0
        count = 0

        # Move right if within bounds
        if my_column < m - 1:
            count += recursivePathCount(my_row, my_column + 1)

        # Move up if within bounds
        if my_row > 0:
            count += recursivePathCount(my_row - 1, my_column)

        # Memoize the result
        memo_table[my_row][my_column] = count
        return count

    # Start the recursion from the bottom-left vertex
    return recursivePathCount(n - 1, 0)

# Example usage:
n = 3
m = 3

# Print the number of shortest paths
count = shortest_paths_counter(n, m)
print(f"Number of shortest paths in a {n}x{m} grid: {count}")
```

Here's a rough explanation of the algorithm:

Initialization: The function shortest_paths_counter(n, m) takes two parameters, n and m, representing the dimensions of the grid. It initializes a memoization table (memo_table) of size n x m to store intermediate results. All entries in this table are initialized to -1.

Recursive Function: The algorithm defines a recursive function recursivePathCount(my_row, my_column) that computes the number of shortest paths from a given position (my_row, my_column) to the top-right vertex. The function takes two parameters, my_row and my_column, representing the current position in the grid.

Base Case: If the current position is at the top-right vertex (i.e., my_row is 0 and my_column is m - 1), we've reached the destination, and there's only one shortest path. In this case, the function returns 1.

Memoization Check: Before performing further calculations, the algorithm checks if the result for the current position has already been computed and stored in the memo_table. If it has, the function returns the memoized result to avoid redundant calculations.

Path Count Calculation: The algorithm initializes a count variable to keep track of the number of shortest paths from the current position. It then recursively explores two possible moves:

Move right (if within bounds): The algorithm recursively calls $recursivePathCount$(my_row, my_column + 1) to count the number of shortest paths when moving right.

Move up (if within bounds): The algorithm recursively calls $recursivePathCount$(my_row - 1, my_column) to count the number of shortest paths when moving up.

Memoization (Again): After calculating the count of paths from the current position, the algorithm stores this count in the memo_table for that position to memoize the result.

Return Result: The recursive function returns the count of shortest paths from the bottom-left vertex to the top-right vertex.

Starting the Recursion: Finally, the main function starts the recursion by calling recursivePathCount(n - 1, 0), which starts from the bottom-left vertex and computes the number of shortest paths.