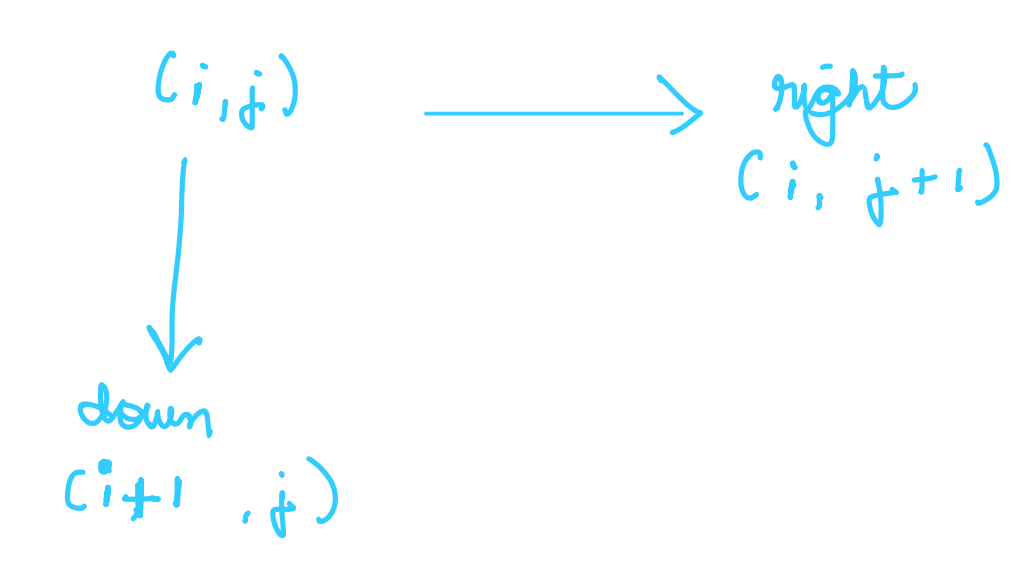
Assignment 6

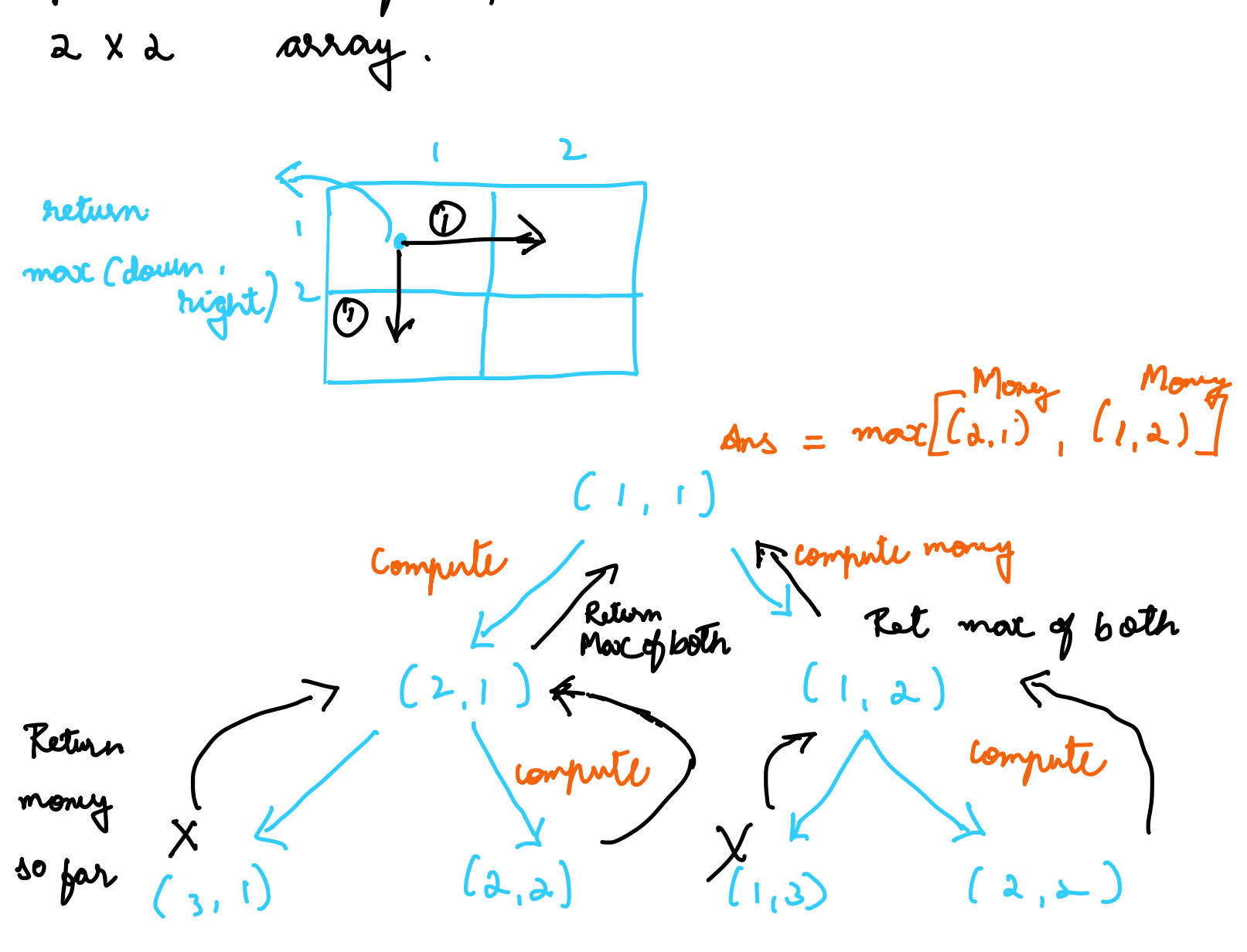
Wednesday, 15 November 2023 5:45 AM

1a) Given $n \times m$ array G, at each position (i, j) where $i = 1$ to n and $j = 1$ to m, we can either go down or right



→ Computing the subproblem

Let us break down the entire problem. Therefore, let us consider a 2 x 2 array.



Define $DP(i,j)$ as the maximum amount of money that can be collected starting from square $(1,1)$ to square (i,j) in the game. The subproblems are defined as finding the maximum amount of money at each position (i,j) on the board.

$$DP(i,j) = \max \begin{cases} 0 & \text{if } i=1 \text{ and } j=1 \\ -\infty & \text{if } i>n \text{ or } j>m \text{ or } G[i][j]=0 \text{ (trap)} \\ \max(DP(i+1,j), DP(i,j+1)) + G[i][j] & \text{if } G[i][j]>0 \text{ (gold)} \\ \frac{1}{2}\cdot\max(DP(i+1,j), DP(i,j+1)) & \text{if } G[i][j]=-2 \text{ (lose half)} \\ 0 & \text{if } G[i][j]=-1 \text{ (lose all)} \end{cases}$$

Quality Function:

The quality function $Q(i,j)$ measures the quality of the subproblem at position (i,j). It takes into account the accumulated gold collected so far and the position factor.

$$Q(i,j) = \text{Accumulated Gold}(i,j) + \text{Position Factor}(i,j)$$

where:

$$\text{Accumulated Gold}(i,j) = \begin{cases} 0 & \text{if } G[i][j] \le 0 \text{ (trap, lose half, or lose all)} \\ G[i][j] & \text{if } G[i][j] > 0 \text{ (gold)} \end{cases}$$

and

$$\text{Position Factor}(i,j) = n + m - i - j$$

This quality function combines the gold collected so far with a position factor, giving higher quality to subproblems closer to the finish.

1b) Recursive Function

func rec (i, j, money) ← initialize money = 0

if i or j is out of bound
return 0

case 1 : G[i][j] = +ve nos > 0
money = money + G[i][j]

case 2 : G[i][j] = 0
exit the function

case 3 : G[i][j] = -1
money = money − money
ie, money becomes 0

case 4 : G[i][j] = -2
money = money / 2

if destination
return money

right = rec (i, j+1, money)
down = (i+1, j, money)

return max (right, down)

Base Case:

If $i = 1$ and $j = 1$, meaning we are at the starting position, the function returns 0. This represents the initial state of the game where no gold has been collected.

Trap Case:

If $i > n$ or $j > m$ (outside the boundaries of the board) or $G[i][j] = 0$ (encountering a trap), the function returns $-\infty$. This signifies that reaching this position is not valid or beneficial.

Gold Case:

If $G[i][j] > 0$ (encountering gold), the function returns the maximum value between moving from the position above ($DP(i+1,j)$) and moving from the position to the left ($DP(i,j+1)$), plus the amount of gold at the current position ($G[i][j]$). This represents the decision to collect the gold and move towards the maximum profit.

Lose Half Case:

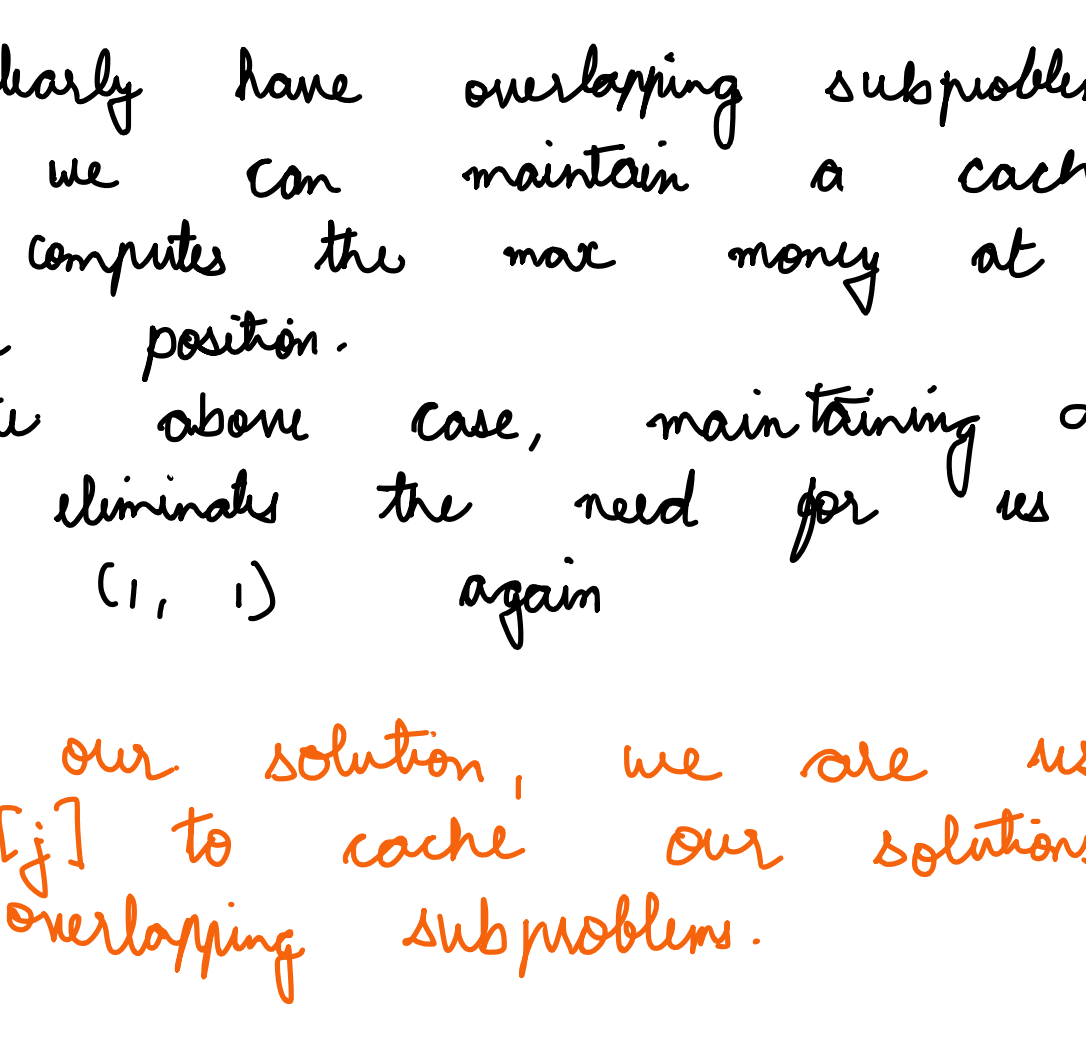
If $G[i][j] = -2$ (encountering the lose-half condition), the function returns half of the maximum value between moving from the position above ($DP(i+1,j)$) and moving from the position to the left ($DP(i,j+1)$). This represents the decision to continue the game but lose half of the collected gold.

Lose All Case:

If $G[i][j] = -1$ (encountering the lose-all condition), the function returns 0. This represents the decision to continue the game but lose all collected gold.

1c) Since we are computing recursively at each $G(i,j)$ in the matrix, the time complexity is exponential.

Let us consider a (2×3) matrix



We clearly have overlapping subproblems. Therefore we can maintain a cache which computes the max money at a particular position.
In the above case, maintaining a cache eliminates the need for us to compute (1, 1) again

In our solution, we are using DP[i][j] to cache our solutions to overlapping subproblems.

This reduces the time complexity to $O(n \times m)$ ∵ at any time there can only be $n \times m$ no of states

→ Reason to Use a cache to Solve Overlapping sub-problems

Here's a theoretical explanation of using a cache and how it helps avoid overlapping subproblems:

1. **Avoiding Redundant Computations:**
   - Recursive algorithms often revisit the same subproblems multiple times, leading to redundant computations.
   - A cache allows the algorithm to store the results of previously solved subproblems, preventing the need to recompute them when encountered again.
2. **Memoization Technique:**
   - When a recursive function is called with a set of parameters, the algorithm first checks the cache to see if it has already computed and stored the result for those parameters.
   - If the result is found in the cache (a cache hit), the algorithm can directly return the cached result without re-executing the computation.
   - If the result is not in the cache (a cache miss), the algorithm proceeds with the computation, stores the result in the cache, and then returns the result.
3. **Efficiently Handling Overlapping Subproblems:**
   - Overlapping subproblems occur when the same subproblems are solved multiple times in the process of solving the larger problem.
   - By caching results, the algorithm remembers solutions to subproblems and reuses them, eliminating the need to solve the same subproblem repeatedly.
4. **Space vs. Time Tradeoff:**
   - While memoization improves the time complexity of an algorithm by avoiding redundant computations, it introduces a space complexity overhead due to the storage of cached results.
   - The tradeoff between time and space depends on the specific characteristics of the problem and the available memory.
5. **Implementation Considerations:**
   - The cache can be implemented using data structures like dictionaries, arrays, or other appropriate structures.
   - It's essential to design the cache such that it uniquely identifies each set of input parameters to ensure correct retrieval of cached results.

1d)
1) Initialize dp[n][m] = -1,
money = 0, i = 1, j = 1
// i and j are used to traverse G

func rec (G, i, j, money)

if i or j out of bound
return money

if dp[i][j] != -1
return dp[i][j]

case 1 : G[i][j] = +ve nos > 0
money = money + G[i][j]

case 2 : G[i][j] = 0 { or set G[i][j] = −∞ to skip the values
exit ( );

case 3 : G[i][j] = -1
money = money − money
ie, money becomes 0

case 4 : G[i][j] = -2
money = money / 2

if destination
return money

right = rec (i, j+1, money)
down = rec (i+1, j, money)

dp[i][j] = max (right, down)

return dp[i][j]

Time complexity : $O(n \times m)$ because at max there can be $n \times m$ no of states

Explanation of Solving the Original Problem:

To solve the original problem of maximizing the amount of money from the start to the finish of the game, we utilize dynamic programming. The approach involves filling a two-dimensional DP table, where each entry $DP[i][j]$ represents the maximum amount of money that can be collected starting from square $(1,1)$ and ending at square (i,j). The table is filled iteratively based on the game rules.

We initialize the DP table with dimensions $(n+1) \times (m+1)$ and fill it with $-\infty$. The base case is set as $DP[n][m] = G[n][m]$, representing the amount of gold in the bottom-right corner.

Next, we fill the last row and last column of the DP table, considering the accumulative gold from the last row and last column, respectively. Finally, we fill the remaining entries in the DP table iteratively, considering the rules for gold, losing half, and losing all.

The maximum amount of money from start to finish is stored in $DP[1][1]$, representing the top-left corner of the DP table.

The running time of this algorithm is $O(n \cdot m)$, where n is the number of rows and m is the number of columns in the board. Each entry in the DP table is computed once, and each computation takes constant time.

Reason for setting $-\infty$ values

Here's a theoretical explanation of using a cache and how it helps avoid overlapping subproblems:

1. **Avoiding Redundant Computations:**
   - Recursive algorithms often revisit the same subproblems multiple times, leading to redundant computations.
   - A cache allows the algorithm to store the results of previously solved subproblems, preventing the need to recompute them when encountered again.
2. **Memoization Technique:**
   - When a recursive function is called with a set of parameters, the algorithm first checks the cache to see if it has already computed and stored the result for those parameters.
   - If the result is found in the cache (a cache hit), the algorithm can directly return the cached result without re-executing the computation.
   - If the result is not in the cache (a cache miss), the algorithm proceeds with the computation, stores the result in the cache, and then returns the result.
3. **Efficiently Handling Overlapping Subproblems:**
   - Overlapping subproblems occur when the same subproblems are solved multiple times in the process of solving the larger problem.
   - By caching results, the algorithm remembers solutions to subproblems and reuses them, eliminating the need to solve the same subproblem repeatedly.
4. **Space vs. Time Tradeoff:**
   - While memoization improves the time complexity of an algorithm by avoiding redundant computations, it introduces a space complexity overhead due to the storage of cached results.
   - The tradeoff between time and space depends on the specific characteristics of the problem and the available memory.
5. **Implementation Considerations:**
   - The cache can be implemented using data structures like dictionaries, arrays, or other appropriate structures.
   - It's essential to design the cache such that it uniquely identifies each set of input parameters to ensure correct retrieval of cached results.

2. Restricted Rod-Cutting Problem Solution

November 24, 2023

Problem Description

Consider a more restrictive version of the rod-cutting problem where you are given a number n (the length of the rod) and a table p_i for $i = 1, \ldots, n$ listing the profit you can make by selling a piece of length i. The goal is to decide how to cut up the given length n rod into pieces in the optimum way to make the most profit, with the restriction that you are not allowed to use more than one piece of the same length.

Dynamic Programming Solution

(a) Define the subproblems and a function measuring the quality of the subproblems.

Define the subproblems as $dp[i][j]$, where i is the index representing the remaining pieces of the rod, and j is the remaining length of the rod that needs to be cut. The mathematical formula for the subproblem is:

$$dp[i][j] = \max(dp[i-1][j], \text{price}[i] + dp[i-1][j - \text{length}[i]])$$

The function measuring the quality of the subproblems is the maximum obtainable value for the given remaining pieces and length.

$$\text{Quality}(i, j) = \max(\text{Quality}(i-1, j), \text{price}[i] + \text{Quality}(i-1, j - \text{length}[i]))$$

Here,

- i is the index representing the remaining pieces of the rod.
- j is the remaining length of the rod that needs to be cut.
- price$[i]$ is the profit obtained by selling a piece of length i.
- length$[i]$ is the length of the piece at index i.

'Writing in an alternative form'

Define $DP(i, S)$ as the maximum profit that can be collected by cutting a rod of length i into pieces, where S is the set of sizes of the pieces used.

$$DP(i,S) = \max \begin{cases} 0 & \text{if } i = 0 \text{ or } S = \{\} \\ -\infty & \text{if } i < 0 \text{ or } S \text{ has repeated sizes} \\ \max_{j=1}^{i} \{DP(i-j, S \cup \{j\}) + \text{profit}[j]\} & \text{otherwise} \end{cases}$$

- Quality(i, j) represents the maximum obtainable value for the given remaining pieces and length.

This formula expresses that the quality of the current subproblem is the maximum value between not cutting the rod at the current index (Quality$(i - 1, j)$) and cutting the rod at the current index (price$[i]$ + Quality$(i - 1, j -$ length$[i]$)).

(b) Give a recursive definition of the function.

```java
static int cutRoad(int[] price, int index, int n, int[][] dp, int
    [][] cuts, HashSet<Integer> usedLengths) {
    if (index < 0) {
        return (n > 0) ? Integer.MIN_VALUE : 0;
    }
    if (dp[index][n] != -1) {
        return dp[index][n];
    }

    int notCut = cutRoad(price, index - 1, n, dp, cuts, usedLengths
    );
    int cut = -5;
    int rodLength = index + 1;

    // Check if cutting the rod is a valid option
    if (rodLength <= n && !usedLengths.contains(rodLength)) {
        usedLengths.add(rodLength);
        cut = price[index] + cutRoad(price, index - 1, n -
    rodLength, dp, cuts, usedLengths);
        usedLengths.remove(rodLength);
    }

    // Update the cuts array if cutting is the better option
    if (cut > notCut) {
        cuts[index][n] = rodLength;
    }
```

```
24
25      dp[index][n] = Math.max(notCut, cut);
26      return dp[index][n];
27  }
```

Listing 1: Recursive function for the Restricted Rod-Cutting Problem

Pseudo-Code

Algorithm 1 Recursive function for the Restricted Rod-Cutting Problem

function CUTROD(price, index, n, dp, cuts, usedLengths)
  if index < 0 **then**
    return Integer.MIN_VALUE if n > 0 else 0
  end if
  if dp[index][n] $\neq -1$ **then**
    return dp[index][n]
  end if
  notCut $\leftarrow$ cutRod(price, index $- 1$, n, dp, cuts, usedLengths)
  cut $\leftarrow -5$
  rodLength $\leftarrow$ index $+ 1$
  if rodLength $\leq$ n and rodLength $\notin$ usedLengths **then**
    add rodLength to usedLengths
    cut $\leftarrow$ price[index] + cutRod(price, index $- 1$, n $-$ rodLength, dp, cuts, usedLengths)
    remove rodLength from usedLengths
  end if
  if cut $>$ notCut **then**
    cuts[index][n] $\leftarrow$ rodLength
  end if
  dp[index][n] $\leftarrow$ max(notCut, cut)
  return dp[index][n]
end function

The `cutRod` function is designed to find the maximum obtainable value by recursively exploring the different ways of cutting a rod under the given restriction. Let's break down the key aspects of the function:

Objective

Given the length of a rod (n), an array of prices for different lengths (*price*), an index representing the remaining pieces of the rod (*index*), and two 2D arrays (*dp* and *cuts*) for memoization, the goal is to determine the maximum profit that can be obtained by cutting the rod under the restriction that each length can only be used once.

Parameters

- *price*: Array representing the profit obtainable for each length.
- *index*: Index representing the remaining pieces of the rod.
- n: Remaining length of the rod that needs to be cut.
- *dp*: Memoization array to store intermediate results.
- *cuts*: Array to store the lengths of pieces used in the optimal solution.
- *usedLengths*: HashSet to keep track of lengths that have been used to ensure each length is used only once.

Base Case

The function has a base case that checks whether the index is less than 0. If true, it means we have considered all pieces, and the function returns a very large negative value if there is still rod length remaining ($n > 0$) or 0 if the rod is fully cut.

```
if (index < 0) {
    return (n > 0) ? Integer.MIN_VALUE : 0;
}
```

Memoization

The function uses memoization to avoid redundant calculations. If the result for the current state (given by *index* and n) has already been computed and stored in *dp*, it is directly returned.

```
if (dp[index][n] != -1) {
    return dp[index][n];
}
```

Recursive Calls

The function explores two possibilities:

1. **Not cutting the rod at the current index:**

$$\text{notCut} = \text{cutRod}(price, index - 1, n, dp, cuts, usedLengths)$$

2. **Cutting the rod at the current index:**

$$\text{cut} = \text{price[index]} + \text{cutRod}(price, index - 1, n - \text{rodLength}, dp, cuts, usedLengths)$$

Validity Check

Before considering the second option (cutting the rod at the current index), the function checks if it's a valid option by ensuring that the length of the piece (rodLength = index + 1) is less than or equal to the remaining length (n) and that the length has not been used before.

```
if (rodLength <= n && !usedLengths.contains(rodLength)) {
    // ...
}
```

Updating Results

The function updates the *cuts* array if cutting is the better option, and then updates the *dp* array with the maximum of the two possibilities.

```
if (cut > notCut) {
    cuts[index][n] = rodLength;
}

dp[index][n] = \max(notCut, cut);
```

Returning Result

Finally, the function returns the maximum obtainable value for the given remaining pieces and length.

```
return dp[index][n];
```

(c) Computing the Function Efficiently

The function is computed efficiently using dynamic programming with memoization. The results of overlapping subproblems are stored in the `dp` array to avoid redundant computations.

In the `cutRod` function, memoization is implemented using a 2D array `dp`. The following code checks whether the result for the current state (given by `index` and `n`) has already been computed and stored in the memoization array. If found, the function directly returns the stored result without further computation.

```
if (dp[index][n] != -1) {
    return dp[index][n];
}
```

This piece of code is crucial for avoiding redundant calculations and improving the efficiency of the algorithm.

Detailed Explanation

In the `cutRod` function, `dp` is a 2D array used for memoization. Its dimensions represent the state of the problem - `index` (representing the remaining pieces) and `n` (representing the remaining length of the rod). The array is initialized with -1, and if a value is found in `dp[index][n]` that is not -1, it means that the result for the corresponding subproblem has already been computed and stored.

Memoization is helpful in dynamic programming problems for several reasons:

1. **Avoiding Redundant Calculations:** By storing previously computed results, the algorithm can avoid recomputing the same subproblems multiple times. This is crucial for efficiency, especially in recursive algorithms with overlapping subproblems.

2. **Improving Time Complexity:** Memoization significantly improves the time complexity of the algorithm. Without memoization, the algorithm might explore an exponential number of overlapping subproblems. Memoization reduces this complexity by storing and reusing results.

3. **Enabling Top-Down Dynamic Programming:** Memoization is often associated with the top-down approach in dynamic programming, where the problem is solved recursively by breaking it down into smaller subproblems. Memoization ensures that each subproblem is solved only once, improving the overall efficiency.

4. **Simplifying Code:** Memoization allows for a more concise and readable code structure. The function can be written without explicitly managing all the details of previously solved subproblems, making the code easier to understand.

In summary, memoization in the `cutRod` function helps optimize the algorithm by storing and reusing previously computed results, reducing redundant calculations, and improving the overall efficiency of the solution to the restricted rod-cutting problem.

(d) Solving the Original Problem

Java Solution.

```
1 import java.util.HashSet;
2
3 public class RestrictedRodCutting {
4
5     static int cutRoad(int[] price, int index, int n, int[][] dp,
      int[][] cuts, HashSet<Integer> usedLengths) {
6         if (index < 0) {
7             return (n > 0) ? Integer.MIN_VALUE : 0;
8         }
```

```
        if (dp[index][n] != -1) {
            return dp[index][n];
        }

        int notCut = cutRoad(price, index - 1, n, dp, cuts,
    usedLengths);
        int cut = -5;
        int rodLength = index + 1;

        // Check if cutting the rod is a valid option
        if (rodLength <= n && !usedLengths.contains(rodLength)) {
            usedLengths.add(rodLength);
            cut = price[index] + cutRoad(price, index - 1, n -
    rodLength, dp, cuts, usedLengths);
            usedLengths.remove(rodLength);
        }

        // Update the cuts array if cutting is the better option
        if (cut > notCut) {
            cuts[index][n] = rodLength;
        }

        dp[index][n] = Math.max(notCut, cut);
        return dp[index][n];
    }

    public static void main(String[] args) {
        int[] arr = {1, 5, 9, 9, 10, 17, 17, 20};
        int size = arr.length;

        int[][] dp = new int[size][size + 1];
        int[][] cuts = new int[size][size + 1];
        HashSet<Integer> usedLengths = new HashSet<>();

        for (int i = 0; i < size; i++) {
            for (int j = 0; j <= size; j++) {
                dp[i][j] = -1;
            }
        }

        System.out.print("Maximum Obtainable Value is: ");
        int maxValue = cutRoad(arr, size - 1, size, dp, cuts,
    usedLengths);
        System.out.println(maxValue);
    }
}
```

Listing 2: Java code for the Restricted Rod-Cutting Problem

You can find the document on Google Drive at the following link:
JAVA FILE LINK

Explanation of the Java Code (Part d)

Let's break down the provided Java code for the restricted rod-cutting problem:

Key Points:

- `cutRoad` **Function:**
  - **Parameters:**
    - `price`: Array representing the profit obtainable for each length.
    - `index`: Index representing the remaining pieces of the rod.
    - `n`: Remaining length of the rod that needs to be cut.
    - `dp`: Memoization array to store intermediate results.
    - `cuts`: Array to store the lengths of pieces used in the optimal solution.
    - `usedLengths`: HashSet to keep track of lengths that have been used to ensure each length is used only once.
  - **Base Case:**
    - If `index` is less than 0, it returns a very large negative value if there is still rod length remaining ($n > 0$) or 0 if the rod is fully cut.
  - **Memoization:**
    - Checks whether the result for the current state (given by `index` and `n`) has already been computed and stored in the memoization array. If found, it directly returns the stored result.
  - **Recursive Calls:**
    - Two possibilities are explored: not cutting the rod at the current index and cutting the rod at the current index. The function recursively calls itself for both options.
  - **Validity Check:**
    - Before considering cutting the rod at the current index, it checks if it's a valid option by ensuring that the length of the piece is less than or equal to the remaining length (n) and that the length has not been used before.
  - **Updating Results:**
    - Updates the `cuts` array if cutting is the better option and then updates the `dp` array with the maximum of the two possibilities.
  - **Returning Result:**
    - Finally, the function returns the maximum obtainable value for the given remaining pieces and length.
- `main` **Method:**
  - Initializes the input array (`arr`), memoization array (`dp`), cuts array (`cuts`), and a HashSet (`usedLengths`).
  - Calls the `cutRoad` function to find the maximum obtainable value.
  - Prints the result.

Tackling the Challenge:

The challenge mentioned that "A subproblem can no longer be described by one number (the size of the rod you are cutting), this gives no information about the sizes of the pieces that have been used." To tackle this, the code uses a 2D array `cuts` to store information about the lengths of pieces used in the optimal solution. This additional array helps reconstruct the solution by keeping track of the lengths of the pieces chosen at each step. This is essential because a simple 1D array wouldn't capture the sizes of individual pieces, and using a HashSet to record all possible combinations would result in an exponential number of subproblems, making the solution impractical. The 2D array `cuts` efficiently records the sizes of pieces used without exploding the number of subproblems.

Time Complexity Explanation

The time complexity of the provided code is $O(n^2)$, where n is the size of the input array.

Reasoning:

- **Recursive Calls:**
  - The primary function, `cutRoad`, is called recursively. The function has two recursive calls, each with reduced input size.
  - For each index i and remaining length n, the function explores two possibilities: cutting or not cutting the rod at the current index.
  - This results in a binary tree of recursive calls with a maximum depth of n.
- **Memoization:**
  - The memoization array, `dp`, is of size size $\times$ (size + 1), where size is the length of the input array.
  - Each cell in the memoization array is computed once, and the function avoids redundant computations by storing and reusing intermediate results.
- **Nested Loop in Main:**
  - The `main` method initializes a 2D array (`dp`) and performs a nested loop over the size of the input array to initialize the memoization array.
  - This nested loop contributes $O(n^2)$ to the overall time complexity.

Conclusion:

Combining the recursive calls and the nested loop, the overall time complexity of the code is $O(n^2)$.