Lecture 15 Review Notes

MA8.401 Topics in Applied Optimization

Monsoon 2023

Contributors:
- Sriteja Reddy Pashya (2021111019)
- Romica Raisinghani (2021101053)

1 Deterministic Dynamic Programming

Deterministic dynamic programming is a technique for solving sequential decision problems. It works by recursively solving a sequence of smaller subproblems, starting from the base case and working towards the final solution. The key insight is that the optimal solution to the overall problem can be constructed from the optimal solutions to the subproblems.

Deterministic dynamic programming problems are characterized by the following properties:

- The problem can be divided into a sequence of stages.
- At each stage, a decision must be made.
- The outcome of each decision is deterministic, meaning that it is known with certainty.
- The objective is to maximize or minimize some function of the decisions made at each stage.

To solve a deterministic dynamic programming problem, we can use the following steps:

1. Identify the stages of the problem.
2. Define the state of the system at each stage.
3. Define the decision that can be made at each stage.
4. Define the reward or cost associated with each decision.
5. Recursively solve the problem for each stage, starting from the base case and working towards the final solution.
6. The optimal solution to the overall problem is the solution that maximizes the reward or minimizes the cost at the final stage.

1.1 The Dynamic Programming Algorithm

Principle of Optimality

Let $\{u_0^*, \ldots, u_{N-1}^*\}$ be an optimal control sequence, which together with x_0 determines the corresponding state sequence $\{x_1^*, \ldots, x_N^*\}$ via the system equation (1). Consider the subproblem whereby we start at x_k^* at time k and wish to minimize the "cost-to-go" from time k to time N,

$$g_k(x_k^*, u_k) + \sum_{m=k+1}^{N-1} g_m(x_m, u_m) + g_N(x_N) \tag{1}$$

over $\{u_k, \dots, u_{N-1}\}$ with $u_m \in U_m(x_m)$, $m = k, \dots, N-1$. Then the truncated optimal control sequence $\{u_k^*, \dots, u_{N-1}^*\}$ is optimal for this subproblem.

The subproblem referred to above is called the tail subproblem that starts at x_k^* .

Definition (Principle of Optimality)
The principle of optimality says that the tail of an optimal sequence is optimal for the tail subproblem.

If the truncated control sequence $\{u_k^*, \dots, uN-1^*\}$ were not optimal as stated, we would be able to reduce the cost further by switching to an optimal sequence for the subproblem once we reach x_k^* (since the preceding choices of controls, $\{u_0^*, \dots, u_{k-1}^*\}$, do not restrict our future choices).

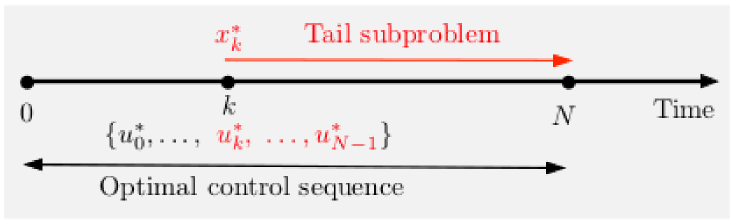


Figure 1: Schematic illustration of the principle of optimality. The tail $\{u_k^*, \dots, u_{N-1}^*\}$ of an optimal sequence $\{u_0^*, \dots, u_{N-1}^*\}$ is optimal for the tail subproblem that starts at the state x_k^* of the optimal state trajectory.

Idea for DP algorithm
Solve all the tail subproblems of a given length using the solution of all the tail subproblems of shorter time length.

Key Idea: To solve tail sub-problem at x_k :
1. Consider every possible u_k and solve the tail subproblem that starts at next state

$$x_{k+1} = f_k(x_k, u_k)$$

This gives the cost u_k

2. Optimize over all possible u_k

1.2 Examples:

1.2.1 Example 1: The Knapsack Problem

The knapsack problem is a classic optimization problem where you are given a set of items, each with a weight and a value, and a knapsack with a limited capacity. The goal is to find the subset of items that has the maximum value and fits in the knapsack.

This problem can be solved using deterministic dynamic programming as follows:

1. The stages of the problem are the different items in the set.
2. The state of the system at each stage is the remaining capacity of the knapsack.
3. The decision that can be made at each stage is to either include the current item in the subset or to exclude it.
4. The reward associated with each decision is the value of the current item.
5. To solve the problem recursively, we can start from the base case, which is the case where there are no more items to choose from. In this case, the optimal solution is to simply return the value of the subset of items that has been selected so far.

For the other cases, we can recursively solve the problem for the next item in the set, assuming that we have already made the optimal decision for all of the previous items. The optimal solution for the current stage is the decision that leads to the maximum value of the subset of items that can be selected.

6. The optimal solution to the overall problem is the solution that maximizes the value of the subset of items that has been selected at the final stage.

1.2.2 Example 2: The Shortest Path Problem

The shortest path problem is another classic optimization problem where you are given a graph with weighted edges and a source vertex and a destination vertex. The goal is to find the shortest path from the source vertex to the destination vertex.

This problem can be solved using deterministic dynamic programming as follows:

1. The stages of the problem are the different vertices in the graph.
2. The state of the system at each stage is the shortest distance from the source vertex to the current vertex.
3. The decision that can be made at each stage is to choose next vertex to visit on the path.
4. The reward associated with each decision is the weight of the edge between the current vertex and the next vertex.
5. To solve the problem recursively, we can start from the base case, which is the case where we have reached the destination vertex. In this case, the optimal solution is to simply return the distance from the source vertex to the destination vertex.

For the other cases, we can recursively solve the problem for the next vertex in the graph, assuming that we have already found the shortest path from the source vertex to the current vertex. The optimal solution for the current stage is the decision that leads to the shortest path from the source vertex to the destination vertex.

6. The optimal solution to the overall problem is the shortest path from the source vertex to the destination vertex.

1.3 Finite Horizon Problem Formulation

In finite horizon problems the system evolves over a finite number N of time steps (also called stages). The state and control at time k of the system will be generally denoted by x_k and u_k, respectively.
In deterministic systems, x_{k+1} is generated nonrandomly, i.e., it is determined solely by x_k and u_k; see Fig. 1.
Thus, a deterministic DP problem involves a system of the form:

$$x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, \ldots, N-1 \tag{2}$$

where,

- k is the time index
- x_k is the state of the system, an element of some state space X_k ,
- u_k is the control or decision variable, to be selected at time k from some given set $U_k(x_k)$, a subset of a control space U_k , that depends on x_k ,
- f_k is a function of (x_k, u_k) that describes the mechanism by which the state is updated from time k to time $k+1$,
- N is the horizon, i.e., the number of times control is applied

In the case of a finite number of states, the system function f_k may be represented by a table that gives the next state x_{k+1} for each possible value of the pair (x_k, u_k). Otherwise a mathematical expression or a computer implementation is necessary to represent f_k .

The state space X_k and control space U_k are arbitrary sets and may depend on k. Similarly, the system function f_k can be arbitrary and may depend on k. The cost incurred at time k is denoted by $g_k(x_k, u_k)$, and the function g_k may depend on k. For a given initial state x_0 , the total cost of a control sequence $\{u_0, \ldots, u_{N-1}\}$ is

$$J(x_0; u_0, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k) \tag{3}$$

where $g_N(x_N)$ is a terminal cost incurred at the end of the process. This is a well-defined number, since the control sequence $\{u_0, \ldots, u_{N-1}\}$ together with x_0 determines exactly the state sequence

$\{x_1, \ldots, x_N\}$ via the system equation (1); see Figure 1. We want to minimize the cost (2) over all sequences $\{u_0, \ldots, u_{N-1}\}$ that satisfy the control constraints, thereby obtaining the **optimal value as a function of** x_0

$$J^*(x_0) = \min_{\substack{u_k \in U_k(x_k) \\ k=0,\ldots,N-1}} J(x_0; u_0, \ldots, u_{N-1}) \tag{4}$$

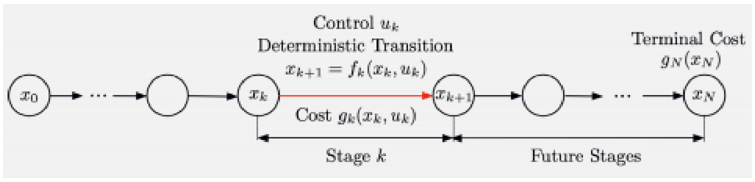


Figure 2: Illustration of a deterministic N -stage optimal control problem. Starting from state x_k , the next state under control u_k is generated non-randomly, according to $x_{k+1} = f_k(x_k, u_k)$, and a stage cost $g_k(x_k, u_k)$ is incurred.

Definition (State space)
The set of all possible x_k is called the **state space** at time k. It can be any set and may depend on k.

Definition (Control space)
Similarly, the set of all possible u_k is called the **control space** at time k. Again it can be any set and may depend on k.

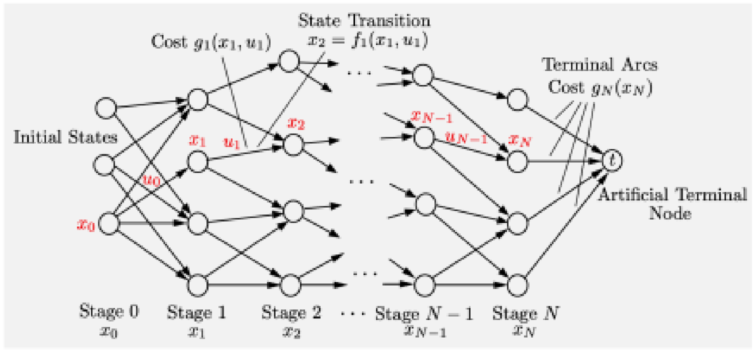


Figure 3: Transition graph for a deterministic finite-state system

Consider the figure 2 shown above.

- Nodes correspond to states x_k.
- Arcs correspond to state-control pairs (x_k, u_k).
- An arc (x_k, u_k) has start and end nodes x_k and $x_{k+1} = f_k(x_k, u_k)$, respectively.

- We view the cost $g_k(x_k, u_k)$ of the transition as the length of this arc.
- The problem is equivalent to finding a shortest path from initial nodes of stage 0 to the terminal node t.
- To handle the final stage, an artificial terminal node t is added.

1.3.1 Algorithms:

There are a number of algorithms that can be used to solve finite horizon DP problems. The most common algorithms are:

- **Value iteration:** This algorithm iteratively builds up a table of optimal values for each state. The algorithm starts by initializing the table to all zeros. Then, at each iteration, the algorithm updates the table for each state as follows:

$$V(x_k) = min_{u_k}[g_k(x_k, u_k) + V(f_k(x_k, u_k))]$$

where $V(x_k)$ is the optimal value for state x_k. The algorithm terminates when the table no longer changes.

- **Policy iteration:** This algorithm iteratively improves a policy by evaluating the current policy and making changes to improve it. The algorithm starts with an initial policy. Then, at each iteration, the algorithm evaluates the current policy by computing the value of each state under the current policy. After evaluating the current policy, the algorithm makes changes to the policy to improve it. The algorithm terminates when the policy no longer changes.
- **Rollout:** This algorithm simulates the system under a given policy and computes the cost. This process is repeated for a number of different policies, and the policy with the lowest cost is selected.

1.3.2 Applications:

Finite horizon DP problems arise in a wide variety of applications, including:

- **Robotics:** Planning the motion of a robot to reach a goal while avoiding obstacles.
- **Financial engineering:** Managing a portfolio of assets to maximize returns or minimize risk over a given time period.
- **Production planning:** Scheduling production to meet demand and minimize costs.
- **Inventory management:** Deciding how much inventory to order and when to order it to meet demand and minimize costs.

- **Energy management:** Optimizing the energy consumption of a building or system.
- **Transportation:**Optimizing the delivery of goods or passengers.
- **Healthcare:** Optimizing the treatment of patients.

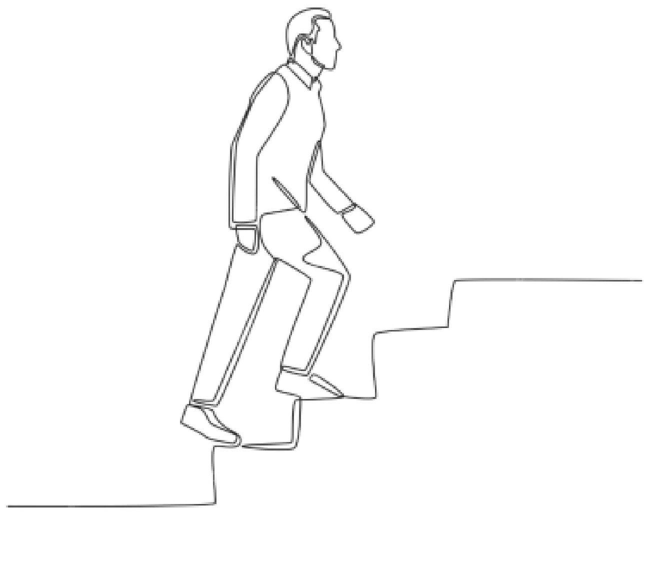
1.3.3 New Ideas in the area of finite horizon DP

- **Deep reinforcement learning:** Deep reinforcement learning algorithms can be used to learn optimal policies for finite horizon DP problems without the need to explicitly model the system dynamics or transition probabilities.
- **Asynchronous DP:** Asynchronous DP algorithms can be used to solve finite horizon DP problems in a distributed manner, which can improve scalability and performance.
- **Transfer learning:** Transfer learning can be used to transfer knowledge from one finite horizon DP problem to another, which can improve the performance of the DP algorithm on the new problem.

1.4 DP Examples: Tail Subproblems and Recursion

1.4.1 Example 1: Climbing Stairs

Problem Statement:
There are n stairs, a person standing at the bottom wants to climb stairs to reach the n^{th} stair. The person can climb either 1 stair or 2 stairs at a time, the task is to count the number of ways that a person can reach at the top. [**Hint**: Formulate a tail sub-problem, and use recursion.]



How to identify a DP problem?
When we see a problem, it is very important to identify it as a dynamic programming problem. Generally (but not limited to) if the problem statement asks for the following:

- Count the total number of ways.
- Given multiple ways of doing a task, which way will give the minimum or the maximum output.

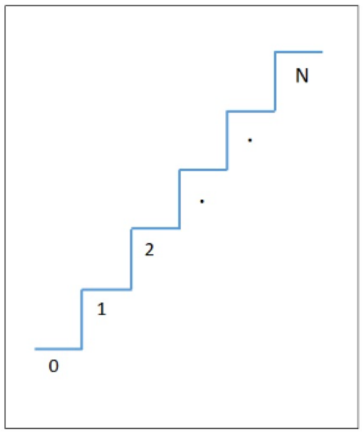
We can try to apply recursion. Once we get the recursive solution, we can go ahead to convert it to a dynamic programming one.

Steps To Solve The Problem After Identification
Once the problem has been identified, the following three steps comes handy in solving the problem:

- Try to represent the problem in terms of indexes.
- Try all possible choices/ways at every index according to the problem statement.
- If the question states
  - Count all the ways – return sum of all choices/ways.
  - Find maximum/minimum- return the choice/way with maximum/minimum output.

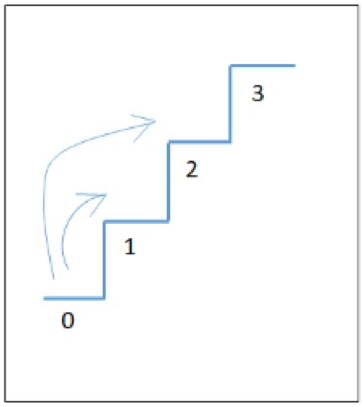
Using these steps to solve the problem "Climbing Stairs"

Step 1: We will assume n stairs as indexes from 0 to N.



Step 2: At a single time, we have 2 choices: Jump one step or jump two steps. We will try both of these options at every index.
Step 3: As the problem statement asks to count the total number of distinct ways, we will return the sum of all the choices in our recursive function.

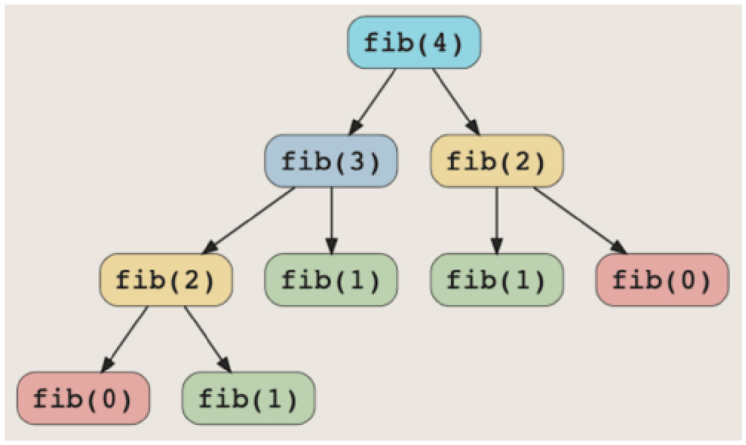
The base case will be when we want to go to the 0^{th} stair, then we have only one option.



There will be one more edge-case when $n = 1$, if we call $ways(n-2)$ we will reach stair numbered -1 which is not defined, therefore we add an extra test case to return 1 (the only way) when $n = 1$.

For all other cases, we return $ways(n) = ways(n-1) + ways(n-2)$. This is similar to calculating the fibonacci series where $fib(n)$ is equivalent to calculating $ways(n)$.

1.5 Overlapping Subproblems



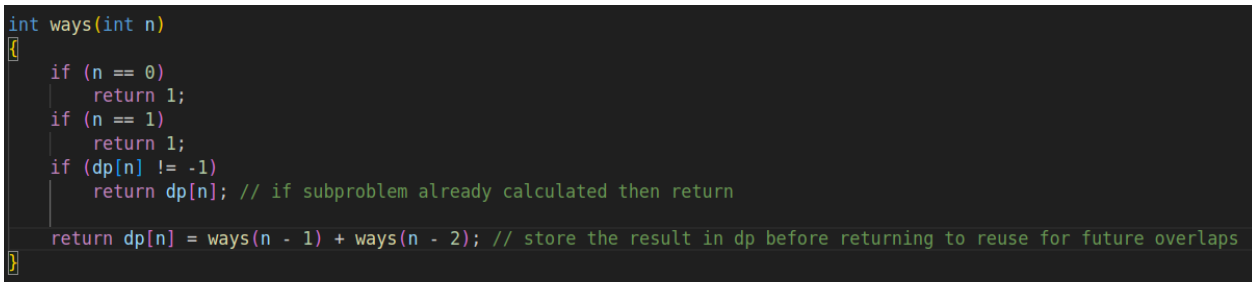
We can see that the function fib(2) is being called 2 times. If we would have stored the value of fib(2), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

- Memoization (Top Down)
- Tabulation (Bottom Up)

Memoization (Top Down):
The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Tabulation (Bottom Up)
The tabulated program for a given problem builds a table in a bottom-up fashion and returns the last entry from the table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions to subproblems bottom-up.

Below is the code to achieve the Memoization (Top Down) approach:



```
int ways(int n)
{
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;
    if (dp[n] != -1)
        return dp[n]; // if subproblem already calculated then return

    return dp[n] = ways(n - 1) + ways(n - 2); // store the result in dp before returning to reuse for future overlaps
}
```

1.5.1 Example 2: Maximum sum of non-adjacent elements

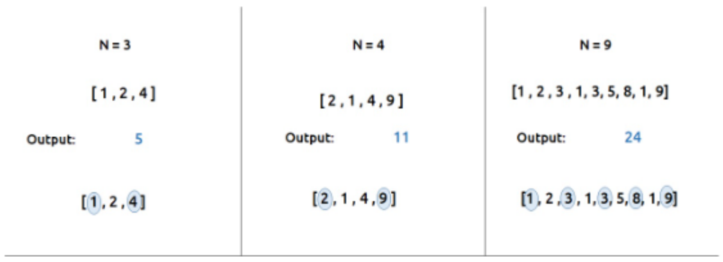
Problem Statement:
Given an array of 'N' positive integers, we need to return the maximum sum of the subsequence such that no two elements of the subsequence are adjacent elements in the array.

Note: A subsequence of an array is a list with elements of the array where some elements are deleted (or not deleted at all) and the elements should be in the same order in the subsequence as in the array.

Look at the image below for more clarity:

Algorithm / Intuition

As we need to find the sum of subsequences, one approach that comes to our mind is to generate all subsequences and pick the one with the maximum sum.



N = 3	N = 4	N = 9
[1 , 2 , 4]	[2 , 1 , 4 , 9]	[1 , 2 , 3 , 1, 3, 5, 8, 1, 9]
Output: 5	Output: 11	Output: 24
[1 , 2 , 4]	[2 , 1 , 4 , 9]	[1 , 2 , 3 , 1, 3, 5, 8, 1, 9]

To generate all the subsequences, we can use the pick/non-pick technique. This technique can be briefly explained as follows:

- At every index of the array, we have two options.
- First, to pick the array element at that index and consider it in our subsequence.
- Second, to leave the array element at that index and not to consider it in our subsequence.

First, we will try to form the recursive solution to the problem with the pick/non-pick technique. There is one more catch, the problem wants us to have only non-adjacent elements of the array in the subsequence, therefore we need to address that too.

Steps to form the recursive solution

Step 1: Form the function in terms of indexes:

- We are given an array which can be easily thought of in terms of indexes.
- We can define our function $f(ind)$ as : Maximum sum of the subsequence starting from index 0 to index ind.
- We need to return $f(n-1)$ as our final answer.

Step 2: Try all the choices to reach the goal.

As mentioned earlier we will use the pick/non-pick technique to generate all subsequences. We also need to take care of the non-adjacent elements in this step.

- If we pick an element then, $pick = arr[ind] + f(ind-2)$. The reason we are doing $f(ind-2)$ is because we have picked the current index element so we need to pick a non-adjacent element so we choose the index '$ind - 2$' instead of '$ind - 1$'.
- Next we need to ignore the current element in our subsequence. So $nonPick = 0 + f(ind-1)$. As we don't pick the current element, we can consider the adjacent element in the subsequence.

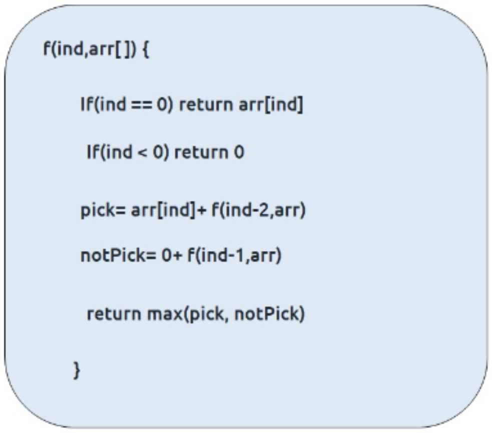
Step 3: Take the maximum of all the choices

As the problem statement asks to find the maximum subsequence total, we will return the maximum of two choices of step2.

Base Conditions The base conditions for the recursive function will be as follows:

- If $ind = 0$, then we know to reach at index=0, we would have ignored the element at index = 1. Therefore, we can simply return the value of $arr[ind]$ and consider it in the subsequence.
- If $ind < 0$, this case can hit when we call $f(ind-2)$ at $ind = 1$. In this case we want to return to the calling function so we simply return 0 so that nothing is added to the subsequence sum.

Our final pseudo-code is shown above.



```
f(ind,arr[ ]) {

    If(ind == 0) return arr[ind]

     If(ind < 0) return 0

    pick= arr[ind]+ f(ind-2,arr)

    notPick= 0+ f(ind-1,arr)

     return max(pick, notPick)

   }
```

Once we form the recursive solution, we can convert it into a dynamic programming one.

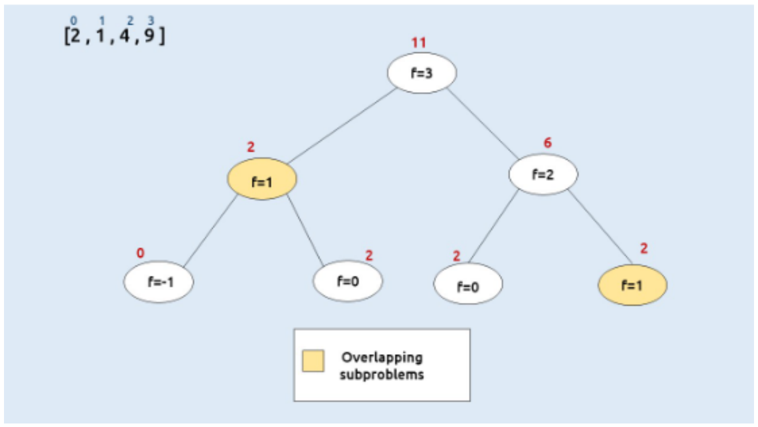
Memoization approach

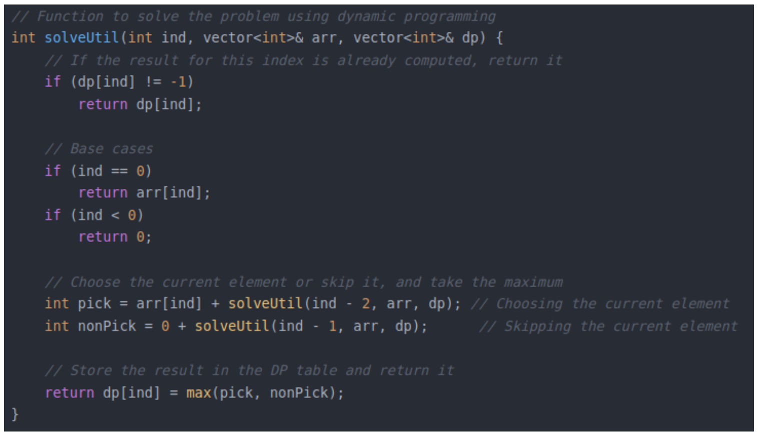
If we observe the recursion tree, we will observe a number of overlapping subproblems. Therefore the recursive solution can be memoized to reduce the time complexity.

Recursion tree diagram: Steps to convert Recursive code to memoization solution:

- Create a $dp[n]$ array initialized to -1.
- Whenever we want to find the answer of a particular value (say n), we first check whether the answer is already calculated using the dp array(i.e $dp[n] \neq -1$). If yes, simply return the value from the dp array.
- If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set $dp[n]$ to the solution we get.

Below is the code to achieve the Memoization (Top Down) approach:





```
// Function to solve the problem using dynamic programming
int solveUtil(int ind, vector<int>& arr, vector<int>& dp) {
    // If the result for this index is already computed, return it
    if (dp[ind] != -1)
        return dp[ind];

    // Base cases
    if (ind == 0)
        return arr[ind];
    if (ind < 0)
        return 0;

    // Choose the current element or skip it, and take the maximum
    int pick = arr[ind] + solveUtil(ind - 2, arr, dp); // Choosing the current element
    int nonPick = 0 + solveUtil(ind - 1, arr, dp);      // Skipping the current element

    // Store the result in the DP table and return it
    return dp[ind] = max(pick, nonPick);
}
```

1.5.2 Example 3: 3-d DP : Ninja and his friends

Problem Statement:
We are given an 'N*M' matrix. Every cell of the matrix has some chocolates on it, mat[i][j] gives us the number of chocolates. We have two friends 'Alice' and 'Bob'. initially, Alice is standing on the cell$(0, 0)$ and Bob is standing on the cell$(0, M - 1)$. Both of them can move only to the cells below them in these three directions: to the bottom cell ($\downarrow$), to the bottom-right cell($\searrow$), or to the bottom-left cell($\swarrow$).

When Alice and Bob visit a cell, they take all the chocolates from that cell with them. It can happen that they visit the same cell, in that case, the chocolates need to be considered only once.

They cannot go out of the boundary of the given matrix, we need to return the maximum number of chocolates that Bob and Alice can together collect.

Algorithm / Intuition

In this question, there are two fixed starting and variable ending points, but as per the movement of Alice and Bob, we know that they will end in the last row. They have to move together at a time to the next row.

Why a Greedy Solution doesn't work?

As we have to return the maximum chocolates collected, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the option that gives us more chocolates. But there is no 'uniformity' in the values of the matrix, therefore it can happen that whenever we are making a local choice that gives us a better path, we actually take a path that in the later stages is giving us fewer chocolates.

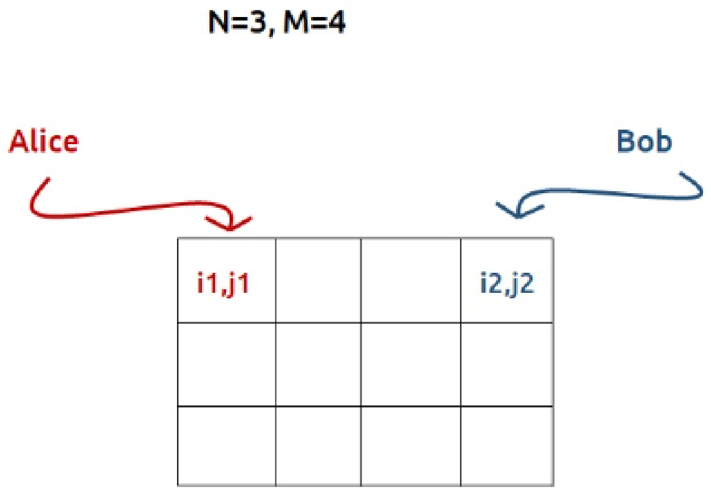
As a greedy solution doesn't work, our next choice will be to try out all the possible paths. To generate all possible paths we will use recursion.

Steps to form the recursive solution:

Step 1: Express the problem in terms of indexes.

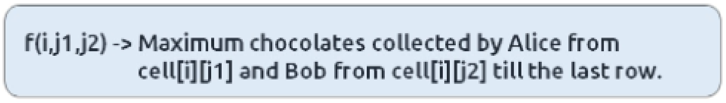
Here we are given two starting points from where Alice and Bob can move.

We are given an '$N \times M$' matrix. We need to define the function with four parameters i_1,j_1,i_2, and j_2 to describe the positions of Alice and Bob at a time.



If we observe, initially Alice and Bob are at the first row, and they always move to the row below them every time, so they will always be in the same row. Therefore two different variables i_1 and i_2, to describe their positions are redundant. We can just use single parameter i, which tells us in which row of the grid both of them are.

Therefore, we can modify the function. It now takes three parameters: i,j_1, and j_2. $f(i, j_1, j_2)$ will give us the maximum number of chocolates collected by Alice and Bob from their current positions to the last position. Base Case:



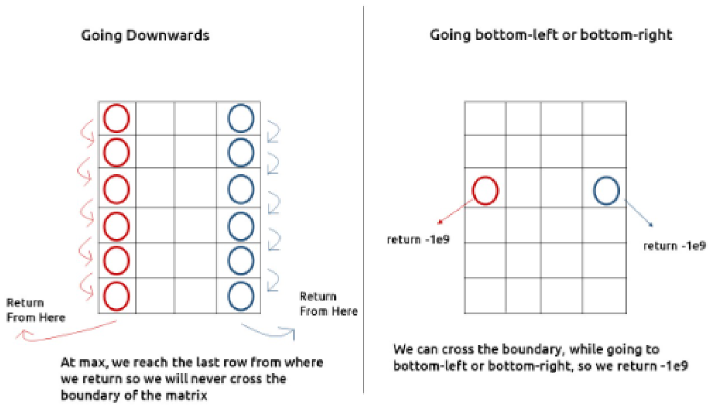
f(i,j1,j2) -> Maximum chocolates collected by Alice from
              cell[i][j1] and Bob from cell[i][j2] till the last row.

There will be the following base cases:

- When $i = N-1$, it means we are at the last row, so we need to return from here. Now it can happen that at the last row, both Alice and Bob are at the same cell, in this condition we will return only chocolates collected by Alice, $mat[i][j1]$(as question states that the chocolates cannot be doubly calculated), otherwise we return sum of chocolates collected by both, $mat[i][j_1] + mat[i][j_1][j_2]$. At every cell, we have three options to go: to the bottom cell ($\downarrow$), to the bottom-right cell($\searrow$) or to the bottom-left cell($\swarrow$)

As we are moving to the bottom cell ($\downarrow$), at max we will reach the last row, from where we return, so we will never go out of the bounding index.

To move to the bottom-right cell($\searrow$) or to the bottom-left cell($\swarrow$), it can happen that we may go out of bound as shown in the figure(below). So we need to handle it, we can return $-1e^9$, whenever we go out of bound, in this way this path will not be selected by the calling function as we have to return the maximum chocolates.


Going Downwards
Return From Here
Return From Here
At max, we reach the last row from where we return so we will never cross the boundary of the matrix
Going bottom-left or bottom-right
return -1e9
return -1e9
We can cross the boundary, while going to bottom-left or bottom-right, so we return -1e9

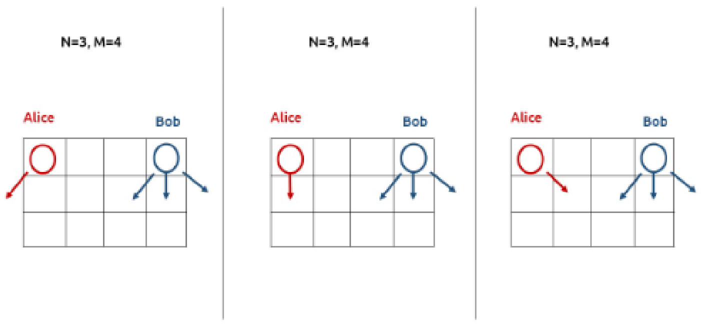
- If $j_1 < 0$ or $j_1 \geq M$ or $j_2 < 0$ or $j_2 \geq M$, then we return $-1e^9$

Step 2: Try out all possible choices at a given index.

At every cell, we have three options to go: to the bottom cell ($\downarrow$), to the bottom-right cell($\searrow$) or to the bottom-left cell($\swarrow$)

Now, we need to understand that we want to move Alice and Bob together. Both of them can individually move three moves but say Alice moves to bottom-left, then Bob can have three different moves for Alice's move, and so on. The following figures will help to understand this: Hence we have a total of 9 different options at every $f(i, j_1, j_2)$ to move Alice and Bob. Now we can manually write these 9 options or we can observe a pattern in them, first Alice moves to one side and Bob tries all three choices, then again Alice moves, then Bob, and so on. This pattern can be easily captured by using two nested loops that change the column numbers(j_1 and j_2).

Note: if ($j_1 = j_2$), as discussed in the base case, we will only consider chocolates collected by one of them otherwise we will consider chocolates collected by both of them.

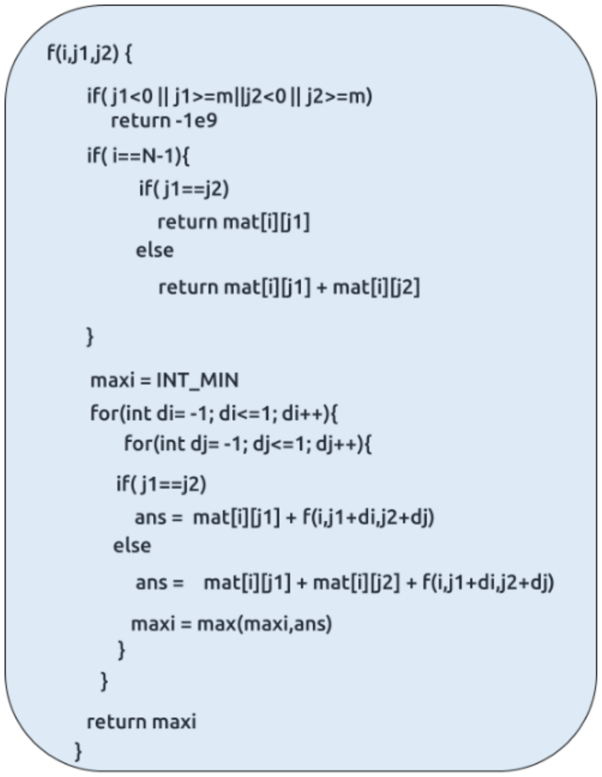


For every single move of Alice, Bob have three different moves

Step 3: Take the maximum of all choices

As we have to find the maximum chocolates collected of all the possible paths, we will return the maximum of all the choices(the 9 choices of step 2). We will take a maxi variable(initialized to $INT-MIN$). We will update maxi to the maximum of the previous maxi and the answer of the current choice. At last, we will return maxi from our function as the answer.

The final pseudocode will be:



```
f(i,j1,j2) {
    if( j1<0 || j1>=m||j2<0 || j2>=m)
      return -1e9
    if( i==N-1){
        if( j1==j2)
          return mat[i][j1]
        else
          return mat[i][j1] + mat[i][j2]
    }
    maxi = INT_MIN
    for(int di= -1; di<=1; di++){
        for(int dj= -1; dj<=1; dj++){
       if( j1==j2)
          ans =  mat[i][j1] + f(i,j1+di,j2+dj)
       else
          ans =   mat[i][j1] + mat[i][j2] + f(i,j1+di,j2+dj)
          maxi = max(maxi,ans)
       }
    }
    return maxi
}
```

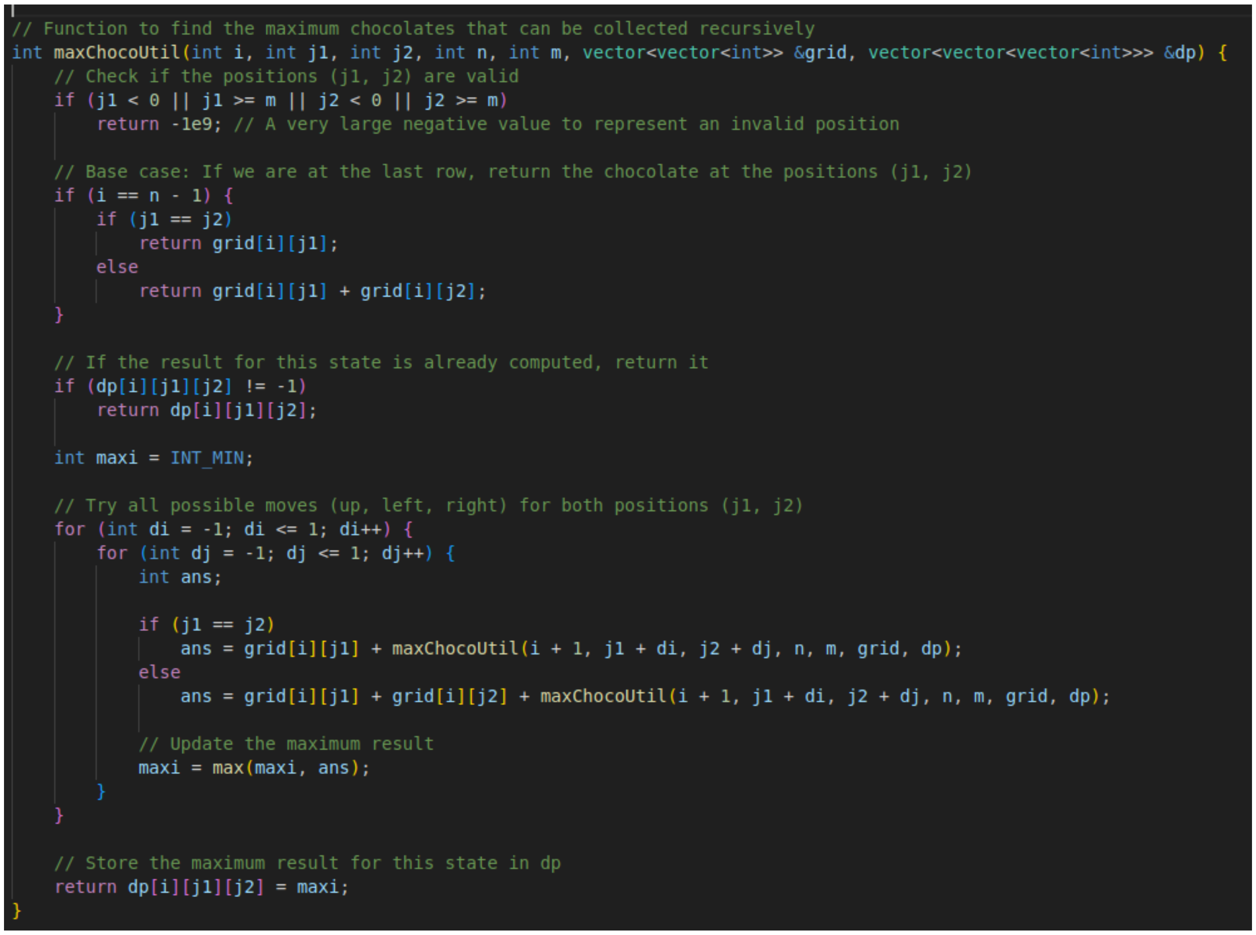
Steps to memoize a recursive solution:

Before moving to the memoization steps, we need to understand the dp array we are taking. The recursive function has three parameters: i,j_1, and j_2. Therefore, we will also need to take a 3-D DP Array. Its dimensions will be [N][M][M] because when we are moving, i can go from 0 to $N-1$, and j_1 and j_2 can go from 0 to $M-1$.

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [N][M][M], initialized to -1.
2. Whenever we want to find the answer of a particular row and column (say $f(i, j_1, j_2)$), we first check whether the answer is already calculated using the dp array(i.e $dp[i][j_1][j_2] \neq -1$). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set $dp[i][j_1][j_2]$ to the solution we get.

Below is the code to achieve the Memoization (Top Down) approach:



```
// Function to find the maximum chocolates that can be collected recursively
int maxChocoUtil(int i, int j1, int j2, int n, int m, vector<vector<int>> &grid, vector<vector<vector<int>>> &dp) {
    // Check if the positions (j1, j2) are valid
    if (j1 < 0 || j1 >= m || j2 < 0 || j2 >= m)
        return -1e9; // A very large negative value to represent an invalid position

    // Base case: If we are at the last row, return the chocolate at the positions (j1, j2)
    if (i == n - 1) {
        if (j1 == j2)
            return grid[i][j1];
        else
            return grid[i][j1] + grid[i][j2];
    }

    // If the result for this state is already computed, return it
    if (dp[i][j1][j2] != -1)
        return dp[i][j1][j2];

    int maxi = INT_MIN;

    // Try all possible moves (up, left, right) for both positions (j1, j2)
    for (int di = -1; di <= 1; di++) {
        for (int dj = -1; dj <= 1; dj++) {
            int ans;

            if (j1 == j2)
                ans = grid[i][j1] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n, m, grid, dp);
            else
                ans = grid[i][j1] + grid[i][j2] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n, m, grid, dp);

            // Update the maximum result
            maxi = max(maxi, ans);
        }
    }

    // Store the maximum result for this state in dp
    return dp[i][j1][j2] = maxi;
}
```

2 Discrete-State Deterministic Scheduling

Discrete-State Deterministic Scheduling is a method used to manage and schedule activities or tasks in various applications, ranging from manufacturing and project management to computer science and transportation. In this example, we'll explore a simplified scenario involving a manufacturing process to illustrate the concept of discrete-state deterministic scheduling.

Scenario: Imagine a small manufacturing facility that produces customized furniture. The production process consists of several discrete stages, and each piece of furniture must go through these stages sequentially. The stages are as follows:

1. **Design and Planning:** Initial design and planning, where the customer's requirements are analyzed, and the necessary materials are determined.

2. **Cutting and Shaping:** The raw materials are cut and shaped according to the design specifications.

3. **Assembling:** The individual components are assembled to create the furniture piece.

4. **Finishing:** The piece is sanded, painted or stained, and any final touches are added.

5. **Quality Control:** A final quality check is performed to ensure the piece meets the required standards.

Each stage has a predetermined processing time:

- Design and Planning: 2 days
- Cutting and Shaping: 3 days
- Assembling: 4 days
- Finishing: 2 days
- Quality Control: 1 day

Additionally, there are specific constraints:

- Only one piece of furniture can be in each stage at a time.
- Once a piece enters a stage, it cannot be interrupted, and it must be completed.
- There's a limit on the number of pieces that can be in the "Assembling" stage at once: a maximum of 2 pieces.

Discrete-State Deterministic Scheduling: To optimize the manufacturing process, we need to determine the order in which the pieces of furniture should be processed to minimize the total time required to complete all orders.

Solution: Let's consider three orders:

Order 1: A custom table
Order 2: A set of chairs

Order 3: A custom bookshelf

To minimize the total processing time, we can employ a discrete-state deterministic scheduling algorithm. The key steps are as follows:

1. Analyze the processing times for each stage and the constraints.
2. Create a schedule based on the processing times and constraints, aiming to minimize idle time and maximize efficiency.
3. Continuously monitor the progress of each order and adjust the schedule as needed.

For example, Order 2, the set of chairs, could enter the "Cutting and Shaping" stage before Order 1 completes the "Design and Planning" stage. This minimizes idle time and ensures efficient resource utilization.

2.1 Applications and Examples

- **Manufacturing and Production:**

  Example: Scheduling the assembly line in an automobile manufacturing plant. Each vehicle follows a discrete sequence of assembly steps, and scheduling ensures that each vehicle moves through these stages efficiently.

- **Project Management:**

  Example: In a construction project, scheduling the tasks involved, such as excavation, foundation laying, framing, electrical work, plumbing, and finishing. The goal is to complete the project on time and within budget.

- **Supply Chain Management:**

  Example: Scheduling the production and transportation of goods in a supply chain to meet customer demands while minimizing inventory and transportation costs.

- **Job Shop Scheduling:**

  Example: Scheduling machines in a job shop to process a set of jobs with varying processing times and machine requirements while minimizing makespan or total completion time.

- **Healthcare and Surgery Scheduling:**

  Example: Scheduling surgical procedures in a hospital to optimize the utilization of operating rooms, minimize patient waiting times, and allocate resources efficiently.

- **Airline Flight Scheduling:**

  Example: Determining the sequence and timing of flights for an airline to optimize aircraft utilization and crew schedules, considering various constraints like crew rest requirements and maintenance schedules.

- **Manufacturing Semiconductor Chips:**

Example: Scheduling the various processing steps involved in semiconductor manufacturing, including photolithography, etching, and quality control, to maximize production yield and minimize cycle time.

- **Print Shop Scheduling:**

  Example: Scheduling print jobs on different printing presses with varying printing speeds and setup times to optimize production efficiency and meet customer deadlines.

- **Education and Class Scheduling:**

  Example: Creating schedules for schools and universities, allocating classrooms and resources, and ensuring that students and teachers can attend classes without conflicts.

- **Retail Inventory Replenishment:**

  Example: Determining when to reorder products from suppliers to maintain optimal inventory levels and meet customer demand without overstocking or understocking.

- **Energy Grid Management:**

  Example: Scheduling the production and distribution of electricity from various sources (e.g., coal, natural gas, renewables) to meet the demands of consumers while minimizing costs and environmental impact.

- **Broadcasting and Television Scheduling:**

  Example: Planning the programming schedule for a TV network, ensuring that shows and advertisements air at specific times and optimizing viewership and advertising revenue.

- **Agricultural Crop Scheduling:**

  Example: Planning planting, harvesting, and irrigation schedules for different crops on a farm to maximize yields and minimize resource use.

3 Link to the scribe:

Click here to edit this scribe