Contents

Chapter 1

Check if the game is valid or not

Check if the game is valid or not - GeeksforGeeks

Three players P1, P2 and P3 are playing a game. But at a time only two players can play the game, so they decided, at a time two players will play the game and one will spectate. When one game ends, the one who lost the game becomes the spectator in the next game and the one who was spectating plays against the winner. There cannot be any draws in any game. Player P1 and P2 will play the first game. Input is the number of games played n and in next line winner of n games.

Examples :

```
Input :
No. of Games : 4
Winner of the Game Gi : 1 1 2 3
Output : YES
Explanation :
Game1 : P1 vs P2 : P1 wins
Game2 : P1 vs P3 : P1 wins
Game3 : P1 vs P2 : P2 wins
Game4 : P3 vs P2 : P3 wins
None of the winners were invalid

Input :
No. of Games : 2
Winner of the Game Gi : 2 1
Output : NO
Explanation :
Game1 : P1 vs P2 : P2 wins
Game2 : P2 vs P3 : P1 wins (Invalid winner)
In Game2 P1 is spectator
```

Approach :

Start with P1 and P2 and continue the game while making the loser new spectator by subtracting the current spectator and winner from total sum of three players i.e. 6. An invalid entry will halt the process.

Below is the implementation for the above approach:

C++

```
 // CPP program to check if the game
// is valid
#include <bits/stdc++.h>
using namespace std;

string check_valid(int a[], int n)
{
    // starting with player P1 and P2
    // so making P3 spectator
    int spec = 3;
    for (int i = 0; i < n; i++) {

        // If spectator wins a game
        // then its not valid
        if (a[i] == spec) {
            return "Invalid";
        }

        // subtracting the current spectator
        // and winner from total sum 6 which
        // makes losing player spectator
        spec = 6 - a[i] - spec;
    }

    // None of the winner is found spectator.
    return "Valid";
}

// Driver program to test above functions
int main()
{
    int n = 4;
    int a[n] = {1, 1, 2, 3};
    cout << check_valid(a, n);
    return 0;
}
```

Java

```
 // java program to check if the game
```

```
// is valid
import java.io.*;

class GFG {



    static String check_valid(int a[], int n)
    {
        // starting with player P1 and P2
        // so making P3 spectator
        int spec = 3;

        for (int i = 0; i < n; i++) {

            // If spectator wins a game
            // then its not valid
            if (a[i] == spec) {
                return "Invalid";
            }

            // subtracting the current spectator
            // and winner from total sum 6 which
            // makes losing player spectator
            spec = 6 - a[i] - spec;
        }

        // None of the winner is found spectator.
        return "Valid";
    }

    // Driver program to test above functions
    public static void main (String[] args) {

        int a[] = {1, 1, 2, 3};
        int n = a.length;

        System.out.println(check_valid(a, n));

        }
}

// This code is contributed by vt_m
```

Python3

```
 # Python3 code to check if the game
# is valid
```

```
def check_valid( a , n ):

    # starting with player P1 and P2
    # so making P3 spectator
    spec = 3
    for i in range(n):

        # If spectator wins a game
        # then its not valid
        if a[i] == spec:
            return "Invalid"

        # subtracting the current spectator
        # and winner from total sum 6 which
        # makes losing player spectator
        spec = 6 - a[i] - spec

    # None of the winner is found spectator.
    return "Valid"

# Driver code to test above functions
n = 4
a = [1, 1, 2, 3]
print(check_valid(a, n))

# This code is contributed by "Sharad_Bhardwaj".
```

C#

```
 // C# program to check if the game
// is valid
using System;

class GFG {

    static String check_valid(int []a, int n)
    {
        // starting with player P1 and P2
        // so making P3 spectator
        int spec = 3;

        for (int i = 0; i < n; i++) {

            // If spectator wins a game
            // then its not valid
            if (a[i] == spec) {
                return "Invalid";
            }

            // subtracting the current spectator
            // and winner from total sum 6 which
            // makes losing player spectator
            spec = 6 - a[i] - spec;
        }

        // None of the winner is found spectator.
        return "Valid";
    }

    // Driver program
    public static void Main ()
    {
        int []a = {1, 1, 2, 3};
        int n = a.Length;

        Console.WriteLine(check_valid(a, n));

        }
}

// This code is contributed by vt_m
```

PHP

```
 <?php
// PHP program to check if
// the game is valid
function check_valid( $a, $n)
{
    // starting with player P1 and P2
    // so making P3 spectator
    $spec = 3;
    for ($i = 0; $i < $n; $i++)
    {

        // If spectator wins a game
        // then its not valid
        if ($a[$i] == $spec)
        {
            return "Invalid";
        }

        // subtracting the current
        // spectator and winner from
        // total sum 6 which makes
        // losing player spectator
        $spec = 6 - $a[$i] - $spec;
    }

    // None of the winner is
    // found spectator.
    return "Valid";
}

// Driver Code
$n = 4;
$a = array(1, 1, 2, 3);
echo check_valid($a, $n);

// This code is contributed by vt_m
?>
```

Output :

```
Valid
```

Improved By : vt_m

Source

https://www.geeksforgeeks.org/check-game-valid-not/

Chapter 2

Choice of Area

Choice of Area - GeeksforGeeks

Consider a game, in which you have two types of powers, A and B and there are 3 types of Areas X, Y and Z. Every second you have to switch between these areas, each area has specific properties by which your power A and power B increase or decrease. We need to keep choosing areas in such a way that our survival time is maximized. Survival time ends when any of the powers, A or B reaches less than 0.
Examples:

```
Initial value of Power A = 20
Initial value of Power B = 8

Area X (3, 2) : If you step into Area X,
                A increases by 3,
                B increases by 2

Area Y (-5, -10) : If you step into Area Y,
                   A decreases by 5,
                   B decreases by 10

Area Z (-20, 5) : If you step into Area Z,
                  A decreases by 20,
                  B increases by 5

It is possible to choose any area in our first step.
We can survive at max 5 unit of time by following
these choice of areas :
X -> Z -> X -> Y -> X
```

This problem can be solved using recursion, after each time unit we can go to any of the area but we will choose that area which ultimately leads to maximum survival time. As

recursion can lead to solving same subproblem many time, we will memoize the result on basis of power A and B, if we reach to same pair of power A and B, we won't solve it again instead we will take the previously calculated result.
Given below is the simple implementation of above approach.

CPP

```
 //  C++ code to get maximum survival time
#include <bits/stdc++.h>
using namespace std;

//  structure to represent an area
struct area
{
    //  increment or decrement in A and B
    int a, b;
    area(int a, int b) : a(a), b(b)
    {}
};

//  Utility method to get maximum of 3 integers
int max(int a, int b, int c)
{
    return max(a, max(b, c));
}

//  Utility method to get maximum survival time
int maxSurvival(int A, int B, area X, area Y, area Z,
                int last, map<pair<int, int>, int>& memo)
{
    //  if any of A or B is less than 0, return 0
    if (A <= 0 || B <= 0)
        return 0;
    pair<int, int> cur = make_pair(A, B);

    //  if already calculated, return calculated value
    if (memo.find(cur) != memo.end())
        return memo[cur];

    int temp;

    //  step to areas on basis of last chose area
    switch(last)
    {
    case 1:
        temp = 1 + max(maxSurvival(A + Y.a, B + Y.b,
                                   X, Y, Z, 2, memo),
                       maxSurvival(A + Z.a, B + Z.b,
                                  X, Y, Z, 3, memo));
        break;
    case 2:
        temp = 1 + max(maxSurvival(A + X.a, B + X.b,
                                  X, Y, Z, 1, memo),
                       maxSurvival(A + Z.a, B + Z.b,
                                  X, Y, Z, 3, memo));
        break;
    case 3:
        temp = 1 + max(maxSurvival(A + X.a, B + X.b,
                                  X, Y, Z, 1, memo),
                       maxSurvival(A + Y.a, B + Y.b,
                                  X, Y, Z, 2, memo));
        break;
    }

    //  store the result into map
    memo[cur] = temp;

    return temp;
}

//  method returns maximum survival time
int getMaxSurvivalTime(int A, int B, area X, area Y, area Z)
{
    if (A <= 0 || B <= 0)
        return 0;
    map< pair<int, int>, int > memo;

    //  At first, we can step into any of the area
    return
        max(maxSurvival(A + X.a, B + X.b, X, Y, Z, 1, memo),
            maxSurvival(A + Y.a, B + Y.b, X, Y, Z, 2, memo),
            maxSurvival(A + Z.a, B + Z.b, X, Y, Z, 3, memo));
}

//  Driver code to test above method
int main()
{
    area X(3, 2);
    area Y(-5, -10);
    area Z(-20, 5);

    int A = 20;
    int B = 8;
    cout << getMaxSurvivalTime(A, B, X, Y, Z);

    return 0;
```

```
}
```

Python3

```
 # Python code to get maximum survival time

# Class to represent an area
class area:
    def __init__(self, a, b):
        self.a = a
        self.b = b

# Utility method to get maximum survival time
def maxSurvival(A, B, X, Y, Z, last, memo):
    # if any of A or B is less than 0, return 0
    if (A <= 0 or B <= 0):
        return 0
    cur = area(A, B)

    # if already calculated, return calculated value
    for ele in memo.keys():
        if (cur.a == ele.a and cur.b == ele.b):
            return memo[ele]

    # step to areas on basis of last chosen area
    if (last == 1):
        temp = 1 + max(maxSurvival(A + Y.a, B + Y.b,
                                   X, Y, Z, 2, memo),
                       maxSurvival(A + Z.a, B + Z.b,
                                   X, Y, Z, 3, memo))
    elif (last == 2):
        temp = 1 + max(maxSurvival(A + X.a, B + X.b,
                                   X, Y, Z, 1, memo),
               maxSurvival(A + Z.a, B + Z.b,
                   X, Y, Z, 3, memo))
    elif (last == 3):
        temp = 1 + max(maxSurvival(A + X.a, B + X.b,
                   X, Y, Z, 1, memo),
               maxSurvival(A + Y.a, B + Y.b,
                   X, Y, Z, 2, memo))

    # store the result into map
    memo[cur] = temp

    return temp

# method returns maximum survival time
def getMaxSurvivalTime(A, B, X, Y, Z):
    if (A <= 0 or B <= 0):
        return 0
    memo = dict()

    # At first, we can step into any of the area
    return max(maxSurvival(A + X.a, B + X.b, X, Y, Z, 1, memo),
           maxSurvival(A + Y.a, B + Y.b, X, Y, Z, 2, memo),
           maxSurvival(A + Z.a, B + Z.b, X, Y, Z, 3, memo))

# Driver code to test above method
X = area(3, 2)
Y = area(-5, -10)
Z = area(-20, 5)

A = 20
B = 8
print(getMaxSurvivalTime(A, B, X, Y, Z))

# This code is contributed by Soumen Ghosh.
```

Output:

```
5
```

Source

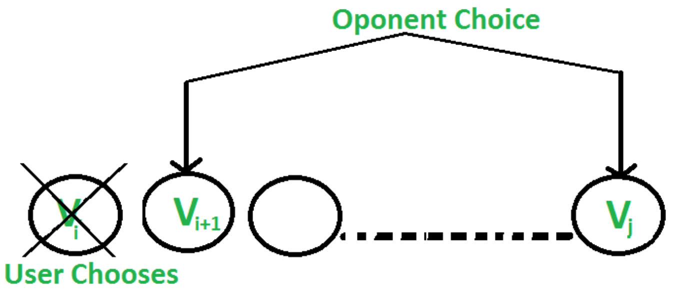
https://www.geeksforgeeks.org/game-theory-choice-area/

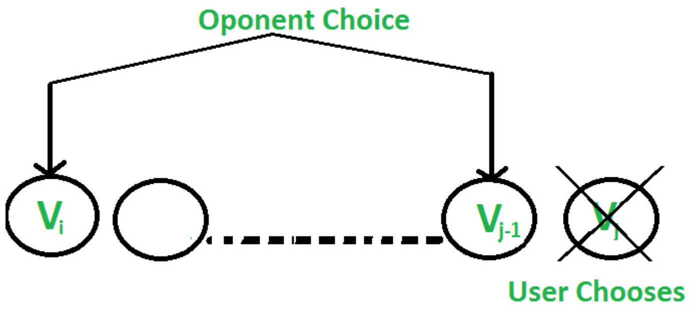
Chapter 3

Coin game of two corners (Greedy Approach)

Coin game of two corners (Greedy Approach) - GeeksforGeeks

Consider a two player coin game where each player gets turn one by one. There is a row of even number of coins, and a player on his/her turn can pick a coin from any of the two corners of the row. The player that collects coins with more value wins the game. Develop a strategy for the player making the first turn, such he/she never looses the game.





Note that the strategy to pick maximum of two corners may not work. In the following example, first player looses the game when he/she uses strategy to pick maximum of two corners.

Example:

```
  18 20 15 30 10 14
First Player picks 18, now row of coins is
  20 15 30 10 14
Second player picks 20, now row of coins is
  15 30 10 14
First Player picks 15, now row of coins is
  30 10 14
Second player picks 30, now row of coins is
  10 14
First Player picks 14, now row of coins is
  10
Second player picks 10, game over.

The total value collected by second player is more (20 +
30 + 10) compared to first player (18 + 15 + 14).
So the second player wins.
```

Note that this problem is different from Optimal Strategy for a Game | DP-31. There the target is to get maximum value. Here the target is to not loose. We have a Greedy Strategy here. The idea is to count sum of values of all even coins and odd coins, compare the two values. The player that makes the first move can always make sure that the other player is never able to choose an even coin if sum of even coins is higher. Similarly, he/she can make sure that the other player is never able to choose an odd coin if sum of odd coins is higher.

Example:

```
  18 20 15 30 10 14
Sum of odd coins = 18 + 15 + 10 = 43
Sum of even coins = 20 + 30 + 14 = 64.
Since the sum of even coins is more, the first
player decides to collect all even coins. He first
picks 14, now the other player can only pick a coin
(10 or 18). Whichever is picked the other player,
the first player again gets an opportunity to pick
an even coin and block all even coins.
```

C++

```
 // CPP program to find coins to be picked to make sure
// that we never loose.
#include <iostream>
```

```
using namespace std;

// Returns optimal value possible that a player can collect
// from an array of coins of size n. Note than n must be even
void printCoins(int arr[], int n)
{
    // Find sum of odd positioned coins
    int oddSum = 0;
    for (int i = 0; i < n; i += 2)
        oddSum += arr[i];

    // Find sum of even positioned coins
    int evenSum = 0;
    for (int i = 0; i < n; i += 2)
        evenSum += arr[i];

    // Print even or odd coins depending upon
    // which sum is greater.
    int start = ((oddSum > evenSum) ? 0 : 1);
    for (int i = start; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int arr1[] = { 8, 15, 3, 7 };
    int n = sizeof(arr1) / sizeof(arr1[0]);
    printCoins(arr1, n);
    cout << endl;

    int arr2[] = { 2, 2, 2, 2 };
    n = sizeof(arr2) / sizeof(arr2[0]);
    printCoins(arr2, n);
    cout << endl;

    int arr3[] = { 20, 30, 2, 2, 2, 10 };
    n = sizeof(arr3) / sizeof(arr3[0]);
    printCoins(arr3, n);

    return 0;
}
```

Java

```
// Java program to find coins to be
// picked to make sure that we never loose.
class GFG
{
```

// Returns optimal value possible
// that a player can collect from
// an array of coins of size n.
// Note than n must be even
static void printCoins(int arr[], int n)
{
// Find sum of odd positioned coins
int oddSum = 0;
for (int i = 0; i < n; i += 2) oddSum += arr[i]; // Find sum of even positioned coins int evenSum = 0; for (int i = 0; i < n; i += 2) evenSum += arr[i]; // Print even or odd coins depending // upon which sum is greater. int start = ((oddSum > evenSum) ? 0 : 1); for (int i = start; i < n; i++) System.out.print(arr[i]+" "); } // Driver Code public static void main(String[] args) { int arr1[] = { 8, 15, 3, 7 }; int n = arr1.length; printCoins(arr1, n); System.out.println(); int arr2[] = { 2, 2, 2, 2 }; n = arr2.length; printCoins(arr2, n); System.out.println(); int arr3[] = { 20, 30, 2, 2, 2, 10 }; n = arr3.length; printCoins(arr3, n); } } // This code is contributed by ChitraNayal [tabby title = "Python 3"]

```
 # Python 3 program to find coins
# to be picked to make sure that
# we never loose

# Returns optimal value possible
# that a player can collect from
# an array of coins of size n.
# Note than n must be even
def printCoins(arr, n) :

    oddSum = 0

    # Find sum of odd positioned coins
    for i in range(0, n, 2) :
        oddSum += arr[i]

    evenSum = 0

    # Find sum of even
    # positioned coins
    for i in range(0, n, 2) :
        evenSum += arr[i]

    # Print even or odd
    # coins depending upon
    # which sum is greater.
    if oddSum > evenSum :
        start = 0
    else :
        start = 1
    for i in range(start, n) :
        print(arr[i], end = " ")

# Driver code
if __name__ == "__main__" :

    arr1 = [8, 15, 3, 7]
    n = len(arr1)
    printCoins(arr1, n)
    print()

    arr2 = [2, 2, 2, 2]
    n = len(arr2)
    printCoins(arr2, n)
    print()

    arr3 = [20, 30, 2, 2, 2, 10]
    n = len(arr3)
    printCoins(arr3, n)

# This code is contributed by ANKITRAI1
```

C#

// C# program to find coins to be
// picked to make sure that we never loose.
using System;

class GFG
{

// Returns optimal value possible
// that a player can collect from
// an array of coins of size n.
// Note than n must be even
static void printCoins(int[] arr, int n)
{

// Find sum of odd positioned coins
int oddSum = 0;
for (int i = 0; i < n; i += 2) oddSum += arr[i]; // Find sum of even positioned coins int evenSum = 0; for (int i = 0; i < n; i += 2) evenSum += arr[i]; // Print even or odd coins depending // upon which sum is greater. int start = ((oddSum > evenSum) ? 0 : 1);
for (int i = start; i < n; i++) Console.Write(arr[i]+" "); } // Driver Code public static void Main() { int[] arr1 = { 8, 15, 3, 7 }; int n = arr1.Length; printCoins(arr1, n); Console.Write("\n"); int[] arr2 = { 2, 2, 2, 2 }; n = arr2.Length; printCoins(arr2, n); Console.Write("\n"); int[] arr3 = { 20, 30, 2, 2, 2, 10 }; n = arr3.Length; printCoins(arr3, n); } } // This code is contributed by ChitraNayal [tabby title="PHP"] $evenSum) ? 0 : 1);
for ($i = $start; $i < $n; $i++) echo $arr[$i]." "; } // Driver Code $arr1 = array(8, 15, 3, 7); $n = sizeof($arr1); printCoins($arr1, $n); echo "\n"; $arr2 = array(2, 2, 2, 2); $n =

sizeof($arr2); printCoins($arr2, $n); echo "\n"; $arr3 = array(20, 30, 2, 2, 2, 10); $n = sizeof($arr3); printCoins($arr3, $n); // This code is contributed by ChitraNayal ?>

Output:

```
15 3 7
2 2 2
30 2 2 2 10
```

Improved By : ANKITRAI1, ChitraNayal

Source

https://www.geeksforgeeks.org/coin-game-of-two-corners-greedy-approach/

Chapter 4

Combinatorial Game Theory | Set 1 (Introduction)

Combinatorial Game Theory | Set 1 (Introduction) - GeeksforGeeks

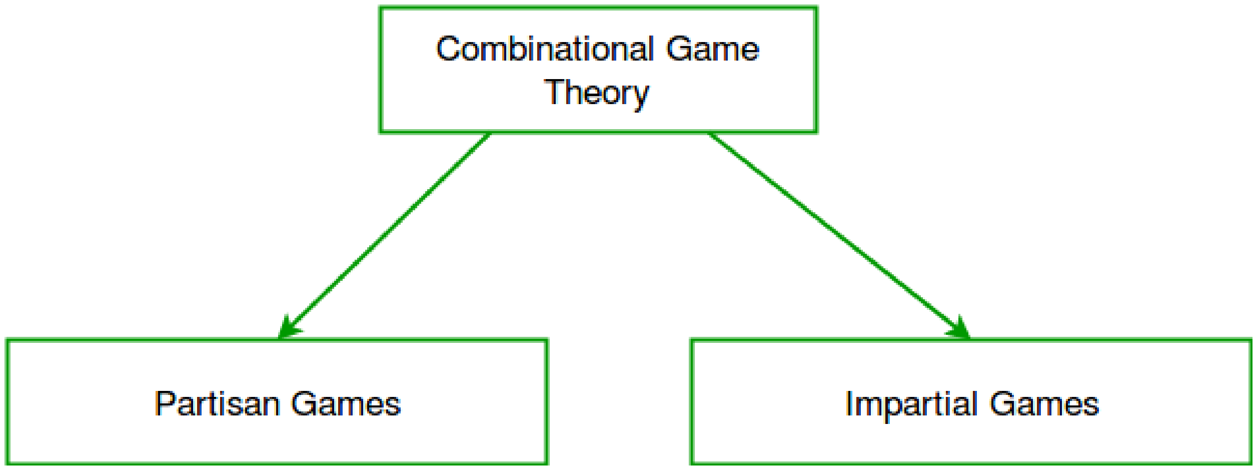
Combinatorial games are two-person games with perfect information and no chance moves (no randomization like coin toss is involved that can effect the game). These games have a win-or-lose or tie outcome and determined by a set of positions, including an initial position, and the player whose turn it is to move. Play moves from one position to another, with the players usually alternating moves, until a terminal position is reached. A terminal position is one from which no moves are possible. Then one of the players is declared the winner and the other the loser. Or there is a tie (Depending on the rules of the combinatorial game, the game could end up in a tie. The only thing that can be stated about the combinatorial game is that the game should end at some point and should not be stuck in a loop. In order to prevent such looping situations in games like chess(consider the case of both the players just moving their queens to-and-fro from one place to the other), there is actually a "50-move rule" according to which the game is considered to be drawn if the last 50 moves by each player have been completed without the movement of any pawn and without any capture. Source : Stackexchange]

On the other hand, Game theory in general includes games of chance, games of imperfect knowledge, and games in which players can move simultaneously.

The specialty of Combinatorial Game Theory (CGT) is that the coding part is relatively very small and easy. The key to the Game Theory problems is that hidden observation, which can be sometimes very hard to find.

Chess, Game of Nim, Tic-Tac-Toe all comes under the category of Combinatorial Game Theory.

We can divide these games into two categories as shown below:



The difference between them is that in **Impartial Games** all the possible moves from any position of game are the **same** for the players, whereas in **Partisan Games** the moves for all the players are **not the same**.

Consider a game like below :
Given a number of piles in which each pile contains some numbers of stones/coins. In each turn, player choose one pile and remove any number of stones (at least one) from that pile. The player who cannot move is considered to lose the game (ie., one who take the last stone is the winner).
As it can be clearly seen from the rules of the above game that the moves are same for both the players. There is no restriction on one player over the other. Such a game is considered to be impartial.
The above mentioned game is famous by the name- Game of Nim which will be discussed in next sections.

In contrast to the above game, let us take an example of **chess**. In this game, one player can only move the black pieces and the other one can only move the white ones. Thus, there is a restriction on both the players. Their set of moves are different and hence such a game is classified under the category of **Partisan Games**.

Partisan Games are much harder to analyze than Impartial Games as in such games Sprague-Grundy Theoremfails.

In the next sections, we will see mostly about Impartial games, like- Game of Nim and its variations, Sprague-Grundy Theorem and many more.

Exercise:
Readers may try solving below simple combinatorial Game Theory problems.
Cycle Race
Game of Chocolates

Sources:
http://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/comb.pdf

https://en.wikipedia.org/wiki/Combinatorial_game_theory
https://en.wikipedia.org/wiki/Impartial_game
https://en.wikipedia.org/wiki/Partisan_game

Source

https://www.geeksforgeeks.org/introduction-to-combinatorial-game-theory/

Chapter 5

Combinatorial Game Theory | Set 2 (Game of Nim)

Combinatorial Game Theory | Set 2 (Game of Nim) - GeeksforGeeks

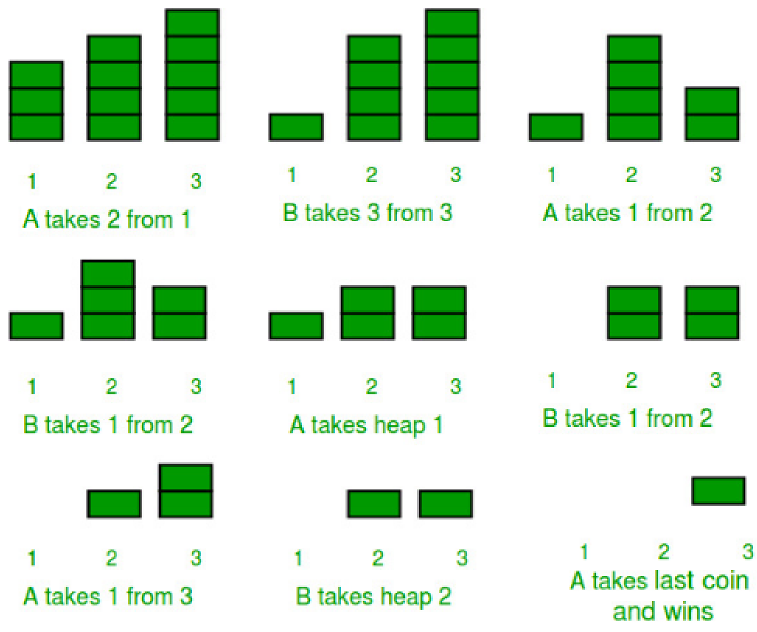
We strongly recommend to refer below article as a prerequisite of this.

Combinatorial Game Theory | Set 1 (Introduction)

In this post, Game of Nim is discussed. The Game of Nim is described by the following rules-

“ *Given a number of piles in which each pile contains some numbers of stones/coins. In each turn, a player can choose only one pile and remove any number of stones (at least one) from that pile. The player who cannot move is considered to lose the game (i.e., one who take the last stone is the winner).* ”

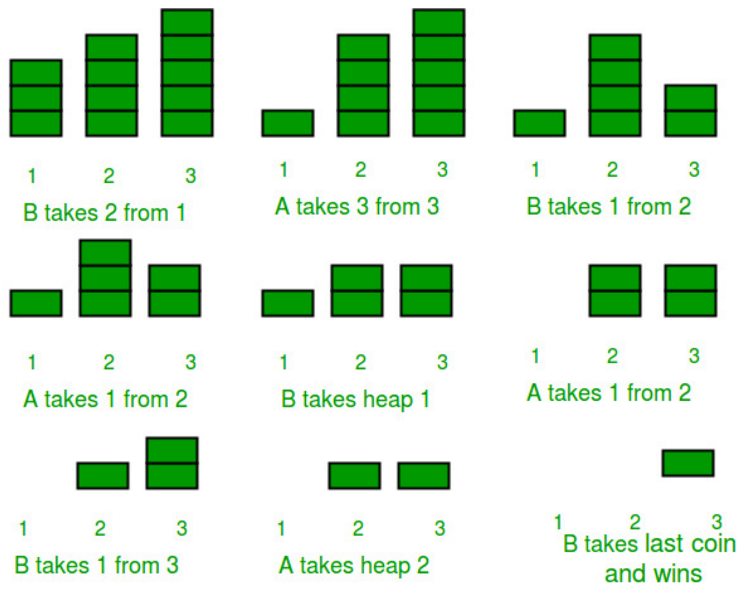
For example, consider that there are two players- **A** and **B**, and initially there are three piles of coins initially having **3, 4, 5** coins in each of them as shown below. We assume that first move is made by **A**. See the below figure for clear understanding of the whole game play.



A Won the match (Note: A made the first move)

So was **A** having a strong expertise in this game ? or he/she was having some edge over **B** by starting first ?

Let us now play again, with the same configuration of the piles as above but this time **B** starting first instead of **A**.

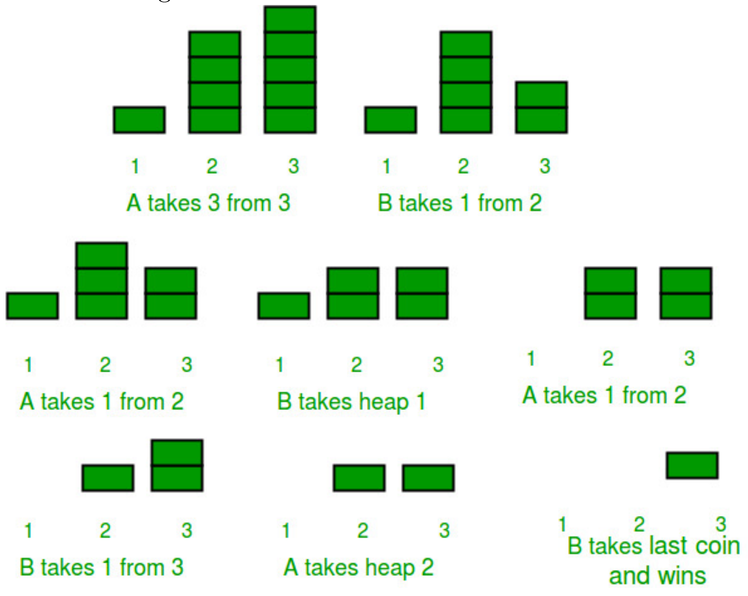


B Won the match (Note: B made the first move)

By the above figure, it must be clear that the game depends on one important factor – Who starts the game first ?

So does the player who starts first will win everytime ?
Let us again play the game, starting from **A** , and this time with a different initial configuration of piles. The piles have 1, 4, 5 coins initially.

Will **A** win again as he has started first ? Let us see.



A made the first move, but lost the Game.

So, the result is clear. **A** has lost. But how? We know that this game depends heavily on which player starts first. Thus, there must be another factor which dominates the result of this simple-yet-interesting game. That factor is the initial configuration of the heaps/piles. This time the initial configuration was different from the previous one.

So, we can conclude that this game depends on two factors-

1. The player who starts first.
2. The initial configuration of the piles/heaps.

In fact, we can predict the winner of the game before even playing the game !

Nim-Sum : The cumulative XOR value of the number of coins/stones in each piles/heaps at any point of the game is called Nim-Sum at that point.

"If both A and B play optimally (i.e- they don't make any mistakes), then the player starting first is guaranteed to win if the Nim-Sum at the beginning of the game is non-zero. Otherwise, if the Nim-Sum evaluates to zero, then player A will lose definitely."

For the proof of the above theorem, see- https://en.wikipedia.org/wiki/Nim#Proof_of_the_winning_formula

Let us apply the above theorem in the games played above. In the first game **A** started first and the Nim-Sum at the beginning of the game was, 3 XOR 4 XOR 5 = 2, which is a non-zero value, and hence **A** won. Whereas in the second game-play, when the initial configuration of the piles were 1, 4, and 5 and **A** started first, then **A** was destined to lose as the Nim-Sum at the beginning of the game wasv1 XOR 4 XOR 5 = 0 .

Implementation:

In the program below, we play the Nim-Game between computer and human(user)
The below program uses two functions
knowWinnerBeforePlaying() : : Tells the result before playing.
playGame() : plays the full game and finally declare the winner. The function playGame() doesn't takes input from the human(user), instead it uses a rand() function to randomly pick up a pile and randomly remove any number of stones from the picked pile.

The below program can be modified to take input from the user by removing the rand() function and inserting cin or scanf() functions.

```
 /* A C program to implement Game of Nim. The program
   assumes that both players are playing optimally */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define COMPUTER 1
#define HUMAN 2

/* A Structure to hold the two parameters of a move

 A move has two parameters-
  1) pile_index = The index of pile from which stone is
                    going to be removed
  2) stones_removed = Number of stones removed from the
                        pile indexed = pile_index  */
struct move
{
    int pile_index;
    int stones_removed;
};

/*
 piles[] -> Array having the initial count of stones/coins
            in each piles before the game has started.
 n       -> Number of piles

 The piles[] are having 0-based indexing*/
```

```
// A C function to output the current game state.
void showPiles (int piles[], int n)
{
    int i;
    printf ("Current Game Status -> ");
    for (i=0; i<n; i++)
        printf ("%d ", piles[i]);
    printf("\n");
    return;
}

// A C function that returns True if game has ended and
// False if game is not yet over
bool gameOver(int piles[], int n)
{
    int i;
    for (i=0; i<n; i++)
        if (piles[i]!=0)
            return (false);

    return (true);
}

// A C function to declare the winner of the game
void declareWinner(int whoseTurn)
{
    if (whoseTurn == COMPUTER)
        printf ("\nHUMAN won\n\n");
    else
        printf("\nCOMPUTER won\n\n");
    return;
}

// A C function to calculate the Nim-Sum at any point
// of the game.
int calculateNimSum(int piles[], int n)
{
    int i, nimsum = piles[0];
    for (i=1; i<n; i++)
        nimsum = nimsum ^ piles[i];
    return(nimsum);
}

// A C function to make moves of the Nim Game
void makeMove(int piles[], int n, struct move * moves)
{
    int i, nim_sum = calculateNimSum(piles, n);
```

```
// The player having the current turn is on a winning
// position. So he/she/it play optimally and tries to make
// Nim-Sum as 0
if (nim_sum != 0)
{
    for (i=0; i<n; i++)
    {
        // If this is not an illegal move
        // then make this move.
        if ((piles[i] ^ nim_sum) < piles[i])
        {
            (*moves).pile_index = i;
            (*moves).stones_removed =
                             piles[i]-(piles[i]^nim_sum);
            piles[i] = (piles[i] ^ nim_sum);
            break;
        }
    }
}

// The player having the current turn is on losing
// position, so he/she/it can only wait for the opponent
// to make a mistake(which doesn't happen in this program
// as both players are playing optimally). He randomly
// choose a non-empty pile and randomly removes few stones
// from it. If the opponent doesn't make a mistake,then it
// doesn't matter which pile this player chooses, as he is
// destined to lose this game.

// If you want to input yourself then remove the rand()
// functions and modify the code to take inputs.
// But remember, you still won't be able to change your
// fate/prediction.
else
{
    // Create an array to hold indices of non-empty piles
    int non_zero_indices[n], count;

    for (i=0, count=0; i<n; i++)
        if (piles[i] > 0)
            non_zero_indices [count++] = i;

    (*moves).pile_index = (rand() % (count));
    (*moves).stones_removed =
             1 + (rand() % (piles[(*moves).pile_index]));
    piles[(*moves).pile_index] =
     piles[(*moves).pile_index] - (*moves).stones_removed;
```

```
        if (piles[(*moves).pile_index] < 0)
            piles[(*moves).pile_index]=0;
    }
    return;
}

// A C function to play the Game of Nim
void playGame(int piles[], int n, int whoseTurn)
{
    printf("\nGAME STARTS\n\n");
    struct move moves;

    while (gameOver (piles, n) == false)
    {
        showPiles(piles, n);

        makeMove(piles, n, &moves);

        if (whoseTurn == COMPUTER)
        {
            printf("COMPUTER removes %d stones from pile "
                   "at index %d\n", moves.stones_removed,
                   moves.pile_index);
            whoseTurn = HUMAN;
        }
        else
        {
            printf("HUMAN removes %d stones from pile at "
                   "index %d\n", moves.stones_removed,
                   moves.pile_index);
            whoseTurn = COMPUTER;
        }
    }

    showPiles(piles, n);
    declareWinner(whoseTurn);
    return;
}

void knowWinnerBeforePlaying(int piles[], int n,
                             int whoseTurn)
{
    printf("Prediction before playing the game -> ");

    if (calculateNimSum(piles, n) !=0)
    {
        if (whoseTurn == COMPUTER)
            printf("COMPUTER will win\n");
        else
            printf("HUMAN will win\n");
    }
    else
    {
        if (whoseTurn == COMPUTER)
            printf("HUMAN will win\n");
        else
            printf("COMPUTER will win\n");
    }

    return;
}

// Driver program to test above functions
int main()
{
    // Test Case 1
    int piles[] = {3, 4, 5};
    int n = sizeof(piles)/sizeof(piles[0]);

    // We will predict the results before playing
    // The COMPUTER starts first
    knowWinnerBeforePlaying(piles, n, COMPUTER);

    // Let us play the game with COMPUTER starting first
    // and check whether our prediction was right or not
    playGame(piles, n, COMPUTER);

    /*
    Test Case 2
    int piles[] = {3, 4, 7};
    int n = sizeof(piles)/sizeof(piles[0]);

    // We will predict the results before playing
    // The HUMAN(You) starts first
    knowWinnerBeforePlaying (piles, n, COMPUTER);

    // Let us play the game with COMPUTER starting first
    // and check whether our prediction was right or not
    playGame (piles, n, HUMAN);    */

    return(0);
}
```

Output : May be different on different runs as random numbers are used to decide next move (for the loosing player).

```
Prediction before playing the game -> COMPUTER will win

GAME STARTS

Current Game Status -> 3 4 5
COMPUTER removes 2 stones from pile at index 0
Current Game Status -> 1 4 5
HUMAN removes 3 stones from pile at index 1
Current Game Status -> 1 1 5
COMPUTER removes 5 stones from pile at index 2
Current Game Status -> 1 1 0
HUMAN removes 1 stones from pile at index 1
Current Game Status -> 1 0 0
COMPUTER removes 1 stones from pile at index 0
Current Game Status -> 0 0 0

COMPUTER won
```

References :
https://en.wikipedia.org/wiki/Nim

Source

https://www.geeksforgeeks.org/combinatorial-game-theory-set-2-game-nim/

Chapter 6

Combinatorial Game Theory | Set 3 (Grundy Numbers/Nimbers and Mex)

Combinatorial Game Theory | Set 3 (Grundy Numbers/Nimbers and Mex) - GeeksforGeeks

We have introduced Combinatorial Game Theory in Set 1 and discussed Game of Nim in Set 2.

Grundy Number is a number that defines a state of a game. We can define any impartial game (example : nim game) in terms of Grundy Number.

Grundy Numbers or Nimbers determine how any Impartial Game (not only the Game of Nim) can be solved once we have calculated the Grundy Numbers associated with that game using Sprague-Grundy Theorem.

But before calculating Grundy Numbers, we need to learn about another term- Mex.

What is Mex?
'Minimum excludant' a.k.a 'Mex' of a set of numbers is the smallest non-negative number not present in the set.

$$\mathrm{mex}(\emptyset) = 0$$
$$\mathrm{mex}(\{1, 2, 3\}) = 0$$
$$\mathrm{mex}(\{0, 2, 4, 6, \ldots\}) = 1$$
$$\mathrm{mex}(\{0, 1, 4, 7, 12\}) = 2$$
$$\mathrm{mex}(\{0, 1, 2, 3, \ldots\}) = \omega$$
$$\mathrm{mex}(\{0, 1, 2, 3, \ldots, \omega\}) = \omega + 1$$

How to calculate Grundy Numbers?
We use this definition- The Grundy Number/ nimber is equal to 0 for a game that is lost

immediately by the first player, and is equal to Mex of the nimbers of all possible next positions for any other game.

Below are three example games and programs to calculate Grundy Number and Mex for each of them. Calculation of Grundy Numbers is done basically by a recursive function called as calculateGrundy() function which uses calculateMex() function as its sub-routine.

Example 1
The game starts with a pile of n stones, and the player to move may take any positive number of stones. Calculate the Grundy Numbers for this game. The last player to move wins. Which player wins the game?

Since if the first player has 0 stones, he will lose immediately, so Grundy(0) = 0

If a player has 1 stones, then he can take all the stones and win. So the next possible position of the game (for the other player) is (0) stones

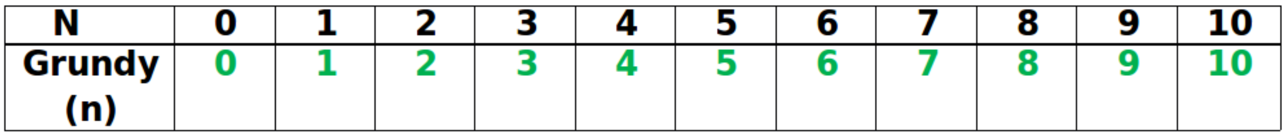
Hence, Grundy(1) = Mex(0) = 1 [According to the definition of Mex]

Similarly, If a player has 2 stones, then he can take only 1 stone or he can take all the stones and wins. So the next possible position of the game (for the other player) is (1, 0) stones respectively.

Hence, Grundy(2) = Mex(0, 1) = 2 [According to the definition of Mex]

Similarly, If a player has 'n' stones, then he can take only 1 stone, or he can take 2 stones…….. or he can take all the stones and win. So the next possible position of the game (for the other player) is (n-1, n-2,….1) stones respectively.

Hence, Grundy(n) = Mex (0, 1, 2, ….n-1) = n [According to the definition of Mex]
We summarize the first the Grundy Value from 0 to 10 in the below table-



N	0	1	2	3	4	5	6	7	8	9	10
Grundy (n)	0	1	2	3	4	5	6	7	8	9	10

```
 /* A recursive C++ program to find Grundy Number for
   a game which is like a one-pile version of Nim.
  Game Description : The game starts with a pile of n stones,
  and the player to move may take any positive number of
  stones. The last player to move wins. Which player wins
  the game? */
#include<bits/stdc++.h>
using namespace std;

// A Function to calculate Mex of all the values in
// that set.
int calculateMex(unordered_set<int> Set)
{
    int Mex = 0;
    while (Set.find(Mex) != Set.end())
        Mex++;

    return (Mex);
}

// A function to Compute Grundy Number of 'n'
// Only this function varies according to the game
int calculateGrundy(int n)
{
    if (n == 0)
        return (0);

    unordered_set<int> Set; // A Hash Table

    for (int i=0; i<=n-1; i++)
            Set.insert(calculateGrundy(i));

    return (calculateMex(Set));
}

// Driver program to test above functions
int main()
{
    int n = 10;
    printf("%d", calculateGrundy(n));
    return (0);
}
```

Output :

```
10
```

The above solution can be optimized using Dynamic Programming as there are overlapping subproblems. The Dynamic programming based implementation can be found here.

Example 2
The game starts with a pile of n stones, and the player to move may take any positive number of stones upto 3 only. The last player to move wins. Which player wins the game? This game is 1 pile version of Nim.

Since if the first player has 0 stones, he will lose immediately, so Grundy(0) = 0

If a player has 1 stones, then he can take all the stones and win. So the next possible position of the game (for the other player) is (0) stones

Hence, Grundy(1) = Mex(0) = 1 [According to the definition of Mex]

Similarly, if a player has 2 stones, then he can take only 1 stone or he can take 2 stones and win. So the next possible position of the game (for the other player) is (1, 0) stones respectively.

Hence, Grundy(2) = Mex(0, 1) = 2 [According to the definition of Mex]

Similarly, Grundy(3) = Mex(0, 1, 2) = 3 [According to the definition of Mex]

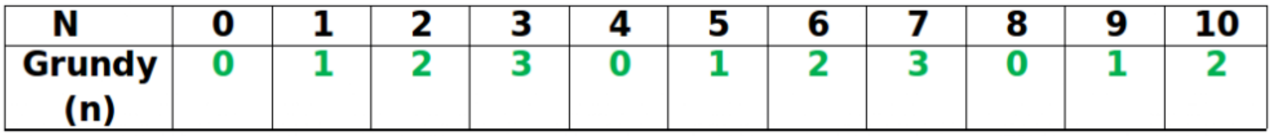
But what about 4 stones ?
If a player has 4 stones, then he can take 1 stone or he can take 2 stones or 3 stones, but he can't take 4 stones (see the constraints of the game). So the next possible position of the game (for the other player) is (3, 2, 1) stones respectively.

Hence, Grundy(4) = Mex (1, 2, 3) = 0 [According to the definition of Mex]

So we can define Grundy Number of any n >= 4 recursively as-

Grundy(n) = Mex[Grundy (n-1), Grundy (n-2), Grundy (n-3)]

We summarize the first the Grundy Value from 0 to 10 in the below table-



N	0	1	2	3	4	5	6	7	8	9	10
Grundy (n)	0	1	2	3	0	1	2	3	0	1	2

```
 /* A recursive C++ program to find Grundy Number for
   a game which is one-pile version of Nim.
  Game Description : The game starts with a pile of
  n stones, and the player to move may take any
  positive number of stones upto 3 only.
  The last player to move wins. */
#include<bits/stdc++.h>
using namespace std;

// A Function to calculate Mex of all the values in
// that set.
int calculateMex(unordered_set<int> Set)
{
    int Mex = 0;

    while (Set.find(Mex) != Set.end())
        Mex++;

    return (Mex);
}

// A function to Compute Grundy Number of 'n'
// Only this function varies according to the game
int calculateGrundy(int n)
{
    if (n == 0)
        return (0);
    if (n == 1)
        return (1);
    if (n == 2)
        return (2);
    if (n == 3)
        return (3);

    unordered_set<int> Set; // A Hash Table

    for (int i=1; i<=3; i++)
            Set.insert(calculateGrundy(n - i));

    return (calculateMex(Set));
}

// Driver program to test above functions
int main()
{
    int n = 10;
    printf("%d", calculateGrundy(n));
    return (0);
}
```

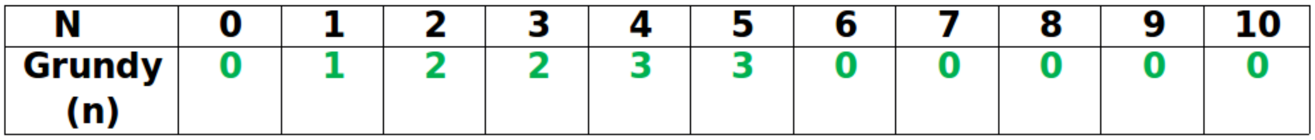
Output :

```
2
```

The above solution can be optimized using Dynamic Programming as there are overlapping subproblems. The Dynamic programming based implementation can be found here.

Example 3
The game starts with a number- 'n' and the player to move divides the number- 'n' with 2, 3 or 6 and then takes the floor. If the integer becomes 0, it is removed. The last player to move wins. Which player wins the game?

We summarize the first the Grundy Value from 0 to 10 in the below table:



N	0	1	2	3	4	5	6	7	8	9	10
Grundy (n)	0	1	2	2	3	3	0	0	0	0	0

Think how we generated this table.

```
 /* A recursive C++ program to find Grundy Number for
```

```
   a game.
 Game Description :  The game starts with a number- 'n'
 and the player to move divides the number- 'n' with 2, 3
 or 6 and then takes the floor. If the integer becomes 0,
 it is removed. The last player to move wins.  */
#include<bits/stdc++.h>
using namespace std;

// A Function to calculate Mex of all the values in
// that set.
int calculateMex(unordered_set<int> Set)
{
    int Mex = 0;

    while (Set.find(Mex) != Set.end())
        Mex++;

    return (Mex);
}

// A function to Compute Grundy Number of 'n'
// Only this function varies according to the game
int calculateGrundy (int n)
{
    if (n == 0)
        return (0);

    unordered_set<int> Set; // A Hash Table

    Set.insert(calculateGrundy(n/2));
    Set.insert(calculateGrundy(n/3));
    Set.insert(calculateGrundy(n/6));

    return (calculateMex(Set));
}

// Driver program to test above functions
int main()
{
    int n = 10;
    printf("%d", calculateGrundy (n));
    return (0);
}
```

Output :

```
0
```

The above solution can be optimized using Dynamic Programming as there are overlapping subproblems. The Dynamic programming based implementation can be found here.

References-
https://en.wikipedia.org/wiki/Mex_(mathematics)
https://en.wikipedia.org/wiki/Nimber

In the next post, we will be discussing solutions of Impartial Games using Grundy Numbers or Nimbers.

Improved By : ozym4nd145

Source

https://www.geeksforgeeks.org/combinatorial-game-theory-set-3-grundy-numbersnimbers-and-mex/

Chapter 7

Combinatorial Game Theory | Set 4 (Sprague – Grundy Theorem)

Combinatorial Game Theory | Set 4 (Sprague - Grundy Theorem) - GeeksforGeeks

Prerequisites : Grundy Numbers/Nimbers and Mex

We have already seen in Set 2 (https://www.geeksforgeeks.org/combinatorial-game-theory-set-2-game-nim/), that we can find who wins in a game of Nim without actually playing the game.

Suppose we change the classic Nim game a bit. This time each player can only remove 1, 2 or 3 stones only (and not any number of stones as in the classic game of Nim). Can we predict who will win?

Yes, we can predict the winner using Sprague-Grundy Theorem.

What is Sprague-Grundy Theorem?
Suppose there is a composite game (more than one sub-game) made up of N sub-games and two players, A and B. Then Sprague-Grundy Theorem says that if both A and B play optimally (i.e., they don't make any mistakes), then the player starting first is guaranteed to win if the XOR of the grundy numbers of position in each sub-games at the beginning of the game is non-zero. Otherwise, if the XOR evaluates to zero, then player A will lose definitely, no matter what.

How to apply Sprague Grundy Theorem ?
We can apply Sprague-Grundy Theorem in any impartial gameand solve it. The basic steps are listed as follows:

1. Break the composite game into sub-games.
2. Then for each sub-game, calculate the Grundy Number at that position.
3. Then calculate the XOR of all the calculated Grundy Numbers.

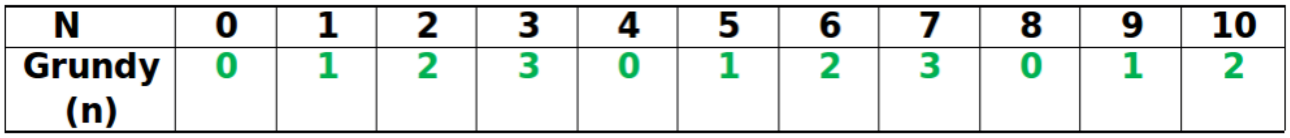
4. If the XOR value is non-zero, then the player who is going to make the turn (First Player) will win else he is destined to lose, no matter what.

Example Game : The game starts with 3 piles having 3, 4 and 5 stones, and the player to move may take any positive number of stones upto 3 only from any of the piles [Provided that the pile has that much amount of stones]. The last player to move wins. Which player wins the game assuming that both players play optimally?

How to tell who will win by applying Sprague-Grundy Theorem?
As, we can see that this game is itself composed of several sub-games.

First Step : The sub-games can be considered as each piles.
Second Step : We see from the below table that

```
Grundy(3) = 3
Grundy(4) = 0
Grundy(5) = 1
```



N	0	1	2	3	4	5	6	7	8	9	10
Grundy (n)	0	1	2	3	0	1	2	3	0	1	2

We have already seen how to calculate the Grundy Numbers of this game in the previous article.

Third Step : The XOR of 3, 4, 5 = 2

Fourth Step : Since XOR is a non-zero number, so we can say that the first player will win.

Below is C++ program that implements above 4 steps.

```
 /*  Game Description-
 "A game is played between two players and there are N piles
 of stones such that each pile has certain number of stones.
 On his/her turn, a player selects a pile and can take any
 non-zero number of stones upto 3 (i.e- 1,2,3)
 The player who cannot move is considered to lose the game
 (i.e., one who take the last stone is the winner).
 Can you find which player wins the game if both players play
 optimally (they don't make any mistake)? "

 A Dynamic Programming approach to calculate Grundy Number
 and Mex and find the Winner using Sprague - Grundy Theorem. */

#include<bits/stdc++.h>
using namespace std;
```

```
/* piles[] -> Array having the initial count of stones/coins
            in each piles before the game has started.
   n       -> Number of piles

   Grundy[] -> Array having the Grundy Number corresponding to
             the initial position of each piles in the game

   The piles[] and Grundy[] are having 0-based indexing*/

#define PLAYER1 1
#define PLAYER2 2

// A Function to calculate Mex of all the values in that set
int calculateMex(unordered_set<int> Set)
{
    int Mex = 0;

    while (Set.find(Mex) != Set.end())
        Mex++;

    return (Mex);
}

// A function to Compute Grundy Number of 'n'
int calculateGrundy(int n, int Grundy[])
{
    Grundy[0] = 0;
    Grundy[1] = 1;
    Grundy[2] = 2;
    Grundy[3] = 3;

    if (Grundy[n] != -1)
        return (Grundy[n]);

    unordered_set<int> Set; // A Hash Table

    for (int i=1; i<=3; i++)
            Set.insert (calculateGrundy (n-i, Grundy));

    // Store the result
    Grundy[n] = calculateMex (Set);

    return (Grundy[n]);
}

// A function to declare the winner of the game
void declareWinner(int whoseTurn, int piles[],
                    int Grundy[], int n)
{
    int xorValue = Grundy[piles[0]];

    for (int i=1; i<=n-1; i++)
        xorValue = xorValue ^ Grundy[piles[i]];

    if (xorValue != 0)
    {
        if (whoseTurn == PLAYER1)
            printf("Player 1 will win\n");
        else
            printf("Player 2 will win\n");
    }
    else
    {
        if (whoseTurn == PLAYER1)
            printf("Player 2 will win\n");
        else
            printf("Player 1 will win\n");
    }

    return;
}


// Driver program to test above functions
int main()
{
    // Test Case 1
    int piles[] = {3, 4, 5};
    int n = sizeof(piles)/sizeof(piles[0]);

    // Find the maximum element
    int maximum = *max_element(piles, piles + n);

    // An array to cache the sub-problems so that
    // re-computation of same sub-problems is avoided
    int Grundy[maximum + 1];
    memset(Grundy, -1, sizeof (Grundy));

    // Calculate Grundy Value of piles[i] and store it
    for (int i=0; i<=n-1; i++)
        calculateGrundy(piles[i], Grundy);

    declareWinner(PLAYER1, piles, Grundy, n);

    /* Test Case 2
    int piles[] = {3, 8, 2};
    int n = sizeof(piles)/sizeof(piles[0]);


    int maximum = *max_element (piles, piles + n);

    // An array to cache the sub-problems so that
    // re-computation of same sub-problems is avoided
    int Grundy [maximum + 1];
    memset(Grundy, -1, sizeof (Grundy));

    // Calculate Grundy Value of piles[i] and store it
    for (int i=0; i<=n-1; i++)
        calculateGrundy(piles[i], Grundy);

    declareWinner(PLAYER2, piles, Grundy, n);   */

    return (0);
}
```

Output :

```
Player 1 will win
```

References :
https://en.wikipedia.org/wiki/Sprague%E2%80%93Grundy_theorem

Exercise to the Readers: Consider the below game.
"A game is played by two players with N integers A1, A2, .., AN. On his/her turn, a player selects an integer, divides it by 2, 3, or 6, and then takes the floor. If the integer becomes 0, it is removed. The last player to move wins. Which player wins the game if both players play optimally?"

Hint : See the example 3 of previous article.

Improved By : vinayb21

Source

https://www.geeksforgeeks.org/combinatorial-game-theory-set-4-sprague-grundy-theorem/

Chapter 8

Find the winner in nim-game

Find the winner in nim-game - GeeksforGeeks

You are given an array A[] of n-elements. There are two players Alice and Bob. A Player can choose any of element from array and remove it. If the bitwise XOR of all remaining elements equals 0 after removal of selected element, then that player looses. This problem is variation of nim-game.
Note : Each players play game alternately. Find out winner if both of the players play optimally. Alice starts the game first. In case case one-element in array consider its value as the XOR of array.

Examples :

> Input : A[] = {3, 3, 2}
> Output : Winner = Bob
> Explanation : Alice can select 2 and remove it that make XOR of array equals to zero also if Alice choose 3 to remove than Bob can choose any of 2/3 and finally Alice have to make his steps.
>
> Input : A[] = {3, 3}
> Output : Winner = Alice
> Explanation : As XOR of array is already zero Alice can't select any element to remove and hence Alice is winner.

Let's start the solution step by step. We have total of three option for the XOR of array and this game.

1. **XOR of array is already 0:** In this case Alice will unable to make a move and hence Alice is winner.
2. **XOR of array is not zero:** Now, in this case we have two options, either size of array will be odd or even.
    - CASE A: If the array size is odd then for sure Bob will win the game.
    - CASE B: If the array size is even then Alice will win the game.

Above conclusion can be proved with the help of mathematical induction.

Let A[] = {1} i.e. size of array is odd and XOR of array is non-zero: In this case Alice can select element 1 and then A[] will become empty and hence XOR of array can be considered as zero. Resulting Bob as winner.

Let size of array is even and XOR of array is non-zero. Now we can prove that Alice can always find an element to remove such that XOR of remaining elements of array will be non-zero.
To prove this lets start from the contradiction i.e. suppose whatever element you should choose XOR of remaining array must be zero.
So, let A1 Xor A2 Xor … An = X and n is even.
As per our contradiction hypothesis, Ai Xor X = 0 for 1<= i <= n.
Calculate XOR of all X Xor Ai (i.e. n equations),
After taking XOR of all n equations we have X Xor X…Xor X (n-times) = 0 as N is even.
Now, also we have A1 Xor A2 Xor.. An = 0 but we know A1 Xor A2…Xor = X. This means we have at least one element in even-size array such that after its removal XOR of remaining elements in non-zero.

Let size of array is even and XOR of array is non-zero. Alice can not remove an element Ai such that xor of remaining number is zero, because that will make Bob win. Now, take the other case when the xor of remaining N?1 number is non-zero. As we know that N?1 is even and from the induction hypothesis, we can say that the position after the current move will be a winning position for Bob. Hence, it is a losing position for Alice.

```
int res = 0;
for (int i = 1; i <= N; i++) {
    res ^= a[i];
}

if (res == 0)
    return "ALice";
if (N%2 == 0)
    return "Alice";
else
    return "Bob";
```

C++

```
 // CPP to find nim-game winner
#include <bits/stdc++.h>
using namespace std;

// function to find winner of NIM-game
string findWinner(int A[], int n)
{
    int res = 0;
    for (int i = 0; i < n; i++)
```

```
        res ^= A[i];

    // case when Alice is winner
    if (res == 0 || n % 2 == 0)
        return "Alice";

    // when Bob is winner
    else
        return "Bob";
}

// driver program
int main()
{
    int A[] = { 1, 4, 3, 5 };
    int n = siseof(A) / sizeof(A[0]);
    cout << "Winner = " << findWinner(A, n);
    return 0;
}
```

Java

```
 // Java to find nim-game winner
class GFG {

    // function to find winner of NIM-game
    static String findWinner(int A[], int n)
    {
        int res = 0;

        for (int i = 0; i < n; i++)
            res ^= A[i];

        // case when Alice is winner
        if (res == 0 || n % 2 == 0)
            return "Alice";

        // when Bob is winner
        else
            return "Bob";
    }

    //Driver code
    public static void main (String[] args)
    {
        int A[] = { 1, 4, 3, 5 };
        int n =A.length;
        System.out.print("Winner = "
                    + findWinner(A, n));
    }
}

// This code is contributed by Anant Agarwal.
```

Python3

```
 # Python3 program to find nim-game winner

# Function to find winner of NIM-game
def findWinner(A, n):

    res = 0
    for i in range(n):
        res ^= A[i]

    # case when Alice is winner
    if (res == 0 or n % 2 == 0):
        return "Alice"

    # when Bob is winner
    else:
        return "Bob"

# Driver code
A = [ 1, 4, 3, 5 ]
n = len(A)
print("Winner = ", findWinner(A, n))

# This code is contributed by Anant Agarwal.
```

C#

```
 // C# to find nim-game winner
using System;

class GFG {

    // function to find winner of NIM-game
    static String findWinner(int []A, int n)
    {
        int res = 0;

        for (int i = 0; i < n; i++)
            res ^= A[i];
        // case when Alice is winner
        if (res == 0 || n % 2 == 0)
            return "Alice";

        // when Bob is winner
        else
            return "Bob";
    }

    //Driver code
    public static void Main ()
    {
        int []A = { 1, 4, 3, 5 };
        int n =A.Length;

        Console.WriteLine("Winner = "
                    + findWinner(A, n));
    }
}

// This code is contributed by vt_m.
```

PHP

```
 <?php
// PHP to find nim-game winner

// function to find
// winner of NIM-game
function findWinner($A, $n)
{
    $res = 0;
    for ($i = 0; $i < $n; $i++)
        $res ^= $A[$i];

    // case when Alice is winner
    if ($res == 0 or $n % 2 == 0)
        return "Alice";

    // when Bob is winner
    else
        return "Bob";
}

// Driver Code
$A = array(1, 4, 3, 5 );
$n = count($A);
```

```
echo "Winner = " , findWinner($A, $n);

// This code is contributed by vt_m.
?>
```

Output :

```
Winner = Alice
```

Improved By : vt_m

Source

https://www.geeksforgeeks.org/find-winner-nim-game/

Chapter 9

Game of N stones where each player can remove 1, 3 or 4

Game of N stones where each player can remove 1, 3 or 4 - GeeksforGeeks

Two players are playing a game with n stones where player 1 always plays first. The two players move in alternating turns and plays optimally. In a single move a player can remove either 1, 3 or 4 stones from the pile of stones. If a player is unable to make a move then that player loses the game. Given the number of stones where n is less than equal to 200, find and print the name of the winner.

Examples:

```
Input : 4
Output : player 1

Input : 7
Output : player 2
```

To solve this problem, we need to find each possible value of n as a winning or losing position. Since above game is one of the impartial combinatorial games, therefore the characterization of losing and winning position is valid.

The characteristic properties of winning and losing states are:

- All terminal positions are losing positions.
- From every winning position, there is atleast one move to a losing position.
- From every losing position, every move is to a winning position.

If a player is able to make a move such that the next move is the losing state than the player is at winning state. Find the state of player 1, if player1 is in winning state then player 1 wins the game otherwise player 2 will win.

Consider the following base positions:

1. **position 0** is the losing state, if the number of stones is 0 than the player1 will unable to make a move therefore player1 loses.
2. **position 1** is the winning state, if the number of stones is 1 than the player1 will remove the stone and win the game.
3. **position 2** is the losing state, if the number of stones is 2 than the player1 will remove 1 stone and then player2 will remove the second stone and win the game.
4. **position 3** is the winning state, if the player is able to take a move such that the next move is the losing state than the player is at winning state, the palyer1 will remove all the 3 stones
5. **position 4** is the winning state, if the player is able to take a move such that the next move is the losing state than the player is at winning state, the palyer1 will remove all the 4 stones
6. **position 5** is the winning state, if the player is able to take a move such that the next move is the losing state than the player is at winning state, the palyer1 will remove 3 stones leaving 2 stones, which is the losing state
7. **position 6** is the winning state, if the player is able to take a move such that the next move is the losing state than the player is at winning state, the palyer1 will remove 4 stones leaving 2 stones, which is the losing state
8. **position 7** is the losing state, if the number of stones is 7 than the player1 can remove 1, 3 or 4 stones which all leads to the losing state, therefore player1 will lose.

Below is the implementation of above approach:

CPP

```
 // CPP program to find winner of
// the game of N stones
#include <bits/stdc++.h>
using namespace std;

const int MAX = 200;

// finds the winning and losing
// states for the 200 stones.
void findStates(int position[])
{
    // 0 means losing state
    // 1 means winning state
    position[0] = 0;
    position[1] = 1;
    position[2] = 0;
    position[3] = 1;
    position[4] = 1;
    position[5] = 1;
    position[6] = 1;
    position[7] = 0;
    // find states for other positions
    for (int i = 8; i <= MAX; i++) {
        if (!position[i - 1] || !position[i - 3]
            || !position[i - 4])
            position[i] = 1;
        else
            position[i] = 0;
    }
}

// driver function
int main()
{
    int N = 100;
    int position[MAX] = { 0 };

    findStates(position);

    if (position[N] == 1)
        cout << "Player 1";
    else
        cout << "Player 2";

    return 0;
}
```

Java

```
 // Java program for the variation
// in nim game
class GFG {

    static final int MAX = 200;

    // finds the winning and losing
    // states for the 200 stones.
    static void findStates(int position[])
    {

        // 0 means losing state
        // 1 means winning state
        position[0] = 0;
        position[1] = 1;
        position[2] = 0;
        position[3] = 1;
        position[4] = 1;
        position[5] = 1;
        position[6] = 1;
        position[7] = 0;

        // find states for other positions
        for (int i = 8; i < MAX; i++) {

            if (position[i - 1]!=1 ||
                    position[i - 3]!=1
                  || position[i - 4]!=1)
                position[i] = 1;
            else
                position[i] = 0;
        }
    }

    //Driver code
    public static void main (String[] args)
    {

        int N = 100;
        int position[]=new int[MAX];

        findStates(position);

        if (position[N] == 1)
            System.out.print("Player 1");
        else
            System.out.print("Player 2");
    }
}

// This code is contributed by Anant Agarwal.
```

Python3

```
 # Python3 program to find winner of
# the game of N stones

MAX = 200

# finds the winning and losing
# states for the 200 stones.
def findStates(position):

    # 0 means losing state
    # 1 means winning state
    position[0] = 0;
    position[1] = 1;
    position[2] = 0;
    position[3] = 1;
    position[4] = 1;
    position[5] = 1;
    position[6] = 1;
    position[7] = 0

    # find states for other positions
    for i in range(8,MAX+1):
        if not(position[i - 1]) or not(position[i - 3]) or not(position[i - 4]):
            position[i] = 1;
        else:
            position[i] = 0;

#driver function
N = 100
position = [0] * (MAX+1)

findStates(position)

if (position[N] == 1):
        print("Player 1")
else:
        print("Player 2")


# This code is contributed by
# Smitha Dinesh Semwal
```

C#

```
 // C# program for the variation
// in nim game
using System;

class GFG {

    static int MAX = 200;

    // finds the winning and losing
    // states for the 200 stones.
    static void findStates(int []position)
    {

        // 0 means losing state
        // 1 means winning state
        position[0] = 0;
        position[1] = 1;
        position[2] = 0;
        position[3] = 1;
        position[4] = 1;
        position[5] = 1;
        position[6] = 1;
        position[7] = 0;

        // find states for other positions
        for (int i = 8; i < MAX; i++)
        {
            if (position[i - 1] != 1
                  || position[i - 3] != 1
                   || position[i - 4]!=1)
                position[i] = 1;
            else
                position[i] = 0;
        }
    }

    // Driver code
    public static void Main ()
    {
        int N = 100;
        int []position = new int[MAX];

        findStates(position);

        if (position[N] == 1)
            Console.WriteLine("Player 1");
        else
            Console.WriteLine("Player 2");
    }
}

// This code is contributed by vt_m.
```

Output:

```
Player 2
```

Improved By : vt_m

Source

https://www.geeksforgeeks.org/game-of-n-stones-where-each-player-can-remove-1-3-or-4/

Chapter 10

Game of Nim with removal of one stone allowed

Game of Nim with removal of one stone allowed - GeeksforGeeks

In Game of Nim, two players take turns removing objects from heaps or the pile of stones. Suppose two players A and B are playing the game. Each is allowed to take only one stone from the pile. The player who picks the last stone of the pile will win the game. Given **N** the number of stones in the pile, the task is to find the winner, if player A starts the game.

Examples :

```
Input : N = 3.
Output : Player A

Player A remove stone 1 which is at the top, then Player B remove stone 2
and finally player A removes the last stone.

Input : N = 15.
Output : Player A
```

For N = 1, player A will remove the only stone from the pile and wins the game.
For N = 2, player A will remove the first stone and then player B remove the second or the last stone. So player B will win the game.

So, we can observe player A wins when N is odd and player B wins when N is even.

Below is the implementation of this approach:

C++

```
 // C++ program for Game of Nim with removal
```

```
// of one stone allowed.
#include<bits/stdc++.h>
using namespace std;

// Return true if player A wins,
// return false if player B wins.
bool findWinner(int N)
{
  // Checking the last bit of N.
  return N&1;
}

// Driven Program
int main()
{
  int N = 15;
  findWinner(N)? (cout << "Player A";):
                 (cout << "Player B";);
  return 0;
}
```

Java

```
 // JAVA Code For Game of Nim with
// removal of one stone allowed
import java.util.*;

class GFG {

    // Return true if player A wins,
    // return false if player B wins.
    static int findWinner(int N)
    {
      // Checking the last bit of N.
      return N & 1;
    }

    /* Driver program to test above function */
    public static void main(String[] args)
    {
        int N = 15;
        if(findWinner(N)==1)
            System.out.println("Player A");
        else
             System.out.println("Player B");

    }
}
```

```
// This code is contributed by Arnav Kr. Mandal.
```

Python3

```
 # Python3 code for Game of Nim with
# removal of one stone allowed.

# Return true if player A wins,
# return false if player B wins.
def findWinner( N ):

    # Checking the last bit of N.
    return N & 1

# Driven Program
N = 15
print("Player A" if findWinner(N) else "Player B")

# This code is contributed by "Sharad_Bhardwaj".
```

C#

```
 // C# Code For Game of Nim with
// removal of one stone allowed
using System;

class GFG {

    // Return true if player A wins,
    // return false if player B wins.
    static int findWinner(int N)
    {
        // Checking the last bit of N.
        return N & 1;
    }

    /* Driver program to test above function */
    public static void Main()
    {
        int N = 15;

        if(findWinner(N) == 1)
            Console.Write("Player A");
        else
            Console.Write("Player B");
    }
}

// This code is contributed by vt_m.
```

PHP

```
 <?php
// PHP program for Game of
// Nim with removal of one
// stone allowed.

// Return true if player A wins,
// return false if player B wins.
function findWinner($N)
{

// Checking the last bit of N.
return $N&1;
}

// Driver Code
$N = 15;

if(findWinner($N))
echo "Player A";
else
echo "Player B";

// This code is contributed by vt_m.
?>
```

Output :

```
Player A
```

Time Complexity: O(1).

Improved By : vt_m

Source

https://www.geeksforgeeks.org/game-nim-removal-one-stone-allowed/

Chapter 11

Game of replacing array elements

Game of replacing array elements - GeeksforGeeks

There are two players A and B who are interested in playing a game of numbers. In each move a player pick two distinct number, let's say *a1* and *a2* and then replace all *a2* by *a1* or *a1* by *a2*. They stop playing game if any one of them is unable to pick two number and the player who is unable to pick two distinct number in an array, looses the game. First player always move first and then second. Task is to find which player wins.

Examples:

```
Input :  arr[] = { 1, 3, 3, 2,, 2, 1 }
Output : Player 2 wins
Explanation:
First plays always looses irrespective
of the numbers chosen by him. For example,
say first player picks ( 1 & 3)
replace all 3 by 1
Now array Become { 1, 1, 1, 2, 2, 1 }
Then second player picks ( 1  2 )
either he replace 1 by 2 or 2 by 1
Array Become { 1, 1, 1, 1, 1, 1 }
Now first player is not able to choose.

Input  : arr[] = { 1, 2, 1, 2 }
Output : Player 1 wins
```

From above examples, we can observe that if number of count of distinct element is even, first player always wins. Else second player wins.

Lets take an another example :

```
  int arr[] =  1, 2, 3, 4, 5, 6
```

Here number of distinct element is even(n). If player 1 pick any two number lets say (4, 1), then we left with n-1 distinct element. So player second left with n-1 distinct element. This precess go on until distinct element become 1. Here **n = 6**

```
Player   :  P1    p2    P1   p2    P1     P2
distinct : [n, n-1, n-2, n-3, n-4, n-5 ]

"At this point no distinct element left,
so p2 is unable to pick two Dis element."
```

Below c++ implementation of above idea :

```
 // CPP program for Game of Replacement
#include <bits/stdc++.h>
using namespace std;

// Function return which player win the game
int playGame(int arr[], int n)
{
    // Create hash that will stores
    // all distinct element
    unordered_set<int> hash;

    // Traverse an array element
    for (int i = 0; i < n; i++)
        hash.insert(arr[i]);

    return (hash.size() % 2 == 0 ? 1 : 2);
}

// Driver Function
int main()
{
    int arr[] = { 1, 1, 2, 2, 2, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Player " << playGame(arr, n) << " Wins" << endl;
    return 0;
}
```

Output:

```
Player 1 Wins
```

Time Complexity : O(n)

Source

https://www.geeksforgeeks.org/game-replacing-array-elements/

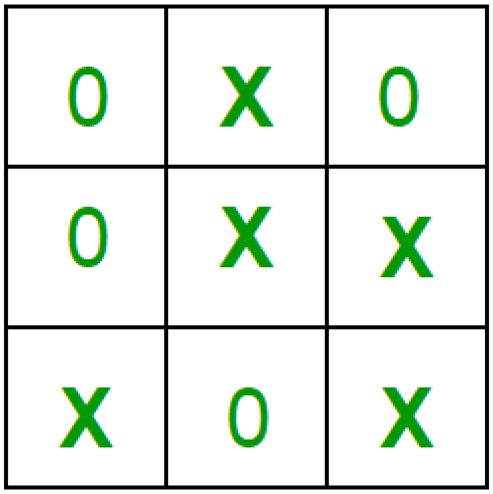
Chapter 12

Implementation of Tic-Tac-Toe game

Implementation of Tic-Tac-Toe game - GeeksforGeeks

Rules of the Game

- The game is to be played between two people (in this program between HUMAN and COMPUTER).
- One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.
- The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).
- If no one wins, then the game is said to be draw.



Implementation
In our program the moves taken by the computer and the human are chosen randomly. We use rand() function for this.

What more can be done in the program?
The program is in not played optimally by both sides because the moves are chosen randomly. The program can be easily modified so that both players play optimally (which will fall under the category of Artificial Intelligence). Also the program can be modified such that the user himself gives the input (using scanf() or cin).
The above changes are left as an exercise to the readers.

Winning Strategy – An Interesting Fact
If both the players play optimally then it is destined that you will never lose ("although the match can still be drawn"). It doesn't matter whether you play first or second.In another ways – " Two expert players will always draw ".
Isn't this interesting ?

```
 // A C++ Program to play tic-tac-toe

#include<bits/stdc++.h>
using namespace std;

#define COMPUTER 1
#define HUMAN 2

#define SIDE 3 // Length of the board

// Computer will move with 'O'
// and human with 'X'
#define COMPUTERMOVE 'O'
#define HUMANMOVE 'X'

// A function to show the current board status
void showBoard(char board[][SIDE])
{
    printf("\n\n");

    printf("\t\t\t  %c | %c  | %c  \n", board[0][0],
                             board[0][1], board[0][2]);
    printf("\t\t\t--------------\n");
    printf("\t\t\t  %c | %c  | %c  \n", board[1][0],
                             board[1][1], board[1][2]);
    printf("\t\t\t--------------\n");
    printf("\t\t\t  %c | %c  | %c  \n\n", board[2][0],
                             board[2][1], board[2][2]);

    return;
}
```

```
// A function to show the instructions
void showInstructions()
{
    printf("\t\t\t  Tic-Tac-Toe\n\n");
    printf("Choose a cell numbered from 1 to 9 as below"
            " and play\n\n");

    printf("\t\t\t  1 | 2  | 3  \n");
    printf("\t\t\t--------------\n");
    printf("\t\t\t  4 | 5  | 6  \n");
    printf("\t\t\t--------------\n");
    printf("\t\t\t  7 | 8  | 9  \n\n");

    printf("-\t-\t-\t-\t-\t-\t-\t-\t-\n\n");

    return;
}


// A function to initialise the game
void initialise(char board[][SIDE], int moves[])
{
    // Initiate the random number generator so that
    // the same configuration doesn't arises
    srand(time(NULL));

    // Initially the board is empty
    for (int i=0; i<SIDE; i++)
    {
        for (int j=0; j<SIDE; j++)
            board[i][j] = ' ';
    }

    // Fill the moves with numbers
    for (int i=0; i<SIDE*SIDE; i++)
        moves[i] = i;

    // randomise the moves
    random_shuffle(moves, moves + SIDE*SIDE);

    return;
}

// A function to declare the winner of the game
void declareWinner(int whoseTurn)
{
    if (whoseTurn == COMPUTER)
        printf("COMPUTER has won\n");
    else
        printf("HUMAN has won\n");
    return;
}

// A function that returns true if any of the row
// is crossed with the same player's move
bool rowCrossed(char board[][SIDE])
{
    for (int i=0; i<SIDE; i++)
    {
        if (board[i][0] == board[i][1] &&
            board[i][1] == board[i][2] &&
            board[i][0] != ' ')
            return (true);
    }
    return(false);
}

// A function that returns true if any of the column
// is crossed with the same player's move
bool columnCrossed(char board[][SIDE])
{
    for (int i=0; i<SIDE; i++)
    {
        if (board[0][i] == board[1][i] &&
            board[1][i] == board[2][i] &&
            board[0][i] != ' ')
            return (true);
    }
    return(false);
}

// A function that returns true if any of the diagonal
// is crossed with the same player's move
bool diagonalCrossed(char board[][SIDE])
{
    if (board[0][0] == board[1][1] &&
        board[1][1] == board[2][2] &&
        board[0][0] != ' ')
        return(true);

    if (board[0][2] == board[1][1] &&
        board[1][1] == board[2][0] &&
         board[0][2] != ' ')
        return(true);

    return(false);
```

```
}

// A function that returns true if the game is over
// else it returns a false
bool gameOver(char board[][SIDE])
{
    return(rowCrossed(board) || columnCrossed(board)
            || diagonalCrossed(board) );
}

// A function to play Tic-Tac-Toe
void playTicTacToe(int whoseTurn)
{
    // A 3*3 Tic-Tac-Toe board for playing
    char board[SIDE][SIDE];

    int moves[SIDE*SIDE];

    // Initialise the game
    initialise(board, moves);

    // Show the instructions before playing
    showInstructions();

    int moveIndex = 0, x, y;

    // Keep playing till the game is over or it is a draw
    while (gameOver(board) == false &&
            moveIndex != SIDE*SIDE)
    {
        if (whoseTurn == COMPUTER)
        {
            x = moves[moveIndex] / SIDE;
            y = moves[moveIndex] % SIDE;
            board[x][y] = COMPUTERMOVE;
            printf("COMPUTER has put a %c in cell %d\n",
                    COMPUTERMOVE, moves[moveIndex]+1);
            showBoard(board);
            moveIndex ++;
            whoseTurn = HUMAN;
        }

        else if (whoseTurn == HUMAN)
        {
            x = moves[moveIndex] / SIDE;
            y = moves[moveIndex] % SIDE;
            board[x][y] = HUMANMOVE;
            printf ("HUMAN has put a %c in cell %d\n",
                    HUMANMOVE, moves[moveIndex]+1);
            showBoard(board);
            moveIndex ++;
            whoseTurn = COMPUTER;
        }
    }

    // If the game has drawn
    if (gameOver(board) == false &&
            moveIndex == SIDE * SIDE)
        printf("It's a draw\n");
    else
    {
        // Toggling the user to declare the actual
        // winner
        if (whoseTurn == COMPUTER)
            whoseTurn = HUMAN;
        else if (whoseTurn == HUMAN)
            whoseTurn = COMPUTER;

        // Declare the winner
        declareWinner(whoseTurn);
    }
    return;
}

// Driver program
int main()
{
    // Let us play the game with COMPUTER starting first
    playTicTacToe(COMPUTER);

    return (0);
}
```

Output:

```


                Tic-Tac-Toe

Choose a cell numbered from 1 to 9 as below and play

              1 | 2  | 3
            --------------
              4 | 5  | 6
            --------------
              7 | 8  | 9
```

```
-    -    -    -    -    -    -    -    -    -

COMPUTER has put a O in cell 6


                |    |
            --------------
                |    | O
            --------------
                |    |

HUMAN has put a X in cell 7


                |    |
            --------------
                |    | O
            --------------
              X |    |

COMPUTER has put a O in cell 5


                |    |
            --------------
                | O  | O
            --------------
              X |    |

HUMAN has put a X in cell 1


              X |    |
            --------------
                | O  | O
            --------------
              X |    |

COMPUTER has put a O in cell 9


              X |    |
            --------------
                | O  | O
            --------------
              X |    | O

HUMAN has put a X in cell 8
```

```
                X |     |
              --------------
                  | O   | O
              --------------
                X | X   | O

COMPUTER has put a O in cell 4


                X |     |
              --------------
                O | O   | O
              --------------
                X | X   | O

COMPUTER has won
```

An Interesting Variant of this game
As said above, if two experienced players are playing the Tic-Tac-Toe, then the game will always draw.
There is another viral variant of this game- Ultimate Tic-Tac-Toe, which aims to make the normal Tic-Tac-Toe more interesting and less predictable.
Have a look at the game here- Link1 Link2

The above article implements simple Tic-Tac-Toe where moves are randomly made. Please refer below article to see how optimal moves are made.
Minimax Algorithm in Game Theory | Set 3 (Tic-Tac-Toe AI – Finding optimal move)

Great discussions on the "winning/never losing" strategy
Quora
Wikihow

Source

https://www.geeksforgeeks.org/implementation-of-tic-tac-toe-game/

Chapter 13

Josephus Problem | (Iterative Solution)

Josephus Problem | (Iterative Solution) - GeeksforGeeks

There are N Children are seated on N chairs arranged around a circle. The chairs are numbered from 1 to N. The game starts going in circles counting the children starting with the first chair. Once the count reaches K, that child leaves the game, removing his/her chair. The game starts again, beginning with the next chair in the circle. The last child remaining in the circle is the winner. Find the child that wins the game.

Examples:

```
Input : N = 5, K = 2
Output : 3
Firstly, the child at position 2 is out,
then position 4 goes out, then position 1
Finally, the child at position 5 is out.
So the position 3 survives.

Input : 7 4
Output : 2
```

We have discussed a recursive solution for Josephus Problem. The given solution is better than the recursive solution of Josephus Solution which is not suitable for large inputs as it gives stack overflow. The time complexity is O(N).

Approach – In the algorithm, we use sum variable to find out the chair to be removed. The current chair position is calculated by adding the chair count K to the previous position i.e. sum and modulus of the sum. At last we return sum+1 as numbering starts from 1 to N.

```
 // Iterative solution for Josephus Problem
```

```
#include <bits/stdc++.h>
using namespace std;

// Function for finding the winning child.
long long int find(long long int n, long long int k)
{
    long long int sum = 0, i;

    // For finding out the removed
    // chairs in each iteration
    for (i = 2; i <= n; i++)
        sum = (sum + k) % i;

    return sum + 1;
}

// Driver function to find the winning child
int main()
{
    int n = 14, k = 2;
    cout << find(n, k);
    return 0;
}
```

Output:

```
13
```

Source

https://www.geeksforgeeks.org/josephus-problem-iterative-solution/

Chapter 14

Minimax Algorithm in Game Theory | Set 1 (Introduction)

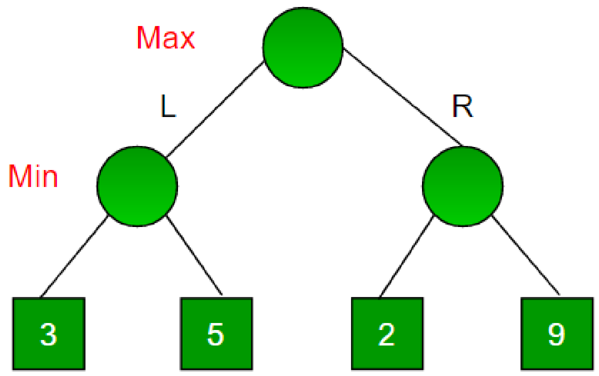
Minimax Algorithm in Game Theory | Set 1 (Introduction) - GeeksforGeeks

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn based games such as Tic-Tac-Toe, Backgamon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to get the lowest score possible while minimizer tries to do opposite.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

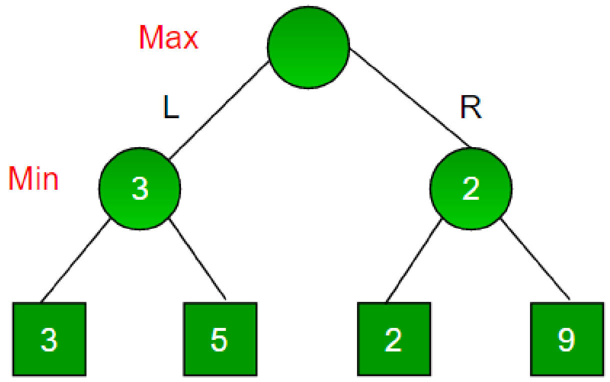
Example:
Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at root, and your opponent at next level. **Which move you would make as a maximizing player considering that your opponent also plays optimally?**



Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

- Maximizer goes LEFT : It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3
- Maximizer goes RIGHT : It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.



Now the game tree looks like below : The above tree shows two possible scores when maximizer makes left and right moves.

Note : Even though there is a value of 9 on the right sub tree, the minimizer will never pick that. We must always assume that our opponent plays optimally.

Below is implementation for the same.

C++

```
 // A simple C++ program to find
// maximum score that
// maximizing player can get.
#include<bits/stdc++.h>
using namespace std;

// Returns the optimal value a maximizer can obtain.
// depth is current depth in game tree.
// nodeIndex is index of current node in scores[].
// isMax is true if current move is
// of maximizer, else false
// scores[] stores leaves of Game tree.
// h is maximum height of Game tree
int minimax(int depth, int nodeIndex, bool isMax,
            int scores[], int h)
{
    // Terminating condition. i.e
    // leaf node is reached
    if (depth == h)
        return scores[nodeIndex];

    //  If current move is maximizer,
    // find the maximum attainable
    // value
    if (isMax)
       return max(minimax(depth+1, nodeIndex*2, false, scores, h),
            minimax(depth+1, nodeIndex*2 + 1, false, scores, h));

    // Else (If current move is Minimizer), find the minimum
    // attainable value
    else
        return min(minimax(depth+1, nodeIndex*2, true, scores, h),
            minimax(depth+1, nodeIndex*2 + 1, true, scores, h));
}

// A utility function to find Log n in base 2
int log2(int n)
{
  return (n==1)? 0 : 1 + log2(n/2);
}

// Driver code
int main()
{
    // The number of elements in scores must be
    // a power of 2.
    int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};
    int n = sizeof(scores)/sizeof(scores[0]);
    int h = log2(n);
    int res = minimax(0, 0, true, scores, h);
    cout << "The optimal value is : " << res << endl;
    return 0;
}
```

Java

```
 // A simple java program to find maximum score that
// maximizing player can get.

import java.io.*;

class GFG {


// Returns the optimal value a maximizer can obtain.
// depth is current depth in game tree.
// nodeIndex is index of current node in scores[].
// isMax is true if current move is of maximizer, else false
// scores[] stores leaves of Game tree.
// h is maximum height of Game tree
 static int minimax(int depth, int nodeIndex, boolean  isMax,
            int scores[], int h)
{
    // Terminating condition. i.e leaf node is reached
    if (depth == h)
        return scores[nodeIndex];

    // If current move is maximizer, find the maximum attainable
    // value
    if (isMax)
    return Math.max(minimax(depth+1, nodeIndex*2, false, scores, h),
            minimax(depth+1, nodeIndex*2 + 1, false, scores, h));

    // Else (If current move is Minimizer), find the minimum
    // attainable value
    else
        return Math.min(minimax(depth+1, nodeIndex*2, true, scores, h),
            minimax(depth+1, nodeIndex*2 + 1, true, scores, h));
}

// A utility function to find Log n in base 2
 static int log2(int n)
{
return (n==1)? 0 : 1 + log2(n/2);
}
// Driver code

    public static void main (String[] args) {
            // The number of elements in scores must be
    // a power of 2.
    int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};
    int n = scores.length;
    int h = log2(n);
    int res = minimax(0, 0, true, scores, h);
    System.out.println( "The optimal value is : "  +res);

    }
}

// This code is contributed by vt_m
```

Python3

```
 # A simple Python3 program to find
# maximum score that
# maximizing player can get
import math

def minimax (curDepth, nodeIndex,
             maxTurn, scores,
             targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                     True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                     True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)
```

```
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))

# This code is contributed
# by rootshadow
```

Output:

```
The optimal value is:  12
```

The idea of this article is to introduce Minimax with a simple example.

- In the above example, there are only two choices for a player. In general, there can be more choices. In that case we need to recur for all possible moves and find maximum/minimum. For example, in Tic-Tax-Toe, the first player can make 9 possible moves.
- In above example, the scores (leaves of Game Tree) are given to us. For a typical game, we need to derive these values

We will soon be covering Tic Tac Toe with Minimax algorithm.

Improved By : rootshadow

Source

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

Chapter 15

Minimax Algorithm in Game Theory | Set 2 (Introduction to Evaluation Function)

Minimax Algorithm in Game Theory | Set 2 (Introduction to Evaluation Function) - GeeksforGeeks

Prerequisite : Minimax Algorithm in Game Theory

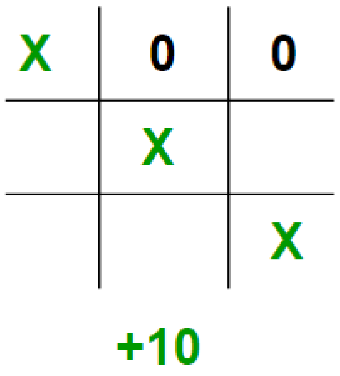
As seen in the above article, each leaf node had a value associated with it. We had stored this value in an array. But in the real world when we are creating a program to play Tic-Tac-Toe, Chess, Backgamon, etc. we need to implement a function that calculates the value of the board depending on the placement of pieces on the board. This function is often known as Evaluation Function. It is sometimes also called Heuristic Function.

The evaluation function is unique for every type of game. In this post, evaluation function for the game Tic-Tac-Toe is discussed. The basic idea behind the evaluation function is to give a high value for a board if **maximizer**'s turn or a low value for the board if **minimizer**'s turn.

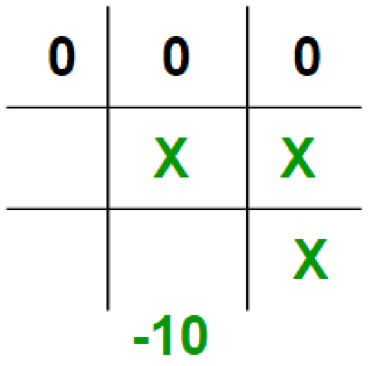
For this scenario let us consider **X** as the **maximizer** and **O** as the **minimizer**.

Let us build our evaluation function :

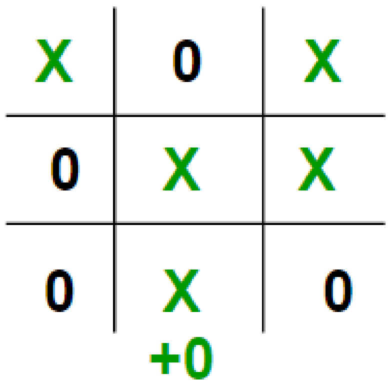
1. If X wins on the board we give it a positive value of +10.



2. If O wins on the board we give it a negative value of -10.



3. If no one has won or the game results in a draw then we give a value of +0.



We could have chosen any positive / negative value other than 10. For the sake of simplicity we chose 10 for the sake of simplicity we shall use lower case 'x' and lower case 'o' to represent the players and an underscore '_' to represent a blank space on the board.

If we represent our board as a 3×3 2D character matrix, like char board[3][3]; then we have to check each row, each column and the diagonals to check if either of the players have gotten 3 in a row.

```
 // C++ program to compute evaluation function for
// Tic Tac Toe Game.
#include<stdio.h>
#include<algorithm>
using namespace std;

// Returns a value based on who is winning
// b[3][3] is the Tic-Tac-Toe board
int evaluate(char b[3][3])
{
    // Checking for Rows for X or O victory.
    for (int row = 0; row<3; row++)
    {
        if (b[row][0]==b[row][1] && b[row][1]==b[row][2])
        {
            if (b[row][0]=='x')
               return +10;
            else if (b[row][0]=='o')
               return -10;
        }
    }

    // Checking for Columns for X or O victory.
    for (int col = 0; col<3; col++)
    {
        if (b[0][col]==b[1][col] && b[1][col]==b[2][col])
        {
            if (b[0][col]=='x')
                return +10;
            else if (b[0][col]=='o')
                return -10;
        }
    }

    // Checking for Diagonals for X or O victory.
    if (b[0][0]==b[1][1] && b[1][1]==b[2][2])
    {
        if (b[0][0]=='x')
            return +10;
        else if (b[0][0]=='o')
            return -10;
    }
    if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
    {
        if (b[0][2]=='x')
```

```
            return +10;
        else if (b[0][2]=='o')
            return -10;
    }

    // Else if none of them have won then return 0
    return 0;
}

// Driver code
int main()
{
    char board[3][3] =
    {
        { 'x', '_', 'o'},
        { '_', 'x', 'o'},
        { '_', '_', 'x'}
    };

    int value = evaluate(board);
    printf("The value of this board is %d\n", value);
    return 0;
}
```

Output :

```
The value of this board is 10
```

The idea of this article is to understand how to write a simple evaluation function for the game Tic-Tac-Toe. In the next article we shall see how to combine this evaluation function with the minimax function. Stay Tuned.

Source

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-2-evaluation-function/

Chapter 16

Minimax Algorithm in Game Theory | Set 3 (Tic-Tac-Toe AI – Finding optimal move)

Minimax Algorithm in Game Theory | Set 3 (Tic-Tac-Toe AI - Finding optimal move) - GeeksforGeeks

Prerequisites: Minimax Algorithm in Game Theory, Evaluation Function in Game Theory

Let us combine what we have learnt so far about minimax and evaluation function to write a proper Tic-Tac-Toe **AI** (**A**rtificial **I**ntelligence) that plays a perfect game. This AI will consider all possible scenarios and makes the most optimal move.

Finding the Best Move :

We shall be introducing a new function called **findBestMove()**. This function evaluates all the available moves using **minimax()** and then returns the best move the maximizer can make. The pseudocode is as follows :

```
function findBestMove(board):
    bestMove = NULL
    for each move in board :
        if current move is better than bestMove
            bestMove = current move
    return bestMove
```

Minimax :

To check whether or not the current move is better than the best move we take the help of **minimax()** function which will consider all the possible ways the game can go and returns

the best value for that move, assuming the opponent also plays optimally
The code for the maximizer and minimizer in the **minimax()** function is similar to **findBestMove()** , the only difference is, instead of returning a move, it will return a value. Here is the pseudocode :

```
function minimax(board, depth, isMaximizingPlayer):

    if current board state is a terminal state :
        return value of the board

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal

    else :
        bestVal = +INFINITY
        for each move in board :
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

Checking for GameOver state :

To check whether the game is over and to make sure there are no moves left we use **isMovesLeft()** function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively. Pseudocode is as follows :

```
function isMovesLeft(board):
    for each cell in board:
        if current cell is empty:
            return true
    return false
```

Making our AI smarter :

One final step is to make our AI a little bit smarter. Even though the following AI plays perfectly, it might choose to make a move which will result in a slower victory or a faster loss. Lets take an example and explain it.

Assume that there are 2 possible ways for X to win the game from a give board state.

- Move **A** : X can win in 2 move

- Move **B** : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves **A** and **B**. Even though the move **A** is better because it ensures a faster victory, our AI may choose **B** sometimes. To overcome this problem we subtract the depth value from the evaluated score. This means that in case of a victory it will choose a the victory which takes least number of moves and in case of a loss it will try to prolong the game and play as many moves as possible. So the new evaluated value will be

- Move **A** will have a value of +10 – 2 = 8
- Move **B** will have a value of +10 – 4 = 6

Now since move **A** has a higher score compared to move **B** our AI will choose move **A** over move **B**. The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value as the minimizer always tries to get, as negative a value as possible. We can subtract the depth either inside the evaluation function or outside it. Anywhere is fine. I have chosen to do it outside the function. Pseudocode implementation is as follows.

```
if maximizer has won:
    return WIN_SCORE - depth

else if minimizer has won:
    return LOOSE_SCORE + depth
```

Below is C++ implementation of above idea.

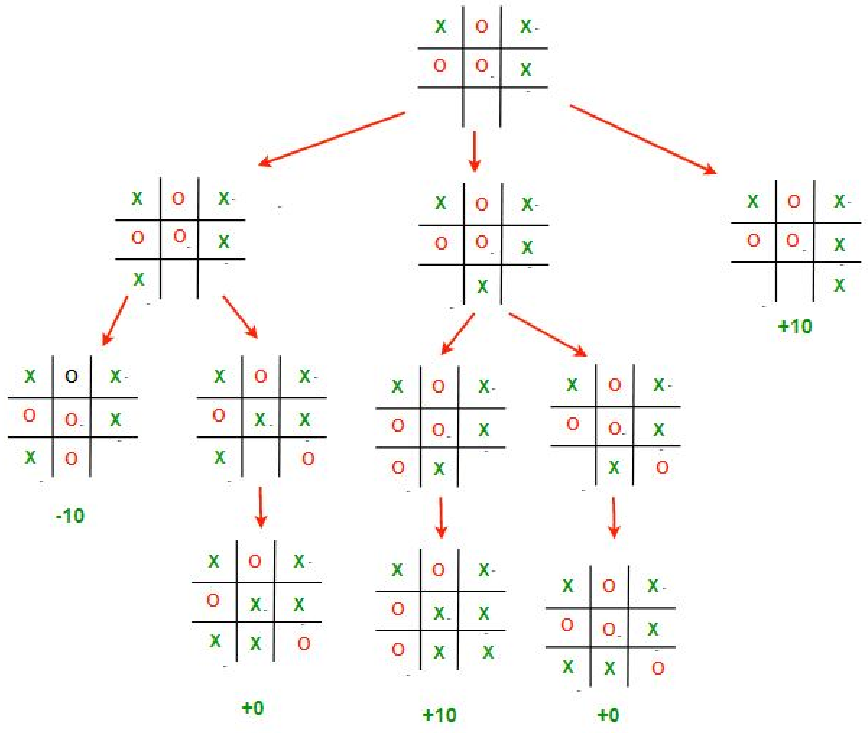
```
 // C++ program to find the next optimal move for
// a player
#include<bits/stdc++.h>
using namespace std;

struct Move
{
    int row, col;
};

char player = 'x', opponent = 'o';

// This function returns true if there are moves
// remaining on the board. It returns false if
// there are no moves left to play.
bool isMovesLeft(char board[3][3])
{
    for (int i = 0; i<3; i++)
        for (int j = 0; j<3; j++)
```

```
            if (board[i][j]=='_')
                return true;
    return false;
}

// This is the evaluation function as discussed
// in the previous article ( http://goo.gl/sJgv68 )
int evaluate(char b[3][3])
{
    // Checking for Rows for X or O victory.
    for (int row = 0; row<3; row++)
    {
        if (b[row][0]==b[row][1] &&
            b[row][1]==b[row][2])
        {
            if (b[row][0]==player)
                return +10;
            else if (b[row][0]==opponent)
                return -10;
        }
    }

    // Checking for Columns for X or O victory.
    for (int col = 0; col<3; col++)
    {
        if (b[0][col]==b[1][col] &&
            b[1][col]==b[2][col])
        {
            if (b[0][col]==player)
                return +10;

            else if (b[0][col]==opponent)
                return -10;
        }
    }

    // Checking for Diagonals for X or O victory.
    if (b[0][0]==b[1][1] && b[1][1]==b[2][2])
    {
        if (b[0][0]==player)
            return +10;
        else if (b[0][0]==opponent)
            return -10;
    }

    if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
    {
        if (b[0][2]==player)
```

```
            return +10;
        else if (b[0][2]==opponent)
            return -10;
    }

    // Else if none of them have won then return 0
    return 0;
}

// This is the minimax function. It considers all
// the possible ways the game can go and returns
// the value of the board
int minimax(char board[3][3], int depth, bool isMax)
{
    int score = evaluate(board);

    // If Maximizer has won the game return his/her
    // evaluated score
    if (score == 10)
        return score;

    // If Minimizer has won the game return his/her
    // evaluated score
    if (score == -10)
        return score;

    // If there are no more moves and no winner then
    // it is a tie
    if (isMovesLeft(board)==false)
        return 0;

    // If this maximizer's move
    if (isMax)
    {
        int best = -1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                // Check if cell is empty
                if (board[i][j]=='_')
                {
                    // Make the move
                    board[i][j] = player;

                    // Call minimax recursively and choose
                    // the maximum value
                    best = max( best,
                        minimax(board, depth+1, !isMax) );

                    // Undo the move
                    board[i][j] = '_';
                }
            }
        }
        return best;
    }

    // If this minimizer's move
    else
    {
        int best = 1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                // Check if cell is empty
                if (board[i][j]=='_')
                {
                    // Make the move
                    board[i][j] = opponent;

                    // Call minimax recursively and choose
                    // the minimum value
                    best = min(best,
                           minimax(board, depth+1, !isMax));

                    // Undo the move
                    board[i][j] = '_';
                }
            }
        }
        return best;
    }
}

// This will return the best possible move for the player
Move findBestMove(char board[3][3])
{
    int bestVal = -1000;
    Move bestMove;
    bestMove.row = -1;
    bestMove.col = -1;

    // Traverse all cells, evalutae minimax function for
    // all empty cells. And return the cell with optimal
    // value.
    for (int i = 0; i<3; i++)
    {
        for (int j = 0; j<3; j++)
        {
            // Check if celll is empty
            if (board[i][j]=='_')
            {
                // Make the move
                board[i][j] = player;

                // compute evaluation function for this
                // move.
                int moveVal = minimax(board, 0, false);

                // Undo the move
                board[i][j] = '_';

                // If the value of the current move is
                // more than the best value, then update
                // best/
                if (moveVal > bestVal)
                {
                    bestMove.row = i;
                    bestMove.col = j;
                    bestVal = moveVal;
                }
            }
        }
    }

    printf("The value of the best Move is : %d\n\n",
            bestVal);

    return bestMove;
}

// Driver code
int main()
{
    char board[3][3] =
    {
        { 'x', 'o', 'x' },
        { 'o', 'o', 'x' },
        { '_', '_', '_' }
    };

    Move bestMove = findBestMove(board);

    printf("The Optimal Move is :\n");
    printf("ROW: %d COL: %d\n\n", bestMove.row,
                                  bestMove.col );
    return 0;
}
```

Output :

```
The value of the best Move is : 10

The Optimal Move is :
ROW: 2 COL: 2
```



This image depicts all the possible paths that the game can take from the root board state. It is often called the **Game Tree**.
The 3 possible scenarios in the above example are :

- **Left Move** : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10

- **Middle Move** : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0
- **Right Move** : If X plays [2,2]. Then he will win the game. The value of this move is +10;

Remember, even though X has a possibility of winning if he plays the middle move, O will never let that happen and will choose to draw instead.

Therefore the best choice for X, is to play [2,2], which will guarantee a victory for him.

We do encourage our readers to try giving various inputs and understanding why the AI chose to play that move. Minimax may confuse programmers as it it thinks several moves in advance and is very hard to debug at times. Remember this implementation of minimax algorithm can be applied any 2 player board game with some minor changes to the board structure and how we iterate through the moves. Also sometimes it is impossible for minimax to compute every possible game state for complex games like Chess. Hence we only compute upto a certain depth and use the evaluation function to calculate the value of the board.

Stay tuned for next weeks article where we shall be discussing about **Alpha-Beta pruning** that can drastically improve the time taken by minimax to traverse a game tree.

Improved By : pk_tautolo

Source

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/

Chapter 17

Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)

Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning) - GeeksforGeeks

Prerequisites: Minimax Algorithm in Game Theory, Evaluation Function in Game Theory

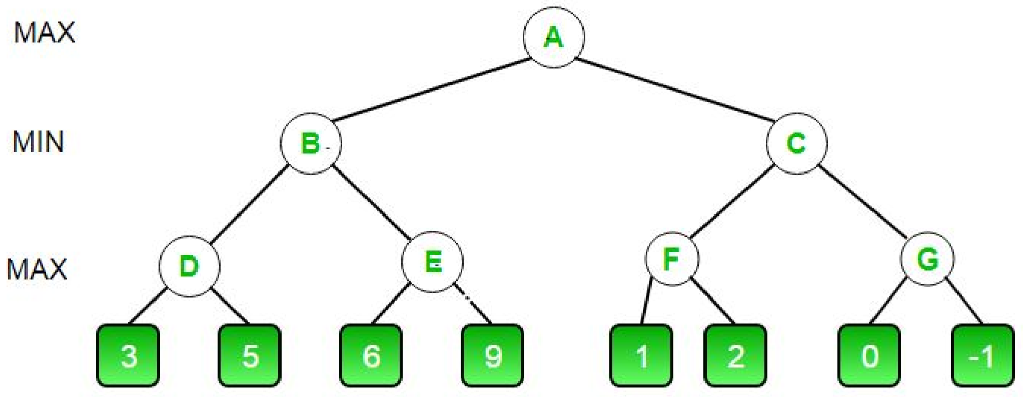
Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.
Alpha is the best value that the **maximizer** currently can guarantee at that level or above.
Beta is the best value that the **minimizer** currently can guarantee at that level or above.

Pseudocode :

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal


// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

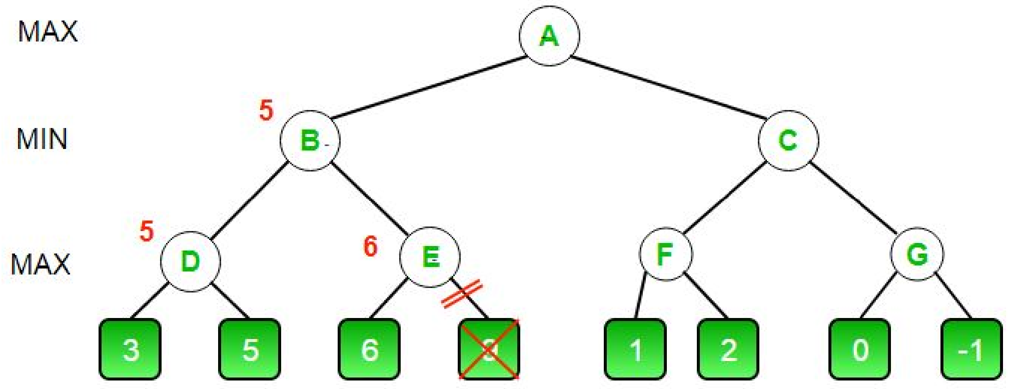
Let's make above algorithm clear with an example.



- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first
- At **B** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- **D** now looks at its right child which returns a value of 5.At **D**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5

- **D** returns a value of 5 to **B**. At **B**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.
- At **E** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**
- Now **E** looks at its left child which is 6. At **E**, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**'s right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we dint have to look at that 9 and hence saved computation time.
- **E** returns a value of 6 to **B**. At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5

So far this is how our game tree looks. The 9 is crossed out because it was never computed.

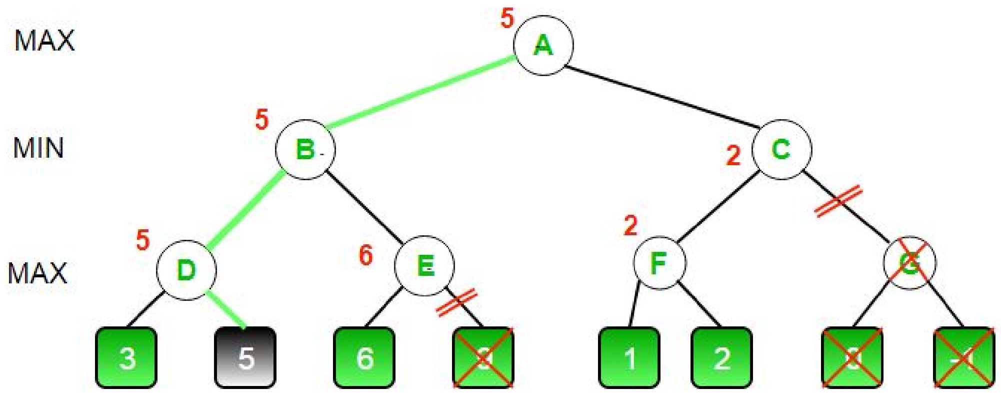


- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 and beta = +INF. **C** calls **F**
- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes false as beta = 2 and alpha = 5. So it breaks and it dose not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you

can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.

- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

This is how our final game tree looks like. As you can see **G** has been crossed out as it was never computed.



CPP

```
 // C++ program to demonstrate
// working of Alpha-Beta Pruning
#include<bits/stdc++.h>
using namespace std;

// Initial values of
// Aplha and Beta
const int MAX = 1000;
const int MIN = -1000;

// Returns optimal value for
// current player(Initially called
// for root and maximizer)
int minimax(int depth, int nodeIndex,
            bool maximizingPlayer,
            int values[], int alpha,
            int beta)
{

    // Terminating condition. i.e
    // leaf node is reached
    if (depth == 3)
        return values[nodeIndex];
    if (maximizingPlayer)
    {
        int best = MIN;

        // Recur for left and
        // right children
        for (int i = 0; i < 2; i++)
        {

            int val = minimax(depth + 1, nodeIndex * 2 + i,
                              false, values, alpha, beta);
            best = max(best, val);
            alpha = max(alpha, best);

            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }
    else
    {
        int best = MAX;

        // Recur for left and
        // right children
        for (int i = 0; i < 2; i++)
        {
            int val = minimax(depth + 1, nodeIndex * 2 + i,
                              true, values, alpha, beta);
            best = min(best, val);
            beta = min(beta, best);

            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }
}

// Driver Code
int main()
{
    int values[8] = { 3, 5, 6, 9, 1, 2, 0, -1 };
    cout <<"The optimal value is : "<< minimax(0, 0, true, values, MIN, MAX);;
    return 0;
```

```
}
```

Java

```
 // Java program to demonstrate
// working of Alpha-Beta Pruning
import java.io.*;

class GFG {

// Initial values of
// Aplha and Beta
static int MAX = 1000;
static int MIN = -1000;

// Returns optimal value for
// current player (Initially called
// for root and maximizer)
static int minimax(int depth, int nodeIndex,
                   Boolean maximizingPlayer,
                   int values[], int alpha,
                   int beta)
{
    // Terminating condition. i.e
    // leaf node is reached
    if (depth == 3)
        return values[nodeIndex];

    if (maximizingPlayer)
    {
        int best = MIN;

        // Recur for left and
        // right children
        for (int i = 0; i < 2; i++)
        {
            int val = minimax(depth + 1, nodeIndex * 2 + i,
                              false, values, alpha, beta);
            best = Math.max(best, val);
            alpha = Math.max(alpha, best);

            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }
    else
    {
        int best = MAX;

        // Recur for left and
        // right children
        for (int i = 0; i < 2; i++)
        {

            int val = minimax(depth + 1, nodeIndex * 2 + i,
                              true, values, alpha, beta);
            best = Math.min(best, val);
            beta = Math.min(beta, best);

            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }
}

    // Driver Code
    public static void main (String[] args)
    {

        int values[] = {3, 5, 6, 9, 1, 2, 0, -1};
        System.out.println("The optimal value is : " +
                            minimax(0, 0, true, values, MIN, MAX));

    }
}

// This code is contributed by vt_m.
```

Output :

```
The optimal value is : 5
```

Improved By : vinayb21, Akshay Aradhya

Source

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Chapter 18

Minimax Algorithm in Game Theory | Set 5 (Zobrist Hashing)

Minimax Algorithm in Game Theory | Set 5 (Zobrist Hashing) - GeeksforGeeks

Previous posts on this topic : Minimax Algorithm in Game Theory, Evaluation Function in Game Theory, Tic-Tac-Toe AI – Finding optimal move, Alpha-Beta Pruning.

Zobrist Hashing is a hashing function that is widely used in 2 player board games. It is the most common hashing function used in transposition table. Transposition tables basically store the evaluated values of previous board states, so that if they are encountered again we simply retrieve the stored value from the transposition table. We will be covering transposition tables in a later article. In this article we shall take the example of chess board and implement a hashing function for that.

Pseudocode :

```
// A matrix with random numbers initialized once
Table[#ofBoardCells][#ofPieces]

// Returns Zobrist hash function for current conf-
// iguration of board.
function findhash(board):
    hash = 0
    for each cell on the board :
        if cell is not empty :
            piece = board[cell]
            hash ^= table[cell][piece]
    return hash
```

Explanation :

The idea behind Zobrist Hashing is that for a given board state, if there is a piece on a given cell, we use the random number of that piece from the corresponding cell in the table.

If more bits are there in the random number the lesser chance of a hash collision. Therefore 64 bit numbers are commonly used as the standard and it is highly unlikely for a hash collision to occur with such large numbers. The table has to be initialized only once during the programs execution.

Also the reason why Zobrist Hashing is widely used in board games is because when a player makes a move, it is not necessary to recalculate the hash value from scratch. Due to the nature of XOR operation we can simply use few XOR operations to recalculate the hash value.

Implementation :

We shall try to find a hash value for the given board configuration.

```
 // A program to illustrate Zobeist Hashing Algorithm
#include <bits/stdc++.h>
using namespace std;

unsigned long long int ZobristTable[8][8][12];
mt19937 mt(01234567);

// Generates a Randome number from 0 to 2^64-1
unsigned long long int randomInt()
{
    uniform_int_distribution<unsigned long long int>
                                 dist(0, UINT64_MAX);
    return dist(mt);
}

// This function associates each piece with
// a number
int indexOf(char piece)
{
    if (piece=='P')
        return 0;
    if (piece=='N')
        return 1;
    if (piece=='B')
        return 2;
    if (piece=='R')
        return 3;
    if (piece=='Q')
        return 4;
    if (piece=='K')
        return 5;
    if (piece=='p')
        return 6;
    if (piece=='n')
        return 7;
    if (piece=='b')
        return 8;
    if (piece=='r')
        return 9;
    if (piece=='q')
        return 10;
    if (piece=='k')
        return 11;
    else
        return -1;
}

// Initializes the table
void initTable()
{
    for (int i = 0; i<8; i++)
      for (int j = 0; j<8; j++)
        for (int k = 0; k<12; k++)
          ZobristTable[i][j][k] = randomInt();
}

// Computes the hash value of a given board
unsigned long long int computeHash(char board[8][9])
{
    unsigned long long int h = 0;
    for (int i = 0; i<8; i++)
    {
        for (int j = 0; j<8; j++)
        {
            if (board[i][j]!='-')
            {
                int piece = indexOf(board[i][j]);
                h ^= ZobristTable[i][j][piece];
            }
        }
    }
    return h;
}

// Main Function
int main()
{
    // Uppercase letters are white pieces
    // Lowercase letters are black pieces
    char board[8][9] =
    {
        "---K----",
        "-R----Q-",
        "--------",
        "-P----p-",
        "-----p--",
        "--------",
        "p---b--q",
        "----n--k"
    };

    initTable();

    unsigned long long int hashValue = computeHash(board);
    printf("The hash value is     : %llu\n", hashValue);

    //Move the white king to the left
    char piece = board[0][3];

    board[0][3] = '-';
    hashValue ^= ZobristTable[0][3][indexOf(piece)];

    board[0][2] = piece;
    hashValue ^= ZobristTable[0][2][indexOf(piece)];


    printf("The new hash vlaue is : %llu\n", hashValue);

    // Undo the white king move
    piece = board[0][2];

    board[0][2] = '-';
    hashValue ^= ZobristTable[0][2][indexOf(piece)];

    board[0][3] = piece;
    hashValue ^= ZobristTable[0][3][indexOf(piece)];

    printf("The old hash vlaue is : %llu\n", hashValue);

    return 0;
}
```

Output :

```
The hash value is     : 14226429382419125366
The new hash vlaue is : 15124945578233295113
The old hash vlaue is : 14226429382419125366
```

Source

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/

Chapter 19

Optimal Strategy for a Game | DP-31

Optimal Strategy for a Game | DP-31 - GeeksforGeeks

Problem statement: Consider a row of n coins of values v1 . . . vn, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Note: The opponent is as clever as the user.

Let us understand the problem with few examples:

1. 5, 3, 7, 10 : The user collects maximum value as 15(10 + 5)
2. 8, 15, 3, 7 : The user collects maximum value as 22(7 + 15)

Does choosing the best at each move give an optimal solution?

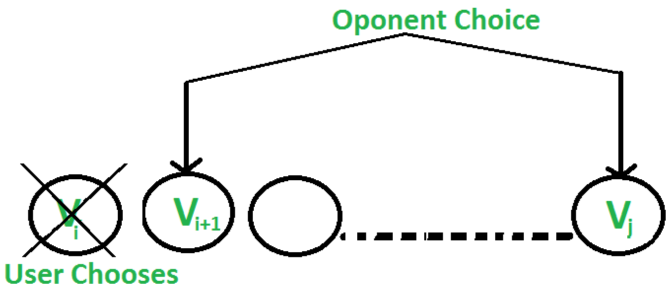
No. In the second example, this is how the game can finish:

1.
…….User chooses 8.
…….Opponent chooses 15.
…….User chooses 7.
…….Opponent chooses 3.
Total value collected by user is 15(8 + 7)

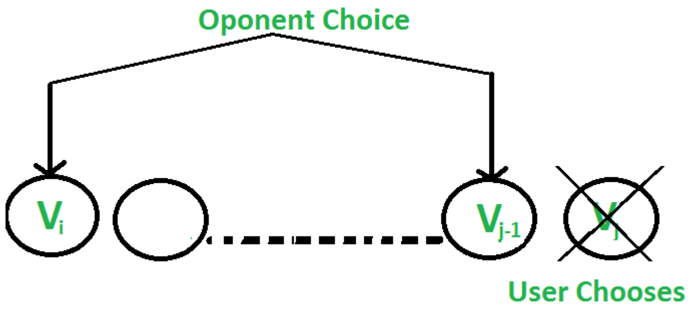
2.
…….User chooses 7.
…….Opponent chooses 8.
…….User chooses 15.
…….Opponent chooses 3.
Total value collected by user is 22(7 + 15)

So if the user follows the second game state, maximum value can be collected although the first move is not the best.

There are two choices:
1. The user chooses the ith coin with value Vi: The opponent either chooses (i+1)th coin or jth coin. The opponent intends to choose the coin which leaves the user with minimum value.
i.e. The user can collect the value Vi + min(F(i+2, j), F(i+1, j-1))



2. The user chooses the jth coin with value Vj: The opponent either chooses ith coin or (j-1)th coin. The opponent intends to choose the coin which leaves the user with minimum value.
i.e. The user can collect the value Vj + min(F(i+1, j-1), F(i, j-2))



Following is recursive solution that is based on above two choices. We take the maximum of two choices.

```
F(i, j)  represents the maximum value the user can collect from
         i'th coin to j'th coin.

    F(i, j)  = Max(Vi + min(F(i+2, j), F(i+1, j-1) ),
                   Vj + min(F(i+1, j-1), F(i, j-2) ))
Base Cases
    F(i, j)  = Vi           If j == i
    F(i, j)  = max(Vi, Vj)  If j == i+1
```

Why Dynamic Programming?
The above relation exhibits overlapping sub-problems. In the above relation, F(i+1, j-1) is

calculated twice.

C++

```
 // C program to find out maximum value from a given sequence of coins
#include <limits.h>
#include <stdio.h>

// Utility functions to get maximum and minimum of two intgers
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }

// Returns optimal value possible that a player can collect from
// an array of coins of size n. Note than n must be even
int optimalStrategyOfGame(int* arr, int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, x, y, z;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http:// goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap) {
        for (i = 0, j = gap; j < n; ++i, ++j) {
            // Here x is value of F(i+2, j), y is F(i+1, j-1) and
            // z is F(i, j-2) in above recursive formula
            x = ((i + 2) <= j) ? table[i + 2][j] : 0;
            y = ((i + 1) <= (j - 1)) ? table[i + 1][j - 1] : 0;
            z = (i <= (j - 2)) ? table[i][j - 2] : 0;

            table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));
        }
    }

    return table[0][n - 1];
}

// Driver program to test above function
int main()
{
    int arr1[] = { 8, 15, 3, 7 };
    int n = sizeof(arr1) / sizeof(arr1[0]);
    printf("%dn", optimalStrategyOfGame(arr1, n));

    int arr2[] = { 2, 2, 2, 2 };
    n = sizeof(arr2) / sizeof(arr2[0]);
    printf("%dn", optimalStrategyOfGame(arr2, n));
    int arr3[] = { 20, 30, 2, 2, 2, 10 };
    n = sizeof(arr3) / sizeof(arr3[0]);
    printf("%dn", optimalStrategyOfGame(arr3, n));

    return 0;
}
```

Java

```
 // Java program to find out maximum
// value from a given sequence of coins
import java.io.*;

class GFG {
    // Utility functions to get maximum
    // and minimum of two intgers
    int max(int a, int b) { return a > b ? a : b; }
    int min(int a, int b) { return a < b ? a : b; }

    // Returns optimal value possible that a player
    // can collect from an array of coins of size n.
    // Note than n must be even
    static int optimalStrategyOfGame(int arr[], int n)
    {
        // Create a table to store solutions of subproblems
        int table[][] = new int[n][n];
        int gap, i, j, x, y, z;

        // Fill table using above recursive formula.
        // Note that the tableis filled in diagonal
        // fashion (similar to http:// goo.gl/PQqoS),
        // from diagonal elements to table[0][n-1]
        // which is the result.
        for (gap = 0; gap < n; ++gap) {
            for (i = 0, j = gap; j < n; ++i, ++j) {
                // Here x is value of F(i+2, j),
                // y is F(i+1, j-1) and z is
                // F(i, j-2) in above recursive formula
                x = ((i + 2) <= j) ? table[i + 2][j] : 0;
                y = ((i + 1) <= (j - 1)) ? table[i + 1][j - 1] : 0;
                z = (i <= (j - 2)) ? table[i][j - 2] : 0;

                table[i][j] = Math.max(arr[i] + Math.min(x, y), arr[j] + Math.min(y, z));
            }
        }

        return table[0][n - 1];
    }
    // Driver program
    public static void main(String[] args)
    {
        int arr1[] = { 8, 15, 3, 7 };
        int n = arr1.length;
        System.out.println("" + optimalStrategyOfGame(arr1, n));

        int arr2[] = { 2, 2, 2, 2 };
        n = arr2.length;
        System.out.println("" + optimalStrategyOfGame(arr2, n));

        int arr3[] = { 20, 30, 2, 2, 2, 10 };
        n = arr3.length;
        System.out.println("" + optimalStrategyOfGame(arr3, n));
    }
}

// This code is contributed by vt_m
```

Output:

```
22
4
42
```

Exercise
Your thoughts on the strategy when the user wishes to only win instead of winning with the maximum value. Like above problem, number of coins is even.
Can Greedy approach work quite well and give an optimal solution? Will your answer change if number of coins is odd? Please see Coin game of two corners

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/optimal-strategy-for-a-game-dp-31/

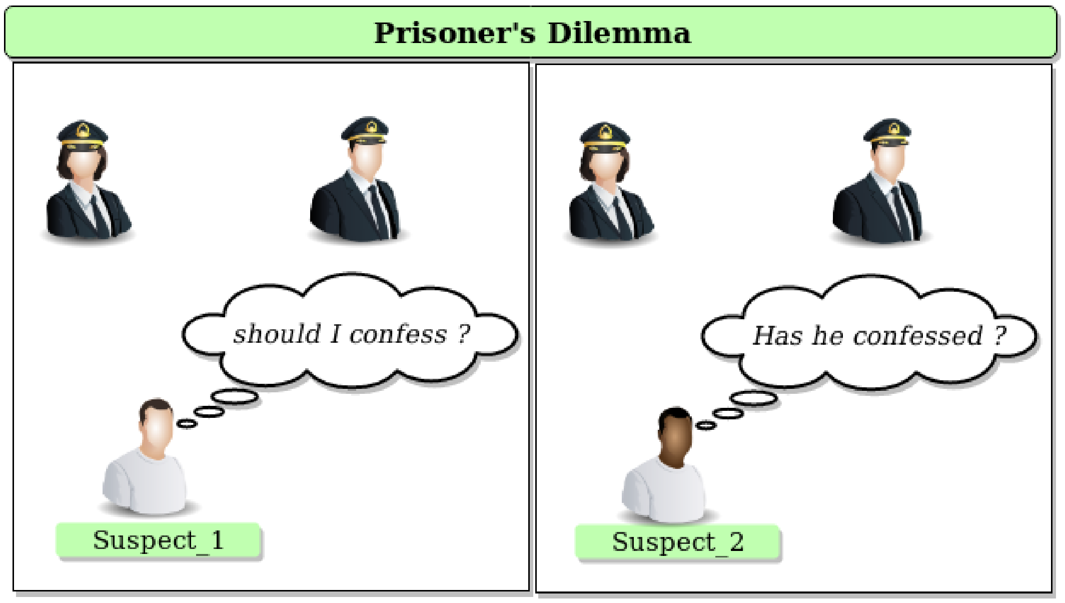
Chapter 20

The prisoner's dilemma in Game theory

The prisoner's dilemma in Game theory - GeeksforGeeks

Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge. They hope to get both sentenced to a year in prison on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to: betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. The offer is:

- If A and B each betray the other, each of them serves 2 years in prison
- If A betrays B but B remains silent, A will be set free and B will serve 3 years in prison (and vice versa)
- If A and B both remain silent, both of them will only serve 1 year in prison (on the lesser charge)



Let's analyze the nature of the dilemma assuming that both understand the nature of the game, and that despite being members of the same gang, they have no loyalty to each other and will have no opportunity for retribution or reward outside the game.

	Prisoner B stays silent	Prisoner B betrays
Prisoner A stays silent	Each serves 1 year	Prisoner A: 3 years Prisoner B: goes free
Prisoner A betrays	Prisoner A: goes free Prisoner B: 3 years	Each serves 2 years

Please try to think over the solution for a while and analyze each case yourself.
By analyzing the table we can see that:
You are always punished less for choosing to betray the other person. However, as a group, both of you fare better by cooperating(remaining silent).
Think over the above statement for a while.
If you have problem in analyzing this then you can watch this video: Khan Academy's explaination

This is the dilemma both the prisoner's face. Should one cooperate or betray?
Even if the best solution would be both the prisoners cooperating with each other but due to uncertainty on each other both of them betray each other getting a lesser optimum solution.
This can be observed in may real-life cases like:

- A pair working on a project. You do best if your competitor does all the work, since you get the same grade. But if neither of you do the work, you both fail.
- Advertising. If both companies spend money on advertising, their market share won't

> change from if neither does. But if one company outspends the other, they will receive a benefit.

The prisoner's dilemma demonstrates that two rational people might not cooperate even if it is in their best interest to do so. Just keep looking around in this beautiful world. Who knows you can find yourself in a prisoner's dilemma one day!

Source

https://www.geeksforgeeks.org/prisoners-dilemma-game-theory/

Chapter 21

Variation in Nim Game

Variation in Nim Game - GeeksforGeeks

Prerequisites:
Sprague Gruncy theorem
Grundy Numbers

Nim is a famous game in which two players take turns removing items from distinct piles. During each turn, a player must remove one or more items from a single, non-empty pile. The winner of the game is whichever player removes the last item from the last non-empty pile.
Now, For each non-empty pile, either player can remove zero items from that pile and have it count as their move; however, this move can only be performed once per pile by either player.
Given the number of items in each pile, determine who will win the game; Player 1 or player 2. If player 1 starts the game and both plays optimally.

Examples:

```
Input : 3 [18, 47, 34]
Output : Player 2 wins
G = g(18)^g(47)^g(34) = (17)^(48)^(33) = 0
Grundy number(G), for this game is zero.
Player 2 wins.

Input : 3 [32, 49, 58]
Output : Player 1 wins
G = g(31)^g(50)^g(57) = (17)^(48)^(33) = 20
Grundy number(G), for this game is non-zero.
Player 1 wins.
```

Approach:
Grundy number for each pile is calculated based on the number of stones.To compensate

the zero move we will have to modify grundy values we used in standard nim game.
If pile size is odd; grundy number is size+1 and
if pile size is even; grundy number is size-1.
We XOR all the grundy number values to check if final Grundy number(G) of game is non zero or not to decide who is winner of game.

Explanation:
Grundy number of a state is the **smallest positive integer that cannot be reached in one valid move**.
So, we need to calculate **mex value** for each n, bottom up wise so that we can induce the grundy number for each n. where n is the pile size and valid move is the move that will lead the current player to winning state.

Winning state: A tuple of values from where the current player will win the game no matter what opponent does. (If G!=0)
Losing state: A tuple of values from where the current player will loose the game no matter what opponent does. (If G=0)

```
For a given pile size n, we have two states:
(1) n with no zero move available,
grundy number will same as standard nim game.
(2) n with zero move available, we can
reach above state and other states with
zero move remaining.

For, n = 0, g(0) = 0, empty pile

For, n = 1, we can reach two states:
(1) n = 0 (zero move not used)
(2) n = 1 (zero move used)
Therefore, g(1) = mex{0, 1} which implies
that g(1)=2.

For, n = 2, we can reach :
(1) n = 0 (zero move not used) state
because this is a valid move.
(2) n = 1 is not a valid move, as it will
lead the current player into loosing state.
Therefore, g(2) = mex{0} which implies
that g(2)=1.

If we try to build a solution bottom-up
like this, it turns out that if n is even,
the grundy number is n - 1 and when it is odd,
the grundy is n + 1.
```

Below is the implementation of above approach:

C++

```
 // CPP program for the variation
// in nim game
#include <bits/stdc++.h>
using namespace std;

// Function to return final
// grundy Number(G) of game
int solve(int p[], int n)
{
    int G = 0;
    for (int i = 0; i < n; i++) {

        // if pile size is odd
        if (p[i] & 1)

            // We XOR pile size+1
            G ^= (p[i] + 1);

        else // if pile size is even

            // We XOR pile size-1
            G ^= (p[i] - 1);
    }
    return G;
}

// driver program
int main()
{
    // Game with 3 piles
    int n = 3;

    // pile with different sizes
    int p[3] = { 32, 49, 58 };

    // Function to return result of game
    int res = solve(p, n);

    if (res == 0) // if G is zero
        cout << "Player 2 wins";
    else // if G is non zero
        cout << "Player 1 wins";

    return 0;
}
```

Java

```
 // Java program for the variation
// in nim game
class GFG {

    // Function to return final
    // grundy Number(G) of game
    static int solve(int p[], int n)
    {

        int G = 0;
        for (int i = 0; i < n; i++) {

            // if pile size is odd
            if (p[i]%2!=0)

                // We XOR pile size+1
                G ^= (p[i] + 1);

            else // if pile size is even

                // We XOR pile size-1
                G ^= (p[i] - 1);
        }

        return G;
    }

    //Driver code
    public static void main (String[] args)
    {

        // Game with 3 piles
        int n = 3;

        // pile with different sizes
        int p[] = { 32, 49, 58 };

        // Function to return result of game
        int res = solve(p, n);

        if (res == 0) // if G is zero
            System.out.print("Player 2 wins");
        else // if G is non zero
            System.out.print("Player 1 wins");
    }
}
```

```
// This code is contributed by Anant Agarwal.
```

Python3

```
 # Python3 program for the
# variation in nim game

# Function to return final
# grundy Number(G) of game
def solve(p, n):
    G = 0
    for i in range(n):

        # if pile size is odd
        if (p[i] % 2 != 0):

            # We XOR pile size+1
            G ^= (p[i] + 1)

        # if pile size is even
        else:

            # We XOR pile size-1
            G ^= (p[i] - 1)

    return G

# Driver code

# Game with 3 piles
n = 3

# pile with different sizes
p = [32, 49, 58]

# Function to return result of game
res = solve(p, n)

if (res == 0): # if G is zero
    print("Player 2 wins")

else: # if G is non zero
    print("Player 1 wins")

# This code is contributed by Anant Agarwal.
```

C#

```
 // C# program for the variation
// in nim game
using System;
class GFG {

    // Function to return final
    // grundy Number(G) of game
    static int solve(int[] p, int n)
    {

        int G = 0;
        for (int i = 0; i < n; i++) {

            // if pile size is odd
            if (p[i] % 2 != 0)

                // We XOR pile size+1
                G ^= (p[i] + 1);

            else // if pile size is even

                // We XOR pile size-1
                G ^= (p[i] - 1);
        }

        return G;
    }

    // Driver code
    public static void Main()
    {

        // Game with 3 piles
        int n = 3;

        // pile with different sizes
        int[] p = { 32, 49, 58 };

        // Function to return result of game
        int res = solve(p, n);

        if (res == 0) // if G is zero
            Console.WriteLine("Player 2 wins");

        else // if G is non zero
            Console.WriteLine("Player 1 wins");
    }
}
```

```
// This code is contributed by vt_m.
```

PHP

```
 <?php
// php program for the variation
// in nim game

// Function to return final
// grundy Number(G) of game
function solve($p,$n)
{
    $G = 0;
    for ($i = 0; $i < $n; $i++)
    {

        // if pile size is odd
        if ($p[$i] & 1)

            // We XOR pile size+1
            $G ^= ($p[$i] + 1);

        else // if pile size is even

            // We XOR pile size-1
            $G ^= ($p[$i] - 1);
    }
    return $G;
}

    // Driver Code
    // Game with 3 piles
    $n = 3;

    // pile with different sizes
    $p= array( 32, 49, 58 );

    // Function to return result of game
    $res = solve($p, $n);

    if ($res == 0) // if G is zero
        echo "Player 2 wins";
    else // if G is non zero
        echo "Player 1 wins";

// This code is contributed by mits
?>
```

Output:

```
Player 1 wins
```

Time Complexity: O(n)

Improved By : Mithun Kumar

Source

https://www.geeksforgeeks.org/variation-nim-game/