

# ICPC SCU LEVEL #0

---

Recursion

# Introduction

- In easy words, Recursion is just a function that calls itself.
- Recursion is a hard technique for beginners
- Recursive thinking is a natural way of thinking and solving problems
- Understanding recursion will help you in advanced topics in future such as graph and tree algorithms, backtracking, dynamic programming, and searching techniques.

# Recall call stack

- When a function X is called inside another function Y , the rest of the Y is not executed until X has finished execution .
- The same is done if another function Z is called inside X , and so on.
- The last called function is the first called function to be removed from that stack ( LAST IN, FIRST OUT)
- In most programming languages , the **main** function is the first to be called.

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}  

```

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun1 : fun2(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun2 : fun3(10)

fun1 : fun2(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun3(10)

fun2 : fun3(10)

fun1 : fun2(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun2 : fun3(10)

fun1 : fun2(10)

main : fun1(10)



# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun1 : fun2(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun1 : fun4(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun4(10)

fun1 : fun4(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

fun1 : fun4(10)

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}  

```

main : fun1(10)

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}  

```

main :  
print("END")

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

main : return 0

# Recall call stack

```
void fun4(int x){cout<<x+4<<endl;}  
void fun3(int x){cout<<x+3<<endl;}  
void fun2(int x){cout<<x+2<<endl; fun3(x);}  
void fun1(int x){cout<<x+1<<endl; fun2(x); fun4(x);}  
int main(){  
    fun1(10);  
    cout<<"END"<<endl;  
    return 0;}
```

Program Execution  
Finishes 😊



# Problem and Subproblem

Sometimes we need to divide the problem into at least two subproblems

Example

Count all the prime palindromes less than or equal to N.

We have two sub-problems

- 1) Check if the number is prime
- 2) Check if the number is palindrome
- 3) The answer is the intersection between the two subproblems

What if the subproblem is similar to the original problem 

**Then :** The solution to this problem is *recursion*

## Problem : Calculate factorial of some integer

$$\text{factorial}(6) = 6 * 5 * 4 * 3 * 2 * 1$$

$$\text{factorial}(5) = 5 * 4 * 3 * 2 * 1$$

$$\text{factorial}(4) = 4 * 3 * 2 * 1$$

$$\text{factorial}(3) = 3 * 2 * 1$$

$$\text{factorial}(2) = 2 * 1$$

$$\text{factorial}(1) = 1$$

Can you see the relation between  $\text{factorial}(n)$  and  $\text{factorial}(n-1)$  ?

# Where's the subproblem 🤔

Say we want to calculate factorial(6)

We can easily do :  $1 * 2 * 3 * 4 * 5 * 6$

**OR**


factorial(5) is a simpler subproblem

Will it help me 🤔 → Yes,  $\text{factorial}(6) = 6 * \text{factorial}(5)$

In the same way → factorial(4) will help us in factorial(5)

In the same way → factorial(3) will help us in factorial(4)

# Base Case Vs Recursive Case

At factorial(1) can we find any smaller subproblems 

We have reached the base case where there's no smaller subproblems.

Every recursive function must have both cases :

- Recursive case  $\Rightarrow$  the smaller repeated subproblem
- Base case  $\Rightarrow$  the subproblem that can't be divided into smaller subproblems, i.e. factorial(1) can't be divided into a smaller subproblem
- Sometimes, the solution involves more than one base case and more than one recursive case, **BUT** we can't have a recursive function without at least one base case and at least one recursive case

## Code </>

```
int factorial(int n){  
    if (n==1) return 1;  
    else return n*factorial(n-1);  
}  
  
int main(){  
    cout << factorial(6) << endl; // 720  
}
```

# Lets Trace

- Call factorial(6)
  - $6 == 1$  ? No
  - Call factorial(5) and multiply the result by 6
    - Call factorial(5)
      - $5 == 1$  ? No
      - Call factorial(4) and multiply the result by 5
        - Call factorial(4)
          - $4 == 1$  ? No
          - Call factorial(3) and multiply the result by 4
            - $3 == 1$  ? No
            - Call factorial(2) and multiply the result by 3
              - $2 == 1$  ? No
              - Call factorial(1) and multiply the result by 2
                - $1 == 1$  ? Yes → Return 1

# Stack tracing

main : factorial(6)

# Stack tracing

factorial(6) : 6 \* factorial(5)

main : factorial(6)



# Stack tracing

factorial(5) : 5 \* factorial(4)

factorial(6) : 6 \* factorial(5)

main : factorial(6)

# Stack tracing

factorial(4) : 4 \* factorial(3)

factorial(5) : 5 \* factorial(4)

factorial(6) : 6 \* factorial(5)

main : factorial(6)

# Stack tracing

factorial(4) : 4 \* factorial(3)

factorial(5) : 5 \* factorial(4)

factorial(6) : 6 \* factorial(5)

main : factorial(6)

factorial(3) : 3 \* factorial(2)

# Stack tracing

factorial(4) : 4 \* factorial(3)

factorial(5) : 5 \* factorial(4)

factorial(6) : 6 \* factorial(5)

main : factorial(6)

factorial(2) : 2 \* factorial(1)

factorial(3) : 3 \* factorial(2)

# Stack tracing

factorial(4) : 4 \* factorial(3)

factorial(5) : 5 \* factorial(4)

factorial(6) : 6 \* factorial(5)

main : factorial(6)

factorial(1) : 1

factorial(2) : 2 \* factorial(1)

factorial(3) : 3 \* factorial(2)

# Stack tracing

factorial(4) : 4 \* factorial(3)

factorial(5) : 5 \* factorial(4)

factorial(6) : 6 \* factorial(5)

main : factorial(6)

factorial(2) : 2 \* 1 = 2

factorial(3) : 3 \* factorial(2)

# Stack tracing

factorial(4) :  $4 * \text{factorial}(3)$

factorial(5) :  $5 * \text{factorial}(4)$

factorial(6) :  $6 * \text{factorial}(5)$

main : factorial(6)

factorial(3) :  $3 * 2 = 6$

# Stack tracing

factorial(4) :  $4 * 6 = 24$

factorial(5) :  $5 * \text{factorial}(4)$

factorial(6) :  $6 * \text{factorial}(5)$

main : factorial(6)



# Stack tracing

factorial(5) :  $5 * 24 = 120$

factorial(6) :  $6 * \text{factorial}(5)$

main : factorial(6)

# Stack tracing


factorial(6) :  $6 * 120 = 720$

main : factorial(6)

# Stack tracing

main : 720

# Stack tracing

Program Execution  
Finishes 

## Problem : Print a flipped triangle

```
void triangle(int n){  
    if (n==0) return;  
    for(int i = 0;i<n;i++)  
        {cout<<"*";}  
    cout<<endl;  
    triangle(n-1);  
}  
  
int main(){triangle(5);}
```

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

# Where's the subproblem 🤔

Say we want to print a flipped triangle of 5 rows

We can easily do it using two nested for loops

**OR**

A triangle with 5 rows is just a single row of 5 stars and a triangle of 4 rows

A triangle with 4 rows is just a single row of 4 stars and a triangle of 3 rows

And so on till we reach 0, it's impossible so we terminate ( base case)

# Stack tracing

main : triangle(5)

# Stack tracing

\*\*\*\*\*

triangle(5) :  
Print 5 stars , call triangle(4)

main : triangle(5)



# Stack tracing

\*\*\*\*\*

\*\*\*\*\*

triangle(4) :  
Print 4 stars , call triangle(3)

triangle(5) :  
Print 5 stars , call triangle(4)

main : triangle(5)

# Stack tracing

triangle(3) :  
Print 3 stars , call triangle(2)

triangle(4) :  
Print 4 stars , call triangle(3)

triangle(5) :  
Print 5 stars , call triangle(4)

main : triangle(5)

\*\*\*\*\*

\*\*\*\*\*

\*\*\*

# Stack tracing

triangle(3) :  
Print 3 stars , call triangle(2)

triangle(4) :  
Print 4 stars , call triangle(3)

triangle(5) :  
Print 5 stars , call triangle(4)

main : triangle(5)

triangle(2) :  
Print 2 stars , call triangle(1)

\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*\*

# Stack tracing

triangle(3) :  
Print 3 stars , call triangle(2)

triangle(4) :  
Print 4 stars , call triangle(3)

triangle(5) :  
Print 5 stars , call triangle(4)

main : triangle(5)

triangle(1) :  
Print 1 star , call triangle(0)

triangle(2) :  
Print 2 stars , call triangle(1)

\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*\*

\*

# Stack tracing

triangle(3) :  
Print 3 stars , call triangle(2)

triangle(4) :  
Print 4 stars , call triangle(3)

triangle(5) :  
Print 5 stars , call triangle(4)

main : triangle(5)

triangle(0) :  
return

triangle(1) :  
Print 1 star , call triangle(0)

triangle(2) :  
Print 2 stars , call triangle(1)

\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*\*

\*

Can you guess the output 🤔

```
void triangle(int n){  
    if (n==0) return;  
    triangle(n-1);  
    for(int i = 0;i<n;i++)  
        {cout<<"*";}   
    cout<<endl;  
}  
  
int main(){triangle(5);}
```

# Exactly , a Regular Triangle 100

```
void triangle(int n){  
    if (n==0) return;  
    triangle(n-1);  
    for(int i = 0;i<n;i++)  
        {cout<<"*";}  
    cout<<endl;  
}  
  
int main(){triangle(5);}
```

```
*  
**  
***  
****  
*****
```

Problem : A coin tossed n times, count the ways that we have at least one head

```
int count_ways(int time, int heads){  
    if (time == n) return heads >= 1;  
    else return count_ways(time+1, heads+1) + count_ways(time+1, heads);  
}
```



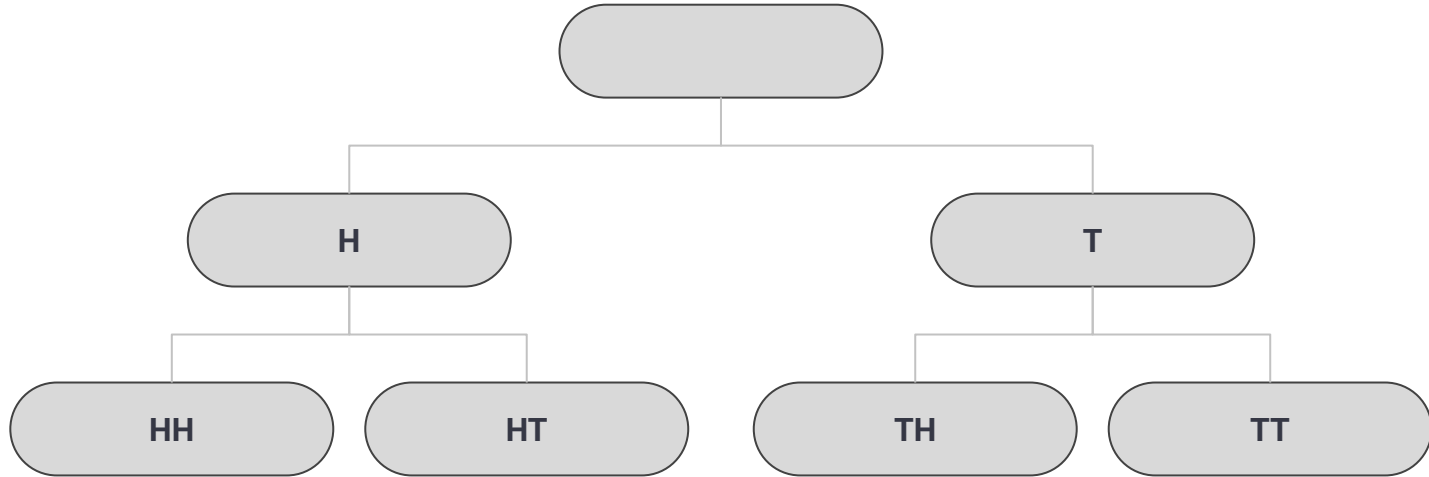
# Where's the subproblem 🤔

Say we tossed the coin twice, so we have two possible outcomes per toss, heads or tails.

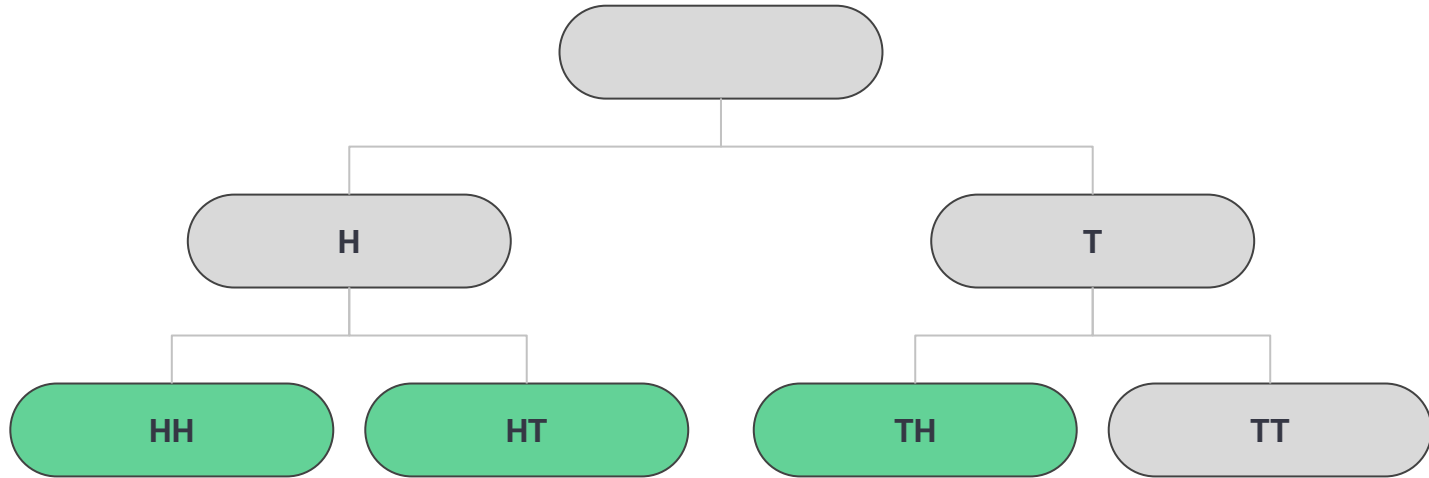
In other words, either the number of heads increase by one or remains constant.

1. Count the number of heads in every outcome
2. If we have at least one head, then we will count this outcome as a way of getting at least one head
3. The answer to the problem is the sum of all of those counted outcomes.

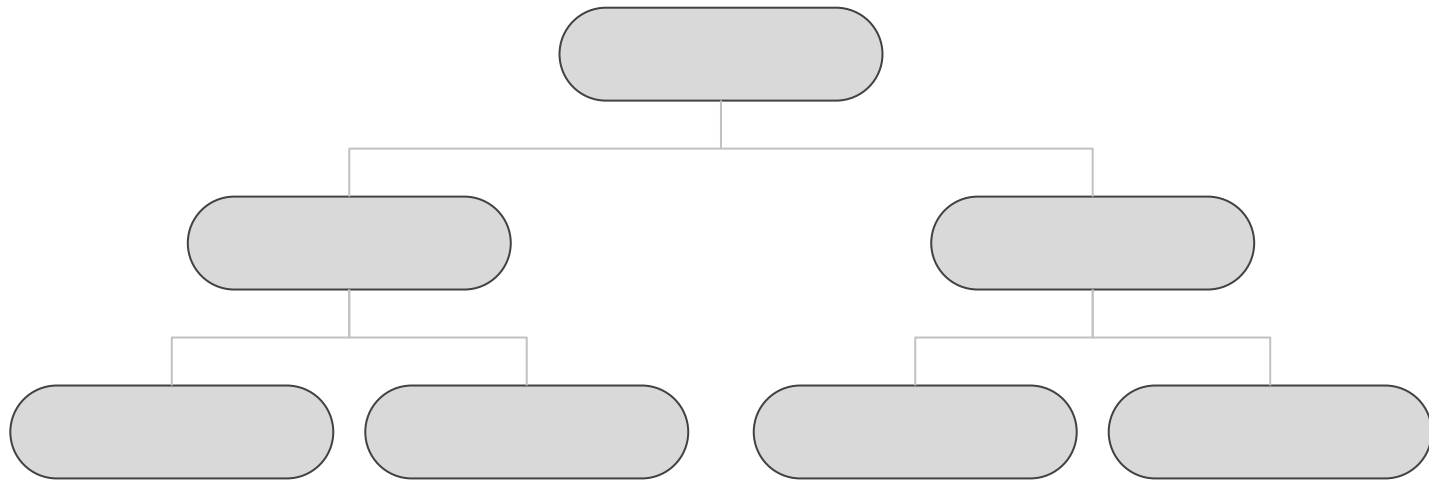
Lets Trace 



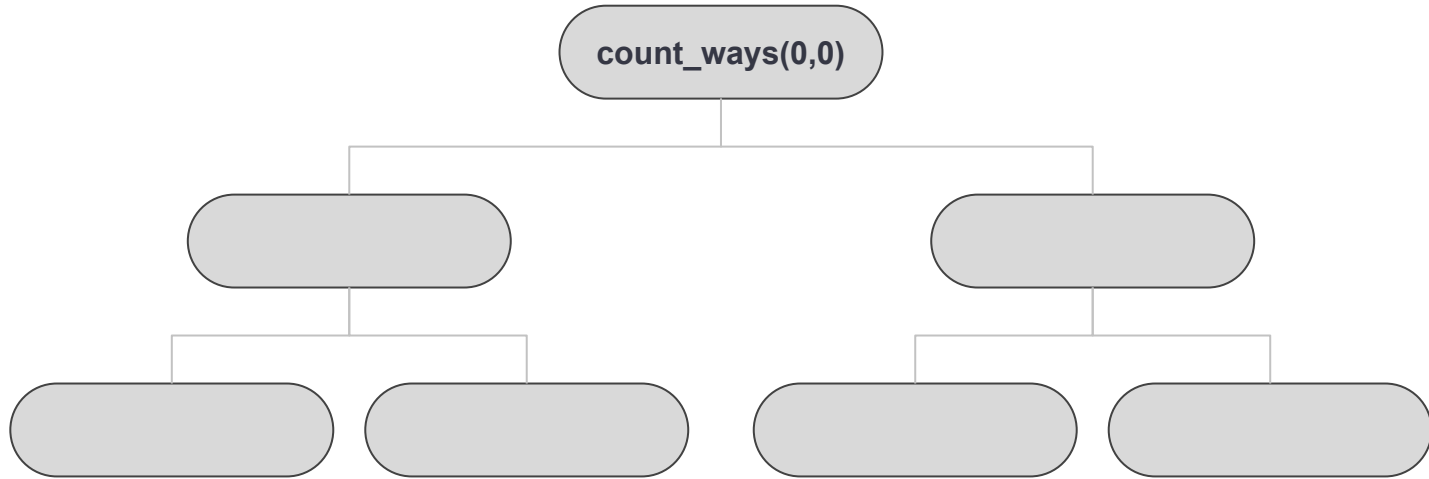
Lets Trace 



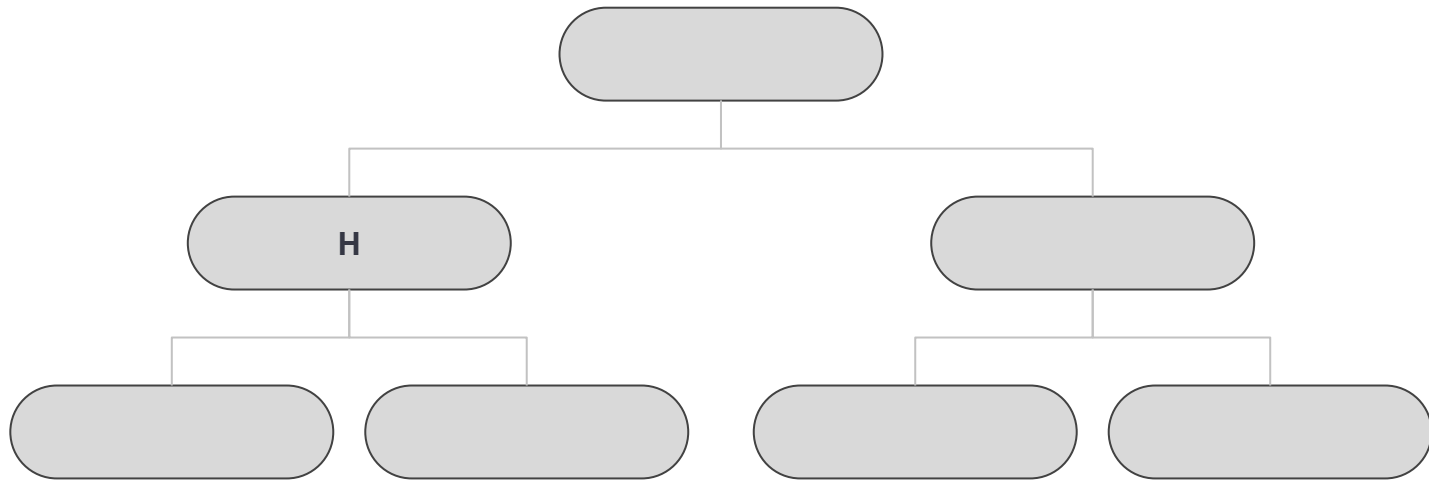
Lets Trace 



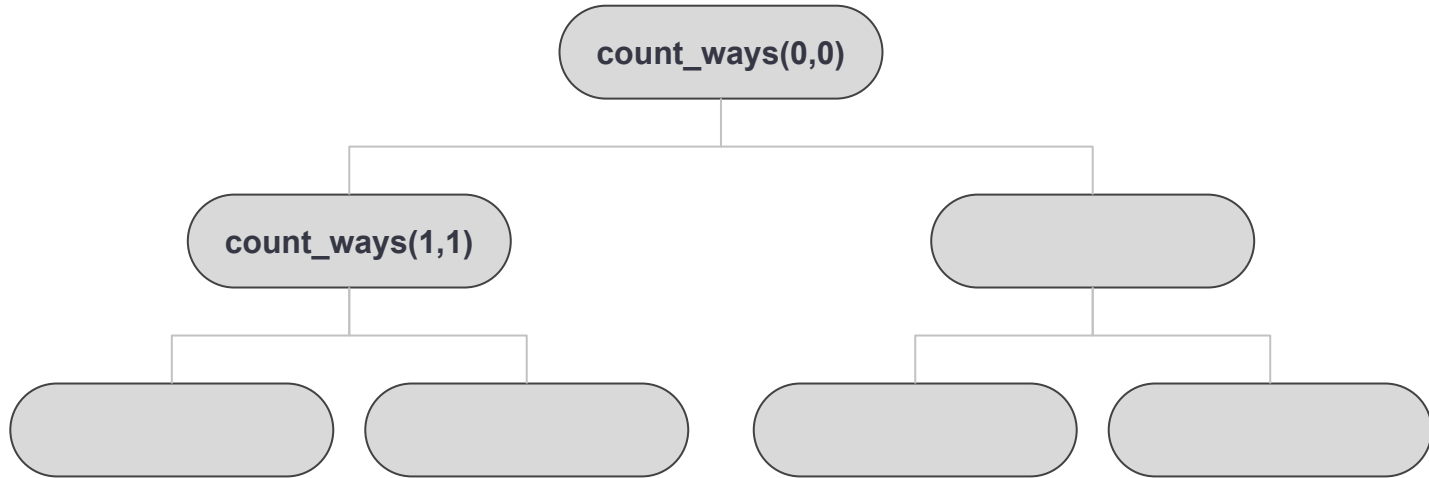
Lets Trace 



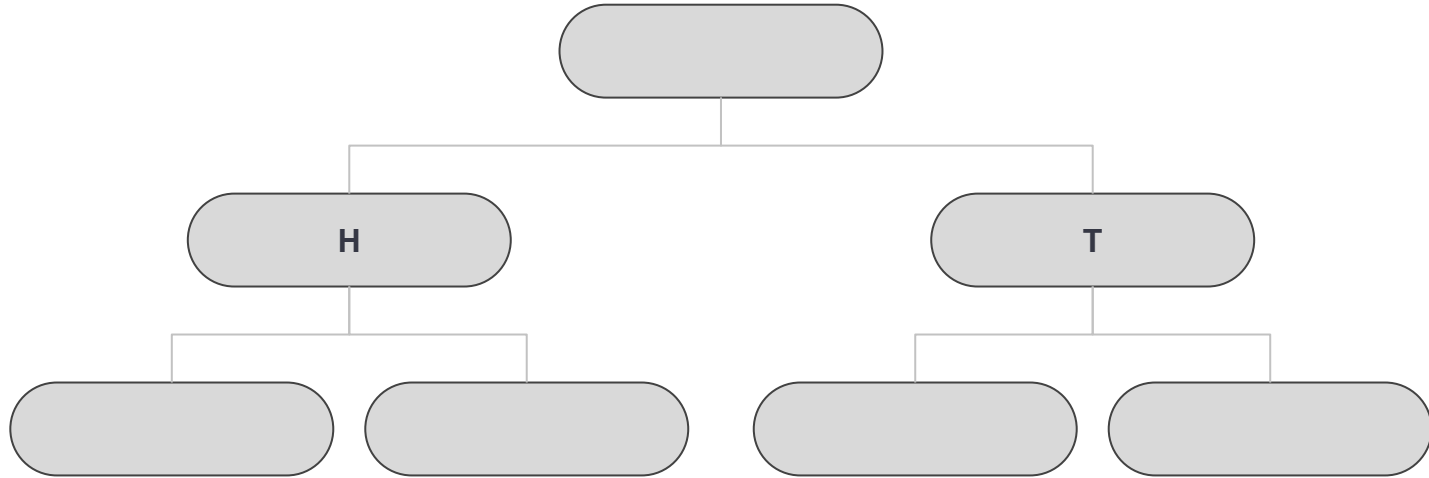
Lets Trace 



Lets Trace 

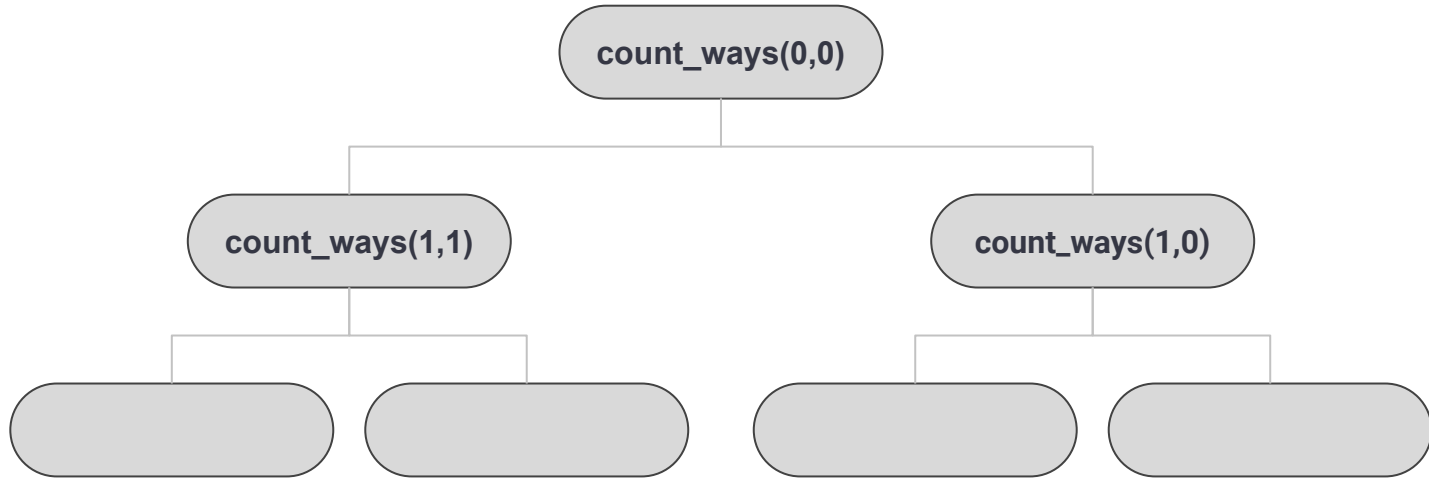


Lets Trace 

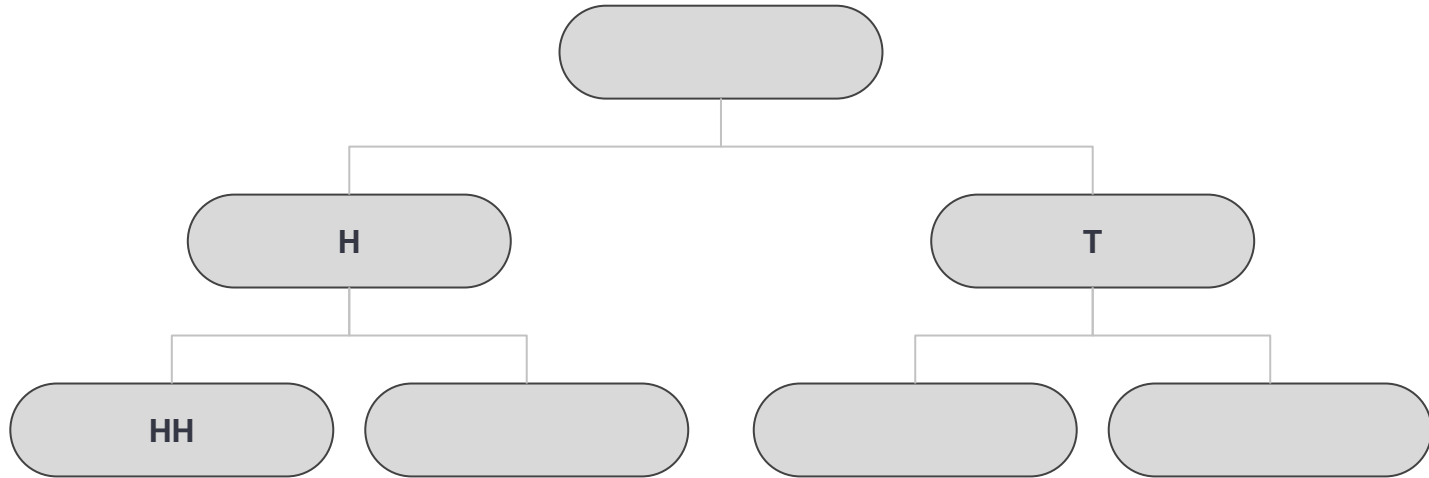




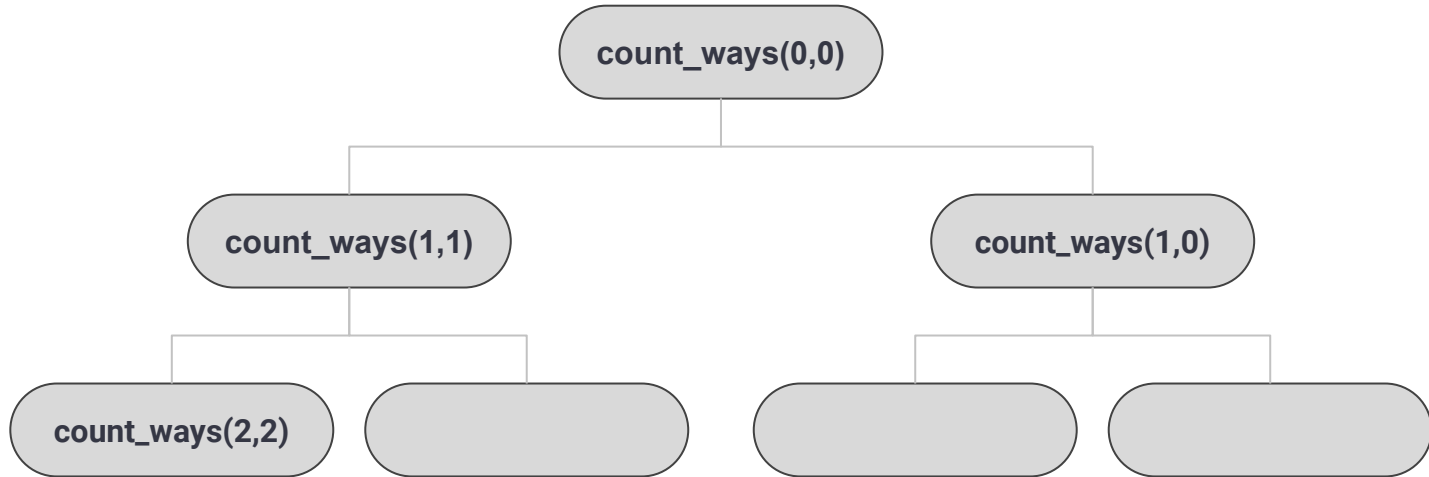
Lets Trace 



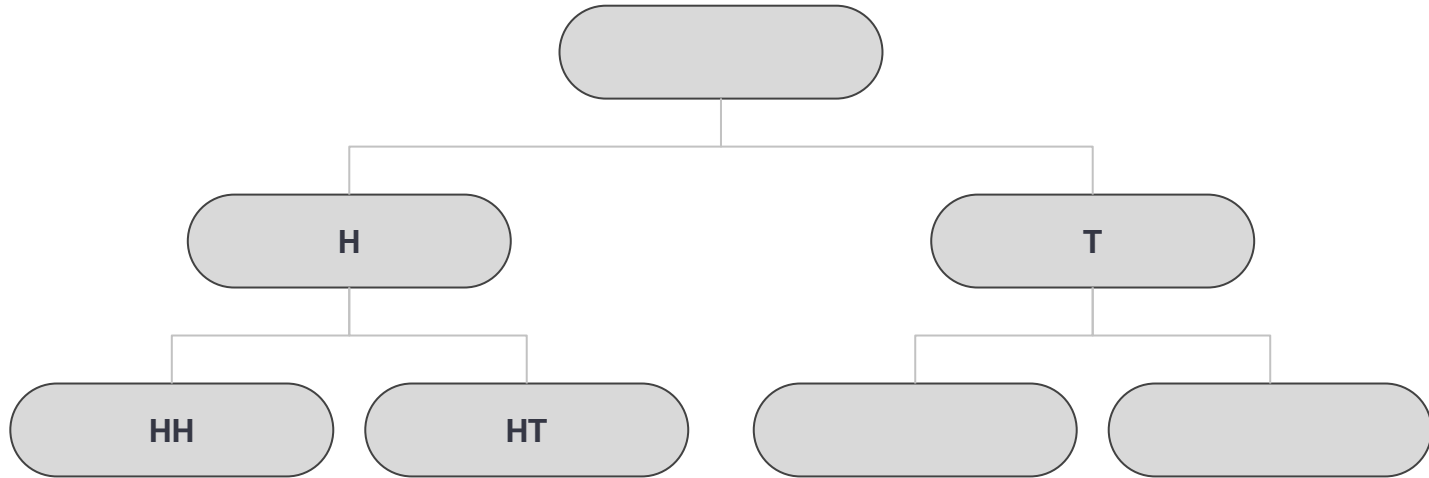
Lets Trace 



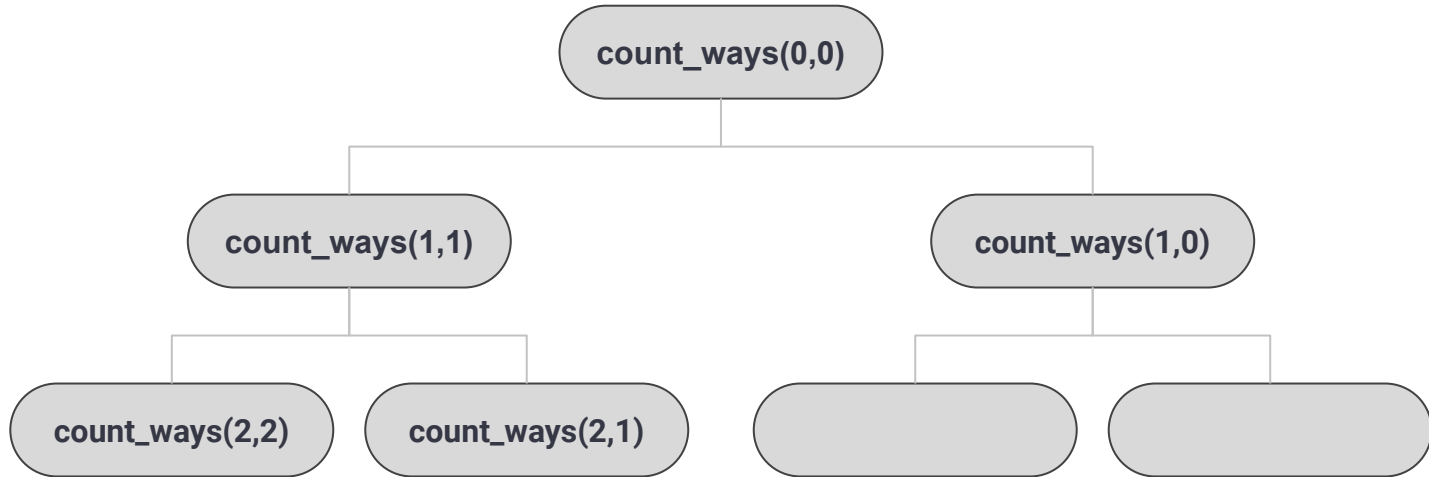
# Lets Trace



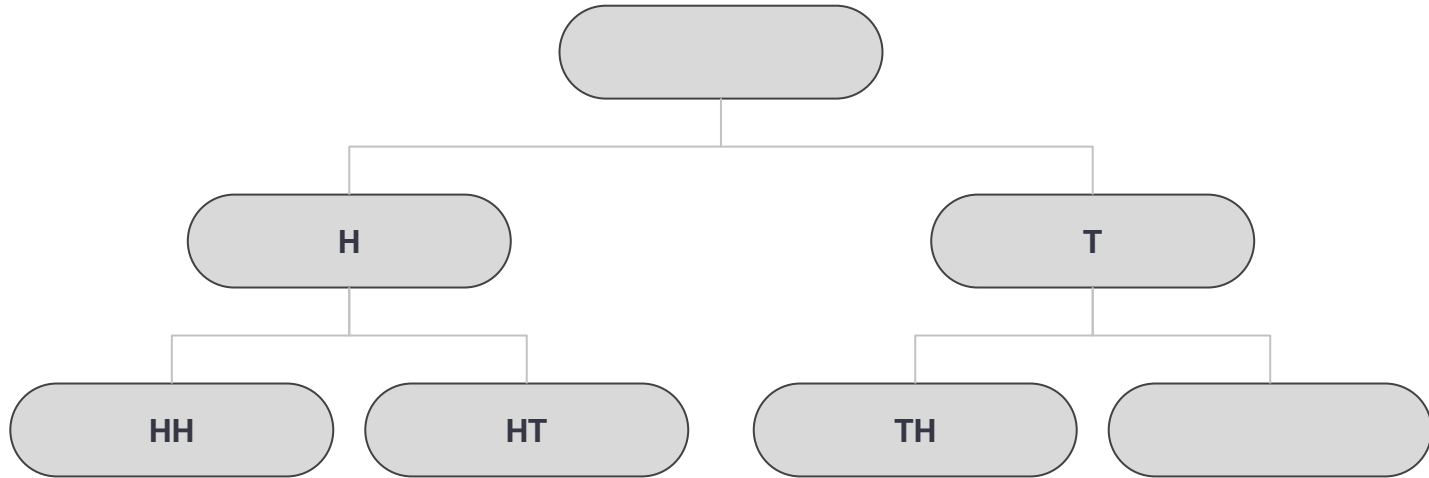
Lets Trace 



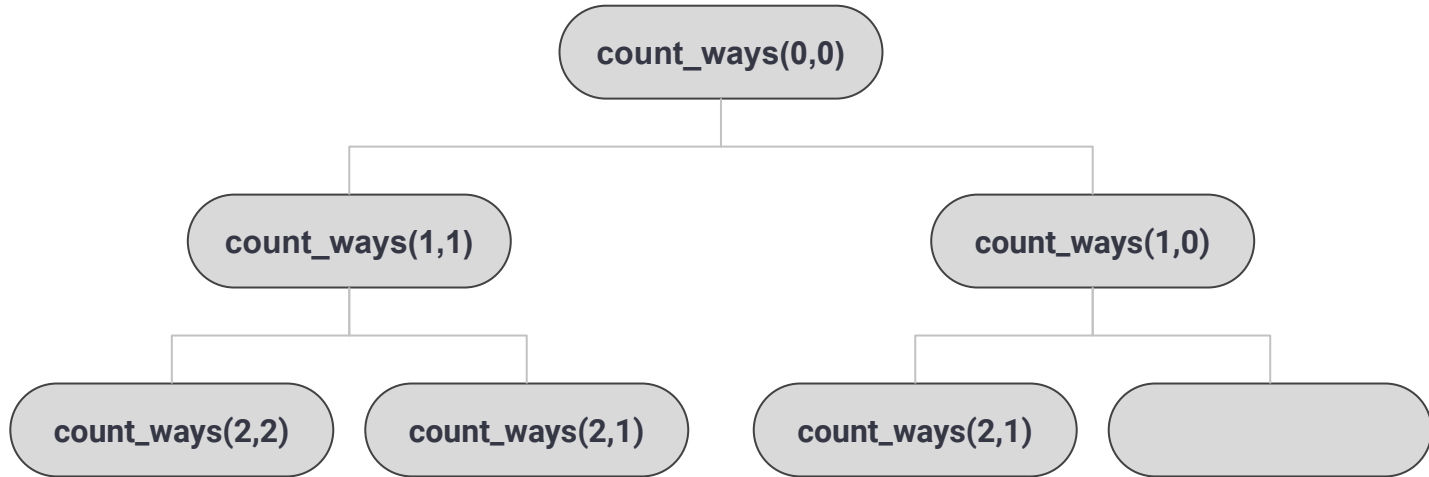
# Lets Trace



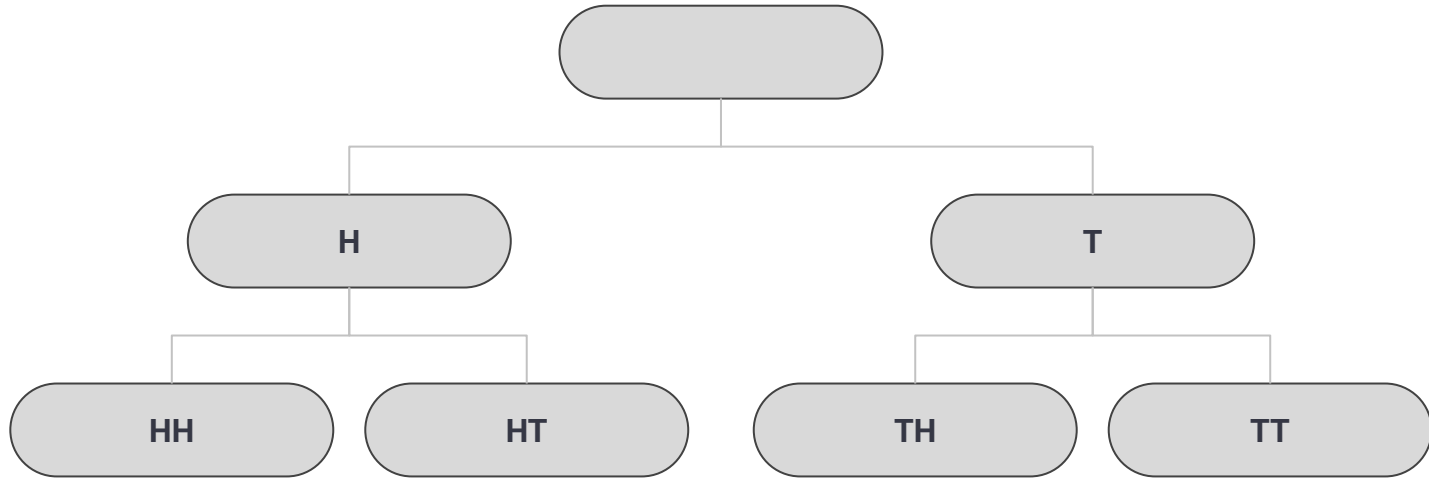
Lets Trace 



# Lets Trace

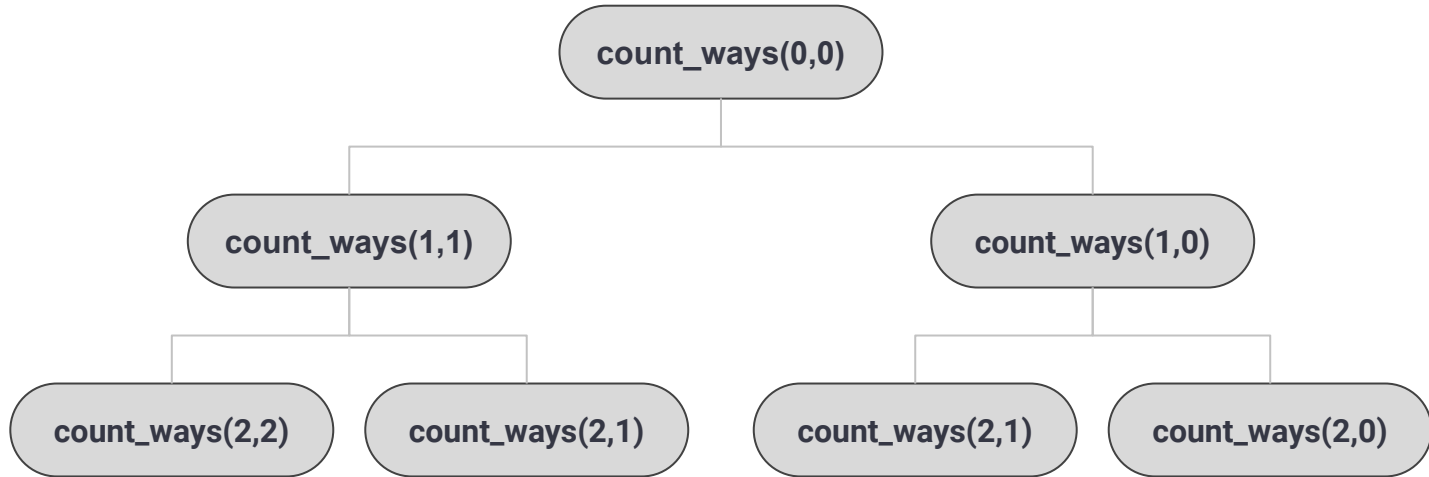


Lets Trace 

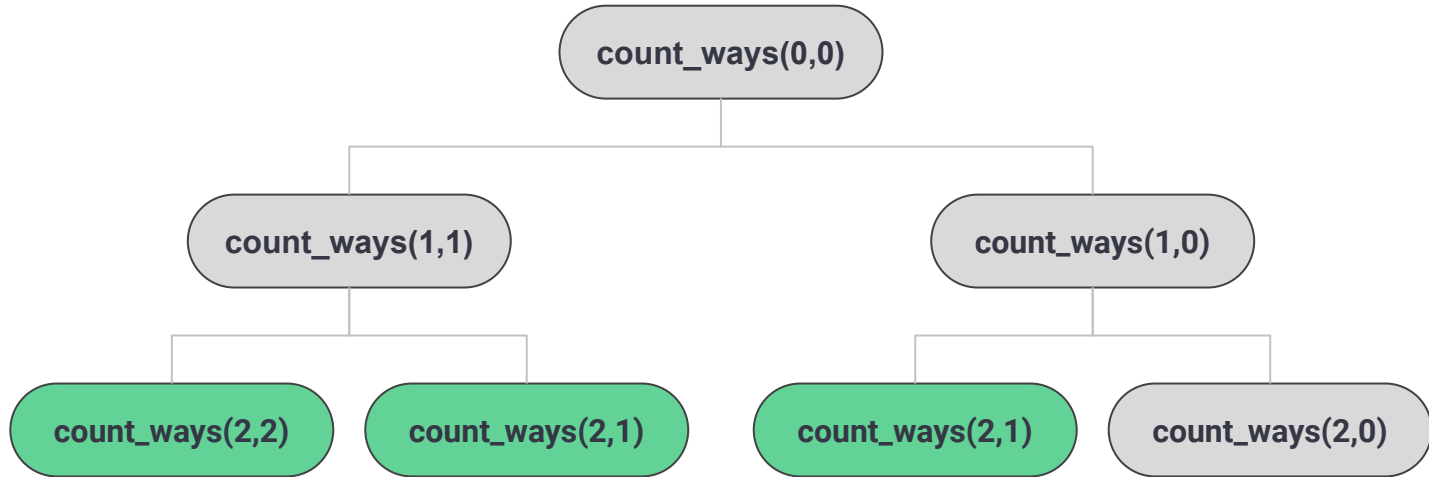




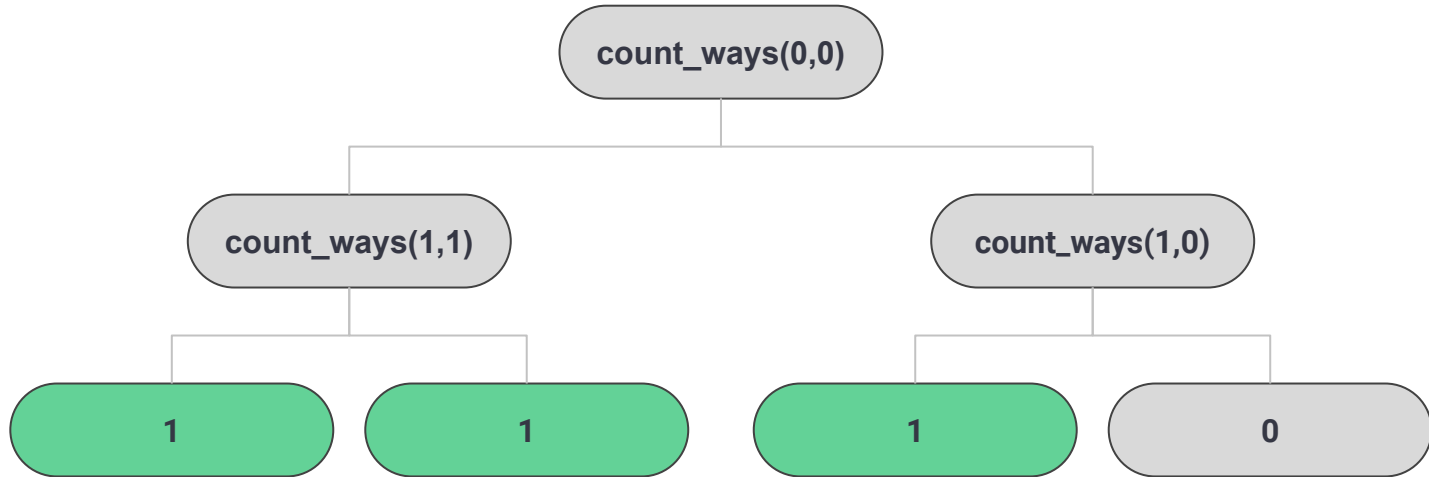
# Lets Trace



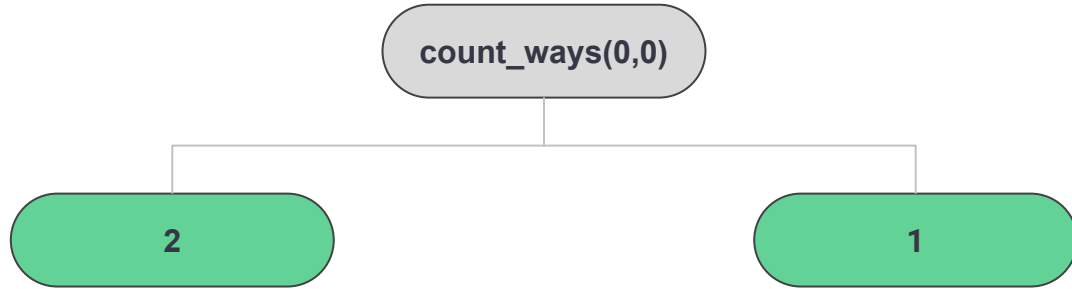
# Lets Trace



Lets Trace 



Lets Trace 



Lets Trace 

3


# Notes

- The order of the recursive call can change the logic of the function, i.e. triangle vs flipped triangle.
- Time complexity of a recursive function can be calculated by the following method
  - Time complexity of a single call \* number of function calls
- The time complexity of a recursive function varies from  $O(n)$  to  $O(C^n)$  where  $C$  is a numeric constant

# Tracing

- Trace every case on papers
  - don't trace all recursive cases ❌
  - Instead , trace one recursive case and when it will reach the base case ,  
Then calculate the answer mentally ✓
- If your program crashes after writing a recursive function , maybe your base case is impossible
  - You get stack overflow (memory / time limit exceeded ) 🚫
  - When tracing, try to find the base case , remember there's always a base case for any recursive function

# Tracing

- You can use visualizers like
  - VisuAlgo
  - Recursion Tree Visualizer
- Use Debuggers 



# Practice Problems

- Print Digits Using Recursion
- Log2
- Max Number
- Number Of Ways
- The Maximum-Path Sum
- Knapsack
- Apple Division

# Thank You

---