Recursion Definition

Wednesday, December 25, 2024 7:16 AM

★ when a function calls itself → until a specific condition is met

Example 1:

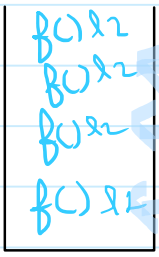
```
main(){
  f()
}
```

```
void f(){
  print("hi")
  f()
}
```

```
void f(){
  print("hi")
  f()
}
```

```
void f(){
  print("hi")
  f()
}
```

infinite loop



=> Stack overflow, segmentation fault

these functions calls are waiting in memory

Example 2: (base condition)

count = 0 1 2

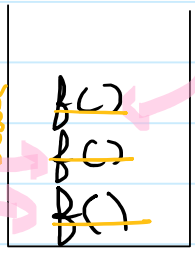
```
f(){
  if (count == 2)
    return
  print(count)
  count++
  f()
}
```

```
f(){
  if (count == 2)
    return
  print(count)
  count++(1)
  f()
}
```

```
f(){
  if (count == 2)
    return
  print(count)
  count++(2)
  f()
}
```

```
f(){
  if (count == 2) yes
    return
  print(count)
  count++
  f()
}
```

```
main(){
  f()
}
```

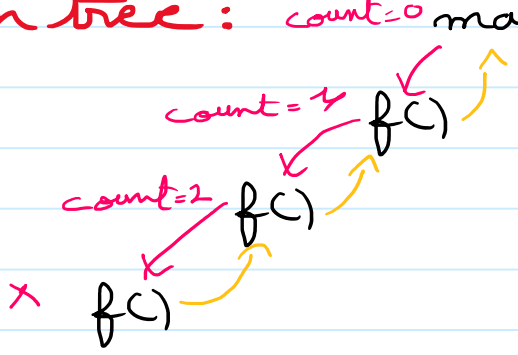


print order: 0, 1, 2

■ : the function waits in stack (memory) and calls next function

■ : function b, tells function a that it's over.

Recursion tree:

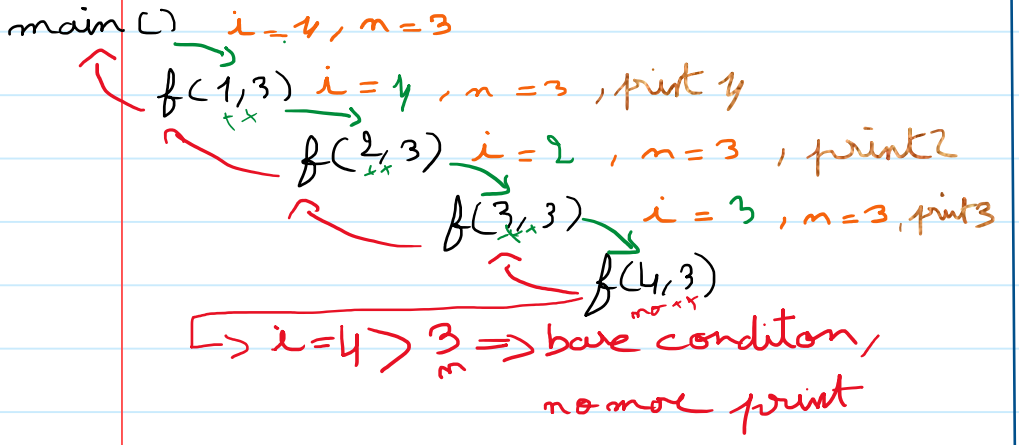


Loops

Sunday, December 29, 2024 10:38 AM

1) print name n times:

Recursion tree:



Solution:

```
void f(i,n){
  if(i>n)
    return
  print('gacie')
  f(i+1,n)
}
main(){
  f(1,3)
}
```

number of terms: last term - first term + 1 = O(n - i + 1) = O(n - 1 + 1) = O(n)

f(3,3)
f(2,3)
f(1,3)
} n times

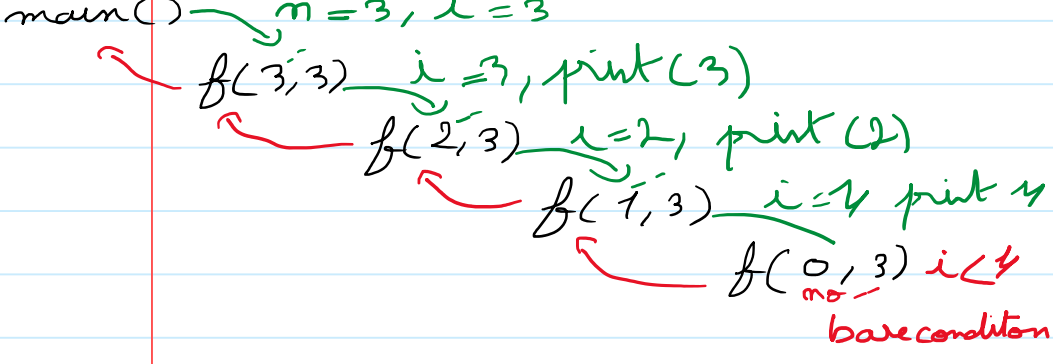
TC = O(n) / S.C = O(n) = stack space

2) print linearly from 1 -> N

same problem as before, just replace print(gacie) by print(i)

3) print from N -> 1

Recursion tree:



Solution

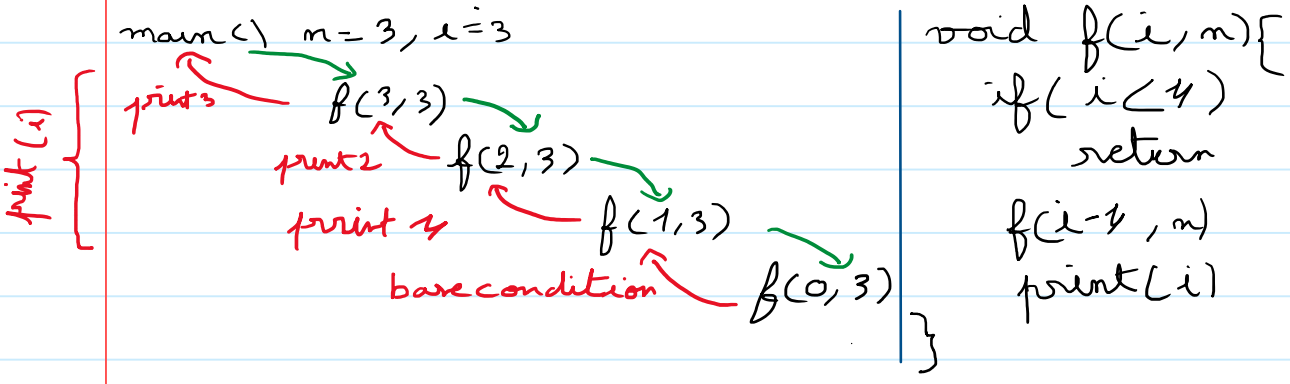
```
void f(i,n){
  if(i<1)
    return
  print(i)
  f(i-1,n)
}
```

Backtracking

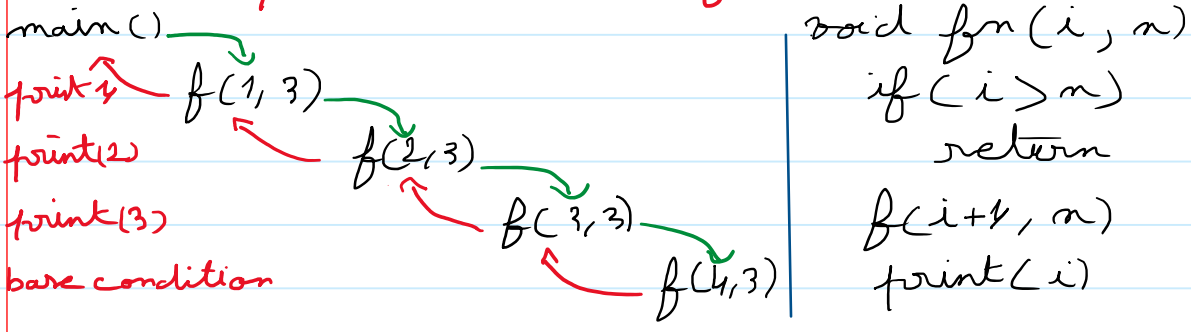
Sunday, December 29, 2024 1:30 PM

4) print linearly from 1 to n linearly

i-1 is not allowed, (backtrack)



4) print from N to 1 by backtrack



Sum of first n natural numbers

Monday, December 30, 2024 6:49 AM

Problem statement :

given a number n, find out the sum of first n numbers.

Examples :

N=5 ; 1+2+3+4+5 = 15

N=6 ; 1+2+3+4+5+6 = 21

Problem analysis :

Solution1 : we loop from 1 to n and sum the i

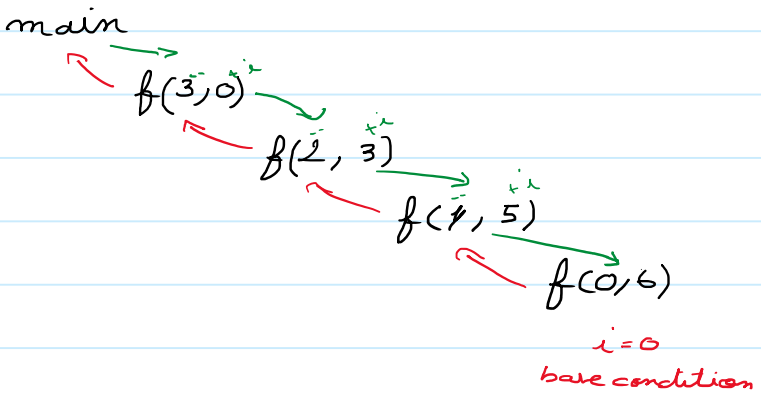
Solution2 : usage of a formula : $\frac{N(N+1)}{2}$

Solution 3 : Recursion

Parameterized way :

```
int TailRecursiveWay(int i, int sum)
{
    if (i < 1) {
        return sum;
    }
    sum = sum + i;
    i = i - 1;
    return TailRecursiveWay(i, sum);
}

int main() {
    cout << TailRecursiveWay(3, 0);
    return 0;
}
```

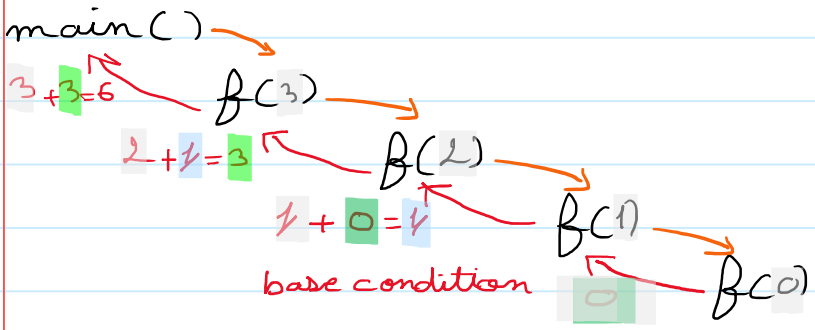


Functional way :

sum(3) = 3 + sum(2), so sum(2) = 2 + sum(1),

sum(1) = 1 + sum(0), and sum(0) = 0

Sum(n) = n + sum(n-1)




```
int NonTailRecursiveWay(const int i)
{
    if (i < 1) {
        return 0;
    }
    return i +
NonTailRecursiveWay(i - 1);
}
int main() {
    cout << NonTailRecursiveWay(3);
    return 0;
}
```

Non-Tail vs Tail Recursive

Tuesday, December 31, 2024 7:34 AM

Here's a table comparing **Non-Tail Recursive** and **Tail Recursive** functions:

Feature	Non-Tail Recursive (functional way)	Tail Recursive (parameterized way)
Definition	Recursive call is not the last operation in the function.	Recursive call is the last operation in the function.
Intermediate Steps	Results are computed after the recursive call returns.	Results are passed as parameters (e.g., accumulators).
Stack Usage	Each recursive call adds a new frame to the call stack.	The current stack frame is reused (if optimized by the language).
Efficiency	Less efficient; may lead to stack overflow for deep recursion.	More efficient with tail-call optimization (TCO).
Parameterization	Often doesn't require extra parameters.	Typically requires additional parameters (e.g., accumulators).
Memory Overhead	Higher memory usage due to stack frames.	Lower memory usage with TCO.
Readability	Often simpler and closer to the mathematical definition.	Slightly more complex due to extra parameters.
Example: Factorial	factorial(n) = n * factorial(n-1)	factorial(n, acc) = factorial(n-1, acc * n)
Error Risk	Higher risk of stack overflow in non-optimized languages.	Safer for large inputs due to reduced stack usage.

same

I see it easier

wins

Example Code for Each

Non-Tail Recursive Factorial:

```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
```

Tail Recursive Factorial:

```
function factorial(n, acc = 1) {
  if (n === 0) return acc;
  return factorial(n - 1, acc * n);
}
```

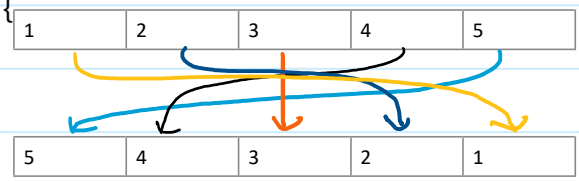
From <https://chatgpt.com/c/67738e12-46c8-800c-9122-af6030b7527e>

Reverse a given Array

Thursday, January 23, 2025 6:43 AM

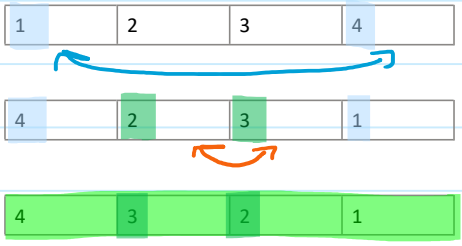
```
int* subArrayMethod(const int size, const int* inputArray) {
    int* resultArray = new int[size];

    for (int i = size -1 ; i>=0; i--) {
        resultArray[size - i - 1] = inputArray[i];
    }
    return resultArray;
}
```



Tc: O(n) S.C: O(n) (new array)

```
const int* optimisedMethod(const int size, int* inputArray) {
    for (int i = 0; i < size / 2; i++) {
        ranges::swap(inputArray[i], inputArray[size - i - 1]);
    }
    return inputArray;
}
```



Tc: O(n/2) S.C = O(1)
(no new data memory created)

```
const int* recursiveWay(int* inputArray, const int leftP, const int rightP) {
    if (rightP < leftP) {
        return inputArray;
    }
    ranges::swap(inputArray[leftP], inputArray[rightP]);
    return recursiveWay(inputArray, leftP+1, rightP - 1);
}
```

TC: O(n/2) SC = O(1)

O(n/2) = O(n)

↳ $\frac{n}{2}$ O(1) = O(n)

Recursive tree

input Arr = IA = {1, 2, 3, 4}

main() → f(IA, 0, 3) → swap(0, 1) → f(IA, 1, 2) → swap(1, 2) → f(IA, 2, 1) no swap

Lp > Rp

2 > 1

base condition