Introduction to Recursion (1)

What is Recursion?

A function calling itself (with different parameters) is called recursion.

For example: `f(n) = f(n-1) + f(n-2)` is an example of recursion.

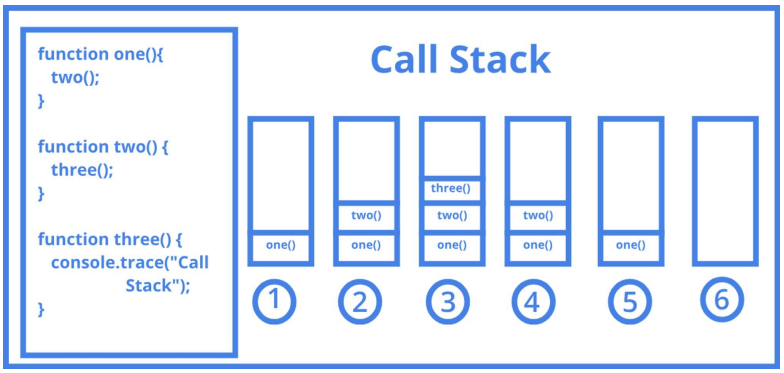
Visual Representation (Recursion Tree)

```
            fib(6)                                  Copy
           /      \
       fib(5)     fib(4)
       /     \       /     \
   fib(4)  fib(3) fib(3) fib(2)
   /     \
fib(3) fib(2)
```

Function call stack

A question that can come to your mind: How does this work? How does fib(3) know that the control has to return to fib(4)? How are the function states maintained?

We can do recursion because of the stack memory.



Structure of a recursive function

1. **State:** States are the function parameters that change during the recursion. For example, in the case of Fibonacci numbers `fib(n)` , n is the state as it changes during every recursive call.
2. **Base condition**: These are very important to terminate a recursion. Otherwise, the recursion would continue infinitely leading to stack size limit exceeded errors. A base condition is hardcoding the answer for some values of the state (generally smaller values) which we can calculate manually.
3. **Transition**: We can call it a subproblem, recursive call, or transition. It is basically how a state depends on other states to calculate its value. Or, calling the next states from the current state.

Complexity Analysis

```
                                fib(n)                                  -> 2^0
                            /           \
                        fib(n-1)          fib(n-2)                      -> 2^1
                       /      \           /       \
                  fib(n-2)    fib(n-3)  fib(n-3)   fib(n-4)            -> 2^2
                  .
                  .
                  .
fib(0)    fib(1)    fib(0)    fib(1)   fib(0)    fib(1) ...     -> 2^n
```

What is Recursion?

A function calling itself (with different parameters) is called recursion.

For example: `f(n) = f(n-1) + f(n-2)` is an example of recursion.

Visual Representation (Recursion Tree)

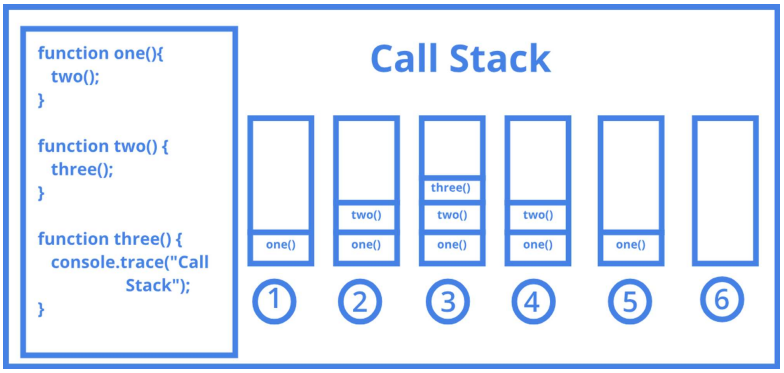
Copy

```
            fib(6)
           /      \
       fib(5)     fib(4)
       /     \       /     \
   fib(4)  fib(3) fib(3) fib(2)
   /     \
fib(3) fib(2)
```

Function call stack

A question that can come to your mind: How does this work? How does fib(3) know that the control has to return to fib(4)? How are the function states maintained?

We can do recursion because of the stack memory.



Structure of a recursive function

1. **State:** States are the function parameters that change during the recursion. For example, in the case of Fibonacci numbers `fib(n)` , n is the state as it changes during every recursive call.
2. **Base condition**: These are very important to terminate a recursion. Otherwise, the recursion would continue infinitely leading to stack size limit exceeded errors. A base condition is hardcoding the answer for some values of the state (generally smaller values) which we can calculate manually.
3. **Transition**: We can call it a subproblem, recursive call, or transition. It is basically how a state depends on other states to calculate its value. Or, calling the next states from the current state.

Complexity Analysis

```
                              fib(n)                                  -> 2^0
                          /           \
                      fib(n-1)          fib(n-2)                      -> 2^1
                     /      \           /       \
                 fib(n-2)   fib(n-3)  fib(n-3)   fib(n-4)            -> 2^2
                 .
                 .
                 .
fib(0)   fib(1)   fib(0)   fib(1)   fib(0)   fib(1) ...     -> 2^n
```

Nearby Squares (1)

Link: https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/nearby-squares-338a4a64/

Solution

What can be the states? For each element, we need to decide if it goes into array B or C.

So essentially, we need 3 things: the current element we are at, the sum of B achieved till now, and the sum of C achieved till now. So the function can be something like

```
f(i, x, y) = MIN(f(i+1, x + arr[I], y), f(i+1, x, y+arr[i]))
```

Copy

The base case will be when we are done with all elements, we need to report the absolute difference

```
f(n, x, y) = abs(x - y)
```

Copy

Code

```
int diff(int i, int x, int y, int arr[], int n) {
   if (i == n) return abs(x - y);

   return diff(i+1, x + arr[i], y, arr, n) + diff(i+1, x, y + arr
}
```

Copy

Can we optimize this further?

If you think about this, the term y is kind of irrelevant. Because we can always derive y at the end using `y = total sum - x`

So, instead, we can write it like this:

```
f(i, x) = MIN(f(i+1, x), f(i+1, x+arr[i])
```

Copy

and base case becomes: `f(n, x) = sum - x`

Code

```
int diff(int i, int x, int arr[], int n, int sum) {
   if (i == n) return abs(x - (sum - x));

   return diff(i+1, x + arr[i], arr, n, sum) + diff(i+1, x, arr, r
}
```

Copy

Link: https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/nearby-squares-338a4a64/

Solution(Not Optimal)

What can be the states? For each element, we need to decide if it goes into array B or C.

So essentially, we need 3 things: the current element we are at, the sum of B achieved till now, and the sum of C achieved till now. So the function can be something like

```
f(i, x, y) = MIN(f(i+1, x + arr[I], y), f(i+1, x, y+arr[i]))
```

Copy

The base case will be when we are done with all elements, we need to report the absolute difference

```
f(n, x, y) = abs(x*x - y*y)
```

Copy

Code

```
int diff(int i, int x, int y, int arr[], int n) {
   if (i == n) return abs(x*x - y*y);

   return diff(i+1, x + arr[i], y, arr, n) + diff(i+1, x, y + arr
}
```

Copy

Can we optimize this further?

If you think about this, the term y is kind of irrelevant. Because we can always derive y at the end using `y = total sum - x`

So, instead, we can write it like this:

```
f(i, x) = MIN(f(i+1, x), f(i+1, x+arr[i])
```

Copy

and base case becomes: `f(n, x) = sum - x`

Code

```
int diff(int i, int x, int arr[], int n, int sum) {
   if (i == n) return abs(x*x - (sum - x) * (sum - x));

   return diff(i+1, x + arr[i], arr, n, sum) + diff(i+1, x, arr, r
}
```

Copy

Solution (Optimal)

As A[i] ≤ 10^8, the above will not work. But we can see N≤20, so 2^N is a valid complexity. For each element, we can have 2 choices - either A or B. So all possible choice combinations are 2^N.

For each element, we need to maintain 2 choices: 0 means it goes to A and 1 means it goes to B. So we get an array like this: [0, 1, 1, 0]

Now when we are done with all elements, we can iterate over this list and find the sum for A and B and then the difference of their squares.

https://p.ip.fi/vUmt

Divide and Conquer (1)

Divide and Conquer is a group of algorithms that rely strongly on recursions. It follows a 2 step idea:

1. Divide a problem into smaller subproblems
2. Combine the solutions of the subproblems to solve the bigger problem

Let's understand this with the **merge sort** algorithm 🙂

Q - You are given an array A of size N. Sort the array

Let's understand how D&C fits in. We need to sort the entire array of N elements. What we can do is, break the array into 2 smaller arrays and if we can sort these 2 smaller arrays, we can try to find a way to combine them to form a single sorted array.

Let's assume we have a `combine` method that takes in 2 sorted arrays and combines them into a single sorted array.

```
combine(arr1, arr2) -> returns arr which is a sorted array of combined arr1 and arr2
```

Now essentially, what we do is

```
1. sort(arr) where arr is of length n
2. Break arr into 2 subarrays arr2 and arr3 of size n/2 each.
3. sort(arr2) and sort(arr3)
4. combine(arr2, arr3) -> Outputs sorted arr
```

So, we have broken down the problem (i.e. sorting an array of length n) into 2 subproblems (sorting an array of length n/2) and then merged these sorted arrays into a single sorted array.

Base case:

When we are left with a single element, there is nothing to divide or sort. So just return in that case.

```
void sort(arr, beg, end) {
     // ---------- BASE CASE ------------
     if (beg == end) {
        // only single element. No need to sort it or combine it
        return;
     }
   // Sorts array from index beg to end
   // -------- DIVIDE STAGE ---------------
   int mid = (beg + end) / 2;
   // ------------ CONQUER STAGE -----------------
   sort(arr, beg, mid); // Sort the first half
   sort(arr, mid+1, end); // Sort the second half
   // ------------ MERGE STAGE -----------------
   combine(arr, beg, mid, end); // Combine the 2 sorted arrays into single sorted ar
}

sort(arr, 0, n-1); // Sort the entire array
```

Read how we can combine efficiently here: https://www.geeksforgeeks.org/merge-two-sorted-arrays/

Q - Let's solve this problem using Divide and Conquer: https://leetcode.com/problems/number-of-1-bits/description/

Divide and Conquer is a group of algorithms that rely strongly on recursions. It follows a 2 step idea:

1. Divide a problem into smaller subproblems
2. Combine the solutions of the subproblems to solve the bigger problem

Let's understand this with the **merge sort** algorithm 🙄

Q - You are given an array A of size N. Sort the array

Let's understand how D&C fits in. We need to sort the entire array of N elements. What we can do is, break the array into 2 smaller arrays and if we can sort these 2 smaller arrays, we can try to find a way to combine them to form a single sorted array.

Let's assume we have a `combine` method that takes in 2 sorted arrays and combines them into a single sorted array.

```
combine(arr1, arr2) -> returns arr which is a sorted array of combined arr1 and arr2
```

Now essentially, what we do is

```
1. sort(arr) where arr is of length n
2. Break arr into 2 subarrays arr2 and arr3 of size n/2 each.
3. sort(arr2) and sort(arr3)
4. combine(arr2, arr3) -> Outputs sorted arr
```

So, we have broken down the problem (i.e. sorting an array of length n) into 2 subproblems (sorting an array of length n/2) and then merged these sorted arrays into a single sorted array.

Base case:

When we are left with a single element, there is nothing to divide or sort. So just return in that case.

```
void sort(arr, beg, end) {
     // ---------- BASE CASE ------------
     if (beg == end) {
        // only single element. No need to sort it or combine it
        return;
     }
   // Sorts array from index beg to end
   // -------- DIVIDE STAGE ---------------
   int mid = (beg + end) / 2;
   // ------------ CONQUER STAGE -----------------
   sort(arr, beg, mid); // Sort the first half
   sort(arr, mid+1, end); // Sort the second half
   // ------------ MERGE STAGE -----------------
   combine(arr, beg, mid, end); // Combine the 2 sorted arrays into single sorted ar
}

sort(arr, 0, n-1); // Sort the entire array
```

Read how we can combine efficiently here: https://www.geeksforgeeks.org/merge-two-sorted-arrays/

Q - Let's solve this problem using Divide and Conquer: https://leetcode.com/problems/number-of-1-bits/description/

How to think recursively? (1)

This is the most challenging part. If we are given a recursive relation, it is easy to translate it into code. However, most of the time, we would have to derive the recurrence relations ourselves.

Generally, I follow a 4 step approach. Let's take an example to understand this

Derive the recurrence relation of nCr (How many ways to choose r items out of n items?)

1. **Identify the states and define the function:** First, identify what is changing with every move. It's pretty obvious, it's either the number of elements remaining or the number of elements we are still left to choose i.e. N and R. At any point of time, we are left with N objects and we need to choose R out of them.
   f(N, R) = Number of ways we can choose R objects out of N
   Also, our final answer should be derivable from this function. In our case, the answer is f(n, r) right?

2. **Assume** that for some value of (N, R) = (x, y), f(x, y) is not calculated yet. However, every other value of f(N, R) where (N, R) ≠ (x, y), f(N, R) is already computed. So out task is to compute f(x, y) using neighboring values.

3. **Calculate f(x, y)**: What is f(x, y)? It is number of ways we can select y objects out of x objects. So currently, we have x objects and we want to choose y objects out of them. Just think of the `xth` object right now. What can we do with it? Either take it or not!
   If we take it, we are left with x-1 objects and we need to choose y-1 objects as we have taken the `xth` object. The number of ways becomes f(x-1, y-1).
   If we don't take it, we are left with x-1 objects, but we still need to choose y objects out of them. The number of ways we can do this is f(x-1, y).
   So overall, the total number of ways is:

```
f(x, y) = f(x-1, y-1) + f(x-1, y)
```
Copy

1. **Calculate the base cases:**
   f(0, y) = 0, f(x, 0) = 1. Think why.

And we are done 🙂

It's very easy to write the code.

```
int combinations(int x, int y) {
   if (x == 0) return 0;
   if (y == 0) return 1;

   return combinations(x-1, y-1) + combinations(x-1, y);
}
```
Copy

Backtracking (1)

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.

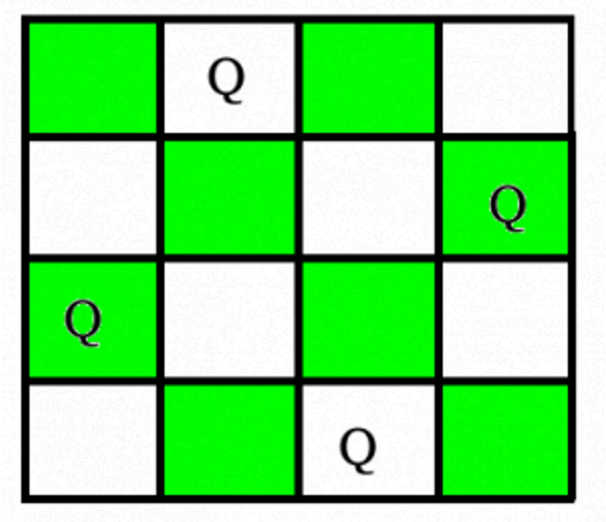
Generally, when we are doing recursion, we are kind of doing a backtracking as, from a point if there are 2 possibilities, first we explore one of them, once done we explore the other one.

But backtracking is generally reserved for use cases where you are undoing some changes manually. For example, let's say in a sudoku solver, you are in some cell currently, let's call it `i,j`

Now, let's say the cell (i, j) can take values v1, v2 and v3. First you choose v1 and update the board using: `board[i][j] = v1` . Then you recursively call solve for the next cell:

`solve(i+1, j)` . Once the recursion for `solve(i+1, j)` completes, we undo the change we had made i.e `board[i][j] = v1` and replace it with the next value v2 i.e.: `board[i][j] = v2` and call `solve` recursively again for the next cell. We continue like this. Once we are done with all possibilities of `(i, j)` , we backtrack by setting `board[i][j] = -1` or some initial value that was present. This appears, as if we never visited cell `(i, j)` and thus the name backtracking.

Let's solve the N Queen problem with backtracking.



You have a N * N board, and you need to place N queens such that no queen is attacking each other

We will use backtracking to solve this problem. First let's identify the states we would need

1. row: The row we are processing right now. We will go in a row-wise fashion, as we can have at most 1 queen per row. So for the row, we will try determining the column(s) where we can place the queen
2. board: The current board condition. We already have placed some queens in the previous rows, the board maintains their locations. Instead of the board, we can also maintain a list of columns where we placed the queens.

▼ Code

```
void solve(int row, int ** board, int n) {                    Copy
   if (row == n) {
      // We have successfully placed n queens. Print the board
      printBoard(board, n);
```

```
            return;
        }

        // Determine where we can place the queen in current row
        for(int col = 0; col<n; col++) {
            if (!isAttacking(row, col, board, n) {
                // isAttacking tells us that if any queen in the current b
                // is attacking the cell row, col
                board[row][col] = 1; // Place the queen
                solve(row+1, board, n); // Recursively solve it for next r
                board[row][col] = 0; // Backtrack / Undo the changes
            }
        }
    }
```

Problem: Staircase (1)

You have N stairs in front of you. At every step, you can either climb up by 1, 2 or 3 stairs. How many ways are there to climb to the top?

▾ Solution

Let's follow the 3 step approach for solving this:

1. Identify the function and states:

```
f(i) = Number of ways to climb to the top if we are at ith stair
```

Copy

Then, f(0) is the answer to the problem

1. Assume, f(i) is calculated for every i except i = k.
2. Try calculating f(k)
   We are at kth stair, what can we do. Either we can go to k+1 or k+2 or k+3.
   Now f(k+1) → No of ways to climb to top from (k+1)th stair
   Similarly, we have f(k+2) and f(k+3).
   As per step 2, these are calculated already, so what will be f(k)?

1. Either we can go to k+1 and climb to top in f(k+1) ways
2. Or we can go to k+2 and climb up in f(k+2) ways
3. Or, go to k+3 and climb up in f(k+3) ways

So total ways: `f(k+1) + f(k+2) + f(k+3)`