Recursion

Wednesday, 10 November 2021 9:07 PM

↳ Function calling itself.

a++;
++a;

```
void solve(n){                    // Initialization ✓
  if(n<=0) return;                // Base/Termination Condition ✓
  print(n);
  solve(n-1);                     // Recursive call ✓
}
```

o/p → 3 2 1 (n=3)

solve(n--); → infinite 3 3 3 - - -

solve(--n); → 3 2 1

```
void solve(n){
  if(n<=0) return;
  solve(n-1);
  print(n);
}
```

o/p → 1 2 3

✓ state Space Tree (SST)

Recursive Stack

solve(-7); ✓

SC = O(n)

max size of recursion stack at any point

Height of SST

```
int fib(n){
  if(n<2) return n;
  return fib(n-1) + fib(n-2);
}
```

SC = O(N)

TC = O(1.618^N) ✓

SC = O(N)

1+2+4+8+ - - - 2^N

= 1(2^(N+1) − 1) / (2 − 1) = 2^(N+1) − 1

state

state transition

Tail Recursion → If the recursive call is the last statement to be executed before function ends.

```
void solve(n);
if(n<=0) return;
print(n);
solve(n-1);
```

SC = O(1)

o/p → 3 2 1

```
int solve(n){
  if(n<=0) return 0;
  return n + solve(n-1);
}
```

o/p → 3 + (2 + (1 + 0)) = 6

Not a Tail recursion because '+' will execute once recursive call is complete.

```
int solve(n, sum){
  if(n<=0) return sum;
  print(n);
  return solve(n-1, sum+n);
}
```

Tail Recursion

solve(3, 0) → solve(2, 0+3) → solve(1, 3+2) → solve(0, 5+1)

Time Complexity of Recursive Function (not always possible to compute)

```
void solve(n){
  if(n<=1) return;
  print(n);
  solve(2*n/3);
}
```

TC = ?

T(n) = T(2n/3) + O(1)

n → n/1.5 → n/(1.5)^2 - - - n/(1.5)^k = 1

n = (1.5)^k ⇒ k = log_1.5(n)

TC = O(log_1.5(n))

n → 2n/3 = n/(3/2) → n/1.5 → (n/1.5) × 2/3

Master Theorem

T(n) = a * T(n/b) + O(n^d)

a > 0 , b > 1 , d ≥ 0

- b^d > a → O(n^d)
- b^d = a → O(n^d log_b(n))
- b^d < a → O(n^log_b(a))

O(n^0) = O(1)

T(n) = T(2n/3) + O(1)

a = 1, b = 3/2 = 1.5, d = 0

b^d = 1.5^0 = 1 = a

O(n^0 log_1.5(n)) = O(log_1.5(n))

T(n) = 2T(n/2) + O(n)

a = 2, b = 2, d = 1

b^d = 2^1 = 2 = a

TC = O(n log_2(n))

T(n) = 3T(n/3) + O(n^3)

a = 3, b = 3, d = 3

b^d = 3^3 = 27 > a

TC = O(n^3)

Q → Given an integer array A & an integer B.

Count of non-empty subsequences of A of size = B having sum <= 1000

A = [400, 500, 300, 800, 100, 50] (0 1 2 3 4 5)

B = 3

Ans = 3

{400, 300, 100}

{400, 500, 100}

{500, 300, 100}

TC <= O(2^n)

SC = O(N)

(i, sum, size) (0, 0, 3)

(1, 400, 2) (1, 0, 3)

(2, 900, 1) (2, 400, 2) (2, 500, 2) (2, 0, 3)

(3, 1200, 0) (3, 900, 1) (3, 1000, 0) (3, 500, 1)

(4, 1100, 0) (4, 900, 1)

(5, 1000, 0) ✓

pruning

( )

```
int solve(i, sum, size){
  if(sum > 1000) return 0;
  if(size == 0) return 1;
  if(i == a.length) return 0;
  return solve(i+1, sum+a[i], size-1)   // select
       + solve(i+1, sum, size);         // reject
}
```