① Asymptotic Analysis
② Introduction about Arrays }

## Asymptotic Analysis

→ Method of defining a limiting behaviour.
⇓
Assuming that the input → ∞

O(n)   O(log n)

$n = 10^9$
$2 > 10^{18}$   $\boxed{10^8}$

$\boxed{f(n) = 2n^2 + n}$
when $n \to \infty$
$f(n) \approx n^2$

We can define two rules for performing this analysis:
① Drop the constants
② Drop the insignificant terms.

$f(n) = 2^n + 5$   $\underset{n \to \infty}{f(n)} = 2^n$
$f(n) = 5n + \log n$   $\underset{n \to \infty}{f(n)} = n$

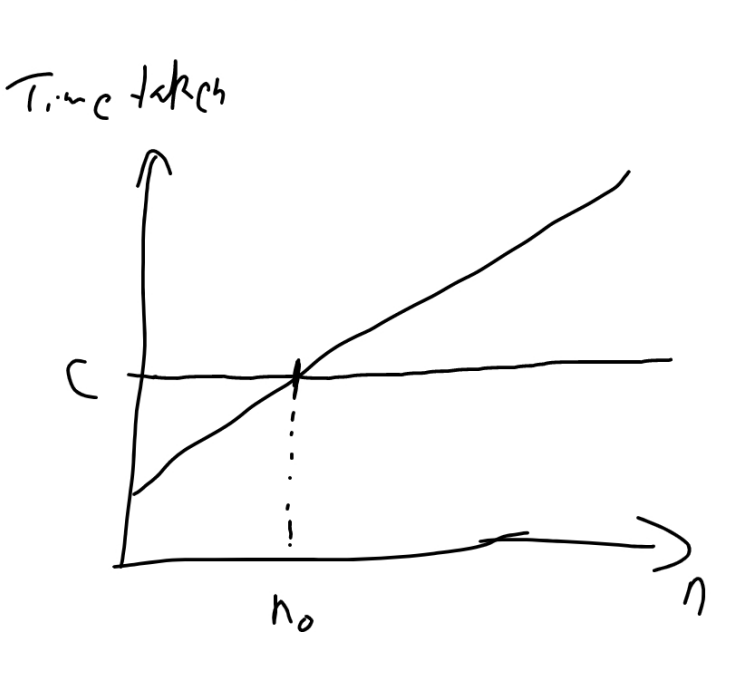
Big-O: Provide an upper bound.
$T(n) = 5n + 6$   Time Complexity of O(n)
$(O(n^2) / O(2^n))$

For interview perspective: Big-O: Provides an exact upper bound.

$T(n) = 5n^2 + 7 \Rightarrow O(n^2)$
$T(n) = n^2 + \log n + n + 1 \Rightarrow O(n^2)$

O(n)   $O(n^2)$

$O(1) < O(\log n) < O(n) < O(n*\log n) < O(n^2) < O(2^n) < O(n^n)$



Algo-1: O(1)   Algo-2: O(n)
2n+3

$f(n) = (n^2 + n^3) + n^{n-1}$
$O(n^n)$

$n^\infty$   $n^{\infty - 1}$

Best-case, worst-case & Average Case:

```
int linear-search ( int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Best case time complexity: O(1)
Worst case time complexity: O(n)
Average case time complexity
no. of comparisons

arr[0] == x : 1
arr[1] == x   2
arr[2] == x   3
⋮
arr[n-1] == x   n
no. element = x   n

(n+1) cases

$$\frac{(1 + 2 + 3 ---- n) + n}{n+1} = \frac{\frac{n(n+1)}{2} + n}{n+1}$$

$$O\left(\frac{\frac{n^2}{2} + \frac{3n}{2}}{n}\right) = O\left(\frac{n}{2} + \frac{3}{2}\right) = \boxed{O(n)}$$

⇓
Average case time complexity for linear search

Worst case
Quick Sort: $O(n^2)$   Avg. case: O(n log n)

Big-omega   Big-Theta   Big-O.
Lower bound   Exact Bound   Upper bound

T(n) = 5n+7

Ω(n)   O(n)
Ω(log n)   O(n*log n)   Θ(n)
Ω(1)   $O(n^2)$

Space Complexity: Space taken up by input + Auxiliary space + space taken up by output.

```
int t;
int m;
vector<int> func (vector<int> arr) {
    vector<int> res(m);
    vector<int> temp(t);
    -------------------
    return res;
}
```

(n)

Space taken up by input: O(n) ]
Space taken up by output: O(m) ] ×
Aux. Space taken: O(t)

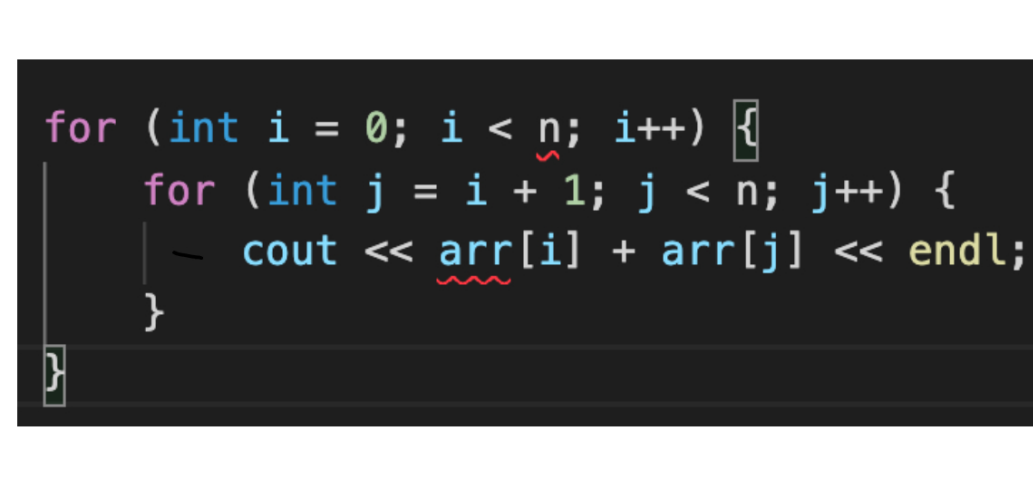
Space Complexity of above func: O(n+m+t)

---

Binary Search: O(log n)

Taken the input: O(n)

O(n + log n) = O(n).



```
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        cout << arr[i] + arr[j] << endl;
    }
}
```

i = 0 ⇒ inner for-loop runs (n-1) iterations
i = 1 ⇒ (n-2)
i = 2 ⇒ (n-3)
⋮
i = n-1 ⇒ (n-(n-1)-1) = 0

0 + 1 + 2 + 3 - ----- + (n-3) + (n-2) + (n-1)

$$O\left(\frac{n*(n-1)}{2} * 1\right) = O(n^2)$$

$\boxed{132}$

"Geeks for geeks"   32-bit is

|·|1|0|1| - - - - - - |0|1|

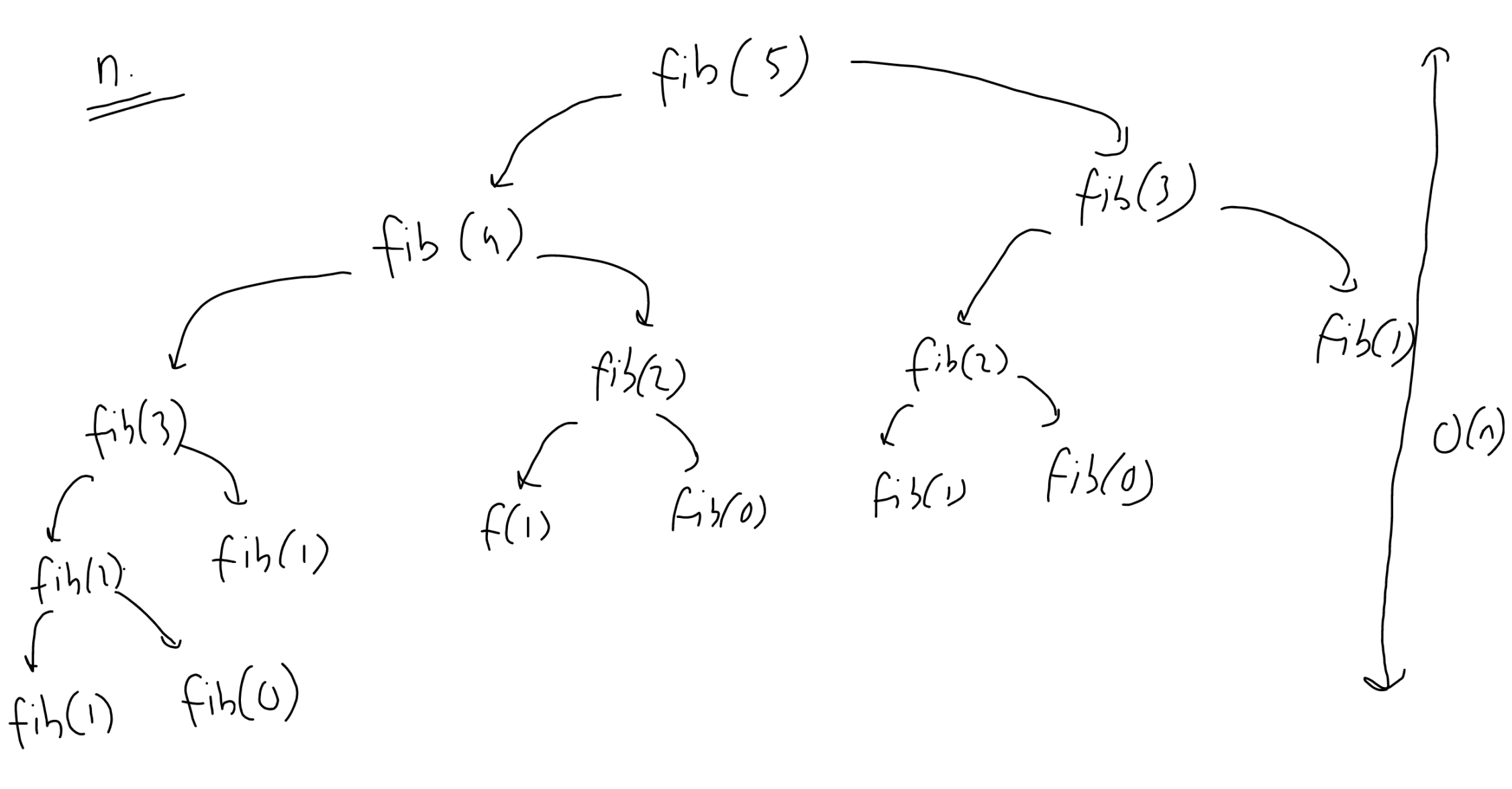
Printing an int: O(1)
Printing a string: O(length(s))

---

Recursion: When a function keeps calling itself till it satisfies some base condition.

$\{f(0) = 0, f(1) = 1\}$   $\{f(n) = f(n-1) + f(n-2)\}$
f(2) = 1   Fibonacci Series:
f(3) = 2
f(4) = 3

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



upper bound
No. of nodes in $0^{th}$ level = 1 = $2^0$
" " " " $1^{st}$ " = 2 = $2^1$
$2^{nd}$ = 4 = $2^2$
$3^{rd}$ = 8 = $2^3$
⋮
$(n-1)^{th}$ level = $2^{n-1}$

Total no. of nodes = $\underbrace{2^0 + 2^1 + 2^2 + ---- + 2^{n-1}}_{GP}$

First term = $2^0 = 1$
No. of terms = n, multiplication factor: 2
Sum of the above GP = $\frac{1(2^n - 1)}{2-1} = 2^n$

Total no. of nodes: $O(2^n)$

Time Complexity of Recursive functions =
{ (No. of nodes in the recursive tree)
×
(Time taken by each function call assuming that the recursion call takes O(1) time) }

For the above Fibonacci function:
No. of nodes in the rec. tree = $O(2^n)$
Time taken by each func. call assuming rec. calls to be O(1) = O(1)
⇒ Time complexity = $O(2^n \times 1) = O(2^n)$

Stack-space occupied by recursion:
(Max. no. of rec. calls at any given point in time) = (ht. of rec. tree)
×
(Space taken by each recursive call)

For the above func,
max. no. of rec. calls at any given point in time: O(n)
Space by each rec call = O(1)
⇒ Overall aux. space = O(n×1) = O(n)

O(n)   $O(n^2)$

```
while (true) {
    cout << "hello";
}
i = 0
while (i < n) {
    cout << "hello";
}
```

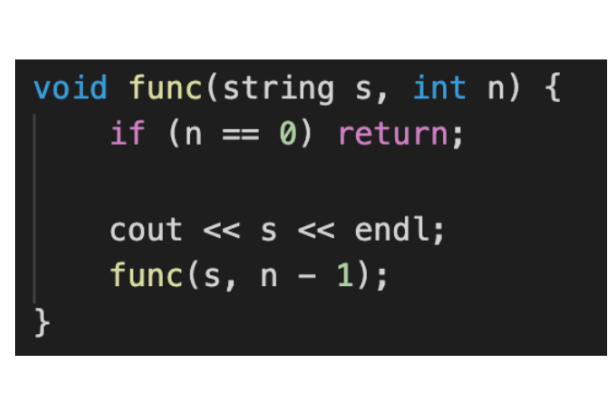
$\boxed{n \to \infty}$

---

Call by value: A new copy of the function params is created for the function.

Call by reference: The original variable gets passed as the function param.

s = "hello"
n = 5



```
void func(string s, int n) {
    if (n == 0) return;

    cout << s << endl;
    func(s, n - 1);
}
```

stdout:
hello
hello
hello
hello
hello

func("hello", 5)
func("hello", 4)
func("hello", 3)
func("hello", 2)
func("hello", 1)
func("hello", 0)

Max. no. of rec. calls at any given time = O(n)
Space occupied by each rec. call = O(m)
⇒ Aux. space complexity = O(n*m)