Asymptotic Analysis.

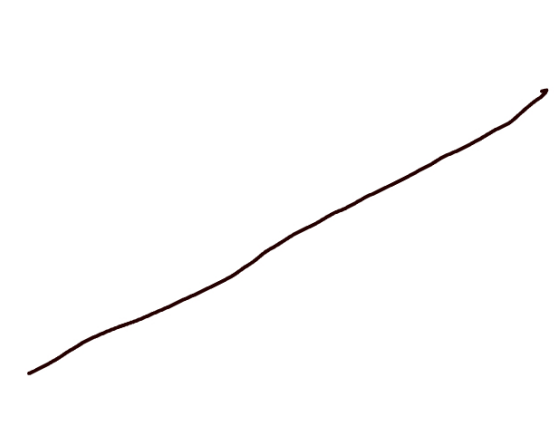
If the input tends to infinity - How will our algorithm perform in that case.

→ 1000 units / sec.

→ 500.

ⓝ ↑

y



$T_1(n) = c * n$ $T_2(n) = c * n^2$

Asymptotic Analysis → It is a method of defining a limiting behavior. → input → ∞

$T(n) = \cancel{5}n + \cancel{2}$ $T(n) = n^2$

$T(n) = \cancel{7}n + \cancel{8}$ $T(n, m) = n^2 + m$

$T(n) = n^2 + \cancel{n}$

$T(n) = n^2 + \cancel{2n}$ 10^{50} 10^{20}

↳ T(n) is asymptotically n^2

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

Big-O : Defines an upper bound to the algo.

$F(n) = 2n + 3$

$T(n) = n \Rightarrow O(n) / O(n \log n) / O(n^2) / O(2^n) / O(n!)$

↑ Tightest upper bound.

Big-Theta. : Provides the exact bound.

$F(n) = 2n + 3$

$T(n) = n \longrightarrow \Theta(n)$

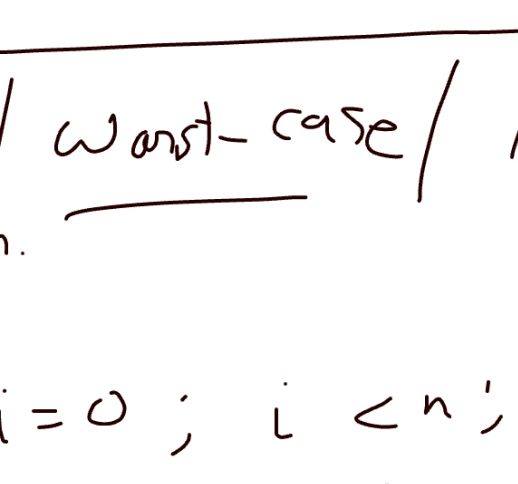
$\Theta(1.67^n)$
$O(2^n)$

Theoretical def. of Big-Θ ≈ Practical def of Big-O

Big-Omega. - Provides a "lower bound".

$F(n) = 2n + 3$

$T(n) = n \longrightarrow \Omega(1) / \Omega(\log n) / \Omega(n)$



Best-case / Worst-case / Avg-case.

```
int arr.
{
    for (int i=0 ; i<n; i++)
        if (arr[i] == target)
            return true;

    return false.
}
```

Best-case. target at index-0

$O(1)$ / $O(\log n)$ - - / $O(n!)$
×

$\Theta(1)$

$\Omega(1)$

~~1, 3, 7, 100~~

Worst-case. $O(n)$

Avg-case.

i = 0 → 1
i = 1 → 2
i = 2 → 3
⋮
i = n-1 → n
not found → n+1

no. of possible cases = n+1

Total time = 1 + 2 + 3 + 4 + - - - - + n+1

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

$TC = O\left(\frac{(n+1)(n+2)}{2(n+1)}\right) = O(n).$

For most cases, worst-case = avg. case.

But not for all.

for eg: Quick Sort → Avg. case: $O(n \log n)$
Worst case: $O(n^2)$.

Space Complexity. → Input space + [Auxiliary space] + Output space. IMP

```
vector<int> func (vector<vector<int>> arr) {    size = n^2
    vector<int> result (m),
    vector<int> temp (m^2)
    ⋮
    return result;
}
```

I/P space : $O(n^2)$
Aux. space : $O(m^2)$
O/P space : $O(m)$

SC = $O(n^2 + m^2)$
AS = $O(m^2)$

---

i = ~~0 1 2 3 4 5 6~~

```
For (i =0; i < n; i++) {
    For (int j = 0; j < m; j++) {
        Cout << i + j << endl;
    }
}
```

i = 0 → m
i = 1 → m
i = 2 → m
⋮
i = n-1 → m

Total = n * m

[["abc", "...", ...]
[ ]
⋮
[ ]]

Printing a string → O(len).

n=1 n = 10^9

32-bit int. → ‾ ‾ ‾ - - - - - ‾

len = 1 len = 10^5

n × m.

l

n ↓ m →

Space taken by 1 string : O(max. len).

No. of strings : n × m

I/P. space : O(n m * max. len).

Comparing 2 ints : O(1) time.

Comparing 2 strings : O(len).

abcdef < abcdeg.

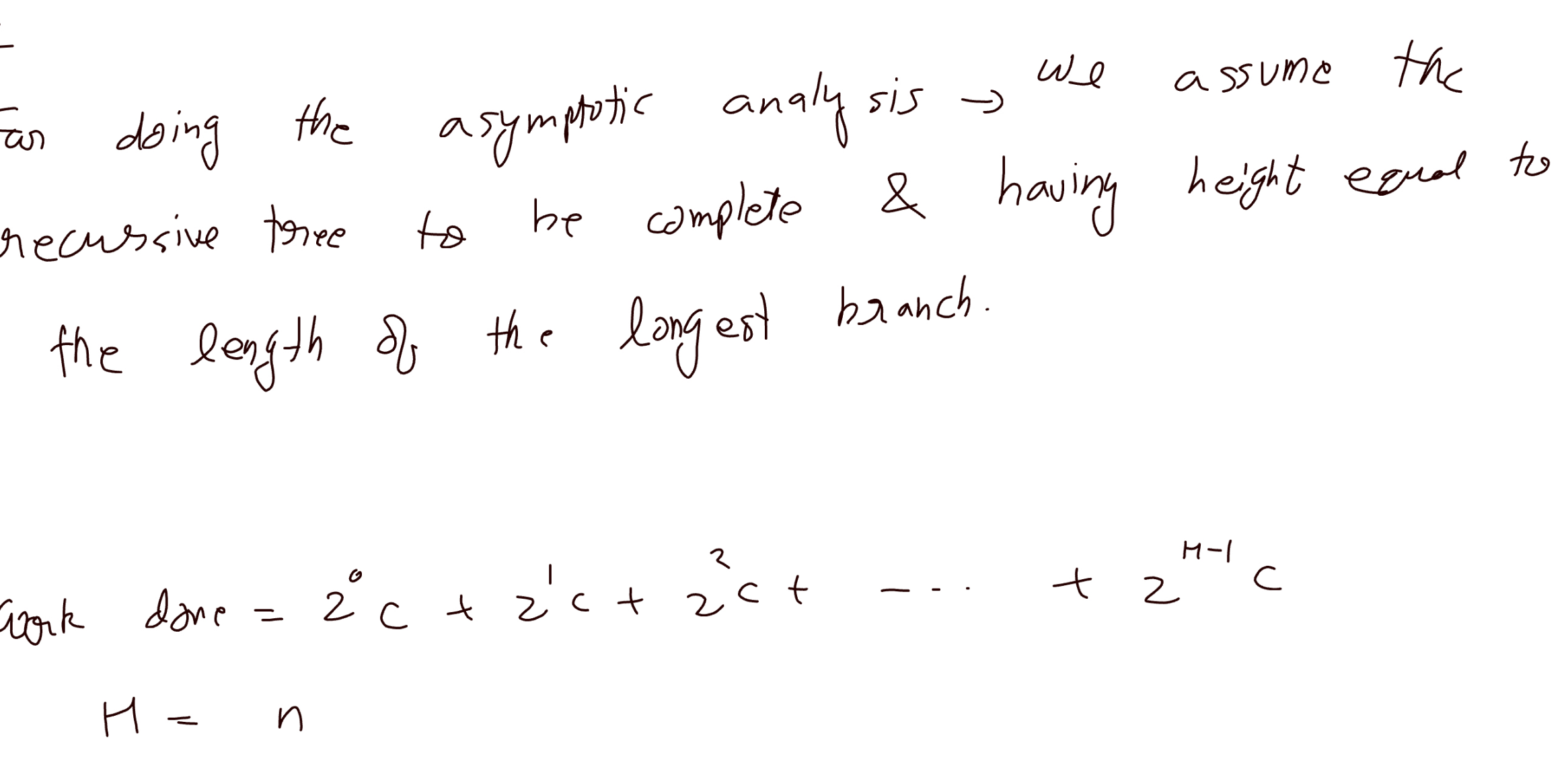
Sorting a list of nos. → O(n log n).
Comparing 2 nos. → O(1).
No. of comparisons → O(n log n)

No. of comparison → O(n log n)
Comparing 2 strings → O(m)
Sorting a list of strings → O(m n log n).

F(n) = F(n-1) + F(n-2) for n ≥ 2
1 for n = 1
0 for n = 0

0, 1, 1, 2, 3, 5, 8, 13, 21 - - -

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



# For doing the asymptotic analysis → we assume the recursive tree to be complete & having height equal to the length of the longest branch.

Work done = $2^0c + 2^1c + 2^2c + - - - + 2^{H-1}c$

H = n

$= 2^0c + 2^1c + 2^2c + - - - + 2^{n-1}c$

↳ GP

$\boxed{\text{Sum of GP} = \frac{a(r^n - 1)}{r - 1}}$ → a = 1st term of GP
r = multiplication factor
n → no. of terms.

Work done = $2^0c + 2^1c + 2^2c + - - - + 2^{n-1}c.$

a = c, r = 2, n = n

$= c\frac{(2^n - 1)}{2 - 1} = O(2^n)$

TC of Rec.

1) Find the time taken by each func. call assuming rec. calls to be O(1), say it is a

2) Find an upper bound to the total no. of func. calls, say this is b.

3) Overall TC: O(ab).