Mathemathical Formulas

1. $\sum_{i=1}^{n} 1 = n$
2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
3. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
4. $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$
5. $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$
6. $\sum_{i=0}^{k} \frac{1}{2^i} = 2 \quad \text{when } k \to +\infty$
7. $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} \quad \text{with } x \neq 1$
8. $\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \sim \ln(n)$
9. $\log(n!) \sim n \log(n)$
10. $\lim_{n \to +\infty} \sum_{k=0}^{n} x^k = \frac{1}{1-x} \quad \text{if } |x| < 1$
11. $\sum_{i=0}^{n} i x^i = \frac{1}{(1-x)^2} \quad \text{if } |x| < 1$
12. $\log(ab) = \log(a) + \log(b)$
13. $\log(\frac{a}{b}) = \log(a) - \log(b)$
14. $\log(a^b) = b \log(a)$
15. $\log_b(a) = \frac{\ln(a)}{\ln(b)}$
16. $a^{\log_b(n)} = n^{\log_b(a)}$
17. $a^n \times a^m = a^{n+m}$
18. $(a^m)^n = a^{m*n}$
19. $a^{(m)^n} \neq (a^m)^n$

How To Calculate Sums

1 Nested Sum

Calculate Nested Sum

We always start from the most inner sum until we finish calculating

Example :

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j}1 &= \sum_{i=1}^{n}\sum_{j=1}^{i}j \\
&= \sum_{i=1}^{n}\frac{i(i+1)}{2} \\
&= \frac{1}{2}\sum_{i=1}^{n}i(i+1) \\
&= \frac{1}{2}(\sum_{i=1}^{n}i^2+\sum_{i=1}^{n}i) \\
&= \frac{1}{2}(\frac{n(n+1)(2n+1)}{6}+\frac{n(n+1)}{2}) \\
&= \frac{n(n+1)(2n+4)}{12}
\end{aligned}$$

2 Adjusting Boundaries To Match A Known Sum

Adjusting Boundaries

We might come across a known sum where the boundaries aren't the same as the rule. All we need to do is calculate the sum with the boundaries of the rule and then add or subtract the term accordingly.

Example:

$$\begin{aligned}\sum_{i=0}^{k-1} 2^i &= \sum_{i=0}^{k} 2^i - \sum_{i=k}^{k} 2^i \\ &= 2^{k+1} - 1 - 2^k\end{aligned}$$

We have a sum that, by the rule, goes till the k-th term, but here it goes up to the $(k-1)$-th term. So, to get the sum up to $(k-1)$, we take the sum up to k and remove the last term.

$$\text{term}(0) + \text{term}(1) + \cdots + \text{term}(k-1) + \text{term}(k) = \textstyle\sum_{i=0}^{k} 2^i$$

$$\sum_{i=0}^{k-1} 2^i + \text{term}(k) = \sum_{i=0}^{k} 2^i$$

$$\sum_{i=0}^{k-1} 2^i = \sum_{i=0}^{k} 2^i - \text{term}(k)$$

Big Notation

Notations

- **Big O** : O upper bound
- **Big Omega**: Ω lower bound
- **Big Theta** : Θ average

Limit Definition

if $\lim\limits_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| = k \quad , \quad k > 0 \quad => \quad f(n) \in \Theta(g(n))$

if $\lim\limits_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| = 0 \quad => \quad f(n) \in O(g(n))$

if $\lim\limits_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| = +\infty \quad => \quad f(n) \in \Omega(g(n))$

Enequalities Definition

$f(n) = \Omega(g(n)) \quad \exists c > 0 \, , \quad c.g(n) \leq f(n) \quad , \forall n \geq n_0$

$f(n) = O(g(n)) \quad \exists c > 0 \, , \quad c.g(n) \geq f(n) \quad , \forall n \geq n_0$

$f(n) = \Theta(g(n)) \quad \exists c_1 > 0 \, , \exists c_2 > 0 \, , \quad c_1.g(n) \leq f(n) \leq c_2.g(n) \quad , \forall n \geq n_0$

Note

- c is a strictly positive real number.
- n is a strictly positive integer.

Example

$\sqrt{2\log_2 n + 3 + 7n} \sim O(\sqrt{n})$

$$\sqrt{2\log_2 n + 3 + 7n} \leq \sqrt{n} \cdot c$$

$$\sqrt{\frac{2\log_2 n + 3 + 7n}{n}} \leq c$$

$$\sqrt{\frac{2\log_2 n}{n} + \frac{3}{n} + 7} \leq c$$

$$\left.\begin{array}{lll} \frac{2\log_2 n}{n} \leq 1 & \Longrightarrow & n \geq 4 \\ \frac{3}{n} \leq 1 & \Longrightarrow & n \geq 3 \end{array}\right\} \longrightarrow \boxed{n_0 = 4}$$

$$\sqrt{1+1+7} \leq c$$

$$\sqrt{9} \leq c \longrightarrow \boxed{c = \sqrt{9}}$$

$2^n \sim \Omega(5^{\log_e(n)})$

$$2^n \geq 5^{\log_e(n)} \cdot c$$

$$2^n \geq n^{\log_e(5)} \cdot c$$

$$2^n \geq n^{1.6} \cdot c$$

$$\frac{2^n}{n^{1.6}} \geq c$$

$\frac{2^n}{n^{1.6}} \geq 1 \Longrightarrow n \geq 1 \longrightarrow \boxed{n_0 = 1}$ and $\boxed{c = 1}$

$n^{2^n} + 6 \cdot 2^n \sim \Theta(n^{2^n})$

$$c_1 \cdot n^{2^n} \leq n^{2^n} + 6 \cdot 2^n \leq c_2 \cdot n^{2^n}$$

$$c_1 \leq 1 + \frac{6 \cdot 2^n}{n^{2^n}} \leq c_2$$

$$\frac{6 \cdot 2^n}{n^{2^n}} \leq 1 \Longrightarrow n \geq 3 \longrightarrow \boxed{n_0 = 3}$$

$$c_1 \leq 2 \leq c_2 \longrightarrow \boxed{c_1 = 1} \text{ and } \boxed{c_2 = 2}$$

Induction

Induction's Steps

1. Verify for $n =$ term
2. Assume true for n
3. Prove for $n + 1$

Example

$a_1 = 1 \quad a_2 = 1$

$a_n = a_{n-1} + a_{n-2} \quad$ if $n \geq 3$

Prove That : $a_n = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}$

Verify For n = 1

$$\left.\begin{array}{l} a_1 = 1 \\ a_1 = \frac{(\frac{1+\sqrt{5}}{2})^1 - (\frac{1-\sqrt{5}}{2})^1}{\sqrt{5}} = 1 \end{array}\right\} \longrightarrow \text{True For } \; n = 1$$

Assume True For n And Prove For n+1

$a_{n+1} = a_n + a_{n-1} \quad$ if $n \geq 3$

Prove That : $a_{n+1} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}$

$$
\begin{aligned}
a_{n+1} &= a_n + a_{n-1} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} \cdot \left(1 + \frac{2}{1+\sqrt{5}}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n} \cdot \left(1 + \frac{2}{1-\sqrt{5}}\right)}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} \cdot \left(\frac{(3+\sqrt{5})(1-\sqrt{5})}{(1+\sqrt{5})(1-\sqrt{5})}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n} \cdot \left(\frac{(3-\sqrt{5})(1+\sqrt{5})}{(1-\sqrt{5})(1+\sqrt{5})}\right)}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} \cdot \left(\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n} \cdot \left(\frac{1-\sqrt{5}}{2}\right)}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}} \\
&= a_{n+1}
\end{aligned}
$$

True For $n + 1$

Conclusion True For n

Order Of Complexity

Order

$$1 \leq \log(n) \leq \sqrt{n} \leq n \leq n\log(n) \leq n^2 \leq n^3 \leq 2^n \leq n! \leq n^n$$

Complexity

Some Terminology

- **Frequency of Execution (F_r)**: The number of times an instruction is executed.
- **Execution Time of Basic Instructions (Δt)**: We assume that all basic instructions (such as print, read, assignment, arithmetic operations, etc.) have the same execution time, denoted as Δt.
- **Function ($f(n)$)**: Represents the total frequency of the program's instructions in relation to the data size n.
- **Exact Execution Time Function ($T(n)$)**: The execution time function in relation to the data size n.
- **Approximate Theoretical Complexity (Asymptotic)**: It approximates $T(n)$ by omitting all constants and taking the term with the highest growth rate.

Example

Algorithm Sum of First N Integers

```
1:  Var
2:  n, sum, i integer;
3:  Begin
4:  sum ← 0;
5:  i ← 1;
6:  print('Input Integer N : ')
7:  Read(n);
8:  while i <= n do
9:      sum ← sum + i;
10:     i ← i + 1;
11: end while
12: print('Sum is ' ,sum);
13: End
```

Time Complexity :

we have $f(n) = \sum fr$ since $f(n)$ is sum of frequency of execution (fr) we need to figure out the fr of each instruction and sum them :

sum ← 0	$fr = 1$ (one affectation)
i ← 1	$fr = 1$ (one affectation)
print('Input Integer N : ')	$fr = 1$ (one print)
Read(n)	$fr = 1$ (one read)
while i <= n **do**	$fr = n + 1$ (check the while condition n+1 times)
sum ← sum + i	$fr = 2n$ (one affectation and one arithmetic operation + (2) inside a while that loops n times (2n))
i ← i + 1	$fr = 2n$ (one affectation and one arithmetic operation + (2) inside a while that loops n times (2n))
print('Sum is ' ,sum)	$fr = 1$ (one print)

$$\begin{aligned} f(n) &= \sum fr \\ &= 1 + 1 + 1 + 1 + (n+1) + 2n + 2n + 1 \\ &= 5n + 6 \\ &= \boxed{5n + 6} \end{aligned}$$

now that we have the complexity function $f(n)$ we need to find the $T(n)$ we have $T(n) = f(n) \times \Delta t$

$$\begin{aligned} T(n) &= f(n) \times \Delta t \\ &= (5n + 6) \times \Delta t \\ &= \underbrace{5\Delta t}_{a} n + \underbrace{\Delta t 6}_{b} \\ &= \boxed{an + b} \end{aligned}$$

In this example the exact theoritical complexity is $T(n) = an + b$ and its approximate theoritical complexity is $an + b \sim O(n)$ we notice that its time complexity is linear

Iterative Algorithm

Iterative Algorithm

Iterative algorithm complexity is calculated by summing the frequency of all instructions.

For Loop

- If the index is decremented or incremented by just 1:
  - The number of iterations is **upper bound - lower bound + 1**
  - Each loop can be represented as a sum and used to calculate complexity
- If the index is decremented or incremented by a constant $c \geq 2$:
  - The number of iterations is $c \cdot k = n \rightarrow k = \frac{n}{c}$
  - Cannot be represented as a sum
- If the index is multiplied or divided by a constant $c \geq 2$:
  - The number of iterations is $c^k = n \rightarrow k = \log_c(n)$
  - Cannot be represented as a sum

While Loop

- Can never be represented as a sum
- If the index is incremented or decremented by just 1, the number of iterations is the difference between the upper and lower bounds.
- If the index is incremented or decremented by a constant $c \geq 2$, the number of iterations is $\frac{n}{c}$.
- If the index is multiplied or divided by a constant $c \geq 2$, the number of iterations is $\log_c(n)$.

Example

Algorithm

```
for (i = n ; i ≥ 0 ; i ← i/2) do
    print('Index i : ',i)
end for
```

Iteration : $\frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \ldots, \frac{n}{2^k}$

Loop stops when $i < 0 \Rightarrow \frac{n}{2^k} < 0$ since it's an natural division it means : $2^k > n \Longrightarrow k > \log_2(n) \Longrightarrow k = \lceil \log_2(n) \rceil = \boxed{\log_2(n) + 1}$

In conslusion $O(\log(n))$

Algorithm

```
r ← 0;
for (i = 1 ; i ≤ n² ; i ← i + 1) do
    for (j = 1 ; j ≤ 2n − 1 ; j ← j + 1) do
        r ← r + 1;
    end for
end for
```

$$\begin{aligned}\sum_{i=1}^{n^2} \sum_{j=1}^{2n-1} 1 &= \sum_{i=1}^{n^2} 2n - 1 \\ &= (2n-1) \cdot n^2 \\ &= \boxed{2n^3 - n^2}\end{aligned}$$

In conslusion $O(n^3)$

Algorithm

```
r ← 0;
for (i = 1 ; i ≤ n ; i ← i + 1) do
    for (j = i + 1 ; j ≤ n ; j ← j + 1) do
        for (k = 1 ; k ≤ j ; j ← j + 1) do
            r ← r + 1;
        end for
    end for
end for
```

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sum_{k=1}^{j}1 &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} j \\
&= \sum_{i=1}^{n}(\sum_{j=1}^{n} j - \sum_{j=1}^{i} j) \\
&= \sum_{i=1}^{n}(\frac{n(n+1)}{2} - \frac{i(i+1)}{2}) \\
&= \frac{1}{2}(\sum_{i=1}^{n} n(n+1) - \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i) \\
&= \frac{1}{2}(n^2(n+1) - \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2}) \\
&= \frac{4n^3 - 3n^2 - 7n}{12}
\end{aligned}$$

In conslusion $O(n^3)$

Algorithm

```
r ← 0; i ← 1
while (i ≤ n) do
    for (j = n^2 ; j ≤ 5 ; j ← j − 1) do
        r ← r + 1;
    end for
    i ← i + 2;
end while
```

For Loop iterates : $n^2 - 5 + 1 = \boxed{n^2 - 4}$

While Loop : $2k > n \Rightarrow k > \frac{n}{2} \Rightarrow k = \lceil \frac{n}{2} \rceil = \boxed{\frac{n}{2} + 1}$

In conslusion $O((n^2 - 4)(\frac{n}{2} + 1)) = O(n^3)$

Algorithm

```
i ← 2; j ← 1
while (i ≤ n) do
    i ← i ∗ i;
end while
while (j ≤ i) do
    j ← 4 ∗ j;
end while
```

First While Iteration : $2, 2^{2^1}, 2^{2^2}, 2^{2^3}, \ldots, 2^{2^k}$

$2^{2^k} > n \Rightarrow k > \log(\log(n)) \Rightarrow k = \lceil \log(\log(n)) \rceil = \boxed{\log(\log(n)) + 1}$

Second While Iteration : $1, 4^1, 4^2, 4^3, \ldots, 4^k$

$4^k > i \Rightarrow 4^k > n \Rightarrow k > \log(n) \Rightarrow k = \lceil \log(n) \rceil = \boxed{\log(n) + 1}$

In Conclusion $O(\log(\log(n)) + 1 + \log(n) + 1) = O(\log(n))$

Recursive Algorithm

Recursive Algorithm

A recursive algorithm is a function that calls itself and includes a base case to terminate the recursion. We analyze its complexity using recursive equations.

Recursive Equations

Divide and Conquer:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- a : Number of recursive calls at once.
- b : Size of subproblem.
- $f(n)$: Cost of composition/decomposition.

Decrement and Conquer:

$$T(n) = aT\left(n - b\right) + c$$

Where:

- a : Number of recursive calls at once.
- b : Size of subproblem.
- c : Cost of performing an operation.

By Substitution

The substitution method involves repeatedly substituting into the recursive equation until the k-th step, where the base case is reached.

Example

$T(0) = 1$
$T(n) = T(n-1) + 2^n$

$$\begin{aligned} T(n) &= T(n-1) + 2^n \\ &= T(n-2) + 2^{n-1} + 2^n \\ &= T(n-3) + 2^{n-2} + 2^{n-1} + 2^n \\ &= T(n-k) + 2^{n-k+1} + \cdots + 2^{n-2} + 2^{n-1} + 2^n \end{aligned}$$

$n - k = 0 \Rightarrow \boxed{k = n}$

$$\begin{aligned} &= T(0) + 2^1 + \cdots + 2^{n-2} + 2^{n-1} + 2^n \\ &= 2^0 + 2^1 + \cdots + 2^{n-2} + 2^{n-1} + 2^n \\ &= \sum_{i=0}^{n} 2^i \\ &= 2^{n+1} - 1 = \boxed{O(2^n)} \end{aligned}$$

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

$$\begin{aligned} T(n) &= 2n + 1 + 2T(\frac{n}{2}) \\ &= 2n + 1 + 2[2(\frac{n}{2} + 1 + 2T(\frac{n}{2^2}))] \\ &= 2n + 2n + 1 + 2 + 2^2 T(\frac{n}{2^2}) \\ &= 2n + 2n + 1 + 2 + 2^2[2\frac{n}{2^2} + 1 + 2T(\frac{n}{2^3})] \\ &= 2n + 2n + 2n + 1 + 2 + 2^2 + 2^3 T(\frac{n}{2^3}) \\ &= 2n \cdot k + 1 + 2 + 2^2 + \cdots + 2^{k-1} + 2^k T(\frac{n}{2^k}) \\ &= 2n \cdot k + \sum_{i=0}^{k-1} 2^i + 2^k T(\frac{n}{2^k}) \end{aligned}$$

$\frac{n}{2^k} = 1 \Rightarrow \boxed{k = \log(n)}$

$$\begin{aligned} &= 2n\log(n) + \sum_{i=0}^{k-1} 2^i + nT(1) \\ &= 2n\log(n) + 2^{k+1} - 1 - 2^k + n \\ &= 2n\log(n) + 2n - 1 - n + n \\ &= 2n\log(n) + 2n - 1 = \boxed{O(n\log(n))} \end{aligned}$$

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

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + \log\log n \\
&= T(\frac{n}{2^2}) + \log\log(\frac{n}{2}) + \log\log n \\
&= T(\frac{n}{2^3}) + \log\log(\frac{n}{2^2}) + \log\log(\frac{n}{2}) + \log\log n \\
&= T(\frac{n}{2^k}) + \log\log(\frac{n}{2^{k-1}}) + \cdots + \log\log(\frac{n}{2}) + \log\log n \\
&= T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} \log\log(\frac{n}{2^i}) \\
&= T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} \log(\log(n) - \log(2^i)) \\
&= T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} \log(\log(n) - i\log(2)) \\
&= T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} \log(\log(n) - i)
\end{aligned}
$$

$\frac{n}{2^k} = 1 \Rightarrow \boxed{k = \log(n)}$

$$
\begin{aligned}
&= T(1) + \sum_{i=0}^{\log(n)-1} \log(\log(n) - i) \\
&= 1 + \log(\log(n)) + \log(\log(n) - 1) + \cdots + \log(\log(n) - \log(n) + 1) \\
&= 1 + \log(\log(n) * (\log(n) - 1) * \cdots * 1) \\
&= 1 + \log((\log(n))!) \\
&= 1 + \log(n)\log(\log(n)) = \boxed{O(\log(n)\log(\log(n)))}
\end{aligned}
$$

Change of Variable

To simplify the recursive equation, we substitute n with a value related to m, transforming $T(n)$ into $S(m)$. This allows us to analyze the complexity in terms of m. Finally, we revert the substitution to express the complexity in terms of n.

Example

$T(1) = 1$
$T(n) = 2T(\sqrt{n}) + \log(n)$

let $n = 2^m$

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m$$

let $T(2^m) = S(m)$

$$S(m) = 2S(\frac{m}{2}) + m$$

This is the same as merge sort so complexity in terms of m is $O(m\log(m))$, in terms of n is $\boxed{O(\log(n)\log(\log(n)))}$

Upper & Lower Bounds

To analyze the complexity of a recursive equation with multiple calls, we calculate both the upper and lower bounds. Ultimately, the result remains consistent.

- **Lower Bound**: Replace larger terms call with smaller ones.
- **Upper Bound**: Replace smaller terms call with larger ones.

Example

Algorithm Fibonnaci

```
1:  function FIB_REC((I/n:Integer): Integer)
2:      Begin
3:      if (n = 0) then
4:          return 0;
5:      else if (n = 1) then
6:          return 1;
7:      else
8:          return Fib_rec(n-1)+Fib_rec(n-2);
9:      end if
10: end function
```

$T(0) = T(1) = 0$
$T(n) = T(n-1) + T(n-2) + 1$

Lower Bound

$T(n) = 2T(n-2) + 1$

$$\begin{aligned} T(n) &= 2T(n-2) + 1 \\ &= 2[2T(n-4) + 1] + 1 \\ &= 2^2 T(n-4) + 2 + 1 \\ &= 2^2[2T(n-6) + 1] + 2 + 1 \\ &= 2^3 T(n-6) + 2^2 + 2 + 1 \\ &= 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i \\ &= 2^k T(n-2k) + 2^{k+1} - 1 - 2^k \end{aligned}$$

$n - 2k = 0 \Rightarrow \boxed{k = \frac{n}{2}}$

$$\begin{aligned} &= 2^{\frac{n}{2}} T(0) + 2^{\frac{n}{2}+1} - 1 - 2^{\frac{n}{2}} \\ &= 2^{\frac{n}{2}} + 2^{\frac{n}{2}+1} - 1 - 2^{\frac{n}{2}} \\ &= 2^{\frac{n}{2}+1} - 1 = O(2^n) \end{aligned}$$

Upper Bound

$T(n) = 2T(n-1) + 1$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2[2T(n-2) + 1] + 1 \\ &= 2^2 T(n-2) + 2 + 1 \\ &= 2^2[2T(n-3) + 1] + 2 + 1 \\ &= 2^3 T(n-3) + 2^2 + 2 + 1 \\ &= 2^3 T(n-3) + 2^2 + 2 + 1 \\ &= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i \\ &= 2^k T(n-k) + 2^{k+1} - 1 - 2^k \end{aligned}$$

$n - k = 0 \Rightarrow \boxed{k = n}$

$$
\begin{aligned}
&= 2^n T(0) + 2^{n+1} - 1 - 2^n \\
&= 2^n + 2^{n+1} - 1 - 2^n \\
&= 2^{n+1} - 1 = O(2^n)
\end{aligned}
$$

$$\boxed{LowerBound = UpperBound = O(2^n) \Rightarrow \text{Complexity} = O(2^n)}$$

Tree

prettyBox

Guess & Test

prettyBox