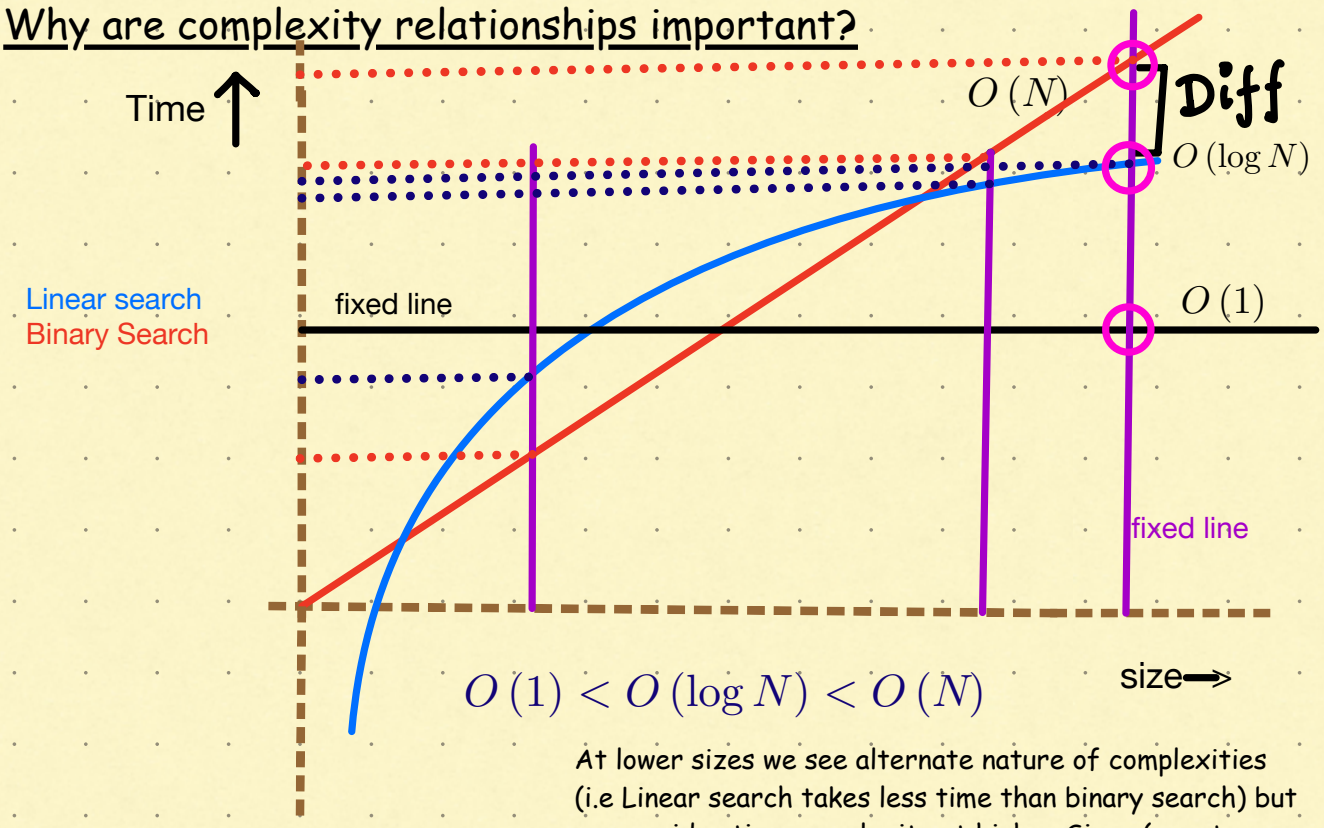
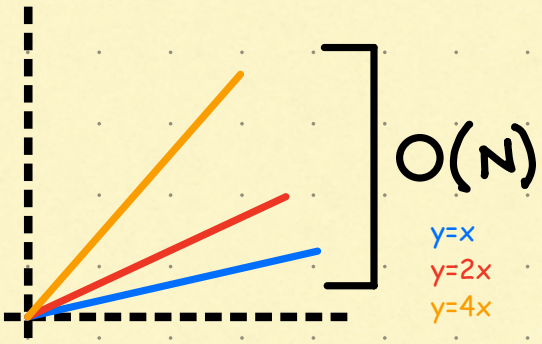
Why are complexity relationships important?



At lower sizes we see alternate nature of complexities (i.e Linear search takes less time than binary search) but we consider time complexity at higher Sizes (worst cases) only , i.e in higher sizes time taken by binary search is less as compared to linear search when seen from the graph (comparison at last 2 fixed purple lines), Hence Binary Search Algorithm is efficient here!

What do we consider when thinking about complexity:

- Always look for worse case complexity (why? -- The chances of a website to crash is more when 2 million users are using it as compared to when 100 users are using it)
- We always look at complexities for large/infinite data
-



*Even though two value of actual time is different, they are all growing linearly.
*We don't care about actual time
** Hence we only need the relationships
i.e if y=3x+5 ,then we ignore all constants in time complexity as we only need the relation in which y depends on x (i.e here linearly dependent y=x)

$$O\left(N^3 + \log N\right)$$

Now consider 1 million as 1m

From 2nd point(i.e we look at complexities of large data)

$$\Rightarrow 1m = \left((1m)^3 + \log(1m)\right)$$

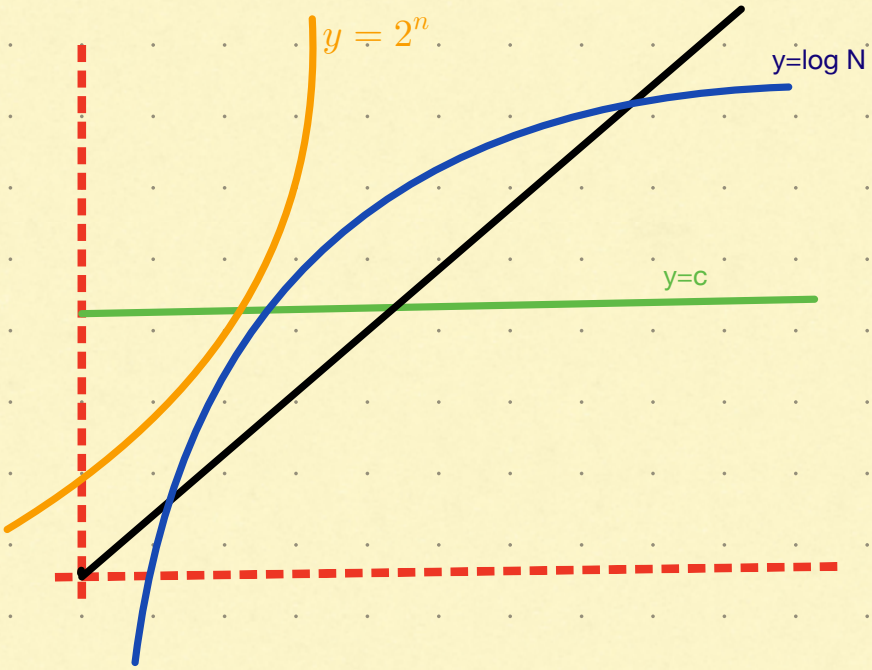
$$= \left((1m)^3 + 6\,\text{sec}\right)$$

$6 <<< 1m$ (ignored)

- Always ignore less dominating terms

Example:

$$O\left(3N^3 + 4N^2 + 5N + 6\right)$$
$$= O\left(N^3 + \cancel{N^2} + \cancel{N}\right)$$
$$= O\left(N^3\right)$$



$$O(1) < O(\log N) < 0(N) < O(2^n)$$

BIG O NOTATION:

WORD DEFINITION:

Big O notation is a mathematical notation that describes the limiting behaviour of a function when the argument tends towards a particular value or infinity

$O\left(N^3\right)$ Upper bound is N^3

MATHEMATICAL DEFINITION:

$$f(N) = O(g(N))$$

$$\boxed{\lim_{N\to\infty} \frac{f(N)}{g(N)} < \infty}$$

$$\underset{g(n)}{O\left(N^3\right)} = \underset{f(n)}{O\left(6N^3 + 3N + s\right)}$$

$$\lim_{N\to\infty} \frac{6N^3 + 3N + S}{N^3}$$

$$= \lim_{N\to\infty} 6 + \frac{3}{N^2} + \frac{S}{N^3} = 6 + \frac{3}{\infty} + \frac{S}{\infty}$$

$$= 6 < \infty$$

We got a finite answer which is capped(less than infinity) , thus there's an upper bound which proves the mathematical definition

BIG OMEGA NOTATION: (OPPOSITE OF BIG O)

WORD DEFINITION:

It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).

Just like O notation provide an asymptotic upper bound, Ω notation provides asymptotic lower bound.

MATHEMATICAL DEFINITION:

$$\lim_{N \to \infty} \frac{f(N)}{g(N)} > 0$$

Qn) What if an algorithm has lower bound and upper bound as N^2

$$= O\left(N^2\right) \ \& \ \Omega\left(N^2\right)$$

THETA NOTATION:(COMBINING BOTH ABOVE)

WORD DEFINITION: $\Theta\left(N^2\right)$

It is defined as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.

MATHEMATICAL DEFINITION:

$$0 < \lim_{N \to \infty} \frac{f(N)}{g(N)} < \infty$$

LITTLE O NOTATION:

WORD DEFINITION:

Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).

MATHEMATICAL DEFINITION:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Big Oh: $f \leq g$; f=O(g)

Little Oh: $f < g$ (strictly decreasing) ; f=o(g)

LITTLE Omega NOTATION:

WORD DEFINITION:

We use little omega(ω) notation to denote a lower bound that is not asymptotically tight

MATHEMATICAL DEFINITION:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Ω	ω
$f = \Omega(g)$ $f \geq g$	$f = \omega(g)$ $f > g$

AUXILIARY SPACE:

Auxiliary Space is the extra space or temporary space used by an algorithm.

The space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criterion than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort, and Heap Sort use O(1) auxiliary space. The space complexity of all these sorting algorithms is O(n) though.

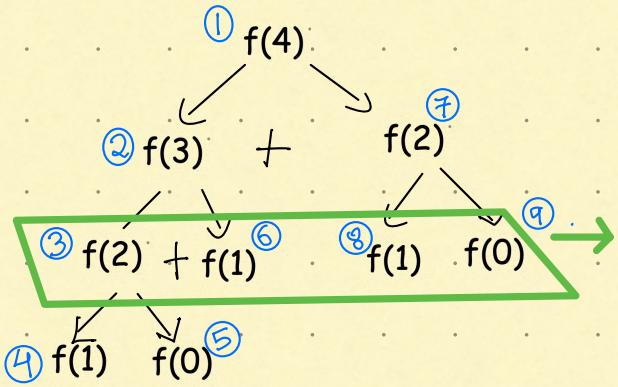
Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n*n, this will require O(n^2) space.

Qn) for (i=1;i<=N;){
        for(j=-1;j<=k ;j++){
          // some operation that takes time t
} i=i+k
}. FIND TIME COMPLEXITY OF THIS ALGORITHM

Solution) Inner Loop: O(kt). time. (for every time its running it takes t time so for k times total time is kt)
Ans = O(kt * (how many times outer loop is running))
Outer Loop: i=1,i+k,1+2k,1+3k.....1+xk
last value must satisfy i<=N i.e 1+xk<=N
==> x= (N-1)/k. where x is number of times outer loop running

Therefore Time complexity is O(kt * (N-1)/k)
= O(Nt)

COMPLEXITY ANALYSIS OF RECURSIVE PROGRAMS



9th recursive call won't even execute if 3rd recursion ain't called yet

** Recursive programs do not have constant space complexity as those function calls are stored in stack memory
**At any particular time no two function calls of recursion will be at stack at the same time[green box]---- only calls that are interlinked will be in the stack at the same time
** Space complexity = height of recursive tree (i.e O(N) here for Nth Fibonacci Number)

2 TYPES OF RECURSION:

1) LINEAR	2) DIV AND CONQUER
$F(N) = F(N-1) + F(N-2)$	$F(N) = F\left(\frac{N}{2}\right) + O(1)$

Let's get started!

DIVIDE AND CONQUER RECURRENCE RELATION:

Form:

① $$T(x) = a_1 T(b_1 x + \varepsilon_1(x)) + a_2 T(b_2 x + \varepsilon_2(x)) + \ldots + a_k T(b_k x + \varepsilon_k(x)) + g(x)$$

$for\ \forall\ x \geq x_0$ (constant)

② $$T(n) = T\left(\frac{N}{2}\right) + c$$

$a_1 = 1$

$\varepsilon_1(x) = 0$

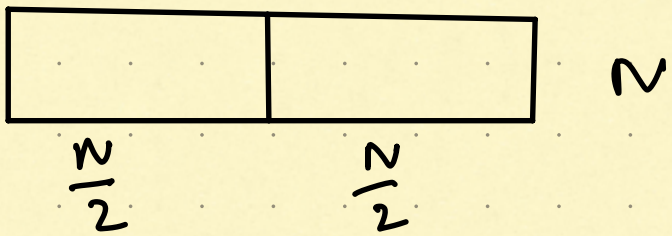
$b_1 = \frac{1}{2}$

$g(x) = c$

On comparing eq 1 and eq2(for BS) we get these values of the listed functions

$$T(N) = 9T\left(\frac{N}{3}\right) + \frac{4}{3}T\left(\frac{5}{6}N\right) + 4N^3$$

$9 \to a_1$, $\frac{N}{3} \to b_1$, $\frac{4}{3} \to a_2$, $\frac{5}{6}N \to b_2$, $4N^3 \to g(N)$



$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + (N-1)$$

sort each

merge

merge sort

How to solve to get complexity?

- PLUG & CHUG
- MASTERS THEOREM

$$F(N) = F\left(\frac{N}{2}\right) + C$$

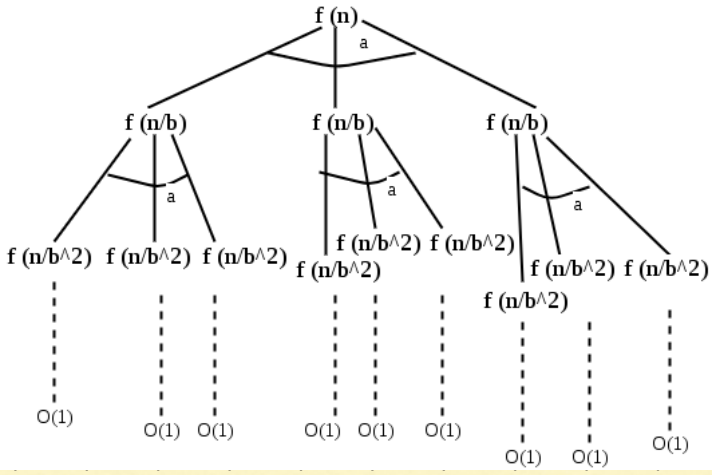
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$ where a >= 1 and b>1

There are following three cases for it:

If $f(n) = O(n^c)$ where $c < \log_b a$ then $T(n) = \Theta\left(n^{\log_b a}\right)$

If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$

If $f(n) = \Omega(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$



In the recurrence tree method, we calculate the total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically the same, then our result becomes height multiplied by work done at any level (Case 2). If work done at the root is asymptotically more, then our result becomes work done at the root (Case 3).

- It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Log(n) cannot be solved using master method.
- Case 2 can be extended for $f(n) = \Theta\left(n^c \log^k n\right)$ if

$f(n) = \Theta\left(n^c \log^k n\right)$ for some constant $k \geq 0$ & $c = \log_b a$ then $T(n) = \Theta\left(n^c \log^{k+1} n\right)$

AKRA BAZZI:

$$T(x) = \theta(x^p + x^p \int_1^x \frac{g(x)}{u^{p+1}} \cdot du)$$

What is P ?

You have to find p such that $a_1 b_1^p + a_2 b_2^p + \ldots = 1$

$$\sum_{i=1}^{k} a_i b_i^P = 1$$

Lets look at some examples say Merge Sort;

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

$a_1 = 2 \quad b_1 = \frac{1}{2} \quad g(n) = n - 1$

$2 * \left(\frac{1}{2}\right)^p = 1 \qquad \therefore p = 1$

Now put value of p in Akra Bazzi formula:

$$T(n) = \theta\left(x^1 + x^1 \int_1^x \frac{u-1}{u^2} du\right) = \theta\left(x + x \int_1^x \frac{1}{u} - \frac{1}{u^2} du\right)$$

$$= \theta\left(x + x\left[\log u + \frac{1}{u}\right]_1^x\right) = \theta(x + x\left[\log x + \frac{1}{x} - 1\right]$$

$$= \theta(x + x\log x + 1 - x) = \theta(x\log x + 1)$$

$$\boxed{= \theta(x \log x)}$$

For array of size N : Merge sort complexity is $= \theta(x \log x)$

$$T(N) = \underset{a_1}{2}T\left(\frac{N}{\underset{b_1}{2}}\right) + \frac{8}{\underset{a_2}{9}}T\left(\frac{3N}{\underset{b_2}{4}}\right) + N^2$$

$$2 \times \left(\frac{1}{2}\right)^p + \frac{8}{9} \times \left(\frac{3}{4}\right)^p = 1 \qquad \therefore p = 2$$

Hit & Trial

$$2 \times \frac{1}{4} + \frac{8}{9} \times \frac{9}{16} = 1$$

$$T(n) = \theta\left(x^2 + x^2 \int_1^x \frac{u^2}{u^3}.dx\right) = \theta\left(x^2 + x^2 \ln x\right)$$

$$= \theta\left(x^2 \ln x\right)$$

If you can't find the value of P:

$$T(x) = 3T\left(\frac{x}{3}\right) + 4T\left(\frac{x}{9}\right) + x^2$$

Lets try p=1

$$3 \times \left(\frac{1}{3}\right)^1 + 4 \times \left(\frac{1}{9}\right)^1 = 1$$

$\Rightarrow 2 > 1 \Rightarrow$ increase the denominator

p>1

Now say p=2 $\quad 3 \times \frac{1}{9} + 4 \times \frac{1}{16} = \frac{7}{12} < 1$

here p<2.

NOTE: When p < power of g(x) then answer = O(g(x))

here g(x) = x^2

p<2 , here ans= O(x^2)

SOLVING LINEAR RECURRENCES:

Ex: F(N) = F(N-1) + F(N-2)

Form:

$$f(x) = a_1 f(x-1) + a_2 f(n-2) + \ldots a_n f(x-n)$$

$$f(x) = \sum_{i=1}^{n} a_i f(x-i)$$ for a_{i}, n is fixed ; n= order of recurrence

SOLUTION for Fibonacci no relation:

f(n) = f(n-1) + f(n-2)

Steps:

1) For some constant alpha , Put: $f(n) = \alpha^n$

$$\Rightarrow \alpha^n = \alpha^{n-1} + \alpha^{n-2}$$

$$\Rightarrow \alpha^n - \alpha^{n-1} - \alpha^{n-2} = 0 \quad \text{—①}$$

div ① by → α^{n-2}

$$\Rightarrow \alpha^2 - \alpha - 1 = 0$$

$$\frac{\alpha^n}{\alpha^{n-2}} = \frac{\alpha^n \times \alpha^2}{\alpha^n} =$$

root of this eqn

$$\alpha = \frac{1 \pm \sqrt{5}}{2}$$

2)

$$f(n) = c_1 \alpha_1^n + c_2 \alpha_2^n = f(n-1) + f(n-2) \rightarrow \text{sol}^n \text{ for Fibo}$$

$$f(n) = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

3) Fact: Number of roots = Number of answers we already have

Hence we have 2 roots $\{\alpha_1, \alpha_2\}$

Thus we have : 2 answers already

$$f(0) = 0, f(1) = 1$$

$$f(0) = 0 = c_1 + c_2 \Rightarrow c_1 = -c_2$$

$$f(1) = C_1\left(\frac{1+\sqrt{5}}{2}\right) + C_2\left(\frac{1-\sqrt{5}}{2}\right)$$

$$1 = C_1\left(\frac{1+\sqrt{5}}{2}\right) - C_1\left(\frac{1-\sqrt{5}}{2}\right)$$

$$C_1 = \frac{1}{\sqrt{5}}, C_2 = -\frac{1}{\sqrt{5}}$$

Putting this in original equation:

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

→ n^{th} Fibo Formula

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\underbrace{\left(\frac{1-\sqrt{5}}{2}\right)^n}$$

↳ as $n \to \infty$ this val $\to 0$, Hence less dominating = ignored

Time Complexity = $O\left(\frac{1+\sqrt{5}}{2}\right)^n$ Golden Ratio In Maths

Q) Equal roots:

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

$$f(n) = \alpha^n \qquad \alpha^n = 2\alpha^{n-1} + \alpha^{n-2}$$

$$\frac{\alpha^n - 2\alpha^{n-1} - \alpha^{n-2}}{\alpha^{n-2}} = 0 \qquad \Rightarrow \alpha^2 - 2\alpha + 1 = 0 \Rightarrow \alpha = 1$$

General Case: If alpha is repeated r times then,

$\alpha^n, n\alpha, n^2\alpha^2, \cdots, n^{r-1}\alpha^n$ are all solutions to the recurrence

Hence I can take 2 roots as : $1, n\alpha^n$

$$f(n) = c_1(\alpha)^n + c_2 n\alpha^n$$

$$f(n) = c_1 + c_2 n$$

$$f(0) = 0 = c_1 \qquad f(1) = 1 = C_1 + C_2$$

$$\boxed{C_2 = 1}$$

$$f(N) = n$$

$$O(N)$$

Non Homogenous Linear Recurrences:

$$f(x) = a_1 f(x-1) + a_2 f(n-2) + \ldots a_n f(x-n) + g(x)$$ extra

How to solve?

1) Replace g(x) by 0 & solve usually.

$$f(n) = 4f(n-1) + 3^n$$
$$\Rightarrow f(n) = 4f(n-1)$$
$$\Rightarrow \alpha^n = 4\alpha^{n-1} = \alpha^n - 4\alpha^{n-1} = 0$$
$$\Rightarrow \alpha - 4 - 0$$
$$\Rightarrow \alpha = 4$$

homogenous sol $f(n) = c_1\alpha^n = c_1 4^n$

2)Take g(n) on one side and find particular solution:
Guess something that is similar to g(n) , if here g(n) = n^2 ,guess a polynomial of degree 2

$f(n) = 4f(n-1) + 3^n$ Guess $f(n) = C \cdot 3^n$

$$C3^n - 4C3^{n-1} = 3^n$$

Particular sol^n $\rightarrow$ $f(n) = -3^{n+1}$

3) Add both solutions together:

$f(n) = c_1 4^n - 3^{n+1}$ $\rightarrow$ $f(1) = 1 \Rightarrow c_1 4 - 3^2 = 1 \Rightarrow c_1 = \frac{5}{2}$

$$\boxed{f(n) = \frac{5}{2}4^n - 3^{n+1}}$$

How to guess particular solution?

- If g(n) is exponential , guess of same type

$g(n) = 2^n + 3^n$ $f(n) = a2^n + b3^n$ → ans

- If it is polynomial then guess of same degree