Divide And Conquer

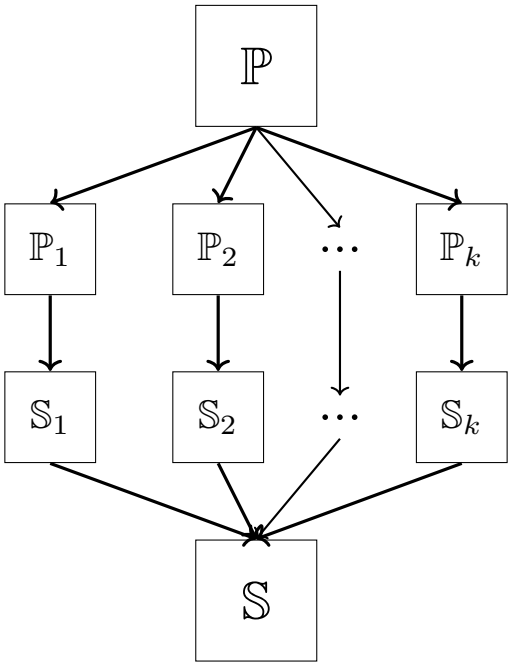
Mohammed Rizin
Unemployed

March 24, 2025

Abstract

Suppose we got a bigger problem, call it P of size n. Sometimes bigger problem are really daunting and we end up doing nothing. Instead, why don't we split the problem into multiple pieces. and solve them and combine the result. Its like we split the work into several pieces and assign to workers independently. So, each worker doesn't feel so hard about solving the problem. So we employ *DivideAndConquer* strategy.

1 Introduction



As discussed earlier, we are dividing a task into several pieces and solve them individually and independently.

1.1 Conditions for Divide and Conquer

1. All the sub-problems should be the same task as the main problem.
2. There should be an known method to combine the solutions at least

Since all the sub-problems are the same task as the parent problem, This is a recursive algorithm.

1.2 Applications of Divide and Conquer

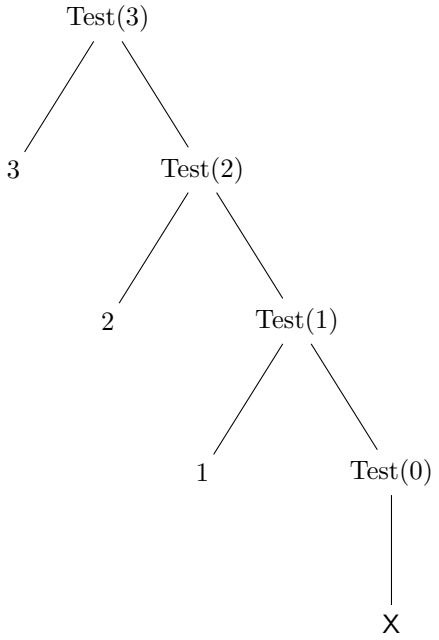
1. Binary search
2. Finding Minimum and Maximum
3. Merge Sort
4. Quick Sort
5. Stressen's Matrix Multiplication

2 Recurrence Relation

To analyze the time complexity of divide and conquer algorithms, we often use recurrence relations. For example:

```
void Test(int n){
    if (n > 0){
        printf("%d", n);
        Test(n-1);
    }
}

Test(3)
```



Since the only statement in the function is printf, which take $O(1)$ time, at each recursive step $O(1)$ is done. If $Test(n)$ is executed, the number of recursive calls are $n + 1$, the number of printf calls are n. So the time complexity of this algorithm is $On(n)$
This won't be possible in every algorithm to open the function and count them and make assumption about n cases. Mathematically, That is not sufficient. Thus, Recurrence Relation Come into play.
Lets take the same code we used before:

Algorithm 2.1 Simple Printing with recursion

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        print(n)                                                ▷ O(1)
        Test(n − 1)                                         ▷ T(n − 1)
    end if
end procedure
```

So,

$$T(n) = \begin{cases} T(n-1)+1 & n > 0, \\ 1 & n = 0 \end{cases}$$

$\because \quad T(n) = T(n-1) + 1,$

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

Substituting(3) in (1),

$T(n) = [T(n-2)+1] + 1 = T(n-2) + 2,$

$T(n) = [T(n-3)+1] + 2 = T(n-3) + 3,$

$T(n) = \quad \vdots \quad + \quad \vdots,$

$T(n) = T(n-k) + k,$

$\because \quad T(0) = 1,$

if (k = n),

$T(n) = T(0) + n,$

$T(n) = 1 + n.$

This algorithm is $O(n)$

Lets look Another example

Algorithm 2.2 Recursion with simple For Loop

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        for i ← 0 to n − 1 do                  ▷ n − 1
            print(n)                           ▷ Everything inside for will be T(n) = n
        end for
        Test(n − 1)                            ▷ T(n − 1)
    end if
end procedure
```

So,

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

We need to take Asymptotic Notation

$$T(n) \simeq T(n-1) + n$$

$$T(n) = \begin{cases} T(n-1)+n & n > 0, \\ 1 & n = 0 \end{cases}$$

$\because \quad T(n) = T(n-1) + n,$

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

Substituting(3) in (1),

$T(n) = [T(n-2) + n - 1] + n = T(n-2) + (n-1) + n,$

$T(n) = [T(n-3) + n - 2] + (n-1) + n = T(n-3) + (n-2) + (n-1) + n,$

$T(n) = \quad \vdots \quad + \quad \vdots,$

$T(n) = T(n-k) + (n-(k-1)) + \cdots + (n-1) + n,$

$\because \quad T(0) = 1,$

if (k = n),

$$T(n) = T(0) + \frac{n(n+1)}{2} = 1 + \frac{n(n+1)}{2},$$

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

This algorithm is $O(n^2)$

Lets look Another example

Algorithm 2.3 Recursion with for loop **Step** $= 2 \cdot i$

```
procedure TEST(int n)                                   ▷ Assume it takes T(n)
    if n > 0 then
        for i ← 0 to n − 1 step 2 · i do                ▷ This is log(n) + 1 Not useful.
            print(n)                                    ▷ This takes log n
        end for
        Test(n − 1)                                     ▷ T(n − 1)
    end if
end procedure
```

Although I know we excluded many of the time complexities simply because we wont using it in asymptotic equation

After taking Asymptotic Notation

$$T(n) = T(n-1) + \log n \tag{1}$$

$$T(n) = \begin{cases} T(n-1) + \log n & n > 0, \\ 1 & n = 0 \end{cases} \tag{2}$$

$$\because \quad T(n) = T(n-1) + \log n$$

$$T(n-1) = T(n-2) + \log(n-1) \tag{3}$$

Substituting eqn (3) in (1),

$$\begin{aligned} T(n) &= [T(n-2) + \log(n-1)] + \log(n) \\ &= T(n-2) + \log(n-1) + \log n \\ &= T(n-2) + \log(n^2 - n), \end{aligned}$$

$$\begin{aligned} T(n) &= [T(n-3) + n - 2] + (n-1) + n \\ &= T(n-3) + \log(n-2) + \log(n-1) + \log n \\ &= T(n-3) + \log((n-2) \cdot (n-1) \cdot (n)) \end{aligned}$$

$$T(n) = T(n-k) + \log\left(\prod_{i=0}^{k-1}(n-i)\right) \tag{4}$$

$$T(n) = T(n-k) + \log\left(\frac{n!}{n-(k-2)!}\right)$$

$$\because \quad T(0) = 1,$$

if (k = n),

$$T(n) = T(0) + \log\left(\frac{n!}{2}\right) = 1 + \cdot \left(\frac{n!}{2}\right)$$

$$T(n) \approx \log n! \approx \log n!$$

Upper bound of $\log n!$ is $O(n \cdot \log n)$

This algorithm is $O(n \cdot \log n)$

$$OR$$

From eqn (4) :

$$T(n) = T(n-k) + \log\left(\prod_{i=0}^{k-1}(n-i)\right)$$

By Binomial Theorem,

$$T(n) = T(n-k) + \log(n^k + \cdots)$$

$$T(n) \approx T(n-k) + k \cdot \log n$$

$$\because \quad T(0) = 1,$$

if (k = n),

$$T(n) = T(0) + n \cdot logn$$

This algorithm is $O(n \cdot \log n)$

By observing above examples, we can observe that for any decreasing recurrence relation function i.e. $(T(n) =$

$T(\underbrace{(n-1)}_{Decreasing} + anything)$ is $O(n \cdot anything)$

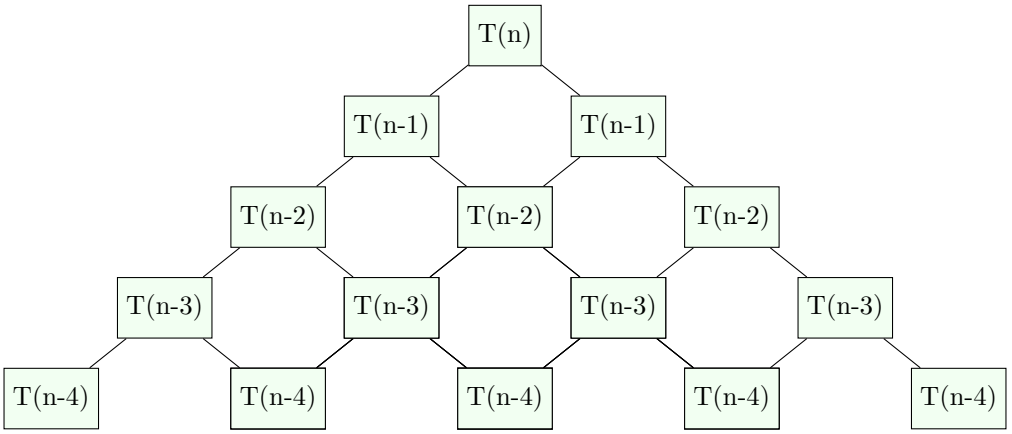
$T(n) = T(n-2) + 1 = 1 + \frac{n}{2} \longrightarrow O(n)$

$T(n) = T(n-100) + n = 1 + \frac{n(n+1)}{200} \longrightarrow O(n^2)$

But what if $T(n) = 2 \cdot T(n-1) + 1$:

Algorithm 2.4 Multiple Recursion

```
procedure TEST(int n)                          ▷ Assume it takes T(n)
    if n > 0 then
        print(n)                                                ▷ O(1)
        Test(n − 1)                                         ▷ T(n − 1)
        Test(n − 1)                                         ▷ T(n − 1)
    end if
end procedure
```



The algorithm time depends on the number of level. At a certain level k the time taken is 2^k. Then $T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

Assume $n = k$, Then it is $2^{n+1} - 1$. That is $O(2^n)$.

$$T(n) = \begin{cases} 2 \cdot T(n-1) + 1 & n > 0, \\ 1 & n = 0 \end{cases}$$

$\because \quad T(n) = T(n-1) + 1,$

$T(n-1) = 2 \cdot T(n-2) + 1,$

Substituting(3) in (1),

$T(n) = 2 \cdot [2 \cdot T(n-2) + 1] + 1 = 4 \cdot T(n-2) + 3,$

$T(n) = 4 \cdot [2 \cdot T(n-3) + 1] + 3 = 8 \cdot T(n-3) + 7,$

$T(n) = \quad \vdots \quad + \quad \vdots,$

$T(n) = 2^k \cdot T(n-k) + 2^k - 1,$

$\because \quad T(0) = 1,$

if (k = n),

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

$T(n) = 2^{n+1} - 1.$

This algorithm is $O(2^n)$

3 Master Theorem for Decreasing

Theorem 3.1. *If $T(n) = a \cdot T(n-1) + f(n)$ where $a > 0$, $b > 0$, and $f(n) = O(n^k)$ where $k \geq 0$, then:*

$$\begin{cases} O(n^k) \text{ or } O(f(n)) & \text{if } 0 < a < 1, \\ O(n^{k+1}) \text{ or } O(n \cdot f(n)) & \text{if } a = 1, \\ O(n^k \cdot a^{\frac{n}{b}}) \text{ or } O(a^{\frac{n}{b}} \cdot f(n)) & \text{if } a > 1. \end{cases}$$

Example 3.1.1. $T(n) = T(n-1) + 1 \to O(n)$

Example 3.1.2. $T(n) = T(n-1) + n \to O(n^2)$

Example 3.1.3. $T(n) = T(n-1) + \log n \to O(n \cdot \log n)$

Example 3.1.4. $T(n) = 2 \cdot T(n-1) + 1 \to O(2^n)$

Example 3.1.5. $T(n) = 2 \cdot T(n-1) + 1 \to O(3^n)$

Example 3.1.6. $T(n) = 3 \cdot T(n-1) + n \to O(n \cdot 3^n)$

4 Recurrence Relation in Dividing Function.

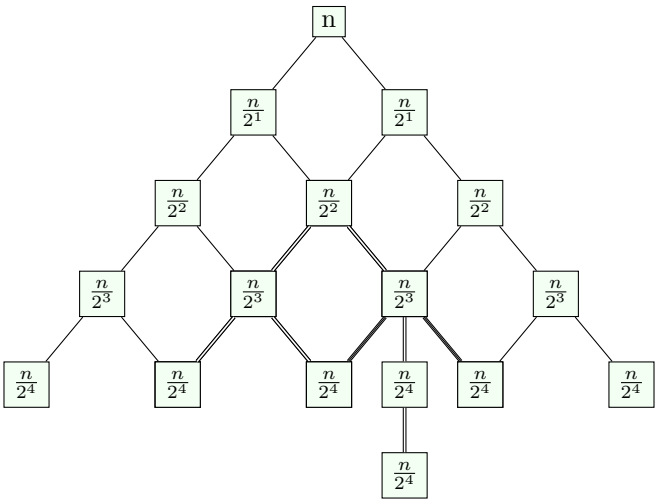
So far we have seen decreasing function and master theorem for decreasing function as well. But what if the value is down by a ratio inside the same function.
If the ratio r is greater than or equal to 1 then This function would go infinity (i.e. we multiply it with positive ratio, and the value never stops). So we use a ratio that $0 < r < 1$, i.e. we divide by some value.
Lets see some examples:

Example 4.0.1. $T(n) = 2 \cdot T(n-1) + 1$ **:**

Algorithm 4.1 Recursion with Division

```
procedure TEST(int n)                                   ▷ T(n)
    if n > 0 then
        for i ← 0 to n − 1 do
            print(n)                                    ▷ O(n)
        end for
        Test(n/2)                                       ▷ T(n/2)
        Test(n/2)                                       ▷ T(n/2)
    end if
end procedure                                           ▷ T(n) = T(n/2) + 1
```

Let us draw the tree: In this tree we are only including the time taken for the purpose of viewing it



If we add the time in each level:

$$\begin{aligned}
&\text{Level 1}: n \\
&\text{Level 2}: \frac{n}{2}+\frac{n}{2}=n \\
&\text{Level 3}: \frac{n}{4}+\frac{n}{4}+\frac{n}{4}+\frac{n}{4}=n \\
&\text{Level 4}: \frac{n}{8}+\frac{n}{8}+\frac{n}{8}+\frac{n}{8}+\frac{n}{8}+\frac{n}{8}+\frac{n}{8}+\frac{n}{8}=n
\end{aligned}$$

Lets Solve it! As we go down we find that at each level it takes n time. The Time Function of this algorithm is :

$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+n & n>1\end{cases} \tag{1}$$

The equation (1) after k step would have $\frac{n}{2^k}$, and there would be k leafs in the tree. As this tree goes on and on. Finally, it will terminate after T(1). Assume $T(\frac{n}{2^k})=T(1)$

$$\begin{aligned}
\frac{n}{2^k}&=1\\
n&=2^k\\
k&=\log_2 n
\end{aligned}$$

So, the total height of the tree is $\log n$ and the time taken in each level is n.
So the total time $T(n)=n\cdot\log n$
So this algorithm have $O(n\cdot\log n)$ □

Let's Solve it by Recurrence Relation:
When we look at equation 1,

$$T(n)=2\cdot T\left(\frac{n}{2}\right)+n \tag{1}$$

$$T\left(\frac{n}{2}\right)=2\cdot T\left(\frac{n}{2}\cdot\frac{n}{2}\right)+\frac{n}{2}=2\cdot T\left(\frac{n}{2^2}\right)+\frac{n}{2} \tag{2}$$

Substituting equation (2) in (1)

$$T(n)=2\cdot[2\cdot T\left(\frac{n}{2^2}\right)+\frac{n}{2}]+n$$

$$T(n)=2^2\cdot T\left(\frac{n}{2^2}\right)+n+n \tag{3}$$

Further if we find $T\left(\frac{n}{2^2}\right)$

$$T(n)=2^3\cdot T\left(\frac{n}{2^3}\right)+3n \tag{3}$$

$$T(n)=\vdots+\vdots \text{ After k steps}$$

$$T(n)=2^k\cdot T\left(\frac{n}{2^k}\right)+kn \tag{4}$$

Assume $\frac{n}{2^k}=1$

$$n=2^k \tag{5}$$

$$k=\log_2 n \tag{6}$$

Subtituting (6)(5) in equation (4)

$$T(n)=n\cdot T(1)+n\cdot\log n$$

$$T(n)=n+n\cdot\log n$$

So the algorithm has a time complexity of $O(n\log n)$

5 Master's Theorem for Dividing Functions

Theorem 5.1. *Master's Theorem for Dividing Functions If* $T(n)=a\cdot T\left(\frac{n}{b}\right)+f(n)$ *where* $a>1$, $b>1$, *and*

$f(n) = O(n^k \cdot \log^p n)$ *where* $k \geq 0$*, then:*

$$\begin{cases} O(n^{\log_b a}) & \log_b a > k, \\ \begin{cases} O(n^k \cdot \log^{p+1} n) & p > -1 \\ O(n^k \cdot \log(\log n)) & p = -1 \\ O(n^k) & p < -1 \end{cases} & \log_b a = k \\ \begin{cases} O(n^k \cdot \log^p n) & p \geq 0 \\ O(n^k) & p < 0 \end{cases} & \log_b a < k \end{cases}$$

Example 5.1.1. $T(n) = 2T(\frac{n}{2}) + 1$

.

$$a = 2 \tag{1}$$
$$b = 2 \tag{2}$$
$$f(n) = \Theta(1)$$
$$f(n) = \Theta(n^0 \cdot \log^0 n) \tag{4}$$

From equation (4),
k = 0
p = 0
Substituting Values from (1) and (2)

$$\log_2 2 = 1 \tag{5}$$
$$k = 0$$
$$Hence, \log_b a > k \text{ (Case 1)}$$
Substituting values from 1 in case equation
Hence $\Theta(n^1)$

□

Example 5.1.2. $T(n) = 4T(\frac{n}{2}) + n$

.

$$\log_2 4 = 2 \tag{1}$$
$$k = 1$$
$$p = 0$$
$$Hence, \log_b a > k \text{ (Case 1)}$$
Substituting values from 1 in case equation
Hence $\Theta(n^2)$

□

Example 5.1.3. $T(n) = 8T(\frac{n}{2}) + n^2$

.

$$\log_2 8 = 3 \tag{1}$$
$$k = 2$$
$$p = 0$$
$$Hence, \log_b a > k \text{ (Case 1)}$$
Substituting values from 1 in case equation
Hence $\Theta(n^2)$

□

Example 5.1.4. $T(n) = 9T(\frac{n}{3}) + n$

.

$$\log_3 9 = 2 \quad (1)$$
$$k = 1$$
$$p = 0$$
$$Hence, \log_b a > k \text{ (Case 1)}$$
$$\text{Substituting values from 1 in case equation}$$
$$\text{Hence } \Theta(n^2)$$

□

Example 5.1.5. $T(n) = 9T(\frac{n}{3}) + n^2$

.

$$\log_3 9 = 2 \quad (1)$$
$$k = 2$$
$$p = 0$$
$$Hence, \log_b a = k \text{ (Case 2)}$$
$$\text{Substituting values from 1 in case equation}$$
$$\text{Hence } \Theta(n^2 \cdot \log n)$$

□

If $\log_b a > k$, then the term $a \cdot T\left(\frac{n}{b}\right)$ dominates the growth of the recurrence relation ($n^{\log_a b}$), as it contributes a higher asymptotic degree compared to $f(n) = O(n^k)$. Consequently, the time complexity of the recurrence is determined by the dominant term, resulting in $T(n) = \Theta(n^{\log_b a})$.

if $\log_b a = k$ and $p > -1$, then the terms $a \cdot T\left(\frac{n}{b}\right)$ and $f(n)$ contribute equally to the growth of the recurrence relation. In this case, the time complexity of the recurrence is determined by the logarithmic factor times $f(n)$. Whatever is the $f(n)$, multiply it by $\log n$.

If $\log_b a = k$ and $\log n$ in the denominator, then it would be take the n term and multiply it with $\log(\log n)$. It would $O(n^k \cdot \log(\log n))$

If $\log_b a = k$ and $\log n$ with power > 1 in the denominator, then it denominator log term would be very negligible. So, we can just remove it. i.e. $\Theta(n^k)$.

If $\log_b a < k$, then the term $a \cdot T\left(\frac{n}{b}\right)$ insignificant to the growth of the recurrence relation. $f(n)$ will contribute higher asymptotic degree compared to former term. Consequently, the time complexity of the recurrence is determined by the dominant term, resulting in $T(n) = \Theta(f(n))$ as long as log term in the numerator. i.e. just take the second term.
If log term is in denominator, then we consider it to be negligible. i.e. $\Theta(n^k)$.

6 Recurrence Relation for Root Function

$T(n) = T(\sqrt{n}) + 1$ **:**

Algorithm 6.1 Recursion with Root

```
procedure TEST(int n)                                   ▷ T(n)
    if n > 2 then
        print(n)                                        ▷ O(1)
        Test(√n)                                        ▷ T(√n)
    end if
end procedure                                           ▷ T(n) = T(√n) + 1
```

After taking Asymptotic Notation

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

$$T(n) = \begin{cases} T(\sqrt{n}) + 1 & n > 2, \\ 1 & n = 2 \end{cases} \tag{2}$$

$$\because \quad T(n) = T(\sqrt{n}) + 1$$

$$T(n^{\frac{1}{2}}) = T(n^{\frac{1}{2^2}}) + 1 \tag{3}$$

Substituting eqn (3) in (1),

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

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

$$T(n) = [T(n^{\frac{1}{2^3}}) + 1] + 2$$

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

$$T(n) = T(n^{\frac{1}{2^k}}) + k \tag{4}$$

$$\because \quad T(2) = 1,$$

From eqn (4) :

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

Asumme n is powers of 2 , $n = 2^m$

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

$$Assume \quad 2^1 = 2^{\frac{m}{2^k}}$$

$$1 = \frac{m}{2^k} \rightarrow m = 2^k$$

$$k = \log_2 m$$

$$m = \log_2 n \rightarrow k = \log_2 \log_2 n$$

This algorithm is $O(\log \log n)$

Additionally, You can use the Master's Theorem for Dividing Funciton to solving the above problem.

7 Conclusion

So, if we have a decreasing function $T(n) = T(n-b) + f(n)$. The term $T(n-b)$ which has $O(n)$ time complexity. The total time complexity will be multiplication of time complexities of both the terms.

if there is coefficient for the first term, the time complexity will no longer be just n, going down each level, the time is doubling from the previous term. If b ¿ 1, only some values under n will be explored. The number of steps would be $\frac{n}{b}$. if a ¿ 1, the doubling happen only $\frac{n}{b}$ times.
But what if a ¡ 1, the first term would add up to 1, according to geometric progression. Time Complexity of second term would be dominant than first or the first term would become 1 as n goes to infinity. $1 \cdot f(n)$ would gives us the total time complexity would be $O(f(n))$. So In general it is the time take taken by first term multiplied by the time taken by second term.

If $\log_b a > k$, then the term $a \cdot T\left(\frac{n}{b}\right)$ dominates the growth of the recurrence relation ($n^{\log_a b}$), as it contributes a higher asymptotic degree compared to $f(n) = O(n^k)$. Consequently, the time complexity of the recurrence is determined by the dominant term, resulting in $T(n) = \Theta(n^{\log_b a})$.

if $\log_b a = k$ and $p > -1$, then the terms $a \cdot T\left(\frac{n}{b}\right)$ and $f(n)$ contribute equally to the growth of the recurrence relation. In this case, the time complexity of the recurrence is determined by the logarithmic factor times $f(n)$. Whatever is the $f(n)$, multiply it by $\log n$.

If $\log_b a = k$ and $\log n$ in the denominator, then it would be take the n term and multiply it with $\log(\log n)$. It would $O(n^k \cdot \log(\log n))$

If $\log_b a = k$ and $\log n$ with power > 1 in the denominator, then it denominator log term would be very

negligible. So, we can just remove it. i.e. $\Theta(n^k)$.

If $\log_b a < k$, then the term $a \cdot T\left(\frac{n}{b}\right)$ insignificant to the growth of the recurrence relation. $f(n)$ will contribute higher asymptotic degree compared to former term. Consequently, the time complexity of the recurrence is determined by the dominant term, resulting in $T(n) = \Theta(f(n))$ as long as log term in the numerator. i.e. just take the second term.
If log term is in denominator, then we consider it to be negligible. i.e. $\Theta(n^k)$.
So In general it is the time take taken by dominant term.