Recurrence Relations

A recurrence relation is a mathematical expression that defines a sequence in terms of its previous terms. In the context of algorithmic analysis, it is often used to model the time complexity of recursive algorithms.

General form of a Recurrence Relation:

an=f(an−1,an−2,....,an−k)an=f(an−1,an−2,....,an−k)

where f is a function that defines the relationship between the current term and the previous terms

Significance of Recurrence Relations

Recurrence Relations play a significant role in analyzing and optimizing the complexity of algorithms. Some of the common uses of Recurrence Relations are:

- Time Complexity Analysis
- Generalizing Divide and Conquer Algorithms
- Analyzing Recursive Algorithms
- Defining State and Transitions for Dynamic Programming.

Common Examples of Recurrence Relations:

- Fibonacci Sequence: F(n) = F(n-1) + F(n-2)
- Factorial of a number n: F(n) = n* F(n-1)
- Merge Sort: T(n) = 2*T(n/2) +O(n)
- Tower of Hanoi: H(n) = 2*H(n-1) +1
- Binary Search: T(n) = T(n/2) +1

Types of Recurrence Relations:

Various types of Recurrence Relations are:

1. **Linear Recurrence Relations:**

Following are some of the examples of recurrence relations based on linear recurrence relation.

T(n) = T(n-1) + n for n > 0 and T(0) = 1

These types of recurrence relations can be easily solved using substitution method.

For example,

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

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

= T(n-k) + (n-(k-1))….. (n-1) + n

Substituting k = n, we get

T(n)=T(0)+1+2+…..+n=n(n+1)/2=O(n2)T(n)=T(0)+1+2+…..+n=n(n+1)/2=O(n2)

2. **Divide and conquer recurrence relations:**

Following are some of the examples of recurrence relations based on divide and conquer.

T(n) = 2T(n/2) + cn

T(n) = 2T(n/2) + √n

These types of recurrence relations can be easily solved using Master Method.

For recurrence relation: T(n) = 2T(n/2) + cn, the values of a = 2, b = 2 and k =1.

Here logb(a)=log2(2)=1=klogb(a)=log2(2)=1=k. Therefore, the complexity will be Θ(nlog2(n))Θ(nlog2(n)).

Similarly for recurrence relation T(n) = 2T(n/2) + √n, the values of a = 2, b = 2 and k =1/2. Herelogb(a)=log2(2)logb(a)=log2(2) = 1 > k. Therefore, the complexity will be Θ(n).

3. **Substitution Recurrences:**

Sometimes, recurrence relations can't be directly solved using techniques like substitution, recurrence tree or master method. Therefore, we need to convert the recurrence relation into appropriate form before solving. For example,

T(n) = T(√n) + 1

To solve this type of recurrence, substitute n = 2m2m as:

T(2m)=T(2m/2)+1LetT(2m)=S(m),S(m)=S(m/2)+1T(2m)=T(2m/2)+1LetT(2m)=S(m),S(m)=S(m/2)+1

Solving by master method, we get

S(m)=Θ(logm)Asn=2morm=log2(n),T(n)=T(2m)=S(m)=Θ(logm)=Θ(loglogn)S(m)=Θ(logm)Asn=2morm=log2(n),T(n)=T(2m)=S(m)=Θ(logm)=Θ(loglogn)

4. **Homogeneous Recurrence Relations:**

A homogeneous recurrence relation is one in which the right-hand side is equal to zero. Mathematically, a homogeneous recurrence relation of order k is represented as:

an=f(an−1,an−2,...,an−k)an=f(an−1,an−2,...,an−k)

with the condition that the above equation equates to 0.

Example: an=2∗an−1−an−2an=2∗an−1−an−2

5. **Non-Homogeneous Recurrence Relations:**

A non-homogeneous recurrence relation is one in which the right-hand side is not equal to zero. It can be expressed as:

an=f(an−1,an−2,...,an−k)+g(n)an=f(an−1,an−2,...,an−k)+g(n)

where g(n) is a function that introduces a term not dependent on the previous terms. The presence of g(n) makes the recurrence non-homogeneous.

Example: an=2∗an−1−an−2+3nan=2∗an−1−an−2+3n

In this case, the term 3n3n on the right-hand side makes the recurrence non-homogeneous.

Ways to Solve Recurrence Relations:

There are mainly three ways of solving recurrences. Please refer analysis of recursion for more details.

1. Substitution Method
2. Recurrence Tree Method
3. Master Method

1. **Substitution Method:**

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

- For example, consider the recurrence T(n) = 2T(n/2) + n.
- We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.
- We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

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

<= 2cn/2Log(n/2) + n

= cnLogn – cnLog2 + n

= cnLogn – cn + n

<= cnLogn

2. **Recurrence Tree Method:**

In this method, we draw a recurrence tree and calculate the time taken by every level of the tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically arithmetic or geometric series.

Consider the recurrence relation, T(n) = T(n/4) + T(n/2) + cn2n2

cn2n2

/ \

T(nn/4) T(nn/2)

If we further break down the expression T(nn/4) and T(nn/2), we get the following recursion tree.

cn2n2

/ \

c(n2n2)/16 c(n2n2)/4

/ \ / \

T(nn/16) T(nn/8) T(nn/8) T(nn/4)

Breaking down further gives us following

cn2n2

/ \

c(n2n2)/16 c(n2n2)/4

/ \ / \

c(n2n2)/256 c(n2n2)/64 c(n2n2)/64 c(n2n2)/16

/ \ / \ / \ / \

To know the value of T(n), we need to calculate the sum of tree nodes level by level. If we sum the above tree level by level, we get the following series T(n) =

cn2(1+5/16+25/256)cn2(1+5/16+25/256) + ….

The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series. We get the sum as (cn2cn2)/(1 – 5/16) which is O(n2n2)

3. **Master Method:**

Master Method is a direct way to get the solution. The master method works only for the following type of recurrences or for recurrences that can be transformed into the following type.

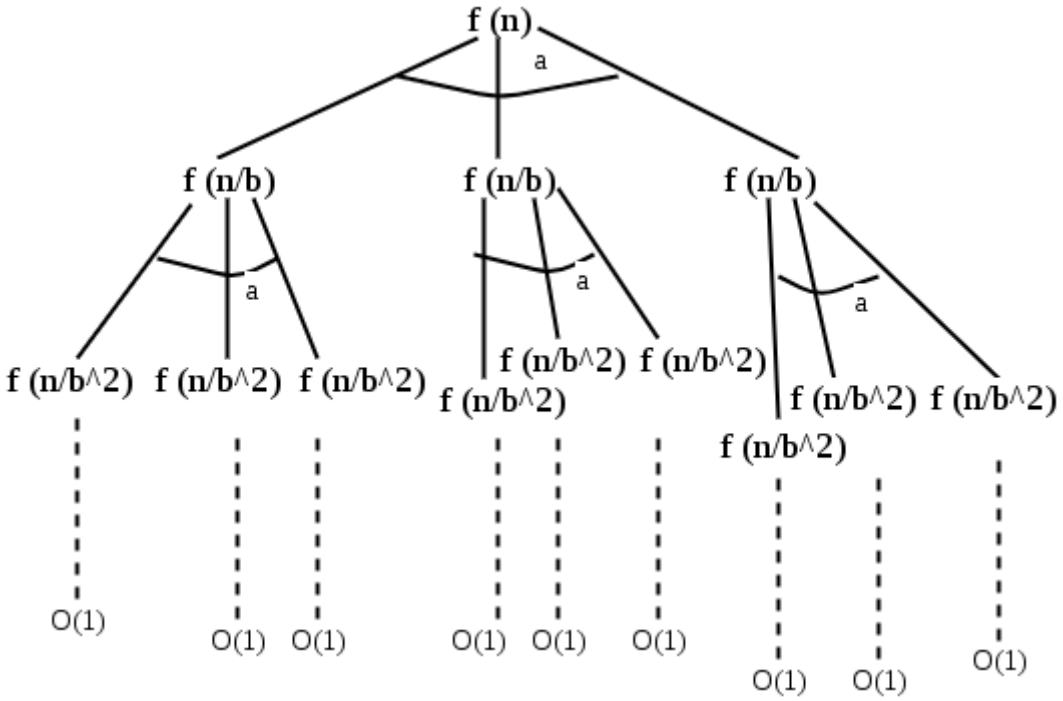
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

There are the following three cases:

- Iff(n)=O(nc)wherec<LogbathenT(n)=Θ(nLogba) Iff(n)=Θ(nc)wherec=LogbathenT(n)=Θ(ncLogn) Iff(n)=Ω(nc)wherec>LogbathenT(n)=Θ(f(n)) Iff(n)=O(nc)wherec<LogbathenT(n)=Θ(nLogba) Iff(n)=Θ(nc)wherec=Logba thenT(n)=Θ(ncLogn) Iff(n)=Ω(nc)wherec>LogbathenT(n)=Θ(f(n))

How does this work?

The master method is mainly derived from the recurrence tree method. If we draw the recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at the root is f(n), and work done at all leaves is Θ(nc) where c is LogbaLogba. And the height of the recurrence tree is LogbnLogbn



In the Recurrence Tree Method, we calculate the total work done. If the work done at leaves is polynomial 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).

Examples of some standard algorithms whose time complexity can be evaluated using the Master Method

- Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and LogbaLogbais also 1. So, the solution is Θ(n Logn)
- Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and LogbaLogb a is also 0. So, the solution is Θ(Logn)