Unit 3 Mathematical Aspects and Analysis of Algorithms

Structure:

3.1 Introduction

In the earlier unit you were introduced to the concepts of analysis framework. In this unit you will be learning about the basic concepts of mathematical analysis of algorithms.

It is essential to check the efficiency of each algorithm in order to select the best algorithm. The efficiency is generally measured by calculating the time complexity of each algorithm. The shorthand way to represent time complexity is asymptotic notation.

For simplicity, we can classify the algorithms into two categories as:

- Non recursive algorithms
- Recursive algorithms

We compute non-recursive algorithm only once to solve the problem.

In this unit, we will mathematically analyze non-recursive algorithms.

Objectives:

After studying this unit you should be able to:

- explain the types of asymptotic notations
- list the basic asymptotic efficiency classes
- describe the efficient analysis of non-recursive algorithms with illustrations

3.2 Asymptotic Notations and Basic Efficiency Classes

To choose the best algorithm we need to check the efficiency of each algorithm. Asymptotic notations describe different rate-of-growth relations between the defining function and the defined set of functions. The order of growth is not restricted by the asymptotic notations, and can also be expressed by basic efficiency classes having certain characteristics.

Let us now discuss asymptotic notations of algorithms

3.2.1 Asymptotic notations

Asymptotic notation within the limit deals with the behavior of a function, i.e. for sufficiently large values of its parameter. While analyzing the run time of an algorithm, it is simpler for us to get an approximate formula for the run-time.

The main characteristic of this approach is that we can neglect constant factors and give importance to the terms that are present in the expression (for T(n)) dominating the function's behavior whenever n becomes large. This allows dividing of un-time functions into broad efficiency classes.

To give time complexity as "fastest possible", "slowest possible" or "average time", asymptotic notations are used in algorithms. Various notations such as Ω (omega), Θ (theta), O (big o) are known as asymptotic notations.

Big Oh notation (O)

'O' is the representation for big oh notation. It is the method of denoting the upper bound of the running time of an algorithm. Big Oh notation helps in calculating the longest amount of time taken for the completion of algorithm.

A function T(n) is said to be in O(h(n)), denoted as $T(n) \in O(h(n))$, if T(n) is bounded above by some constant multiple of h(n) for all large n, i.e., if there exist some positive constant C and some non negative integer n_0 such that $T(n \leq C\ h(n)$ for all $n \geq n_0$.

The graph of C h(n) and T(n) can be seen in the figure 3.1. As n becomes larger, the running time increases considerably. For example, consider $T(n)=13n^3+42n^2+2n\log n+4n$. Here as the value on n increases n^3 is much larger than n^2, nlogn and n. Hence it dominates the function T(n) and we can consider the running time to grow by the order of n^3. Therefore it can be written as $T(n)=O(n^3)$. The value of n for T(n) and C h(n) will not be less than n_0.Therefore values less than n_0 are considered as not relevant.

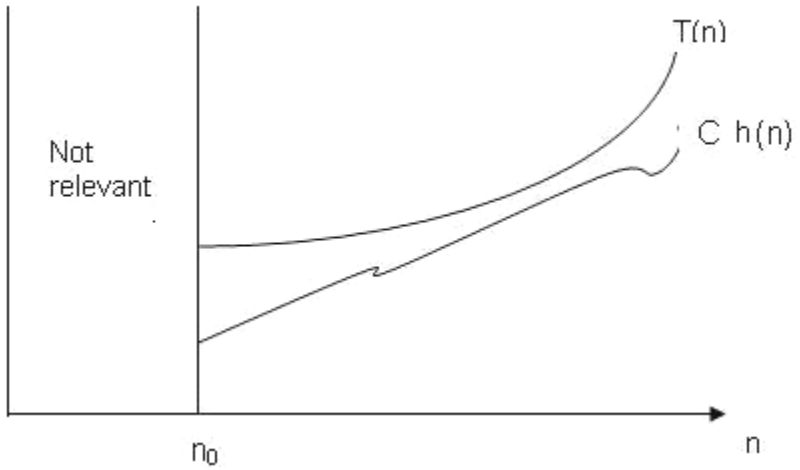

Figure 3.1: Big Oh Notation T(n) Є O(h(n))

Example:

Consider function $T(n)=1n+2$ and $h(n)=n^2$. Determine some constant C so that $t(n) \leq C*h(n)$.

$T(n)=n+2$ and $h(n)=n^2$, find C for n=1, Now

$T(n)=n+2$

$=(1) + 2$

$T(n)=3$ and $h(n)=n^2=1^2=1$

$T(n)>h(n)$

If n=3

Then $T(n)=(3)+2 = 5$ and $h(n)=n^2=9$

$T(n)<h(n)$

Hence for n>2, T(n)<h(n), Therefore Big Oh notation always gives the existing upper bound.

Let us next discuss another asymptotic notation, the omega notation.

Omega notation (Ω)

'Ω' is the representation for omega notation. Omega notation represents the lower bound of the running time of an algorithm. This notation denotes the shortest amount of time that an algorithm takes.

A function T(n) is said to be in Ω(h(n)), denoted as T(n)∈ Ω(h(n)), if T(n) is bounded below by some constant multiple of h(n) for all large n, i.e., if there exist some positive constant C and some non-negative integer n0 such that T(n) ≥ C h(n) for all n ≥ n0.

The graph of C h(n) and T(n) can be seen in the figure 3.2.

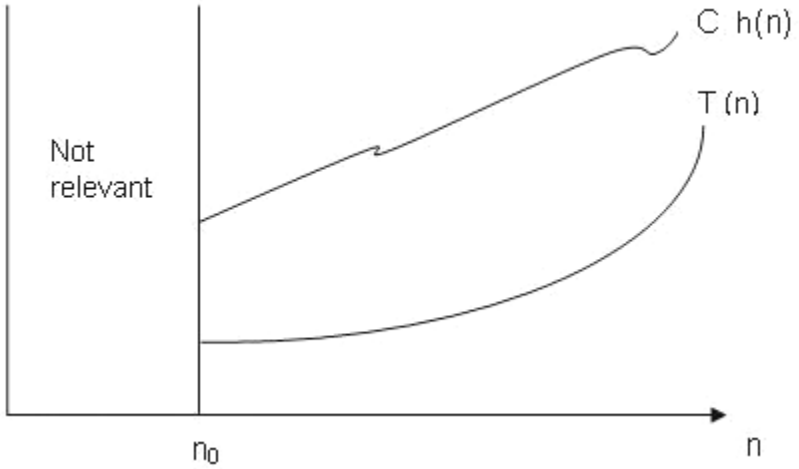

Figure 3.2: Omega Notation T(n) Є Ω(h(n)

Example:

Consider $T(n)=3n^2+2$ and $h(n)=5n$

Then assume n = 0

If n=0, T(n)=2 h(n)=0 i.e. T(n)>h(n)

Assume n=1

Then T(n)=5 and h(n)=5 i.e. T(n)=h(n)

Assume n=3

Then T(n)=29 and h(n) = 15 i.e. T(n)>h(n)

Therefore for n>3, T(n)>C h(n)

Theta notation (Θ)

The depiction for theta notation is 'Θ'. This notation depicts the running time between the upper bound and lower bound.

A function T(n) is said to be in Θ(h(n)), denoted as $T(n) \in \Theta(h(n))$, if T(n) is bounded both above and below by some positive constant multiples of h(n) for all large n, i.e., if there exist some positive constant C_1 and C_2 and some non-negative integer n_0 such that $C_2h(n) \leq T(n) \leq C_1h(n)$ for all $n \geq n_0$

The graph of C_1 h(n), C_2 h(n) and T(n) can be seen in the figure 3.3.

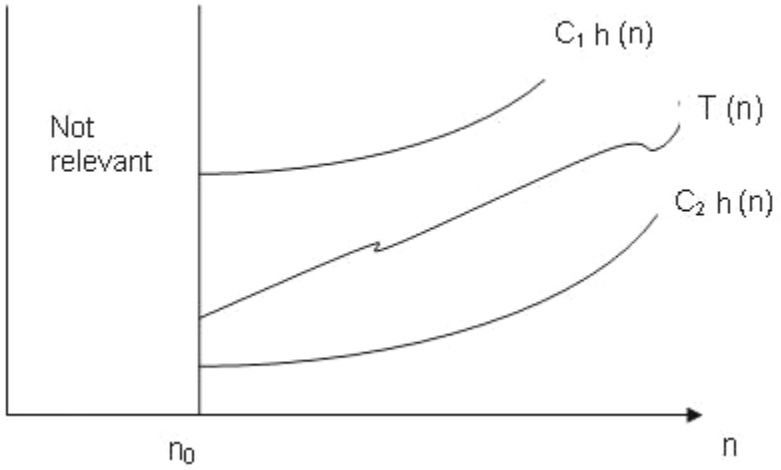

Figure 3.3: Theta Notation T(n) Є Θ(h(n)

Example:

Assume T(n)=2n+5 and h(n)=4n similarly T(n)=2n+5 and h(n)=6n

Where n≥2 so now 4n<2n+5<6n

Here C_1=4, C_2=6 and n_0=2

Theta notation is more accurate than Big Oh notation and Omega notation.

We will next discuss some rules that help us to analyze algorithms.

The maximum rule

Maximum rule is a useful tool that proves that one function is in the order of another function.

Let F, h: N → RZ^0 be two arbitrary functions. The maximum rule says that O(F(n)+h(n))=O(max (F(n), h(n))).

Let p, q: $N \rightarrow RZ^0$ be labeled for each natural number n by $p(n)=F(n) +h(n)$ and $q(n)=\max (F(n), h(n))$, and let an arbitrary function $T: N \rightarrow RZ^0$ be defined then the maximum rule states that

$T(n) \in O(p(n))$ if and only if $T(n) \in O(q(n))$.

Maximum rule is applied for even Theta notation.

Example:

Consider an algorithm that progress in three steps: initialization, processing and finalization.

These steps take time in $\Theta(n^2)$, $\Theta(n^3)$ and $\Theta(n\log n)$ respectively. It is therefore clear that to complete the algorithm, time taken is $\Theta(n^2+n^3+n\log n)$. From the maximum rule

$$\begin{aligned}\Theta(n^2 + n^3 + n \log n) &= \Theta(\max (n^2, n^3+ n\log n)) \\ &= \Theta(\max (n^2, \max (n^3, n\log n))) \\ &= \Theta(\max (n^2, n^3)) \\ &= \Theta(n^3)\end{aligned}$$

From the above example, it is clear that the maximum rule simplifies to any finite constant number of functions. We do not use this rule if some of the functions often tend to be negative. When the rule is used then the risk reasoning will be as below:

$\Theta(n) = \Theta(n + n^2 - n^2) = \Theta(\max (n, n^2 - n^2) = \Theta(n^2)$

Example:

Consider the function

$$\begin{aligned}\Theta(T (n)) &= \Theta(\max (12 n^3 \log n, 6 n^2, \log^2 n, 36) \\ &= \Theta(12 n^3 \log n) \\ &= \Theta(n^3 \log n)\end{aligned}$$

The above reasoning is not correct as the function $6n^2$ is always negative. The following reasoning is correct:

$$\begin{aligned}\Theta(T (n)) &= \Theta(\max (11n^3 \log n + n^3 \log n - 6n^2 + \log^2 n + 36) \\ &= \Theta(\max (11n^3 \log n, n^3 \log n - 6n^2, \log^2 n, 36) \\ &= \Theta(11n^3 \log n) \\ &= \Theta(n^3 \log n)\end{aligned}$$

Even if $n^3 \log n - 6n^2$ is negative for small values of n, for large values of n (i.e. n≥4), it is always non-negative.

The limit rule

This powerful rule states that, given arbitrary functions F and h: N → RZ^0

1. If $\lim_{n\to\infty} \frac{F(n)}{h(n)} \in$ R+ then $F(n) \in \Theta(h(n))$

2. If $\lim_{n\to\infty} \frac{F(n)}{h(n)} = 0$ then $F(n) \in O(h(n))$ but $F(n) \notin \Theta(h(n)$

3. If $\lim_{n\to\infty} \frac{F(n)}{h(n)} = \infty$ then $F(n) \in \Omega(h(n))$ but $F(n) \notin \Omega(h(n))$

The L'Hôpital's rule

L'Hôpital's rule uses derivatives to help compute limits with indeterminate forms. Application of the rule converts an indeterminate form to a determinate form, allowing easy computation of the limit.

In simple cases, L'Hôpital's rule states that for functions F(n) and h(n), if:

$$\lim_{n\to x} F(n) = \lim_{n\to x} h(n) = 0 \quad \text{or} \quad \lim_{n\to x} F(n) = \lim_{n\to x} h(n) = \pm\infty$$

then:

$$\lim_{n\to x} \frac{F(n)}{h(n)} = \lim_{n\to x} \frac{F'(n)}{h'(n)}$$

where F'(n) and h'(n) are the derivatives of F(n) and h(n) respectively.

Example:

Let F(n)=log n and h(n)=$\sqrt{n}$ be two functions. Since both F(n) and h(n) tend to infinity as n tends to infinity, de l'Hôpital's rule is used to compute

$$\lim_{n\to\infty} \frac{F(n)}{h(n)} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to 1} \frac{1/n}{1/2\sqrt{n}} = \lim_{n\to\infty} \frac{2}{\sqrt{n}}$$

Therefore it is clear that $F(n) \in O(h(n))$ but $F(n) \notin \Theta(h(n))$.

Conditional asymptotic notation

When we start the analysis of algorithms it becomes easy for us if we restrict our initial attentions to instances whose size is satisfied in certain conditions (like being a power of 2). Consider n as the size of integers to be

multiplied. The algorithm moves forward if n=1, that needs microseconds for a suitable constant 'a'. If n>1, the algorithm proceeds by multiplying four pairs of integers of size n/2 (or three if we use the better algorithm). To accomplish additional tasks it takes some linear amount of time.

This algorithm takes a worst case time which is given by the function $T: N \rightarrow R^{\geq 0}$ recursively defined where R is a set of non negative integers.

$T(1) = a$

$T(n) = 4T([n/2]) + bn$ for $n > 1$

Conditional asymptotic notation is a simple notational convenience. Its main interest is that we can eliminate it after using it for analyzing the algorithm. A function $F: N \rightarrow R^{\geq 0}$ is non decreasing function if there is an integer threshold n_0 such that $F(n) \leq F(n+1)$ for all $n \geq n_0$. This implies that mathematical induction is $F(n) \leq F(m)$ whenever $m \geq n \geq n_0$.

Consider $b \geq 2$ as an integer. Function F is b-smooth if it is non-decreasing and satisfies the condition $F(bn) \in O(F(n))$. Other wise, there should be a constant C (depending on b) such that $F(bn) \leq C\ F(n)$ for all $n \geq n_0$. A function is said to be smooth if it is b-smooth for every integer $b \geq 2$.

Most expected functions in the analysis of algorithms will be smooth, such as log n, nlogn, n^2, or any polynomial whose leading coefficient will be positive. However, functions those grow too fast, such as nlogn, 2n or n! will not be smooth because the ratio F(2n)/F(n) is unbounded.

Which shows that $(2n)^{\log (2n)}$ is not approximate of $O(n^{\log n})$ because a constant never bounds $2n^2$. Functions that are bounded above by a polynomial are usually smooth only if they are eventually non-decreasing. If they are not eventually non-decreasing then there is a probability for the function to be in the exact order of some other function which is smooth. For instance, let b(n) represent the number of bits equal to 1 in the binary expansion of n, for instance b(13) = 3 because 13 is written as 1101 in binary. Consider F(n)=b(n)+log n. It is easy to see that F(n) is not eventually non-decreasing and therefore it is not smooth because $b(2^k - 1) = k$ whereas $b(2^k) = 1$ for all k. However $F(n) \in \Theta(\log n)$ is a smooth function.

A constructive property of smoothness is that if we assume f is b-smooth for any specific integer $b \geq 2$, then it is actually smooth. To prove this, consider

any two integers a and b (not smaller than 2). Assume that f is b-smooth. It is important to show that f is a-smooth as well. Consider C and n_0 as constants such that F(bn)≤C F(n) and F(n)≤F(n+1) for all n≥n_0. Let i= [$\log_b a$]. By definition of the logarithm $a = b^{\log_b a} \leq b^{\log_b a} = b^i$.

Consider n≥n_0. It is obvious to show by mathematical induction from b-smoothness of F that F(b^in)≤C^i F(n). But F(an)≤F(b^in) because F is eventually non-decreasing and approximate to b^in ≥ an ≥ n_0. It implies that F (an) ≤ CˆF(n) for Cˆ=C^i, and therefore F is a-smooth.

Smoothness rule

Smooth functions seem to be interesting because of the smoothness rule. Consider F: N → $R^{\geq 0}$ (where $R^{\geq 0}$ is non negative integers) as a smooth function and T: N → $R^{\geq 0}$ as an eventually non-decreasing function. Consider an integer where b≥2. The smoothness rule states that T(n)∈Θ(F(n)) whenever T(n)∈Θ(F(n) | n is a power of b). We apply this rule equally to O and Ω notation. The smoothness rule assumes directly that T(n)∈Θ(n^2) if n^2 is a smooth function and T(n) is eventually non-decreasing. The first condition is immediate since the function is approximate to n^2 (which is obviously nondecreasing) and $(2n)^2=4n^2$. We can demonstrate the second function from the recurrence relation using mathematical induction. Therefore conditional asymptotic notation is a stepping stone that generates the final result unconditionally. i.e. T(n)=Θ(n^2).

Some properties of asymptotic order of growth

- If F1(n)∈O(h1(n)) and F2(n)∈O(h2(n)) , then F1(n)+F2(n)∈O(max{h1(n),h2(n)})
- Implication: We determine the overall efficiency of the algorithm with a larger order of growth.
  - For example, $-6n^2+2n\log n \in O(n^2)$

Asymptotic growth rate

- O(h(n)): Class of function F(n) that grows no faster than h(n)
- Ω(h(n)): Class of function F(n) that grows at least as fast as h(n)
- Θ(h(n)): Class of function F(n) that grows at same rate as h(n)

Comparing growth rate using limits

The limit-based approach is convenient for comparing the growth rate of asymptotic functions.

$\lim_{n\to\infty} \frac{T(n)}{h(n)}$ = 0 (order of growth of T(n)<order of growth of h(n))

$\lim_{n\to\infty} \frac{T(n)}{h(n)}$ = C>0 (order of growth of T(n)=order of growth of h(n))

$\lim_{n\to\infty} \frac{T(n)}{h(n)}$ = ∞ (order of growth of T(n)>order of growth of h(n))

3.2.2 Basic asymptotic efficiency classes

We have previously analyzed that we can obtain a different order of growth by means of constant multiple (C in C*h(n)). We used different types of notations for these orders of growth. But we do not restrict the classification of order of growth to Ω, Θ and O. There are various efficiency classes and each class possesses certain characteristic which is shown in table 3.1.

Table 3.1: Basic Asymptotic Efficiency Classes

Growth order	**Name of the efficiency class**	**Explanation**	**Example**
1	Constant	Specifies that algorithm's running time is not changed with increase in size of the input.	Scanning the elements of array
log n	Logarithmic	For each iteration of algorithm a constant factor shortens the problem's size.	Performing the operations of binary search
N	Linear	Algorithms that examine the list of size n.	Performing the operations of sequential search
n logn	n-log-n	Divide and conquer algorithms.	Using merge sort or quick sort elements are sorted
n^2	Quadratic	Algorithms with two embedded loops.	Scanning the elements of matrix

n^3	Cubic	Algorithms with three embedded loops.	Executing matrix multiplication
2^n	Exponential	Algorithms that generate all the subsets which are present in n – element sets	Generating all the subsets of n different elements
n!	Factorial	Algorithms that generate all the permutations of an n-element set	All the permutations are generated

Activity 1

Determine a constant p for a given function $F(n) \leq p*h(n)$ where $F(n)=2n+3$ and $h(n)=n^2$.

Self Assessment Questions

1. _____ is more accurate than Big Oh notation and Omega notation.
2. _____ asymptotic notation is a simple notational convenience.
3. _____ depicts the running time between the upper bound and lower bound.

3.3 Mathematical Analysis of Non-Recursive Algorithms

In the previous section we studied and analyzed the types of asymptotic notations involved in algorithms and examined the rules and limitations of the algorithms with examples.

In this section we are going to deal with the mathematical analysis of non-recursive algorithms.

We execute non-recursive algorithms only once to solve the problem.

3.3.1 Analyzing efficiency of non recursive algorithms

The steps involved in analyzing the efficiency of non-recursive algorithms are as follows:

- Decide the input size based on the constraint n
- Identify the basic operations of algorithm
- Check the number of times the basic operation is executed. Find whether the execution of basic operation is dependent on input size n or not. If the basic operation is depending on worst case, best case and average case then analysis of algorithm needs more attention.

- Set up summation formula for the number of times the basic operation is implemented.
- Simplify the sum using standard formula and rules.

The commonly used summation rules are listed next.

Summation formula and rules:

1) $\sum_{i=1}^{n} 1 = 1+1+1\text{----------}1 = n \in \Theta(n)$

2) $\sum_{i=1}^{n} i = 1+2+3\text{----------}n = \frac{n(n+1)}{2} \in \Theta(n^2)$

3) $\sum_{i=1}^{n} i^k = 1+2^k+3^k\text{-------}n^k = \frac{n^k(n^k+1)}{k+1} \in \Theta(n^{k+1})$

4) $\sum_{i=1}^{n} a^i = 1+a+\text{ ---------}a^n = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$

5) $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$

6) $\sum_{i=1}^{n} a_i \pm b_i = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$

7) $\sum_{i=k}^{n} 1 = n-k+1$ where n and k are upper and lower limits

Example 1

We will now discuss an example for identifying the element which has the minimum value in a given array.

We can find the element with the help of the general plan. Let us now see the algorithm for finding the minimum element in an array.

Algorithm for finding the minimum element in an array

Algorithm Min_Element (A [0------n-1])

// Input: Array (A [0------n-1])

// Output: Returns the smallest element in the given array

Min_value ←A [0]

```
For i←1 to n-1 do
{
  If (A[i]) <Min_value) then
    Min_value A ← [i]
}
return Min_value
```

Mathematical analysis

Step 1: The total number of elements in the given array is n, i.e. the size of the input is n.

Step 2: The operation done here is comparison in loop in order to find the smaller value.

Step 3: We implement the comparison recurrence of the loop. In this case as comparison is done for each loop there is no need for analysis of best case, worst case and average case.

Step 4: Consider h(n) as the iteration for the comparison to execute. We compare the algorithm is every time the loop gets executed which implies that for each value of i comparison is done. Therefore comparison is made for i=1 to n-1 times.

h(n) = Single comparison made for each i

Step 5: Simplifying the sum i.e.

$$h(n)=\sum_{i=1}^{n-1}1$$

$h(n) = n-1 \in \Theta(n)$ {Using the rule $\sum_{i=1}^{n}1=n$}

Therefore the efficiency of above algorithm is Θ(n)

Let us now trace the algorithm for finding the minimum element in the array.

Algorithm tracing for finding the minimum element in an array

Let us consider n=4

(A [0,1,3,10])// A is an array of elements in which we are going to find the //smallest element

```
Min_value ←A [0]//Min_value=0
For i←1 to 4-1 do// this loop iterates from i=1 to i=4-1
{
  If (A[1]) <Min_value) then// if 1<0
    Min_value← A[1]
}
return Min_value // Min_value = 0
```

Example 2

Let us next discuss an algorithm for counting the number of bits in an integer.

Algorithm for counting the number of bits in an integer

```
Algorithm Binary (p)
// Input: p is the decimal integer
// Output: Number of digits
countA← 1
While (p>1)
{
   countA ← countA+1
   p ← [p/2]
}
Return countA
```

Mathematical analysis:

Step 1: let the size of the input be p.

Step 2: The while loop indicates the basic operation and checks whether p>1. Execution of while loop is done whenever p>1 is true. It gets executed once more when p>1 is false. When p>1 is false the statements inside the while loop is not executed.

Step 3: The value of n gets halved whenever the loop gets repeated. Hence the efficiency of the loop is $\log_2 p$.

Step 4: The total number of times the while loop gets executed is given by $[\log_2 p] + 1$.

Let us now trace the algorithm for counting the number of elements in an integer.

Algorithm tracing for counting the number of bits in an integer

Let us consider p=16// p is a decimal integer
// Input: p is the decimal integer
// Output: Number of bits
countA ← 1
While (p>1)
{
  countA ← countA+1// countA=2 in first iteration of this loop
  p ← [p/2]// p=8 in first iteration of this loop
}
Return countA//the value returned will be 5

3.3.2 Matrix multiplication

In general, to multiply 2 matrices given, the number of rows in the first matrix should be the same as the number of columns in the second matrix. In other words two matrices can be multiplied only if one is of dimension m×n and the other is of dimension n×p where m, n, and p are natural numbers {m,n,p∈N}. The resulting matrix will be of dimension m×p.

A square matrix is one in which the number of rows and columns are the same. We will now discuss the algorithm for matrix multiplication of two square matrices with n elements each.

Algorithm for matrix multiplication:

Algorithm MatrixMultiplication(A[0.. n - 1, 0.. n - 1], B[0.. n - 1, 0.. n - 1])
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two *n*-by-*n* matrices *A* and *B*
//Output: Matrix C = AB
for i = 0 to n - 1 do
  for j = 0 to n - 1 do
C[i, j] = 0
for k = 0 to n - 1 do

```
            C[i, j] = C[i, j] + A[i, k] * B[k, j]
        end
    end
end
return C
```

The tracing for the matrix multiplication algorithm is given below.

Algorithm tracing for matrix multiplication of two 4X4 matrices:

```
Algorithm MatrixMultiplication(A[0.. 3, 0..3 ], B[0..3, 0..3])
//Multiplies two square matrices of order n by the definition-based
algorithm
//Input: Two 4-by-4 matrices A and B
//Output: Matrix C = AB
for i = 0 to 4 - 1 do
    for j = 0 to 4 - 1 do
C[i, j] = 0
for k  0 to 4 - 1 do
      C[0, 0] = C[0,0j] + A[0, 0] * B[0, 0] // this executes for k = 0 to 3
end
    end
end
return C
```

Let us now analyze the steps involved in matrix multiplication:

- Let the input size be n
- Addition and multiplication are the two operations that take place in the innermost loop according to the following rules:
  - Both get executed exactly once for each repetition of the innermost loop
  - Choice is of no importance between two operations for the algorithm's basic operation
  - The total number of basic operation (multiplication) executions are dependent on n

- The basic operation count formula includes:
  - Multiplication is performed once on each repetition of the innermost loop (k) and finding the total number of multiplications are done for all pairs of i and j
  - The algorithm computes n^2 elements of the product matrix. Each element is calculated as the scalar (dot) product of an n-element row of A and an n-element column of B, which accepts n multiplications. Therefore the number of basic operations performed to multiply two matrices of n elements can be given by C(n) which is of the order n^3

    i.e. $$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

Let us next discuss the time efficiency of recursive algorithms.

3.3.3 Time efficiency of non-recursive algorithms

Time efficiency approximation depends on the type of definition that is needed to describe the steps involved in an algorithm. The time required to perform a step should always bound above by a constant. In some instances, count of addition of two numbers might be as one step. In such cases approximation of time efficiency becomes critical. This consideration might not justify certain situations. If the numbers involved in a computation are randomly large, then the time required for performing single addition is no longer considered as constant.

The steps involved in mathematical analysis of nonrecursive algorithms are:

- Decide the parameter n based on the input size
- Identify the basic execution of algorithm
- Determine the worst, average, and best case for the size of input (n)
- Set up a sum for C(n) which represents the loop structure of the algorithm
- Simplify the sum using standard formulas

3.3.4 Tower of Hanoi puzzle

Let us now discuss a non-recursive algorithm for the Towers of Hanoi problem.

Tower of Hanoi or Towers of Hanoi is a mathematical game or puzzle that has three pegs A, B, and C (refer figure 3.4) and a number of disks of

different sizes which can slide onto any peg. Initially, we arrange all the disks in a neat stack in ascending order of size on one peg putting the smallest disc at the top. This makes a conical shape as can be seen in the figure 3.4.

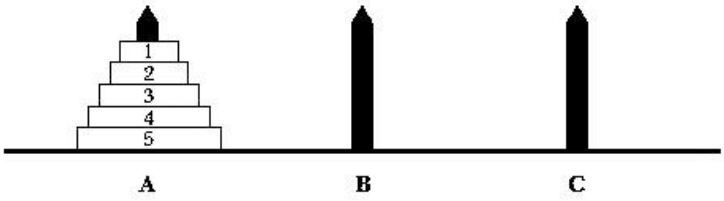

Figure 3.4: The Tower of Hanoi

A French mathematician Eduardo Lucas invented the puzzle in 1883. There is a tale about a Vietnamese temple which comprises of a large room with three time-worn posts in it surrounded by sixty four golden disks. The priests of Hanoi used to move these disks according to the rules of the puzzle during ancient times. The puzzle is therefore known as the Tower of Brahma puzzle. According to the tale, the world will end when the last move of the puzzle is accomplished.

This puzzle is played with any number of disks. The main aim of the puzzle is to move all the disks to another peg according to the following rules:

- We can move only one disk at a time.
- In each move we can lift the upper disk from one of the pegs and slide it onto another peg on top of the other disks which are already present on that peg.
- We need to ensure that no disk is placed on the top of a smaller sized disk.

The list of moves for a tower has much regularity. While counting, the moves that start from 1 and the ordinal of the disk that moves during the move m is divided by 2. Therefore every odd move will have the smallest disk. The non-recursive algorithm solution is simpler than the recursive algorithm.

In varying moves:

- Move the smallest disk to the peg from which it has not recently come from.
- Move the other disk legally (there will be only one option)

With this knowledge, we can recover a set of disks in the middle of an optimal solution having no state information other than the positions of each disk.

- Examining the smallest top disk (that is not disk 0), and noting what will be it's only (legal) move.
- If that move is the disk's 'natural' move, then the disc is not moved since the (last disc 0) move, and that move should be handled.
- If that move is not the disk's 'natural' move, then move disk 0.

Proven statement 1: In minimal length solution of towers of Hanoi puzzle the first move is always with the smallest disk.

Proof: A single move is always combination of two consecutive moves with a smallest ring.

Algorithm for moving the rings in clock wise direction in one post:

If n is odd then d: = clockwise else d: = counterclockwise

Repeat

Move the smallest ring in one post in the direction d till all rings are on same post.

Make the legal move that will not include the smallest ring until all the rings come to the same post.

3.3.5 Conversion of recursive algorithm to non-recursive algorithm

Let us now discuss how to convert recursive algorithms to non-recursive algorithms.

We declare and initialize a stack (recursion stack) on insertion of the code at the beginning of the function. Generally, the stack holds the values of parameters, local variables, and a return address for each recursive call. We use separate stacks for each value. A label of 1 is attached to the first executable statement. If it is a value returning function, then we need to change all appearances of the function name on the left hand side of assignment statements by a new variable (say z) of the similar type as the function.

A set of instructions replaces every recursive call which performs the following:

1. Store the values of all pass by value parameters and local variables in the stack. Declare the pointer to the top of the stack as global.
2. Create i-th new label, ‘i’, and store ‘i’ in the stack. The value i is used as return address in this label.
3. Compute the arguments of this call that relate to pass by value parameters and assign these values to the right formal parameters.
4. Insert an absolute branch to the beginning of the function.
5. If it is a void function then add the label created in step 2 to the statement followed by the unconditional branch. If this statement has already labels then replace it and change all references to it.
6. If it is a value returning function then go according to the absolute branch by code for using the value of the variable z in the same way as the function value was used earlier. The label created in step 2 gives the first statement of the code.
7. Following the above steps removes all recursive calls from the function. Finally, it is the time to head the end statement of the function by code to do the following:
8. Assign the value of z to the function name if the recursion stack is empty. If it is a value returning function, then it computes the value till the end of the function. Void functions are executed till the end of function.
9. Restore all pass by value parameters and local variables if the stack is not empty, then all pass by value parameters and local variables are restored. They will be at the top of the stack by using the return label from the top of the stack and executing a branch to this label. This is done using a case statement.
10. Additionally, any label (if any) attached to the end of the function statement is moved to the code’s first statement for step 8 and 9.

Activity 2

Write an algorithm for counting even number of bits in an integer

Self Assessment Questions

4. Tower of Hanoi is a _____ puzzle.
5. The time required to perform a step should always bound above by a _____.
6. _____ is of no importance between two operations for the algorithm's basic operation.

3.4 Summary

It is very important to obtain the best algorithm for analysis. For selecting the best algorithm, checking the efficiency of each algorithm is essential. The shorthand way for representing time complexity is asymptotic notation.

Asymptotic notation within the limit deals with the character of a function that is a parameter with large values. The main characteristic of this approach is that constant factors are neglected and importance is given to the terms that are present in the expression (for T(n)) dominating the function's behavior whenever n becomes large. This helps in classification of run-time functions into broad efficiency classes. The different types of asymptotic notations are Big Oh notation, Omega notation and Theta notation.

We classify algorithms broadly into recursive and non-recursive algorithms. In this unit we have analyzed non-recursive algorithms mathematically with suitable examples. Non-recursive algorithm is an algorithm which is performed only once to solve the problem.

3.5 Glossary

Term	Description
Recursive algorithm	It is an algorithm which calls itself with smaller inputs and obtains the inputs for the current input by applying simple operations to the returned value of the smaller input.
Runtime	The time when a program or process is being executed is called as runtime.
Notation	It is the activity of representing something by a special system of characters.

3.6 Terminal Questions

1. Explain Big Oh notation with suitable example.
2. Define and explain Theta notation.
3. Explain conditional asymptotic notation with an example.
4. What are the various types of basic efficiency classes?
5. Explain Towers of Hanoi puzzle for non recursive algorithm with an example.
6. How recursive algorithm is converted in to non recursive algorithm?

3.7 Answers

Self Assessment Questions

1. Theta notation
2. Conditional
3. Theta notation
4. Mathematical
5. Constant
6. Choice

Terminal Questions

1. Refer section 3.2.1 – Asymptotic notations
2. Refer section 3.2.1 – Asymptotic notations
3. Refer section 3.2.1 – Asymptotic notations
4. Refer section 3.2.2 – Basic efficiency classes
5. Refer section 3.3.5 – Towers of Hanoi
6. Refer section 3.3.6 – Conversion of recursive algorithm in to non recursive algorithm

References

- A. A. Puntambekar (2008). *Design and Analysis of Algorithms*, First edition, Technical publications, Pune.
- James Andrew Storer. *An Introduction to Data Structures and Algorithms,* Brandies university Waltham, U.S.A.

E-Reference

- http://www.cmpe.boun.edu.tr/~akin/cmpe160/recursion.html
- www.cs.utsa.edu/~bylander/cs3343/chapter2handout.pdf
- www.scuec.edu.cn/jsj/jpkc/algorithm/res/ppt/Lecture03.ppt