FUNDAMENTALS OF ALGORITHM DESIGN AND MACHINE LEARNING

LECTURE SCRIBE NOTES

DATE: 7th JAN, 2025

TOPICS COVERED:

- Exact Time Analysis
  - Overview
  - Examples
- Asymptotic Analysis
  - Overview
  - Big-Oh, Big-Omega, Big-Theta
  - Function class comparison
  - Examples
- Recursive Relations Examples

REESHAV SAMANTA

24BM6JP45

PGDBA BATCH-10

1. Exact Time Analysis

Overview

Exact time analysis involves determining the precise number of fundamental operations executed by an algorithm, rather than just estimating asymptotically. It provides a detailed count of operations like comparisons, iterations, recursive calls, and variable assignments, helping to accurately evaluate the algorithm's efficiency. It helps us to get an idea about the asymptotic time bounds.

Examples

1.

```
sum(A,n){
  s = 0
  for i = 0 to n
    s = s + A[i]
  return s
}
```

This program calculates the sum of all elements in the array *A* of size *n*.

In the solution we will see how many times each line has run.

```
sum(A,n){
  s = 0            Runs 1 time
  for i = 0 to n   Runs (n+1) times
    s = s + A[i]   Runs n times
  return s         Runs 1 time
}
```

- The assignment s = 0 and return statement runs once each.
- *for* loop runs for (n+1) times as there are n+1 comparisons. The present value of *i* is compared with 0 through n. The loop stops when *i = n.*
- The operation inside the *for* loop runs n times as there are n iterations.

Exact time = (1 + (n+1) + n + 1) = 2n + 3

Time Complexity = O(n)

2.

```
add(A,B,n){
  for i = 0 to n
   {
    for j = 0 to n
      c[i,j] = A[i,j] + B[i,j]
   }
}
```

This program calculates the sum of two *n x n* matrices *A* and *B.*

```
add(A,B,n){
  for i = 0 to n                  Runs n+1 times
    for j = 0 to n                Runs n(n+1) times
      c[i,j] = A[i,j] + B[i,j]    Runs n*n times
}
```

- By the same logic in Ex. 1, the first loop runs (n+1) times and everything within that runs n times, making it n(n+1) comparisons in the second loop.
- The final addition operation runs n × n times due to two nested loops.

Exact time = $n^2 + n(n+1) + (n+1) = 2n^2 + 2n + 1$

Time Complexity = $O(n^2)$

3.

```
mul(A,B,n){
 for i = 0 to n                        Runs n+1 times
   for j = 0 to n                      Runs n(n+1) times
     c[i,j] = 0                        Runs n*n times
     for k = 0 to n                    Runs n*n(n+1) times
       c[i,j] = c[i,j] + A[i,k]*B[k,j]  Runs n*n*n times
 }
```

This program calculates the product of two $n \times n$ matrices *A* and *B*.

- There are three nested loops running n times each. This hints that the time complexity will be $O(n^3)$.

Exact time = $n^3 + n^2(n+1) + n^2 + n(n+1) + (n+1) = n^3 + 3n^2 + 2n + 1$

Time Complexity = $O(n^3)$

4.

```
for i = 0 to n    Runs n+1 times
  pf("IITKGP")    Runs n times
```

This program prints "IITKGP" *n* number of times.

Exact time = n + (n+1) = 2n + 1

Time Complexity = O(n)

5.

```
for i = 0 to n, step = 2    Runs n/2 + 1 times
  pf("IITKGP")              Runs n/2 times
```

This program prints "IITKGP" with a step of 2. Thus making it n/2 no. of times.

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

Time Complexity = O(n)

6.

```
p = 0
for (i = 1, p ≤ n, i++)
  p = p + i
```

This program increments the variable *p* by *i,* which is incremental in nature (*i)*.

- The stopping condition is given as *p > n.*

i	*p*
1	1
2	1+2
3	1+2+3
…	…
k	1+2+..+k

p > n
1 + 2 + … + k > n
$\frac{k(k+1)}{2} > n$
$k^2 + k > 2n$
$k^2 > n$
$k > \sqrt{n}$

Time Complexity = O($\sqrt{n}$)

7.

```
for (i = 1, i < n, i = i*2)
  pf("IITKGP")
```

The program keeps printing 'IITKGP' for each iteration of the loop, with the value of *i* doubling in each step until it becomes greater than or equal to *n.*

- The stopping condition is given as *i > n.*
  i = 1 x 2 x 2 x…x 2 ≥ n
  $2^k \geq n$
  $k \geq \log_2 n$
- The *for* loop runs for [$\log_2$ n] no. of times. Here, [.] denotes GIF.

Time Complexity = O($\log_2$ n)

8.

```
for (i = n, i ≥ 1, i = i/2)
  pf("IITKGP")
```

Let us say that the program runs for *k* no. of iterations.

- The stopping condition is given as *i < 1.*

$i = n \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} \times \ldots \times \frac{1}{2} = \frac{n}{2^k} < 1$

$2^k > n$

$k > log_2 n$

Time Complexity = $O(\log_2 n)$

9.

```
for (i = 0, i < n, i = i++)
  for(j = 0, j < i, j++)
     pf("IITKGP")
```

- No. of iterations is given as:

i	j	#iterations
1	0	1
2	0	
	1	2
3	0	
	1	
	2	3
…	…	…
n		n

No. of iterations = $1+2+3+..+n = \frac{n(n+1)}{2}$

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

Time Complexity = $O(n^2)$

10.

```
for (i = 0, i*i < n, i = i++)
  pf("IITKGP")
```

- Stopping condition is given as $i^2 \geq n$, or $i \geq \sqrt{n}$
- No. of iterations = $[\sqrt{n}]$, where [.] denotes GIF

Time complexity = $O(\sqrt{n})$

11.

```
p = 0
for(i = 1, i < n, i = i*2)
  p++                          Runs p = [log2n] times

for(j = 1, j < p, j = j*2)
  pf(" ")                      Runs log2p = log2(log2n) times
```

- The loops are NOT nested and dependant only in terms of the value of *p*.

- We will use the concept used in Q.7. and we will see that $p = \log_2 n$.
- The second loop similarly runs for $\log_2 p$ times.

Exact time = $1 + (p + 1) + p + (\log_2 p + 1) + \log_2 p = 1 + (\log_2 n + 1) + \log_2 n + (\log_2(\log_2 n) + 1) + \log_2(\log_2 n)$

Time Complexity = $O(\log_2 n)$

12.

```
for(i = 0, i < n, i++)          Runs n+1 times
  for(j = 1, j < n, j = j*2)    Runs n*([log2n]+1) times
     pf(" ")                    Runs n*[log2n] times
```

- This involves two nested loops, the outer having arithmetic increments, and the inner having geometric increments, doubling the value of *j* in each iteration until it is greater than or equal to *n*.

Exact time = $2n\log_2 n + 2n + 1$

Time Complexity = $O(n\log_2 n)$

13.

```
a = 1             Runs 1 time
while(a < b)      Runs [log2b]+1 times
  pf("IITKGP")    Runs [log2b] times
  a = a * 2       Runs [log2b] times
```

This program prints "IITKGP" till the value of *a* is less than *b*.

- Suppose there are *k* iterations in the *while* loop.
- Stopping condition:
  $a = 1 \times 2 \times 2 \times \ldots \times 2$ (*k times*) $= 2^k \geq b$
  $k \geq \log_2 b$

Exact time = $3\log_2 b + 2$

Time Complexity = $O(\log_2 b)$

SUMMARY

• for(i = 0, i < n, i++)	O(n)
• for(i = 0, i < n, i = i + 2)	O(n)
• for(i = n, i > 1, i--)	O(n)
• for(i = 1, i < n, i = i*2)	O(log2n)
• for(i = 1, i < n, i = i*3)	O(log3n)
• for(i = n, i > 1, i = i/2)	O(log2n)

2. Asymptotic Analysis

Overview

Asymptotic Analysis helps in evaluating the efficiency of algorithms by analysing their growth rates as the input size *n* approaches infinity. The three key notations are:

1. **Big-Oh (O)**: Upper Bound
2. **Big-Omega (Ω)**: Lower Bound
3. **Big-Theta (Θ)**: Tight/Average Bound

Big-Oh (O) – Upper Bound

Definition: For a given function g(n), O(g(n)) denotes the set of functions f(n) satisfying the following property:

There exist positive constants *c* and n_0 such that:

$0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Represents the upper bound of a function *f(n)*

Example

$4n^2 + 2n + 1 \leq c \cdot (4n^2 + 2n^2 + n^2)$

$f(n) \leq n^2$

$f(n) = O(n^2)$

Big-Omega (Ω) – Lower Bound

Definition: For a given function g(n), Ω(g(n)) denotes the set of functions f(n) satisfying the following property:

There exist positive constants *c* and n_0 such that:

$0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$

Represents the lower bound of a function *f(n)*

Example

$4n + 3 \geq n$

$f(n) \geq n$

$f(n) = \Omega(n)$

Big-Theta (Θ) – Lower Bound

Definition: For a given function g(n), Θ(g(n)) denotes the set of functions f(n) satisfying the following property:

There exist positive constants c_1, c_2 and n_0 such that:

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n > n_0$

Represents the tight (exact) bound of a function *f(n)*

Example

$n^2 \leq 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$

$n^2 \leq f(n) \leq 9n^2$

$f(n) = \Theta(n^2)$

NOTE: If *f(n) = O(g(n))* and *f(n) = Ω(g(n)),* then *f(n) = Θ(g(n))* provided *g(n)* is the same in both *O(g(n))* and *Ω(g(n)).*

Function Classes for Asymptotic Analysis

Function classes in asymptotic analysis are used to categorize functions based on their growth rates as the input size *n* increases. These classes help in understanding the relative efficiency of algorithms by comparing their time or space complexities. Commonly analysed classes include constant (O(1)), logarithmic ($O(\log_2 n)$), linear (O(n)), quadratic ($O(n^2)$), exponential ($O(2^n)$), and factorial (O(n!)) complexities. Understanding these growth patterns allows us to predict algorithm behaviour for large input sizes and choose the most efficient solutions.

$$\mathbf{1 < \log_2 n < \sqrt{n} < n < n \log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n}$$

This sequence represents the increasing order of growth rates for common functions used in asymptotic analysis, from constant to exponential and beyond.

Examples

1. *f(n) = n^2logn + n*

   Checking for upper bound
   $n^2\log n + n \leq c \cdot (n^2\log n + n^2\log n)$
   $n^2\log n + n \leq 2c \cdot (n^2\log n)$
   f(n) = O(n^2logn)

   Checking for lower bound
   $n^2\log n + n \geq 1 \cdot (n^2\log n)$
   f(n) = Ω(n^2logn)

Checking for tight bound

$n^2 \log n \leq n^2 \log n + n \leq 2c \cdot (n^2 \log n)$

f(n) = Θ(n²logn)

2. *f(n) = log(n!)*

   log(1.1.1…1) ≤ log(1.2.3…n) ≤ log(n.n….n)

   1 ≤ log(n!) ≤ nlogn

   Upper Bound

   f(n) = O(nlogn)

   Lower Bound

   f(n) = Ω(1)

3. *f(n) = n!*

   Checking for upper bound

   n! = 1.2.3…n ≤ n.n….n

   n! ≤ n^n

   f(n) ≤ n^n

   f(n) = O(n^n)

   Checking for lower bound

   n! = 1.2.3…n ≥ 1.1…1

   n! ≥ 1

   f(n) ≥ 1

   f(n) = Ω(1)

4. $f(\mathrm{N}) = log \binom{N}{N/2}$

   Using Stirling's Approximation,

   $$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n,$$

   $$\log(N!) \sim N \log N - N + \tfrac{1}{2} \log(2\pi N),$$

   $$log \binom{N}{N/2} \approx N\, log\, 2 - \frac{1}{2} log\, N + const. = N\, log\, 2 + log\left(1/\sqrt{\mathrm{N}}\right)$$

   We know that $log\left(1/\sqrt{\mathrm{N}}\right)$ grows much slower than $N\, log\, 2$

   f(N) = O(N)

Examples on recursive relations

1.

```
void ABC(int n)
  if(n > 0)        Runs 1 time
    pf(“ ”)        Runs 1 time
    ABC(n-1)       Runs T(n-1) times
```

The program runs T(n) no. of iterations (steps)

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

So, T(n) is given as:

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

Writing the recursive relation:

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

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

…

T(2) = T(1) + 1

T(1) = T(0) + 1

Summing all the above equations:

T(n) = T(0) + n

T(n) = n + 1

Time Complexity = O(n)

2.

```
void ABC(int n)
  if(n > 0)                  Runs 1 time
    for(i = 0, i < n, i++)   Runs (n+1) times
      pf(“ ”)                Runs n times
    ABC(n-1)                 Runs T(n-1) times
```

The program runs T(n) no. of iterations (steps)

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

So, T(n) is given as:

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

Writing the recursive relation:

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

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

…

T(2) = T(1) + 2

T(1) = T(0) + 1

Summing all the above equations:

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

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

Time Complexity = $O(n^2)$

3.

```
ABC(int n)
 if(n > 0)    Runs 1 time
   pf(“ ”)    Runs 1 time
   ABC(n-1)   Runs T(n-1) times
   ABC(n-1)   Runs T(n-1) time
```

The program runs T(n) no. of iterations (steps)

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

So, T(n) is given as:

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

Writing the recursive relation:

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

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

$2^2T(n-2) = 2^3T(n-3) + 2^2$

…

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

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

Summing all the above equations:

T(n) = 2^nT(0) + (1 + 2 + 2^2 + … + 2^{n-1})

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

Time Complexity = O(2^n)

4.

```
ABC(int n)
  if(n > 1)    Runs 1 time
    pf(“ ”)    Runs 1 time
    ABC(n/2)   Runs T(n/2) time
```

The program runs T(n) no. of iterations (steps)

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

So, T(n) is given as:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

Writing the recursive relation:

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

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

…

T(n/2^{k-2}) = T(n/2^{k-1}) + 1

T(n/2^{k-1}) = T(n/2^k) + 1

Summing all the above equations:

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

The recurrence reaches base case when $\frac{n}{2^k} = 1$ *or k = $\log_2 n$*

Thus, T(n) = T(1) + $\log_2 n$

T(n) = 1 + $\log_2 n$

Time Complexity = O($\log_2 n$)

5. Merge Sort

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

Writing the recursive relation:

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

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

…

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

$2^{k-1}T(n/2^{k-1}) = 2^kT(n/2^k) + n$

Summing all the above equations:

$T(n) = 2^kT(n/2^k) + nk$

The recurrence reaches base case when $\frac{n}{2^k} = 1$ *or* $k = \log_2 n$

$T(n) = n + n\log_2 n$

Time Complexity = $O(n\log_2 n)$

6.

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

Writing the recursive relation:

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

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

…

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

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

Summing all the above equations:

$T(n) = 2^kT(n/2^k) + n \cdot \frac{1-\left(\frac{1}{2}\right)^k}{1-\frac{1}{2}}$

The recurrence reaches base case when $\frac{n}{2^k} = 1$ *or* $k = \log_2 n$

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

Time Complexity = $O(n)$

7.

```
void ABC(int n)
  if (n > 0)                          Runs 1 time
    for(i = 1, i < n, i = i*2)        Runs [log2n]+1 times
        pf(“ ”)                       Runs [log2n] times
    ABC(n-1)                          Runs T(n-1)
```

The program runs T(n) no. of iterations (steps)

T(n) = T(n-1) + 2$\log_2$n + 2 = T(n-1) + O($\log_2$n)

So, T(n) is given as:

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

Writing the recursive relation:

T(n) = T(n-1) + $\log_2$n

T(n-1) = T(n-2) + $\log_2$(n-1)

…

T(2) = T(1) + $\log_2$2

T(1) = T(0) + $\log_2$1

Summing all the above equations:

T(n) = T(0) + $\log_2$1 + $\log_2$2 + $\log_2$3 + $\log_2$4 + … + $\log_2$(n-1) + $\log_2$n

T(n) = 1 + $\log_2$(n!)

Time Complexity = O(nlogn)

[Check pg.8 Ex.2. for $\log_2$(n!) = O(nlogn)]