Unit - 1 Part - 1 Notes
Introduction

Basic Formula:-

Properties of Logarithm (item 1):-

1. $log_a 1 = 0$
2. $log_a a = 1$
3. $log_a x^y = y log_a x$
4. $log_a xy = log_a x + log_a y$
5. $log_a \frac{x}{y} = log_a x - log_a y$
6. $a^{log_b x} = x^{log_b a}$
7. $log_a x = \frac{log_b x}{log_b a} = log_a b \ log_b x$
8. $a^{log_a x} = x$

Combinatorics (item 2)

1. Number of permutations of an n-element set: P(n) = n!
2. Number of k-combinations of an n-element set: C(n,k) = $\frac{n!}{k!(n-k)!}$
3. Number of subsets of an n-element set: 2^n

Useful finite Summation Identities ($a \neq 1$) (item 3)

- $\sum_{i=l}^{u} 1 = 1 + 1 + 1 + \ldots + 1 = u - l + 1, \ l \leq u$
- $\sum_{i=1}^{n} lg \ i \approx nlgn$
- $\sum_{k=0}^{n} a^k = \frac{1 - a^n}{1 - a}, \ a < 1$
- $\sum_{k=0}^{n} a^k = \frac{a^n - 1}{a - 1}, \ a > 1$

- $\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$
- $\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{k=0}^{n} k^3 = \frac{(n(n+1))^2}{4}, \ a < 1$
- $\sum_{k=0}^{n} k^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$
- $\sum_{k=0}^{n} kx^k = \frac{x-(n+1)x^{n+1}+nx^{n+2}}{(x-1)^2}, \ x > 1$
- $\sum_{k=0}^{n} kx^k = \frac{x-(n+1)x^{n+1}+nx^{n+2}}{(1-x)^2}, \ x < 1$

Useful infinite identities (item 4)

- $\sum_{k=1}^{\infty} a^k = \frac{a}{1-a}, \ a < 1$
- $\sum_{k=1}^{\infty} a^k = \infty, \ a > 1$

Sum Manipulation Rules (item 5)

1. $\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i, where\ l \leq m < u$

4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Miscellaneous

1. $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n as\ n \to \infty\ (Stirling's\ formula)$ see item 3 equation 2

2. Modular arithmetic (n, m are integers, p is a positive integer)
   1. (n+m) mod p = (n mod p + m mod p) mod p
   2. (nm) mod p = ((n mod p)(m mod p)) mod p

1. **Analysis of Algorithm:-**
   Analysis of algorithm is a method using which we find the complexity of algorithms. There are two types of algorithm.
   a. Non - Recursive.
   b. Recursive.

2. **Analysis of Non-recursive algorithm:-**
   Steps in mathematical analysis of non recursive algorithms include:
   1. Decide on parameter n indicating input size.
   2. Identify algorithm's basic operation.
   3. Determine worst, average and best case for input of size n.
   4. Set up summation for C(n) reflecting algorithm's loop structure.
   5. Simplify summation using standard formulas (see page 1).
   Example:- Selection Sort
   Input: an array A[0…n-1]
   Output: Array A[0…n-1] sorted in ascending order.
   Code:-

```
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
            if A[j] < [min]
```

min ← j
swap(A[i], A[min])

basic operation: comparison

Inner loop:

$$S(i) = \sum_{j=i+1}^{n-1} 1 = (n-1) - (i+1) + 1 = n - i - 1$$

Outer loop:

$$op = \sum_{i=0}^{n-2} = \sum_{i=0}^{n-2} (n-i-1) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$op = (n-1) * (n-2) - \frac{(n-2) * (n-1)}{2}$$

$$op = \frac{(n-2) * (n-1)}{2} = O(n^2)$$

3. Analysis of Recursive Algorithm:-

Steps in mathematical analysis of recursive algorithms:

1. Decide on parameter n indicating input size
2. Identify algorithm's basic operation
3. Determine worst, average, and best case for input of size n
4. Set up a recurrence relation and initial condition(s) for T(n)-the number of times the basic operation will be executed for an input of size n (alternatively count recursive calls)
5. Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution.

Methods for Solving Recurrence Relations

No universal method exists that would enable us to solve every recurrence relation. (This is not surprising, because we do not have such a method even for solving much simpler equations in one unknown f (x) = 0 for an arbitrary function f (x).) There are several techniques, however, some more powerful than others, that can solve a variety of recurrences.

1. **Method of Forward Substitutions** Starting with the initial term (or terms) of the sequence given by the initial condition(s), we can use the recurrence equation to generate the few first terms of its solution in the hope of seeing a pattern that

can be Short Tutorial on Recurrence Relations 481 expressed by a closed-end formula. If such a formula is found, its validity should be either checked by direct substitution into the recurrence equation and the initial condition.

For example, consider the recurrence

T(n) = 2T(n – 1) + 1 for n > 1, T(1) = 1.

We obtain the few first terms as follows:

T(1) = 1,

T(2) = 2 T(1) + 1 = 2 . 1 + 1 = 3,

T(3) = 2 T(2) + 1 = 2 . 3 + 1 = 7,

T(4) = 2 T(3) + 1 = 2 . 7 + 1 = 15.

It is not difficult to notice that these numbers are one less than consecutive powers of 2: x(n) = 2^n – 1 for n = 1, 2, 3, and 4.

As a practical matter, the method of forward substitutions works in a very limited number of cases because it is usually very difficult to recognize the pattern in the first few terms of the sequence

2. **Method of Backward Substitutions** This method of solving recurrence relations works exactly as its name implies: using the recurrence relation in question, we express T(n – 1) as a function of T(n – 2) and substitute the result into the original equation to get T(n) as a function of T(n – 2). Repeating this step for T(n – 2) yields an expression of T(n) as a function of T(n – 3). For many recurrence relations, we will then be able to see a pattern and express T(n) as a function of t(n – i) for an arbitrary i = 1, 2,... . Selecting i to make n – i reach the initial condition and using one of the standard summation formulas often leads to a closed-end formula for the solution to the recurrence.

   As an example, let us apply the method of backward substitutions to recurrence.

   Thus, we have the recurrence equation

   T(n) = T(n – 1) + 1 for n > 1, T(1) = 1.

   Replacing n by n – 1 in the equation yields

   T(n – 1) = T(n – 2) + n – 1;

   after substituting this expression for T(n – 1) in the initial equation,

   we obtain

   T(n) = [T(n – 2) + n – 1] + n = T(n – 2) + (n – 1) + n.

   Replacing n by n – 2 in the initial equation yields T(n – 2) = T(n – 3) + n – 2; after substituting this expression for T(n – 2), we obtain

   T(n) = [T(n – 3) + n – 2] + (n – 1) + n = T(n – 3) + (n – 2) + (n – 1) + n.

   Comparing the three formulas for T(n), we can see the pattern arising after i

such substitutions:1 T(n) = T(n – i) + (n – i + 1) + (n – i + 2) + ... + n.

Since initial condition is specified for n = 0, we need n – i = 0,
i.e., i = n, to reach it:
T(n) = T(0) + 1 + 2 + ... + n = 0 + 1 + 2 + ... + n = n(n + 1)/2.

Important recurrence type:-

- **Linear:** one (constant) operation reduces problem size by one.
  T(n) = T(n-1) + c
  T(1) = d
  Solution: T(n) = (n-1)c + d

- **Quadratic:** A pass through input reduces problem size by one
  T(n) = T(n-1)+cn
  T(1) = d
  Solution: T(n) = $\left(\frac{n(n+1)}{2} - 1\right) c + d$

- **Logarithmic:** One (constant) operation reduces problem size by half.
  T(n) = T(n/2) + c
  T(1) = d
  Solution: T(n) = c log n + d

- **n log n:** A pass through input reduces problem size by half.
  T(n) = 2T(n/2) + cn
  T(1) = d
  Solution: T(n) = cn log n + dn

Example 1

```
fact(n)
     if (n <= 1)
          return n
     else
          return n*fact(n-1)
```

In this equation one multiply operation is reducing the size of input by one hence the recurrence relation will be:-

T(n) = T(n-1) + 1
T(1) = 1 ……..(This tells us that base case takes constant number of operations)

Solution:-

T(n) = T(n-1) + 1 …… eq 1
putting n —> n-1 in eq 1
T(n-1) = T(n-2) + 1……. eq 2
putting n —> n-2 in eq 1
T(n-2) = T(n-3) + 1…… eq 3
put value of T(n-1) from eq 2 to eq 1
T(n) = T(n-2)+1+1
= T(n-2) + 2 …….. eq 4
put value of T(n-2) from eq 3 to eq 4
T(n) = T(n-3)+1+2
= T(n-3) + 3
now we can generalise as we can see the pattern here n is reduced by one and the operation is increased by one
T(n) = T(n-k) + k
base condition is T(1) so we make n-k = 1 i.e k = n-1
T(n) = T(n-(n-1)) + n-1
T(n) = T(1) + n-1
= 1+ n-1 = n = O(n).

Example 2: Binary Search
// initially called with low = 0, high = N-1

```
  BinarySearch(A[0..N-1], value, low, high) {
      if (high < low)
          return not_found // value would be inserted at
index "low"
      mid = (low + high) / 2
      if (A[mid] > value)
          return BinarySearch(A, value, low, mid-1)
      else if (A[mid] < value)
          return BinarySearch(A, value, mid+1, high)
      else
          return mid
  }
```

Seeing this algorithm we can say that by seeing one element the algorithm is discarding half of the array so the recurrence relation would be:-

T(n) = T(n/2)+1 ……. eq 1

T(1) = 1

put n —> n/2 in eq 1

T(n/2) = T(n/4) + 1 …… eq 2

put n —> n/4 in eq 1

T(n/4) = T(n/8) + 1 …… eq 3

put value of T(n/2) from eq 2 to eq 1

T(n) = T(n/4)+1+1

T(n) = T(n/4) + 2 …… eq 4

put value of T(n/4) from eq 3 to eq 4

T(n) = T(n/8) +1 + 2

T(n) = T(n/8) + 3

now seeing the pattern we can generalise the equation

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

we know that T(1) = 1, so we need to reduce n/2^k to 1

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

2^k = n

k = $log_2 n$

therefore:

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

T(n) = 1 + $log_2 n$ = O($log_2 n$).

Practice problems:-

1. **Program:-**

```
for (i =1; i < n; i=i*2)
{
   for(j = 1; j < √n ; j = j+1)
   {
      //something
   }
```

}

Answer = $\sqrt{n}\ logn$

2. $T(n) = T(n/4) + 1\ for\ n > 1, T(1) = 1,\ answer\ =\ O(log_4 n)$.
3. $T(n) = 2T(n/2) + 1,\ for\ n > 1,\ T(1) = 1,\ answer = O(n)$.
4. $T(n) = 2T(n-1) + 1\ for\ n > 1,\ T(1) = 1,\ answer = O(2^n)$.
5. $T(n) = T(n-1) + n,\ for\ n > 1,\ T(1) = 1,\ answer\ =\ O(n^2)$.
6. $T(n) = 2T(n-1) + n,\ for\ n > 1,\ T(1) = 1,\ answer\ =\ O(2^n)$.
7. $T(n) = 2T(n/2) + n,\ for\ n > 1,\ T(1) = 1,\ answer = O(n\ log_2 n)$.

Solve the 6^{th} problem carefully, use equation 9 item 3.

In Question 1 you cannot show it using $\sum$ notation. It can be solved if you just see how many times the inner loop will run based on updation.