Designing and Analysis of Algorithms

Course Code: ECS 5101/CS514

- Dr Rahul Mishra
- IIT Patna

- **Lecture 3**

Standard Notations and common functions

1. Monotonicity

* A function $f(n)$ is <u>monotonically increasing</u> if $m \le n$ implies $f(m) \le f(n)$.

* Similarly, it is <u>monotonically decreasing</u> if $m \le n$ implies $f(m) \ge f(n)$.

* A function $f(n)$ is <u>strictly increasing</u> if $m < n$ implies $f(m) < f(n)$

* <u>Strictly decreasing</u> if $m < n$ implies $f(m) > f(n)$.

Standard Notations and common functions

2. Floor and Ceiling Functions

* For any real number x, we denote the greatest integer less than or equal x by $\boxed{\lfloor x \rfloor}$ (read "the floor of x") $\Rightarrow$ "greatest integer that does not exceed x" $\lfloor \quad \rfloor$ floor value

* The least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x") $\lceil \quad \rceil$
$\Rightarrow$ "least integer that is not less than x"
Ceiling value

* For any real x decimal

i) $x-1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x+1$ e.g., $3.14-1 < \lfloor 3.14 \rfloor \le 3.14 \le \lceil 3.14 \rceil < 3.14+1$

$\Rightarrow 2.14 < 3 \le 3.14 \le 4 < 4.14$

* For any integer n

ii) $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ e.g., $n = 5$

$\lceil 2.5 \rceil + \lfloor 2.5 \rfloor = 3 + 2 = 5$

Standard Notations and common functions

3. Modular function vs Arithmetic

* For any integer a and any positive integer n, the value $a \bmod n$ is the remainder (or residue) of the quotient a/n.

read as: "a modulo n"

* More exactly $k \pmod M$ is the unique integer r such that

$$\boxed{k = Mq + r \quad \text{where} \quad 0 \leq r < M.}$$

* When k is positive, simple divide k by M to obtain remainder r. Thus,

$25 \pmod 7 = 4$; $25 \pmod 5 = 0$, $35 \pmod{11} = 2$, $3 \pmod 8 = 3$

* ▶ IF $(a \bmod n) = (b \bmod n)$, we write $\boxed{a \equiv b \pmod n}$ and say that a is equivalent to b, modulo n.

* $\Rightarrow$ $\boxed{\text{Congruent}}$ $\rightarrow$

The mathematical congruence relation is defined as follows:

$a \equiv b \pmod M$ if and only if M divides $b - a$.

Standard Notations and common functions

8. Polynomials

Given a nonnegative integer d, a polynomial in n of degree d, is a function p(n) of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i \quad ; \quad \boxed{\text{sometimes } n \text{ is } x.}$$

Where the constants $a_0, a_1, \dots a_d$ are the coefficients of the polynomial and $\boxed{a_d \neq 0.}$

* A polynomial is asymptotically positive if and only if $a_d > 0$

* For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$

* For any real constant $a \geq 0$, the function n^a is monotonically increasing

* For any real constant $a \leq 0$, the function n^a is monotonically decreasing.

* A function $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for some constant k.

Standard Notations and common functions

Control structures

Three types of logic, or flow of control, called

i) Sequence logic, or sequential flow

ii) Selection logic, or conditional flow

iii) Iteration logic, or repetitive flow

→ $\boxed{\text{Sequence logic}}$ discussed in previous algorithmic example →

→ Unless instructions are given to the contrary, the modules are executed in the obvious sequence.

Module A ①
↓
Module B ②
↓
Module C ③

Standard Notations and common functions

Selection Logic (Conditional Flow)

* Selection logic employs a number of conditions which lead to a selection of one out of several alternative modules.

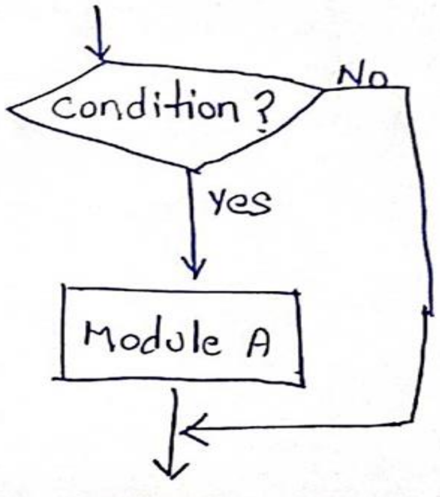
* The structures which implement this logic are called conditional structure or IF statement.

e.g.: End of such a structure by the statement →

[End of IF structure]

1> Single Alternatives →

IF condition, then:
    [Module A]
[End of IF structure]



Single

2> Double Alternatives

3> Multiple Alternatives

Standard Notations and common functions

Iteration Logic (Repetitive Flow)

Each type begins with a Repeat statement and is followed by a module, called the body of the loop.

* The repeat-for loop uses an index variable, such as K, to control the loop. The loop usually have the form:

Repeat for K = R to S by T:

[Module]

[End of loop.]

R ← initial value

S ← end value/test value

T ← increment

IF T is positive ← increment

IF T is negative ← decrement

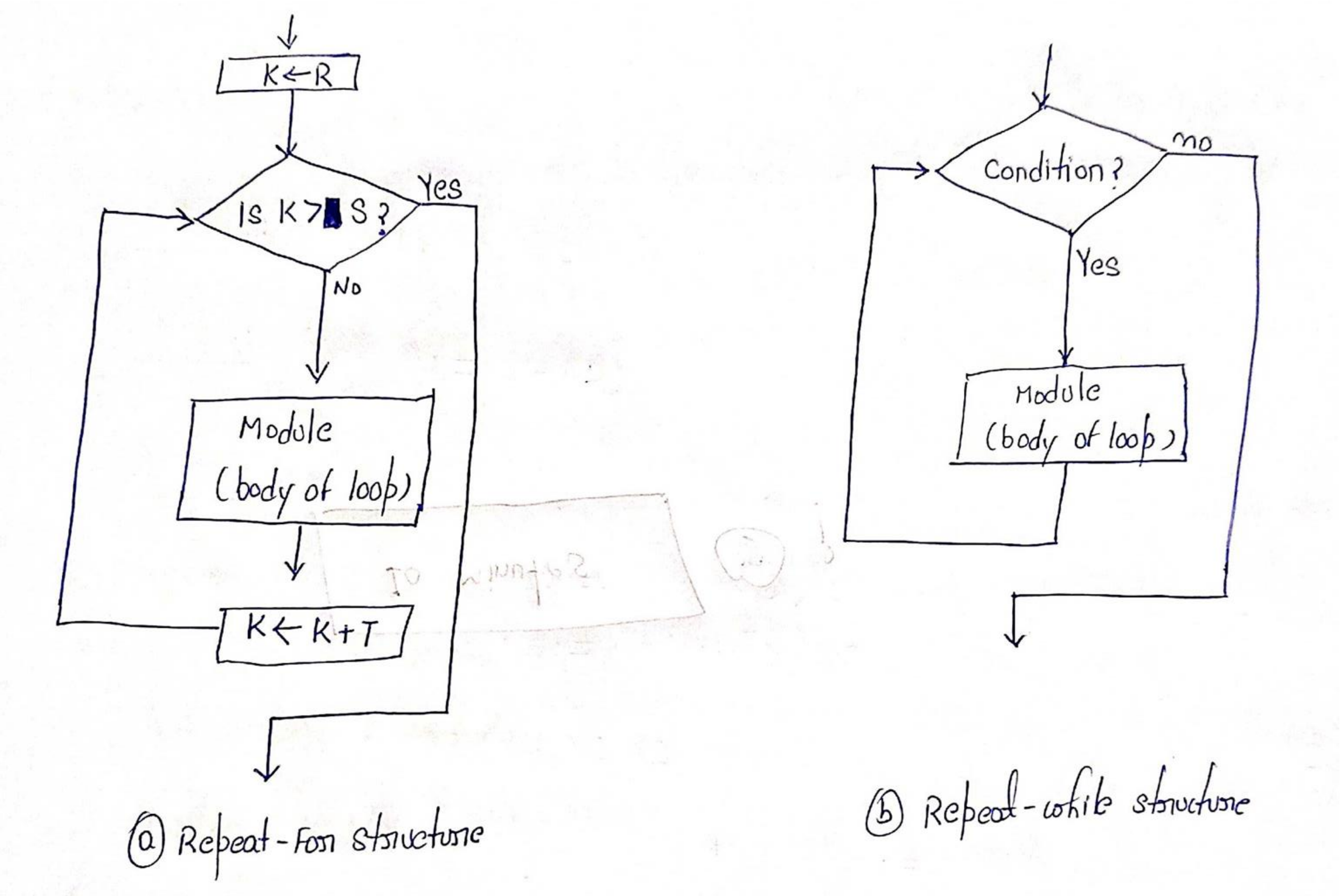
* repeat-while loop uses a condition to control the loop.

Repeat while condition:

[Module]

[End of loop.]

Standard Notations and common functions



Algorithm Analysis and Runtime

Algorithmic analysis and runtime

- **Algorithm analysis** is the process of evaluating the performance of an algorithm in terms of its **efficiency and scalability**.

- The primary goal of algorithm analysis is to understand how an algorithm will behave as the size of the input data increases and to identify any bottlenecks or performance issues that may arise.

- One common approach to algorithm analysis is to measure the **running time of the algorithm as a function of the input size**. This can be done empirically by running the algorithm on inputs of different sizes and measuring the time it takes to complete each run.

- Alternatively, the time complexity of the algorithm can be analyzed theoretically, by analyzing the **number of operations the algorithm performs as a function of the input size**.

Algorithmic analysis and runtime

- Other factors that can affect the performance of an algorithm include the **use of memory**, **the use of parallelism**, and the **use of heuristics** or **other optimization techniques**.

- These factors can also be analyzed using algorithm analysis techniques, such as space complexity analysis, parallelism analysis, and optimization analysis.

- Algorithm analysis is a critical tool for understanding the performance of algorithms and for designing efficient algorithms for complex problems.

- It is widely used in computer science, engineering, and other fields to optimize performance and solve complex problems.

Algorithmic analysis and runtime

* Suppose M is an algorithm, and suppose n is the size of the input data.
* The time and space used by the algorithm M are the two main measures for the efficiency of M.
* The time is measured by counting the number of key operations – in sorting and searching algorithms, e.g., the number of comparisons.
* Specifically, key operations are so defined that the time for other operations is much less than or at most proportional to the time for the key operations.
* The space measured by counting the maximum of memory needed by the algorithm.
* “The complexity of an algorithm $\boxed{M}$ is the function $\boxed{f(n)}$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data.

Algorithmic analysis and runtime

* The storage space required by an algorithm is simply a multiple of the data size n.

* Unless otherwise stated or implied, the term "complexity" shall refer to the running time.

* The running time of an algorithm depends not only on the size n of the input data but also on the particular data.

e.g.

Suppose we are given an English short story "TEXT" and suppose we want to search through TEXT for the first occurrence of a given 3-letter word W. If W is the 3-letter word "the," then it is likely that W occurs near the beginning of TEXT, so $f(n)$ will be small. On the other hand, if W is the 3-letter word "zoo" then W may not appear in TEXT at all, so $f(n)$ will be large.

Algorithmic analysis and runtime

1) <u>Worst case</u> : → the maximum value of $f(n)$ for any possible input

2) <u>Average case</u>: the expected value of $f(n)$

3) <u>Best case</u>: Sometimes, we also consider the minimum possible value of $f(n)$.

* Analysis of average case assumes a certain probabilities distribution for the input data;
↳ one such assumption might be that all possible permutations of an input dataset are equally likely.

→ The average case also uses the following concept in probability theory

→ Let the number $n_1, n_2, \ldots, n_k$ occur with respective probabilities $p_1, p_2, \ldots, p_k$

Expected value $\boxed{E = n_1 p_1 + n_2 p_2 + \cdots + n_k p_k}$

Algorithmic analysis and runtime

(Linear Search) A linear array DATA with N elements and a specific ITEM of information are given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC = 0.

1. [Initialize] Set K := 1 and LOC := 0.
2. Repeat Steps 3 and 4 while LOC = 0 and K ≤ N.
3. If ITEM = DATA[K], then: Set LOC: = K.
4. Set K := K + 1. [Increments counter.]

   [End of Step 2 loop.]
5. [Successful?]

   If LOC = 0, then:

   Write: ITEM is not in the array DATA.

   Else:

   Write: LOC is the location of ITEM.

   [End of If structure.]
6. Exit.

Algorithmic analysis and runtime

- The complexity of the search algorithm is given by the number **C** of comparisons between
  ITEM and **DATA[K].**

- We seek **C(n)** for the worst case and the average case.

Worst case:
The worst case occurs when **ITEM** is the last element in the array **DATA** or is not there.

$$C(n) = n$$

Accordingly, **C(n) = n** is the worst-case complexity of the linear search algorithm.

Algorithmic analysis and runtime

- We assume that the **ITEM** does appear in the **DATA** and that it is equally likely to occur at any position in the array.

- The number of comparisons can be any of the numbers: **1, 2, 3, …, n**, and each number occurs with the probability **p= 1/n**.

$$
\begin{aligned}
C(n) &= 1.\ 1/n + 2.\ 1/n + \ldots + n.\ 1/n \\
&= (1+2+3+ \ldots + n).\ 1/n \\
&= n(n+1)/2\ .\ 1/n = (n+1)/2.
\end{aligned}
$$

- This agrees with the intuitive feeling that the average number of comparisons needed to find the location of ITEM is approximately equal to half the number of elements in the DATA list.

Algorithmic analysis and runtime

- The complexity of average case of an algorithm is usually much more complicated to analyses than that of the worst case.
- The probabilistic distribution that one assumes for the average case may not actually apply to real situations.
- Thus, unless otherwise stated or implied, the complexity of an algorithm shall mean the function which gives the running time of the worst case in terms of the input size.
- Moreover, the complexity of the average case for many algorithms is proportional to the worst case.

Growth of function

* Suppose M is an algorithm, and suppose n is the size of the input data.

* The complexity $\boxed{f(n) \text{ of } M}$ increases as $\boxed{n}$ increases.

* It is usually the rate of increase of $\boxed{f(n)}$ that we want to examine.

* This is usually done by comparing $f(n)$ with some standard function

$$\boxed{\log_2 n, \quad n, \quad n\log_2 n, \quad n^2, \quad n^3, \quad 2^n}$$

n \ $g(n)$	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

"Rate of Growth of Standard Functions"

Asymptotic Analysis

- ***Dictionary meaning:*** *"Asymptotic function approaches a given value as an expression containing a variable tends to infinity."*

- We are concerned with how the **running time** of an algorithm increases with the size **of the input in the limit, as the size of the input increases without bounds.**

- An algorithm that is asymptotically more efficient will be the best choice for all but very small input.

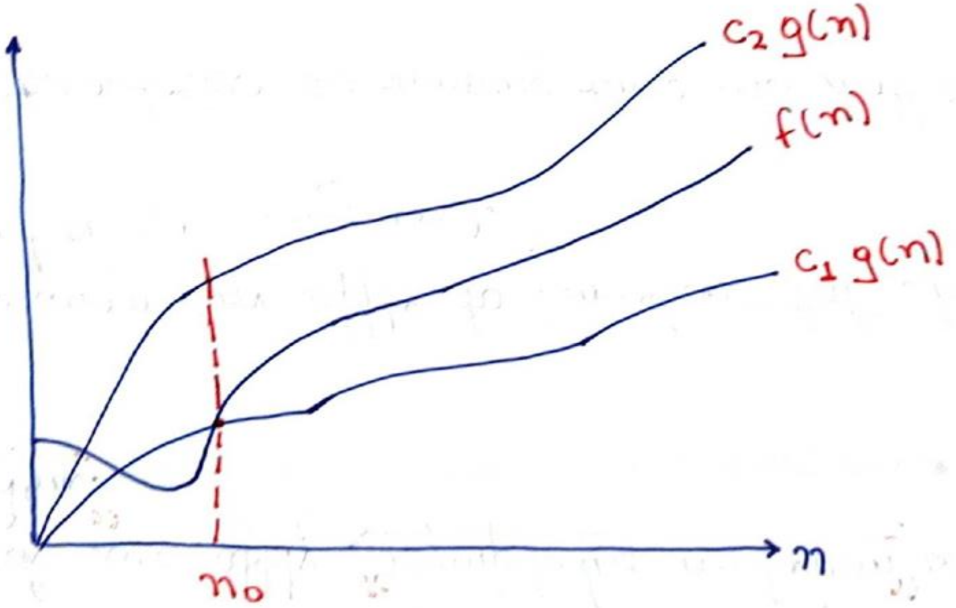
- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domain is the set of natural numbers, **N** = {0,1,2,…}.

- They are used for defined worst-case running-time function **T(n),** which usually is defined only on integer input sizes.

Asymptotic Analysis: $\theta - notation$

Let us define what this notation mean. For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions.

* $\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \geq n_0 \}$.



$f(n) = \Theta(g(n))$

Asymptotic Analysis: $\theta - notation$

Θ - Notation

* A function $\boxed{f(n)}$ belongs to the set $\boxed{\Theta(g(n))}$ if there exist positive constants $\boxed{c_1}$ and $\boxed{c_2}$ such that it can be "sandwiched" between $\boxed{c_1 g(n) \text{ and } c_2 g(n)}$, for sufficiently large $\boxed{n}$.
  * → (read as f of n is theta of g of n)

* Since $\Theta(g(n))$ is a set, we can write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$.

* Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion.

* An intuitive picture of functions $\boxed{f(n) \text{ and } g(n)}$, where $f(n) = \Theta(g(n))$.

* For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $\boxed{c_1 g(n)}$ and at or below $\boxed{c_2 g(n)}$.

* $g(n)$ is an asymptotically tight bound for $f(n)$.

Asymptotic Analysis: $\theta - notation$

The definition of θ(g(n)) require that every member $f(n) \in \theta\left(g_{(n)}\right)$ by asymptotically non-negative, that is, that f**(n)** be non-negative whenever **n** is sufficiently large.

Example → $f(n) = 18n + 9$

since $f(n) \geq 18n$ and $f(n) \leq 27n$,

for $n \geq 1$

~~we have $f(n) = \theta(n)$ and~~

- $g(n) = 3n$

$$c_1 3n \leq \underbrace{18n + 9}_{f(n)} \leq c_2 3n$$

$c_1 = 6 \quad c_2 = 9$