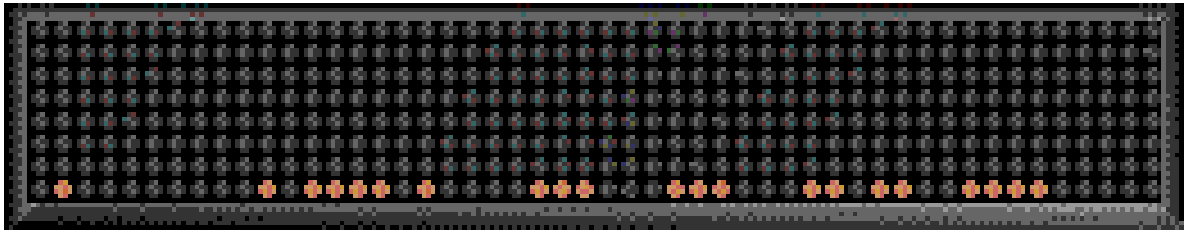
DATA STRUCTURES (CSN-102)



ENJOY YOUR JOURNEY!

About me

- Dr. Neetesh Kumar,
- Assistant Professor, CSE,
- Indian Institute of Technology-Roorkee (IIT-R),
- Haridwar Highway, Uttarakhand, India -247667.
- Web: https://www.iitr.ac.in/~CSE/Neetesh_Kumar
- Google Scholar: https://scholar.google.co.in/citations?user=Ut_l01AAAAAJ&hl=en
- Write to me: Neetesh@cs.iitr.ac.in

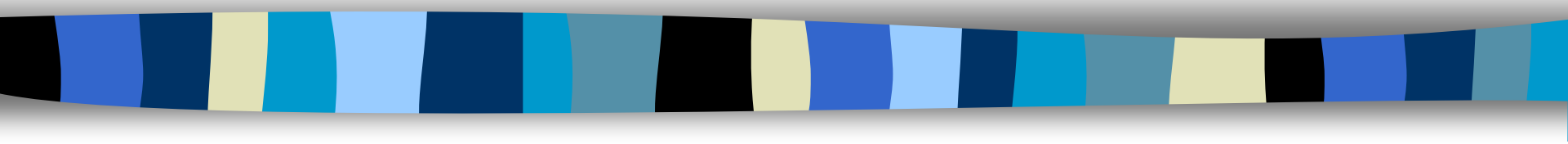
List of TAs

- Nitin Tyagi: nitin_t@cs.iitr.ac.in
- Anshul Pundhir: anshul_p@cs.iitr.ac.in
- Ashish Kumar : ashish_k@cs.iitr.ac.in
- Nisha Singh Chauhan: nisha_sc@cs.iitr.ac.in
- Vibha Bharilya: vibha_b@cs.iitr.ac.in
- I. Mani Verma: im_varma@cs.iitr.ac.in
- Anuj Sachan: anuj_s@cs.iitr.ac.in

Prerequisite

- Basic Concepts of C/C++/Java
- General Overview of Computer and its Architecture
- Basic Mathematics
- Most Importantly Your Attention ☺

Today's Agenda



An Introduction to the Course

Course Strategy

- 3 Lectures in a Week
- 1 Tutorial for Each Batch in a Week
- One Tutorial for Solving the Problems and Clearing Out Doubts
- Next Alternative Tutorial for Assignment
- Approximately 7 Tutorial for Doubts Clearing and Problem Solving
- Approximately 7 for Programming based Assignments

Marks Distribution

Particular	Marks	Remark
ETE	50%	One ETE
MTE	25%	One MTE
Tutorials	20%	6 to 7 Assignments
Quiz / Surprise Quiz	5%	One Quiz

Why This Course ?

- You will be able to evaluate the quality of a program
- Understanding the running time and memory space
- You will be able to write fast programs
- You will be able to solve new problems
- You will be able to give non-trivial methods to solve problems.
- (Your program will be faster than others.)

Goals of the Course

- Become familiar with most of the fundamental data structures in computer science
- Improve ability to solve problems abstractly
  - data structures are the building blocks
- Improve ability to analyze your algorithms
  - prove correctness
  - gauge (and improve) time complexity
- To improve overall programming skills by using effective data structures

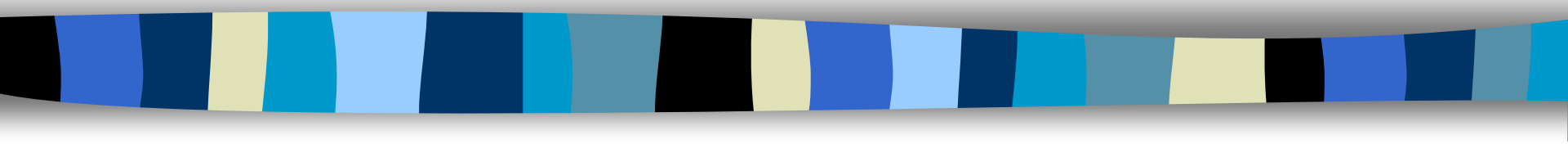
Broad Syllabus

- Review of C Concepts: Structures, Union, Pointers, Memory Allocation
- Asymptotic Notation
- Arrays, Dictionary, Abstract Data Type
- List, Stack, and Queue
- Sorting and Searching
- Trees: General, Binary, Binary Search Tree, AVL, Red-Black, B-Tree, etc.,
- Graphs: Basic Graph and Property, Representation, BFS, DFS, Topological, Dijkstra, Prims etc.

Books

- Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
- Data Structures Using C, Aaron M. Tenenbaum, Pearson Education India,
- Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed "Fundamentals of Data Structures in C", Computer Science Press, 1992.

Understanding Data Structures



Observation

- All programs manipulate data
  - programs *process*, *store*, *display*, *gather*
  - data can be *information*, *numbers*, *images*, *sound*
- Each program must decide how to store data
- Choice influences program at every level
  - execution speed
  - memory requirements
  - maintenance (debugging, extending, etc.)

Algorithm

- Definition
  An *algorithm* is a finite set of instructions or steps that accomplishes a particular task.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

What is a Data Structure?

Algorithm: (OR)

– An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.

• Program:

– An implementation of an algorithm in some programming languages

• Data Structure:

– Organization of data needed to solve the problem

Data Type

- Data Type

A data type, in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. A string, for example, is a data type that is used to classify text and an integer is a data type used to classify integers. **OR**

A *data type* is a collection of *objects* and a set of *operations* that act on those objects.

Abstract Data Type

- Abstract Data Type
  Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations.

OR

- An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Specification vs. Implementation

- Operation specification
  - function name
  - the types of arguments
  - the type of the results
- Implementation independent

*Structure 1.1: Abstract data type *Natural_Number* (p.17)

structure Natural_Number is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all x, y $\in$ *Nat_Number*; *TRUE*, *FALSE* $\in$ *Boolean*

and where +, -, <, and == are the usual integer operations.

Nat_No Zero ( ) ::= 0

Boolean Is_Zero(x) ::= **if** (x) **return** *FALSE*
else return *TRUE*

Nat_No Add(x, y) ::= **if** ((x+y) <= *INT_MAX*) **return** x+y
else return *INT_MAX*

Boolean Equal(x,y) ::= **if** (x== y) **return** *TRUE*
else return *FALSE*

Nat_No Successor(x) ::= **if** (x == *INT_MAX*) **return** x
else return x+1

Nat_No Subtract(x,y) ::= **if** (x<y) **return** 0
else return x-y

end *Natural_Number*

::= is defined as

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

An instance characteristic is a measure (typically with integer value) of the magnitude of an instance of the problem.

***Program 1.9:** Simple arithmetic function (p.19)

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

$S_{abc}(I) = 0$

***Program 1.10:** Iterative function for summing a list of numbers (p.20)

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

$S_{sum}(I) = 0$

Recall: pass the address of the first element of the array & pass by value

***Program 1.11:** Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
 }
```

$S_{rsum}(I)=S_{rsum}(n)=6n$

Assumptions:

***Figure 1.1:** Space needed for one recursive call of Program 1.11 (p.21)

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition

$$T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$$

- ADD is an add operation
- n is number of instance characteristics
- ADD(n) is number of add operations
- Ca is time required for an ADD operation

Methods to compute the step count

- Introduce variable count into programs
- Tabular method
  - Determine the total number of steps contributed by each statement
    step per execution × frequency
  - add up the contribution of all statements

Iterative summing of a list of numbers

***Program 1.12:** Program 1.10 with count statements (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;              /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;         /* last execution of for */
   return tempsum;
   count++;         /* for return */
}
```

2n + 3 steps

***Program 1.13:** Simplified version of Program 1.12 (p.23)

```
float sum(float list[ ], int n)
{
    float tempsum = 0;
    int i;
    for (i = 0; i < n; i++)
        count += 2;
    count += 3;
    return 0;
}
```

2n + 3 steps

Recursive summing of a list of numbers

***Program 1.14:** Program 1.11 with count statements added (p.24)

```
float rsum(float list[ ], int n)
{
        count++;        /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

2n+2

Matrix addition

***Program 1.15:** Matrix addition (p.25)

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                        int c [ ] [MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < rows; i++)
       for (j= 0; j < cols; j++)
         c[i][j] = a[i][j] +b[i][j];
 }
```

***Program 1.16:** Matrix addition with count statements (p.25)

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int row, int cols )
{
   int i, j;
   for (i = 0; i < rows; i++){
        count++; /* for i for loop */
        for (j = 0; j < cols; j++) {
           count++; /* for j for loop */
           c[i][j] = a[i][j] + b[i][j];
           count++; /* for assignment statement */
        }
        count++;    /* last time of j for loop */
   }
  count++;         /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

***Program 1.17:** Simplification of Program 1.16 (p.26)

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                        int c[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         count += 2;
         count += 2;
   }
   count++;
}
```

$2rows \times cols + 2rows + 1$

Tabular Method

***Figure 1.2:** Step count table for Program 1.10 (p.26)

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

***Figure 1.3:** Step count table for recursive summing function (p.27)

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

***Figure 1.4:** Step count table for matrix addition (p.27)

Statement	s/e	Frequency	Total steps

Void add (int a[ ][MAX_SIZE]. . .)
{
int i, j;
for (i = 0; i < row; i++)
for (j=0; j< cols; j++)
c[i][j] = a[i][j] + b[i][j];
}
	0
	0
	0
	1
	1
	1
	0
		0
		0
		0
		rows+1
		rows. (cols+1)
		rows. cols
		0
			0
			0
			0
			rows+1
			rows. cols+rows
			rows. cols
			0

Total			2rows. cols+2rows+1

Exercise 1

***Program 1.18:** Printing out a matrix (p.28)

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < row; i++) {
      for (j = 0; j < cols; j++)
          printf("%d",  matrix[i][j]);
      printf( "\n");
   }
}
```

Exercise 2

***Program 1.19:Matrix multiplication function**(p.28)

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE])
{
   int i, j, k;
   for (i = 0; i < MAX_SIZE; i++)
      for (j = 0; j< MAX_SIZE;  j++) {
          c[i][j] = 0;
      for (k = 0; k < MAX_SIZE; k++)
            c[i][j]  +=  a[i][k] * b[k][j];
      }
}
```

Exercise 3

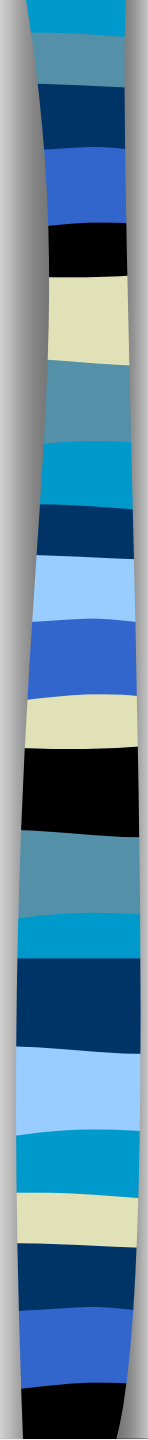
***Program 1.20:Matrix product function**(p.29)

```
void prod(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE],
                                                    int rowsa, int colsb, int colsa)
{
   int i, j, k;
   for (i = 0; i < rowsa; i++)
      for (j = 0; j< colsb;  j++) {
          c[i][j] = 0;
      for (k = 0; k< colsa; k++)
            c[i][j]  +=  a[i][k] * b[k][j];
      }
}
```

Exercise 4

***Program 1.21:Matrix transposition function** (p.29)

```
void transpose(int a[ ][MAX_SIZE])
{
   int i, j, temp;
   for (i = 0; i < MAX_SIZE-1; i++)
      for (j = i+1; j < MAX_SIZE;  j++)
          SWAP (a[i][j], a[j][i], temp);
}
```



Switch to Time Complexity

- O(1): constant
- O(n): linear
- O(n^2): quadratic
- O(n^3): cubic
- O(2^n): exponential
- O(logn)
- O(nlogn)

*Figure 1.8: Plot of function values(p.39)

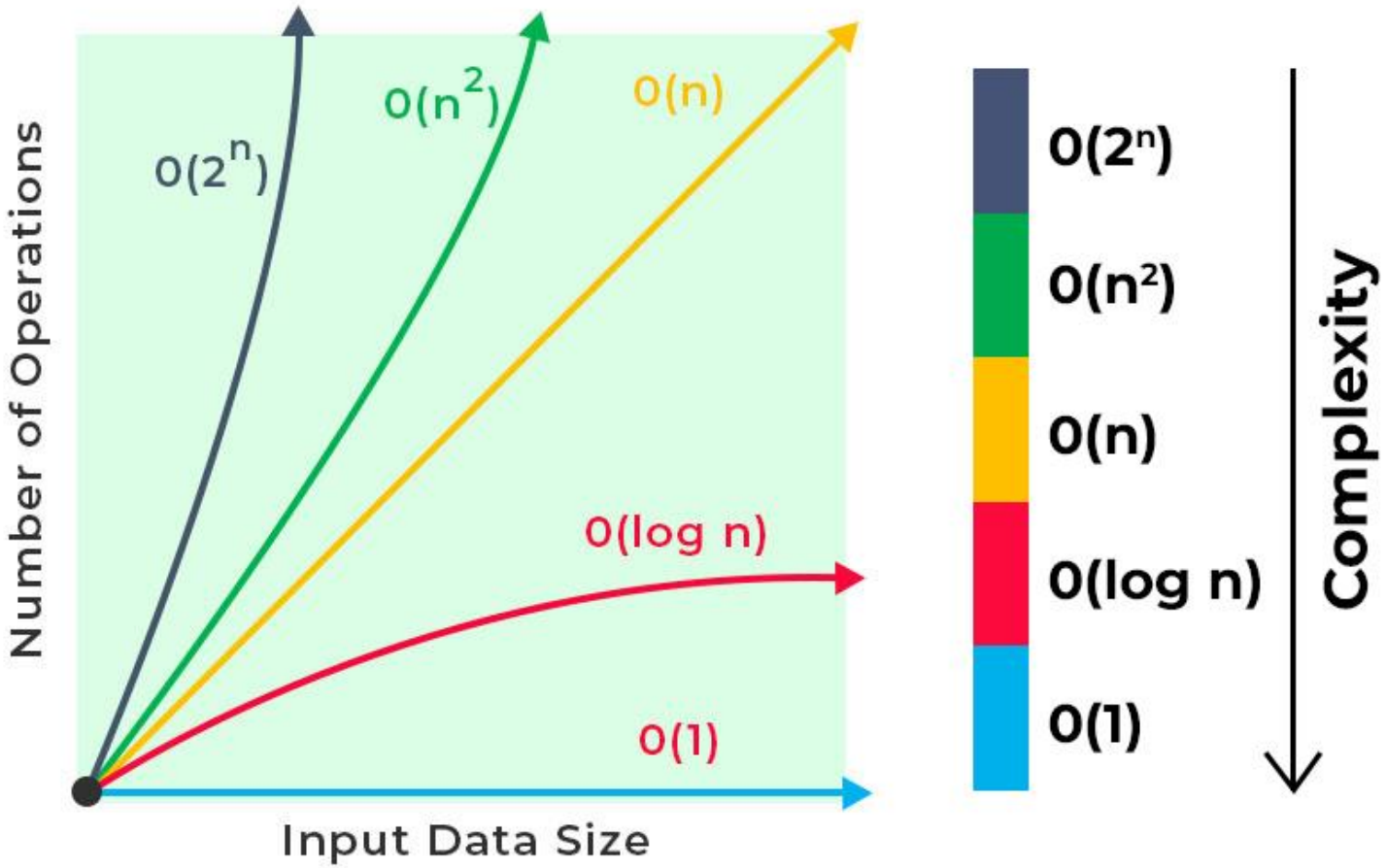


*Figure 1.9: Times on a 1 billion instruction per second computer (p.40)

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$ yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^7$ yr	$3.17*10^{43}$ yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years