Introduction: Basic Terminology, Elementary Data Organization

→ Data are simply values or sets of value.

→ A data item refers to a single unit of values.

→ Data items that are divided into subitems are called group items.

→ those that are not divided into subitems are called elementary items.

Eg. employee name
- First name
- middle name
- last name

Mobile number
↳ treated as single item

→ Collections of data are frequently organized into a hierarchy of fields, records and files.

→ An entity is something that has certain attributes or properties which may be assigned values.

→ Entities with similiar attributes form an entity set.

Entity :	Employee		
Attributes :	Name	Age	Mobile
Values :	GAIL	34	7123456780

→ The term information is sometimes used for data with given attributes (processed data)

→ Data may be organized in terms of logical or mathematical model of a particular organization of data is called a data structure.

Data Structures
- Primitive Data Structure
  - Integer
  - Real
  - Character
  - Boolean
- Non-Primitive Data Structure
  - Linear Data Structure
    - Arrays
    - Linked List
    - Stacks
    - Queues
  - Non-linear Data Structure
    - Trees
    - Graphs

Reference: Lipschutz, "Data Structures" Schaum Outline series, Chapter 1

→ Data Structure includes following four operations

1. Traversing: Accessing each record exactly once
2. Searching: Finding the location of the record
3. Inserting: Adding a new record to the structure
4. Deleting: Removing a record from the structure

Two operations for special situations.

1. Sorting: Arranging the records in some logical order
2. Merging: Combining the records in two different sorted files into a single sorted file.

Algorithm, and Efficiency of Algorithm

→ An algorithm is a well-defined list of step for solving a particular problem.

→ the time and space it uses are two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size.

Complexity of Algorithm:

→ Suppose M is an algorithm, n is the size of input data.

→ The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size n of input data.

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

→ The two cases one usually investigates in the complexity theory as follows:

1. Worst Case: the maximum value of f(n) for any possible input
2. Average Case: the expected value of f(n).

Sometimes Consider the minimum possible value of f(n), Called the best Case.

Reference: Lipschutz, "Data Structures" Schaum outline series, Chapter 1 & 2.

COMPLEXITY OF ALGORITHM

Asymptotic Notations

Big Oh 'O' | Big omega 'Ω' | Theta θ | Small oh 'o' | Small omega ω

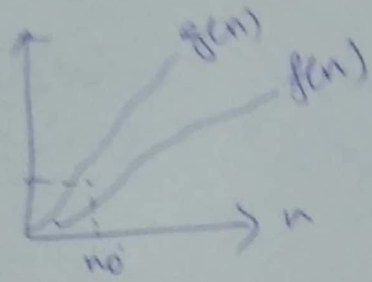
Big Oh 'O' (upper bound)

$3n^2 + 2n + 5$

f(n) & g(n) are two functions

$f(n) \le c\, g(n)$

for $c > 0$, $n_0 \geq 1$



$n^2 + 3n + 2 \le c\, O(n^2)$ ✓ — $O(n^4)$ LUB

$n^2 + 3n + 2 = O(n^3)$ ✓ — $O(n^3)$ loose UB

$n^2 + 3n + 2 = O(n)$ ✗ — $O(n^2)$ Tight upper bound

$\le c\,n$ — $O(n)$ ✗

Big Omega 'Ω' (Lower bound)

$\cancel{n^2}$ $f(n) \geq c\, g(n)$

$n^2 + 2n + 3 \geq cn^2$ ✓

$\Omega(n^2)$

$n^2+2n+3 \geq cn$

$\Omega(n)$ ✓

$\cancel{\Omega(n^2)}$

fcc fi·c-o-i hic

$n^2+2n+3 \geq c(1)$

$\Omega(1)$ ✓

$\Omega(n^2) \rightarrow$ Tight Lower bound.

$\Omega(n) \rightarrow$ Loose " "

$\Omega(1) \rightarrow$ " " "

$\Omega(n^3)$ ✗

Theta

TLB = TUB = Θ (exactly one)

$\Theta(n^2)$ ✓ $\Theta(n)$ ✗ $\Theta(n^4)$ ✗

$\Theta(n^3)$ ✗ $\Theta(1) =$ ✗

$\cancel{c_1\ f(g}$

$c_1 g(n_1) \leq f(n) \leq c_2 g(n_2)$

✓ $c_1(n^2) \leq n^2+3n+5 \leq c_2 n^2$

✗ $c_1 n^3 \leq n^2+3n+5 \leq c_2 n^3$

✗ $c_1 n \leq n^2+3n+5 \leq c_2 n$

Small 'o' and small 'ω' only relay on TUB and TLB respectively.

Asymptotic Notation:

→ Rate of Growth. Big O Notation:

Suppose, M is an algorithm

n is the size of input data

so, the complexity $f(n)$ of M increases as n increases.

Suppose $f(n)$ & $g(n)$ are functions defined on positive integers with the property that $f(n)$ is bounded by some multiple of $g(n)$ for almost all n. i.e. Suppose there exist a positive integer n_0 and a positive number M such that $\forall\ n > n_0$

$$|f(n)| \leq M|g(n)|$$

then, $f(n) = O(g(n))$ ("$f(n)$ is of order of $g(n)$")

→ this defines an upper bound function $g(n)$ for $f(n)$ which represents the time/space complexity of the algorithm.

→ Omega Notation (Ω):

It is used when the function $g(n)$ defines a lower bound for the function $f(n)$

Suppose there exists a positive integer n_0 and positive number M such that $|f(n)| \geq M|g(n)|\ \forall\ n \geq n_0$

then $f(n) = \Omega(g(n))$ ("$f(n)$ is of omega of $g(n)$")

→ Theta Notation (Θ):

It is used when the function $f(n)$ is bounded both from above & below by the function $g(n)$.

Suppose there exist two positive constants c_1 and c_2 and a positive integer n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|\ \forall\ n \geq n_0$

Time space trade-off

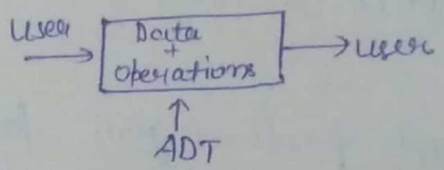
A space-time or time-memory trade off is a case where an algorithm or program trades increased space for decreased time. The utility of a given space-time tradeoff is affected by related fixed and variable cost (Eg CPU speed, RAM space, hard-drive space)

Abstract Data Type

→ refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation.

→ with an ADT, we know a specific data type can do, but how it actually does it is hidden.

→ It allow us to use the functions while hiding the implementation.



E.g. place the list on an ADT. The users should not be aware of the structure that we use i.e. whether it is tree, or graph or something else. As long as they are able to insert and retrieve data, it does not make a difference as to how we store the data

→ It can thus be further defined as a data declaration packaged together with the operations that are meaningful for the data type. Encapsulate the data, operate the data and then hide them from the user.

Array and Linked List can be used to implement an ADT list.

Lipschutz "Data Structures" Schaum Series. (1.10-1.12, 2.19)

Array

→ Data structures are classified as either linear or non-linear.
A DS is said to be linear if its elements form a sequence (linear list).
There are two basic ways of representing such linear structures, first is array (linear relationship between the elements represented by means of sequential memory locations). Other way is to have the linear relationship between the elements represented by means of pointers or links. i.e. linked list.

Linear Array

→ It is a list of a finite number n of homogeneous data elements (i.e. data elements of same type) such that:

(a) the elements of the array are referenced respectively by an index set consisting of n consecutive numbers.

(b) The elements of the array are stored respectively in successive memory locations.

→ the number n of elements is called the length or size of array.

→ $\boxed{\text{length of array} = UB - LB + 1}$

UB → upper bound (largest index)

LB → lower bound (smallest index)

→ element of array A can be ~~obtained~~ denoted by the subscript notation

$A_1, A_2, \ldots A_n$

or by the bracket notation A[1], A[2], A[3], ... A[N]

→ the no. K in A[K] is called a subscript or an index and A[K] is called a subscripted variable.

A[6]:- A

111	222	333	444	555	666
1	2	3	4	5	6

Representation of Linear Array in Memory

Let, LA be a linear array in the memory of computer

$LOC(LA[K])$ = address of element $LA[K]$ of array LA.

$Base(LA)$ = address of first element of LA

So,

$$\boxed{Loc(LA[K]) = Base(LA) + w(K - LB)}$$

where,

w = number of words per memory cell for array

E.g. A → 1932 through 1984

$Base(A) = 200$, $w = 4$ words per memory cell.

$Loc(A[1932]) = 200$.
$Loc(A[1933]) = 204$
$Loc(A[1934]) = 208$
$LOC(A[1935]) = 212$

Add^r of array element for $K = 1965$

$$Loc(A[1965]) = Base(A) + w(1965 - LB)$$
$$= 200 + 4(1965 - 1932)$$
$$= 332$$

Multidimensional Array

→ the LA also called 1-D array, since each element in the array is referenced by a single subscript.

→ When array elements are referenced by two or more subscripts, this is called multidimensional array.

Two dimensional Array

A 2-D $m \times n$ array A is a collection of $m \cdot n$ data elements such that each element is specified by a pair of integers (such as J, K) called subscript, with the property that

$$1 \le J \le m \quad \text{and} \quad 1 \le K \le n$$

$$A_{J,K} \text{ or } A[J, K]$$

2-D array are called matrices in mathematics and tables in business applications.

→ A[J, K], elements appear in row J and column K. (row is a horizontal list and column is a vertical list of elements.)

A[3, 4] or Array A of 3×4 size

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & A_{11} & A_{12} & A_{13} & A_{14} \\ 2 & A_{21} & A_{22} & A_{23} & A_{24} \\ 3 & A_{31} & A_{32} & A_{33} & A_{34} \end{array}$$

		LB	UB
$A \to m \times n$,	first dimension of A contains the index set	1, ...	m
	second " " " " " " "	1, ...	n

R₁: Lipschutz, "Data Structures", (4.1 – 4.7)

R₂: Tenenbaum, "Data Structure using C" (4.5 – 4.7)

Storage Representation:

1. Row major order:

→ we can consider 2-D array as 1-D array since it has elements with a single dimension.

→ 2-D array can be assumed as a single column with many rows this representation is called row-major order

Address of element of m^{th} row and n^{th} column =

addr (a[m,n]) = (total no. of rows present before the m^{th} row × size of row)
+ (total no. of elements present before the n^{th} elements in the m^{th} row × size of element)

The total no. of rows present before m^{th} row = $(m - lb1)$

size of row = total no. of elements present in row × size of element

Total no. of elements in a row = $ub2 - lb2 + 1$

So,

$$addr\,(a[m,n]) = ((m - lb1) \times (ub2 - lb2 + 1) \times size) + ((n - lb2) \times size)$$

R1: Lipsschutz, " Data structures with C" (4.30 - 4.32)

R2: Tenenbaum, " Data Structures using C" (36-37)

Column Major Order

→ Also represent a 2-D array as one single row of column and map it sequentially, this is called Column major order representation

$$\boxed{\begin{array}{l} addr(a[m,n]) = (\text{Total no. of Columns present before } n^{th} \text{ column} \times \text{column size}) \\ \quad + (\text{Total no. of elements present before } m^{th} \text{ element of } n^{th} \text{ column} \times \text{each element size}) \end{array}}$$

Column placed before n^{th} Column = $(n - lb2)$
↳ second dimension LB.

No. of elements in column = $(ub1 - lb1 + 1)$

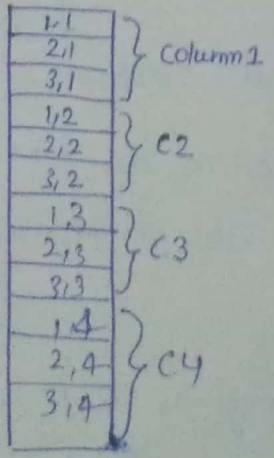
Therefore,

$$\boxed{addr(a[m,n]) = ((n - lb2) \times (ub1 - lb1 + 1) \times size) + ((m - lb1) \times size)}$$

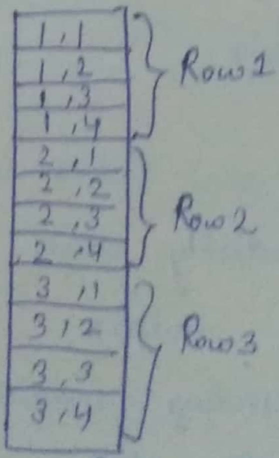
Representation of 2-D Array

→ A programming language will store the array A either column by column or row by row.

A(3×4)



(a) Column-major order



(b) Row major order

$$\boxed{Loc(A[J,K]) = Base(A) + w[M(K-1) + (J-1)]}$$ Column major order

$$\boxed{Loc(A[J,K]) = Base(A) + w[N(J-1) + (K-1)]}$$ Row-major order

Application of Array:

→ Arrays are used to implement mathematical vectors and matrices
→ also used to implement other data structures such as heap, hash table, queue, deque, stack, string
→ dynamic memory allocation

Sparse Matrices:

→ Matrices with a relatively high proportion of zero entries are called sparse matrices.

→ A matrix, where all entries above the main diagonal are zero or equivalently, where non-zero entries can only occur on or below the main diagonal is called a lower triangular matrix.

$$\begin{bmatrix} 4 & & & & \\ 3 & -5 & & & \\ 2 & 0 & 1 & & \\ 4 & 9 & 8 & 7 & \\ 3 & 2 & 1 & 0 & 5 \end{bmatrix} \qquad \begin{bmatrix} 1 & -3 & & & & \\ 4 & 2 & 9 & & & \\ & 6 & 3 & -2 & & \\ & & 8 & 4 & 7 & \\ & & & 6 & 5 & 0 \\ & & & & -2 & 6 \end{bmatrix}$$

(a) Triangular Matrix (b) Tridiagonal matrix

→ A matrix, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a tridiagonal matrix.

→ natural method of representing matrices in memory as 2-D array may not be suitable for sparse matrices.

Representation of 2D array in memory

Let A be 2D array represented by $m \times n$

$LOC(LA[K]) = Base(LA) + w(K-1)$ } 1D

Col^n major

$$LOC(A[J,K]) = Base(A) + w[M(K-1)+(J-1)]$$

Row major

$$LOC(A[J,K]) = Base(A) + w[N(J-1)+(K-1)]$$

Base (A) = ~~A[1,1]~~ address of first element A[1,1]

M = rows
N = col^n

Q. $\overset{M}{25} \times \overset{N}{4}$, $w = 4$

Base [Score] = 200

row major

SCORE [12, 3]

$$LOC[SCORE[12,3]) = 200 + 4[4(12-1)+(4-1)]$$

$$= 384$$

address of 3rd col^n of 12th row.

Sparse Matrix Representation

A sparse matrix can be represented by using two representations

1. Triplet Representation
2. Linked Representation

1. **Triplet Representation**: Consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores total rows, total columns and total non-zero values in matrix

E.g

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 9 & 0 \\ 1 & 0 & 8 & 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 & 2 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 5 \\ 4 & 0 & 0 & 2 & 0 & 0 & 0 \end{array} \rightarrow$$

Row	Column	Values
5	6	6
0	4	9
1	1	8
2	2	4
3	5	5
4	2	2

Total row, Total Col^n, Total non-zero values

2. **Linked Representation**: use linked list DS to represent a sparse matrix. In this LL, use 2 different nodes namely header node and element node. Header node consists of 3 fields and element node consists of 5 fields as

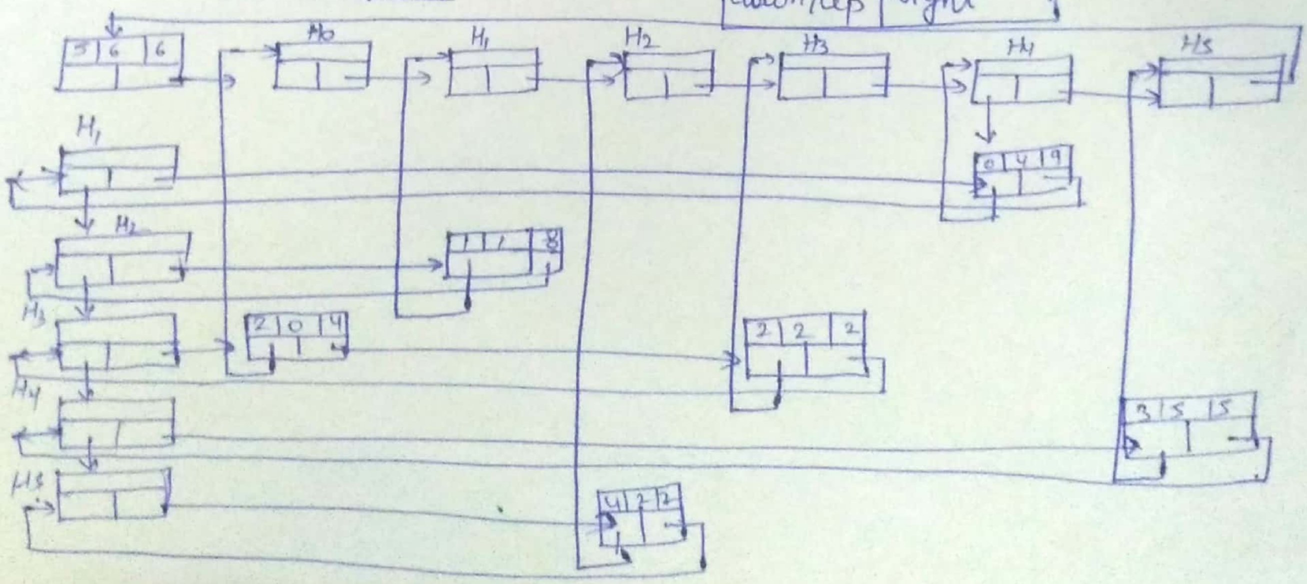
Header Node

Indexed Value	
down	right

Element Node

row	column	value
down/up	right	

E.g



Traverse Linear Array

1. Set $K = LB$
2. Repeat Steps 3 and 4 while $K \le UB$
3. Apply PROCESS to $LA[K]$
4. Set $K = K+1$
5. Exit

Insertion

Insert (LA, N, K, ITEM)

1. Set $J = N$
2. Repeat Steps 3 and 4 while $J \geq K$
3. Set $LA[J+1] = LA[J]$
4. Set $J = J-1$
5. Set $LA[K] = ITEM$
6. Set $N = N+1$
7. Exit

⇒ Here LA is a linear array with N elements and K is +ve integer such that $K \le N$. This algo insert an element ITEM into the K^{th} pos^n in LA.

⇒ In reverse order so that data not get erased.

Deletion from linear array

DELETE (LA, N, K, ITEM)

1. Set ITEM = LA[K]
2. Repeat for J = K to N-1
3. Set LA[J] = LA[J+1]
4. Set N = N-1
5. Exit.

⇒ LA is a linear array with N elements and K is a +ve integer such that $K \leq N$.

Find number of non zero elements in Sparse Matrix.

1. Set NUM = 0
2. Repeat for I = 1 to N
3. Repeat for J = 1 to N
   if A[I, J] ≠ 0 then
   Set NUM = NUM + 1

4. Return.