

UNIT I

Introduction to Algorithm and Data Structures

Prof. Pujashree Vidap
PICT Pune



UNIT -1

Introduction to Algorithm and Data Structures

- Introduction to Problem to Data Structure
- Data Structure Classification
- Algorithms- Flowchart, Pseudo code.
- Complexities of an Algorithms and Asymptotic Notation
- Algorithmic Strategies



SOME QUESTION FOR U !!!!!!!!!!!!!!!

- Why are we studying any programming language?
- Advantage and disadvantage of the languages you have studied earlier?
- What do you think efficiency of the program depends on?
- What is your approach of solving a particular problem?
- Do you know anything about any data structures?



Problem to Solve

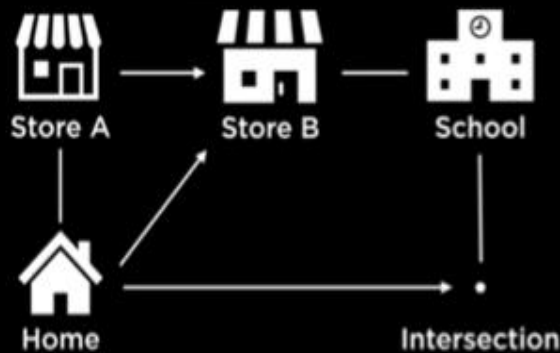
- Suppose we have to sort 10 Numbers.
or
- Suppose I want to bring something from market and have to lock my home before going

Points to Note:

- What do u think the steps I need to follow to solve this problem efficiently?
- Do u think efficiency of the program will be effected if data is not stored properly?
- Example: Library management, Bank application to manage customer account, Google Map



From Problem to Algorithm and Data Structure



Home	(49.2, -123.4)
Store A	(49.3, -123.4)
Store B	(49.3, -123.3)
School	(49.3, -123.2)
Intersection	(49.2, -123.2)

(Home, Store A)
(Store A, Home)
(Home, Store B)
(Home, Intersection)
(Store A, Store B)
(Store B, School)
(School, Store B)
(Intersection, School)

Home	(Store A, Store B, Intersection)
Store A	(Store B)
Store B	(School)
School	(Store B, School)
Intersection	(School)

Algorithm for finding shortest path

- Find all places we can go from home
- From each of those places find all path
- Keep track of distances
- Repeat this process untill we reach to school
- Compare the distances we have traveled
- Find the shortest distance among above found distances



Important steps of problem solving

1. Select the appropriate data structure.
2. Design suitable algorithm.



Data Structure

- Data structure describes the way the data is organized and stored in computer memory for convenient processing.

OR

- Data Structure can be defined as a particular way of storing and organizing data in a computer so that it can be stored or accessed efficiently.



Concept of Data Structure

Data must be organized for the following basic reasons-:

- o **Suitable representation**
- o **Ease of retrieval**
- o **Operations allowed**



So the overall performance of a program depends upon the following two factors:

- Choice of right data structure.
- Design of suitable algorithm to work upon the chosen data structure.



Choice 1:

[illegible]name

list

rollnomarks

percent

grade

Parallel list for student data

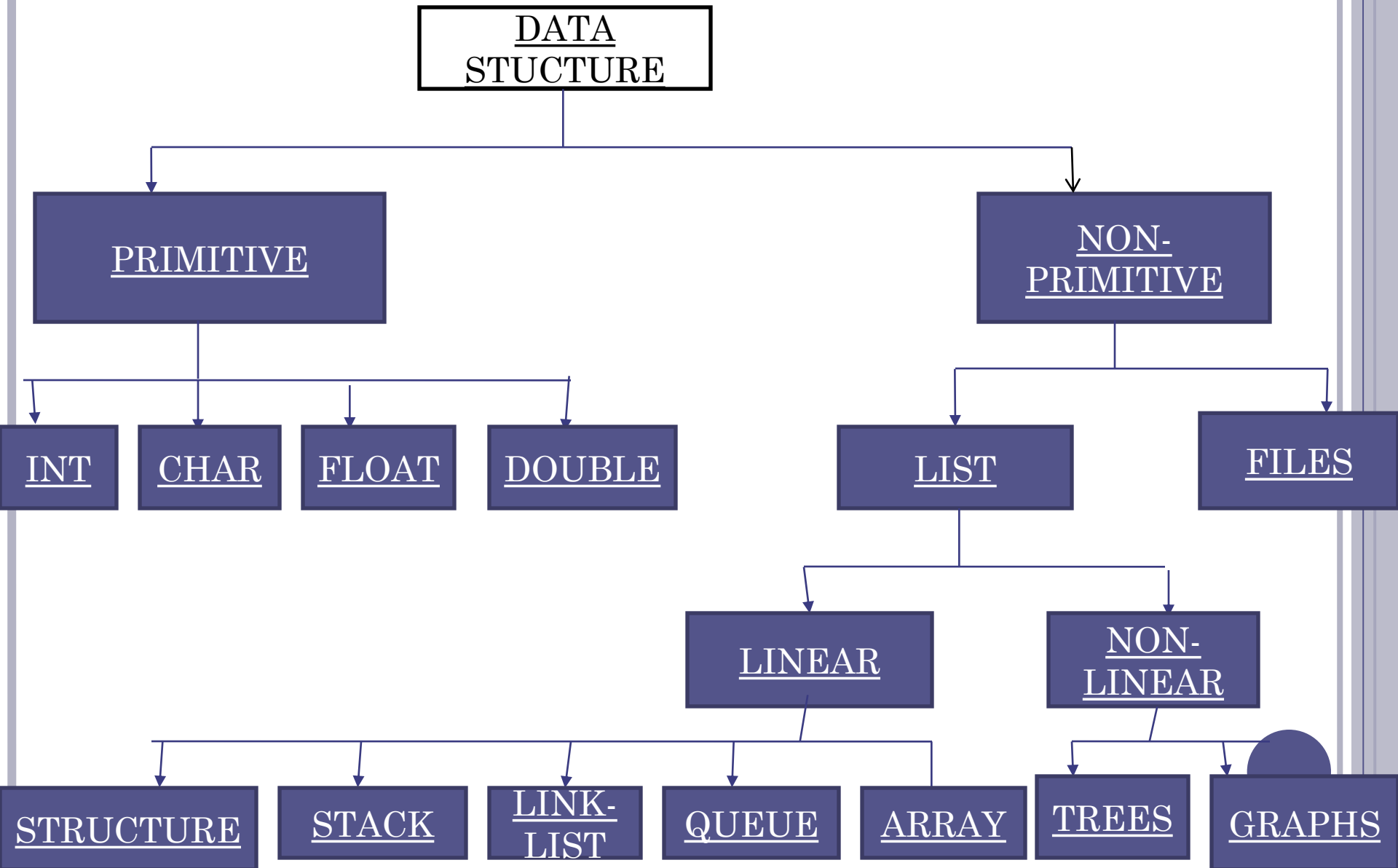


Choice II:

```
struct student
{
char name[15];
int roll;
int marks;
float percent;
char grade;
}s[10];
```



TYPE OF DATA STRUCTURES



Primitive D.S

Basic data types such as int, char, float are known as **Primitive D.S.**

It has predefined way of storing data by the system and the set of operations that can be performed on these data are also predefined.

Eg. int, char, float.

Non Primitive D.S

That data types that are derived from primary data structures are known as **Non Primitive D.S.** They are used to store groups of values.

Eg. array, structure, union, link list, stack, queues etc..



Linear Data Structure

These represents a sequence of items and the elements follow a linear ordering.

The Linear D.S has following properties:

- Each element is followed by one other element.
- No two elements are followed by the same element.



Types Of Linear DS:

ARRAYS:

An array is used to store elements of the same type.

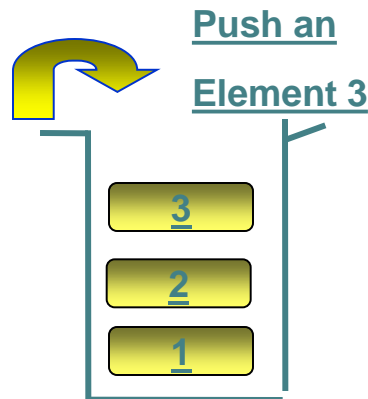
The number of elements that can be stored in a array can be calculated as: $(\text{upperbound} - \text{lowerbound}) + 1$

```
int a[10]
```



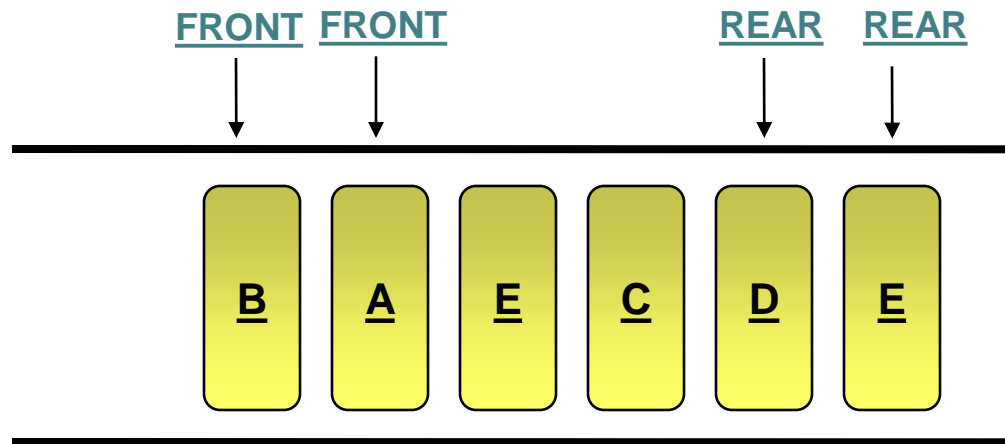
STACKS

Collection of elements like array with a special feature that deletion and insertion of elements can be done only from one end, called the TOP (Top of stack). Due to this property it is also called LIFO data structure i.e. Last-In-First-Out data structure.



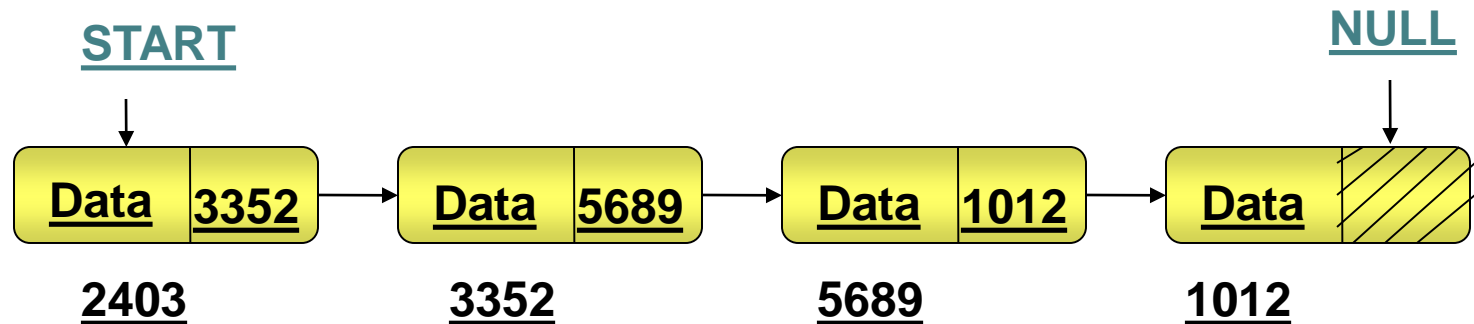
QUEUES

Queues are **First -In-First-Out (FIFO)** data. In a queue new elements are added to the queue from one end called Rear end, and the elements are deleted from another end called Front end. Eg. the queue in a ticket counter.



LINK LIST

A link list is a linear collection of elements called nodes. The linear order is maintained by pointers. Each node is divided into two or more parts; one data field and another pointer field.



STRUCTURE

- Collections of related variables (aggregates) of under one name.
- Can contain variables of different data types
- Commonly used to define records to be stored in files

Eg. struct ADate

```
{  
    int month;  
    int day;  
    int year;  
} Date;
```

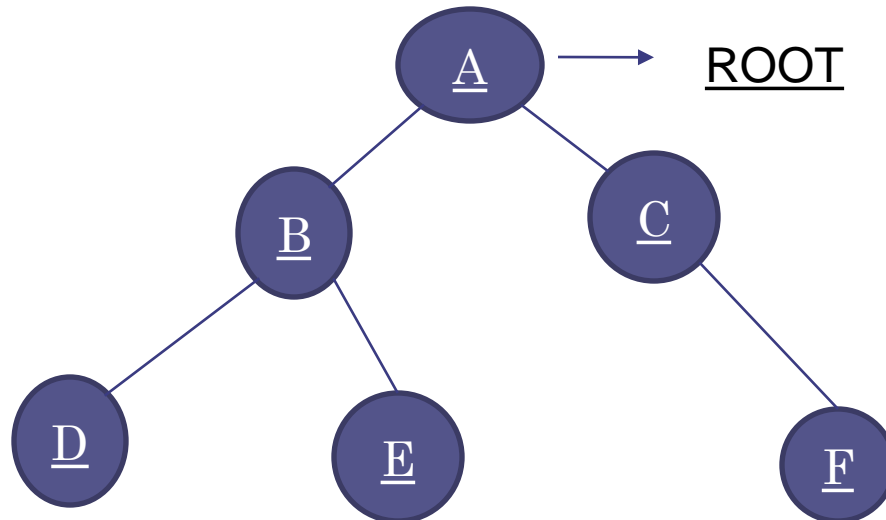


Types of Non Linear DS

TREES

A tree can be defined as finite set of data items called nodes.

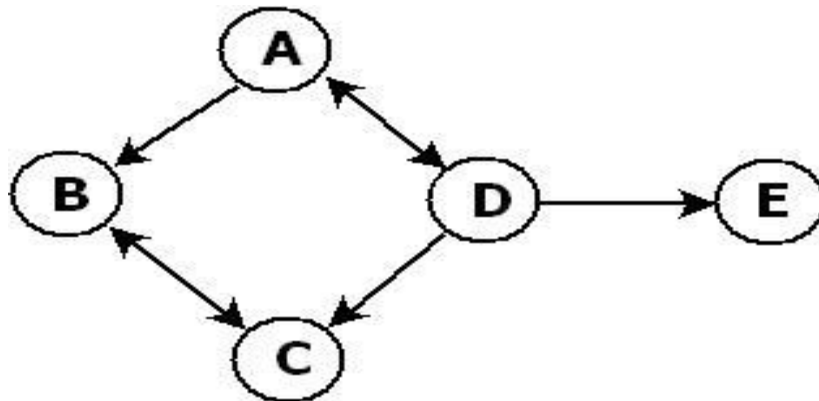
- It represents the hierarchical relationship between various elements.
- Each node is connected to its children by edges.
- There is a special data item at the top called Root of tree.



GRAPHS

Graph is non-linear data structure capable of representing kinds of physical structures.

A graph $G(V,E)$ is a set of vertices V and a set of edges E connecting vertices.



ARRAY

- Write a program to reverse an array

STRUCTURE

- Create a structure Student containing name, rollno, marks in 5 subjects. Allow user to enter details of five students. Calculate the % of each student and display the details of first two toppers.



Problem statement

Create an structure Account that the bank might use to represent customer bank accounts. Include the data members Name, Account number and Account balance. Provide three member functions in a class, member function Credit() should add an amount to current balance and member function Debit() should withdraw the money from the account and ensures that the debit amount does not exceed the account's balance. If it does, the balance should left unchanged and the function should print a message indicating “ Debit Amount Exceeded Account Balance “. Member function getBalance() should display the current balance with customers information . Create a Program that display information of five Customers.



HOME ASSIGNMENT

- Design a structure BIGBAZAR to store order information. Each order information has an 11-character item code, 20- character item name, price and an integer quantity. Provide the following member functions in a class:-
 - 1. functions to read these records from standard input getdata(),
 - 2. to calculate the total price calculate()
 - 3. sort functions to arrange data by ItemCode sort(),
 - 4. to display the same and display the totalprice display().
 - Also use function displaycount() to display the number of records.



ALGORITHM



ALGORITHM

An *Algorithm*, can be defined as a finite sequence of instructions, that if followed accomplishes a particular task.

An algorithm refer to a method that can be used by a computer for a solution of problem



HOW TO DEVELOP AN ALGORITHM

1. Understand the problem.
2. Identify the **output** of the problem.
3. Identify **inputs** required by the problem and choose the associated data structure.
4. Design a **logic** that will produce the desired output from the given inputs.
5. **Test** the algorithm for different sets of input data.
6. Repeat steps 1 to 5 till the algorithm produces the desired results for all types of input and rules.



CHARACTERISTICS OF AN ALGORITHM

INPUT: This part of the algorithm reads the data of the given problem.(zero or more input externally supplied)

OUTPUT: It must produce the desired output.(one quantity at least)

DEFINITENESS: Each instruction is clear and unambiguous .for e.g. add 6 or 7 to x.

FINITENESS: The algorithm must come to an end after a finite number of steps.

EFFECTIVENESS:.. It should also be basic enough to carried out with pencil and paper. Each instruction should be definite and feasible.



Pseudo code and flowcharts

TOOL FOR ALGORITHM WRITING



PSEUDO CODE

- *Pseudo code* is an artificial and informal language that helps programmers develop algorithms.
- Pseudo code may be an informal English, combinations of computer languages and spoken language. Whatever works for you.



PSEUDOCODE & ALGORITHM

- **Example 1:**

Write an algorithm to determine a student's final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.



PSEUDOCODE & ALGORITHM

Algorithm:

- *Input a set of 4 marks*
- *Calculate their average by summing and dividing by 4*
- *if average is below 60*
 Print “FAIL”
 else
 Print “PASS”



PSEUDOCODE & ALGORITHM

Pseudocode:

```
1: read M1,M2,M3,M4
2: GRADE := (M1+M2+M3+M4)/4
3: if (GRADE < 60) then
    write "FAIL"
4.  else
    write "PASS"
```



ALGORITHM SPECIFICATIONS

Pseudo code conventions

1. Comments begin with // and continue until the end of line
2. Blocks are indicated by { } and statements are delimited by ;
3. Identifiers begin with letter. Data types are not explicitly declared. The type will be clear from context .
4. Global and local will also be clear from context
5. Assignment operator is used for assigning. $A := 3$;
6. Boolean values used are true and false. Logical operators- and, or, not. All Relational operators are used. $>, <, =, \geq, \leq, \neq$



7. Elements of 1D & 2D array are accessed using $A[i]$ & $A[i, j]$ respectively. Array index start from 0.

8. while(condition) do
 {--- }

9. repeat

 until(condition);

10. for $i:=1$ to 5 do
 {
 }

11. if condition then statement else statement;



12. Case statement

```
case{  
    :condition1:statement  
    :condition2:statement  
    else: statement }  
}
```

13. Input and output is done using instruction read and write

14. Algorithm Max(A , n) { }

Note:

- Algorithm may or may not return any value.
- Simple variables in procedure are passed by value .
- Array and record name is treated as pointers



Designing an Algorithm

SEQUENCE STRUCTURE:

Algorithm to compute area of rectangle

Algorithm AreaRect(l, b, area)

```
{ // l is length of rectangle, b is breadth of  
  // rectangle and area will hold the calculated  
  // area of rectangle.
```

1. read l, b;
2. area := l*b;
3. write area;

```
}
```



SELECTION (conditional control)

Algorithm to find greatest of 3 numbers

Algorithm greatest(num1,num2,num3,res)

{ //num1, num2, num3 are 3 nos. to be compared
//and res will hold the largest value.

1. read num1,num2,num3;
 2. if(num1>num2 and num1>num3)then res:=num1;
 3. elseif(num2>num1 and num2>num3) then
 - 3.1 res:=num2;
 4. elseif(num3>num1 and num3>num2) then
 - 4.1 res:=num3;
 5. else res:=num1 ;
 6. write res;
- }



ITERATIONS (or repetitions)

Algorithm to find largest of 10 numbers

Algorithm largest (A, n, large)

{ // A is an array of n nos. and large will hold the
//largest no.

1. read size of list n;
 2. for i:=0 to n-1 do
 - 2.1 read A[i];
 3. set large := A[0]
 4. for i:=1 to n-1 do
 - 4.1 if (A[i]>large)
 - 4.1.1 large := A[i]
 5. write large;
- }



Algorithm to calculate sum of even numbers present in a given list and also count the even numbers present in the list

Algorithm count_even(a,n,e_counter,sum)

```
{ // a is an array of n nos, a counter variable e_counter is used to
  //count even nos in given list, variable sum will hold the sum of
  // 10 nos
1.sum:=0,e_counter:=0;
2.read n;
3.for i:=0 to n-1 do
    3.1 read a[i];
4.for i:=0 to n-1 do
    4.1 if (a[i]%2= 0) then
        4.1.1 write a[i] is even;
        4.1.2 e_counter:=e_counter+1;
        4.1.3 sum := sum+a[i];

5. write sum, e_counter
}
```



- The **variable acting as accumulator** is **SUM** as it is holding sum of 10 nos
- Counter variable is **e_counter** as it is counting no. of even nos in the list and **i** as it is counting the no of times operation is performed.





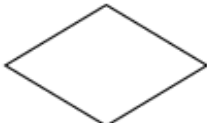
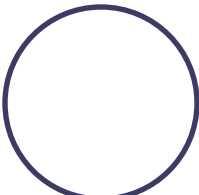


FLOWCHARTS

- Graphically depict the logical steps to carry out a task and show how the steps relate to each other.
- A flowchart is simply a graphical representation of steps.
- It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes.

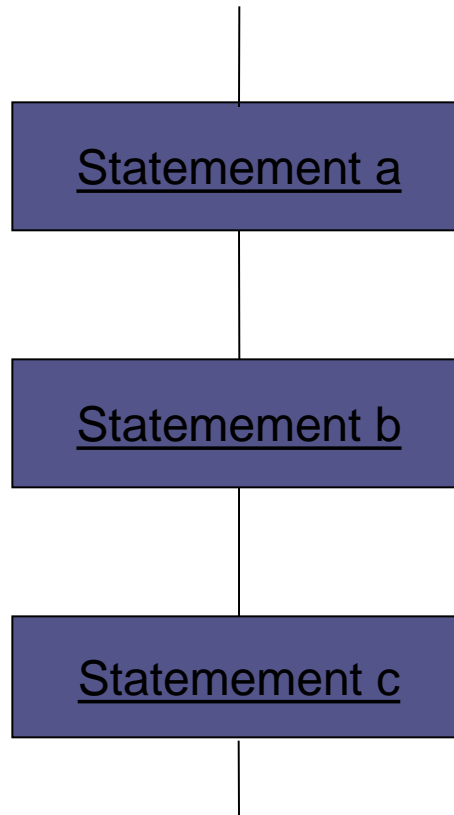


FLOWCHART SYMBOLS

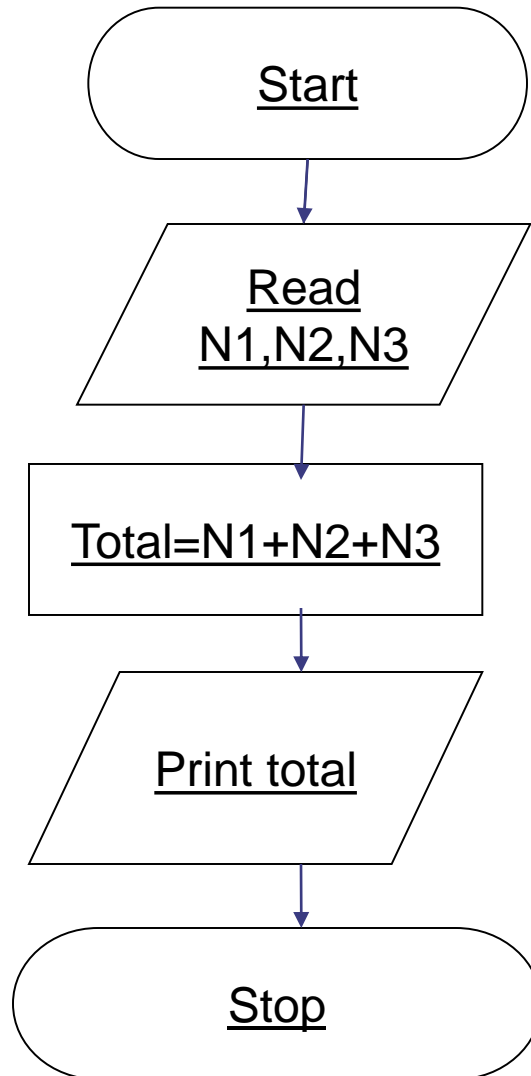
Symbol	Name	Meaning
	<i>Flowline</i>	Used to connect symbols and indicate the flow of logic.
	<i>Terminal</i>	Used to represent the beginning (Start) or the end (End) of a task.
	<i>Input/Output</i>	Used for input and output operations, such as reading and displaying. The data to be read or displayed are described inside.
	<i>Processing</i>	Used for arithmetic and data-manipulation operations. The instructions are listed inside the symbol.
	<i>Decision</i>	Used for any logic or comparison operations. Unlike the input/output and processing symbols, which have one entry and one exit flowline, the decision symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is "yes" or "no."
	<u>connector</u>	<u>Used to join different flowlines</u>



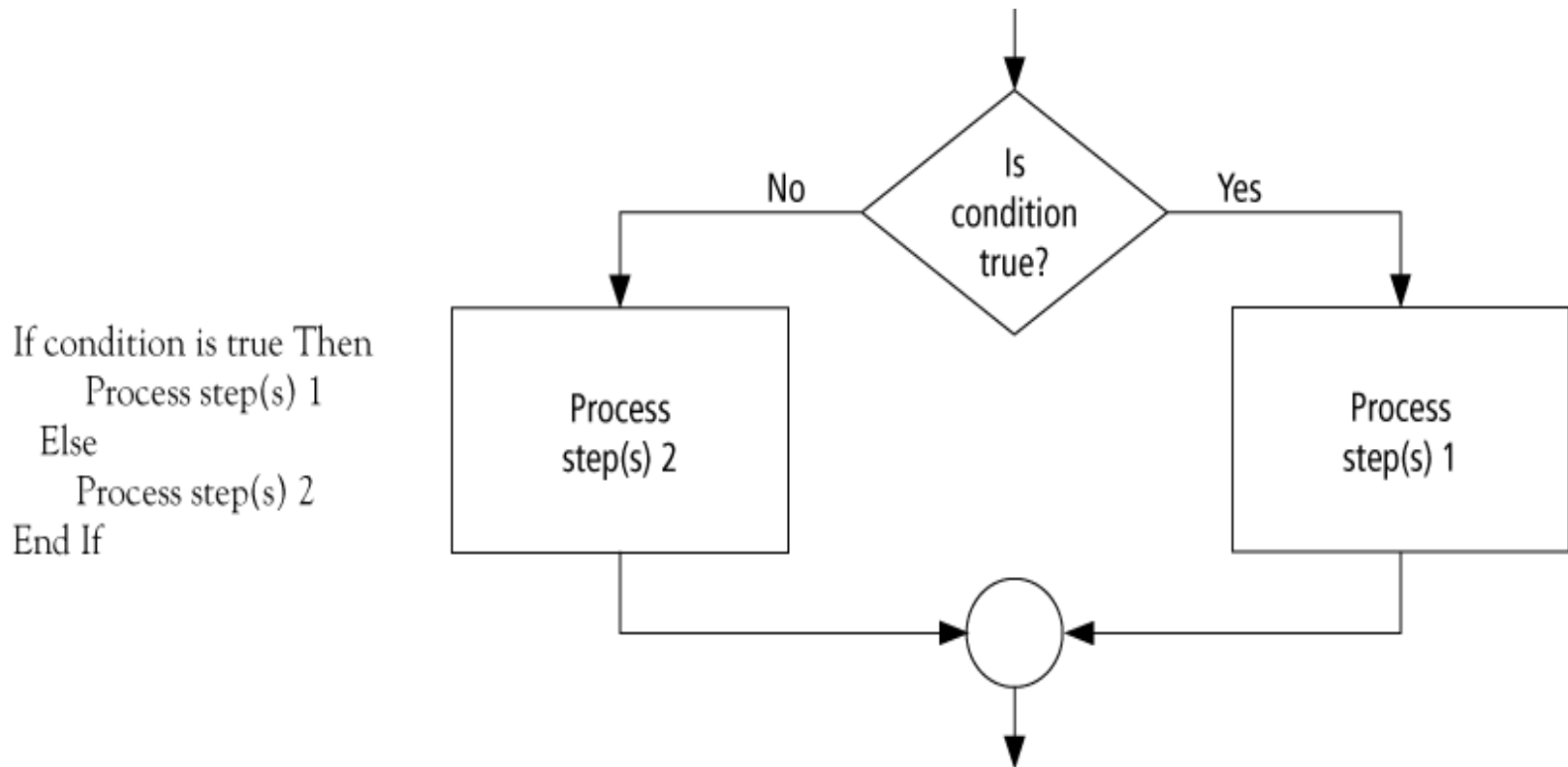
1. SEQUENCE



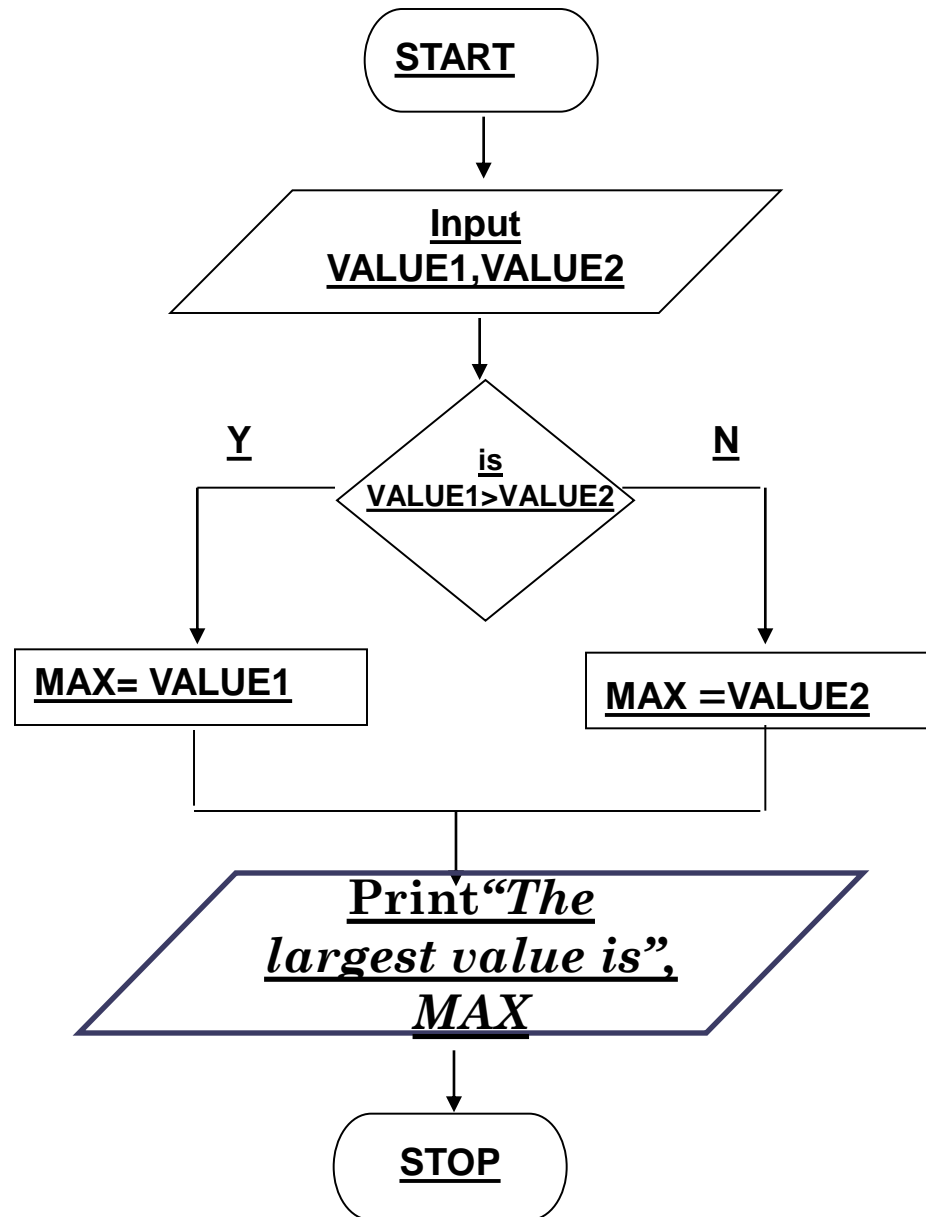
EXAMPLE ADD THREE NUMBERS



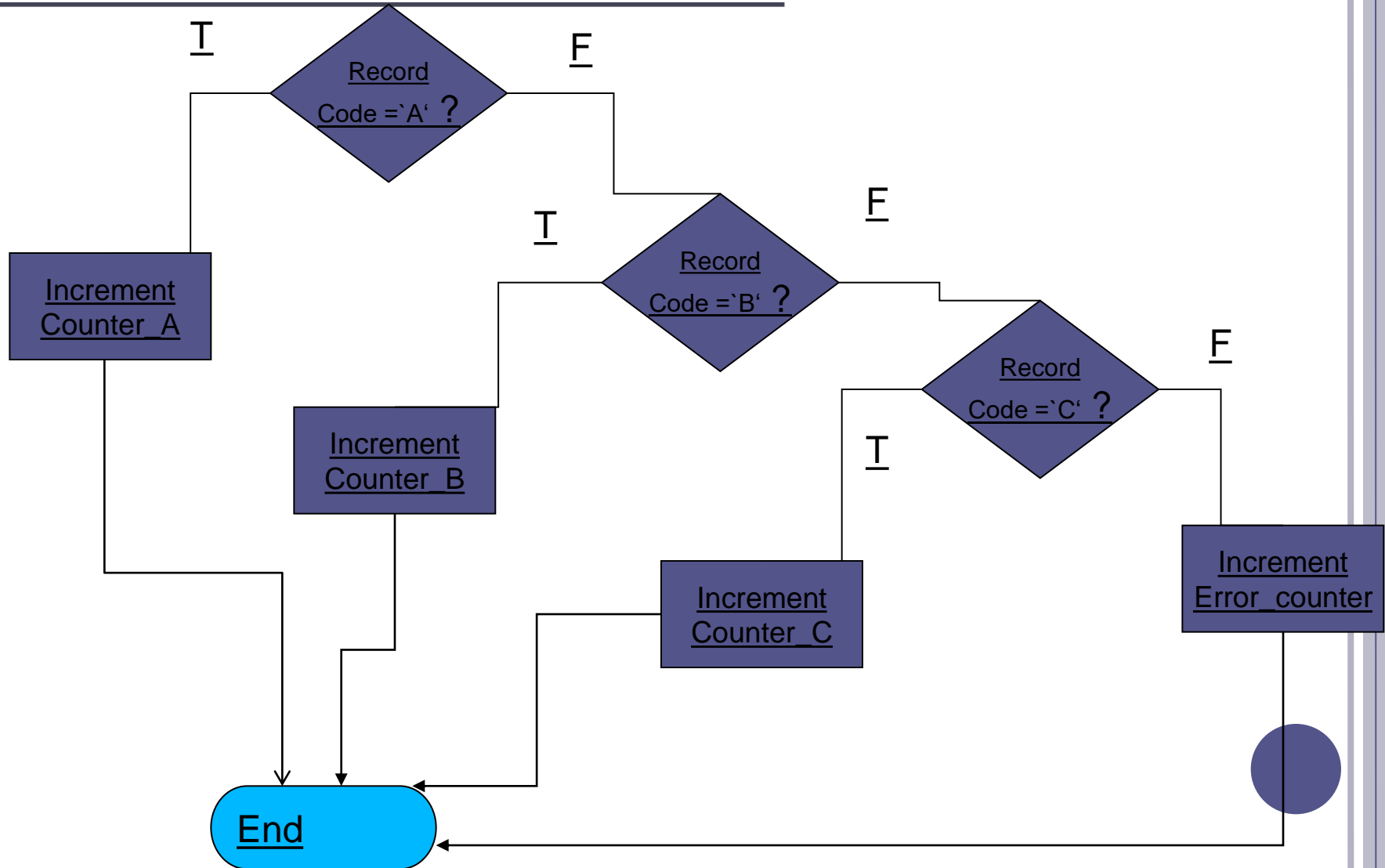
DECISION FLOW CHART



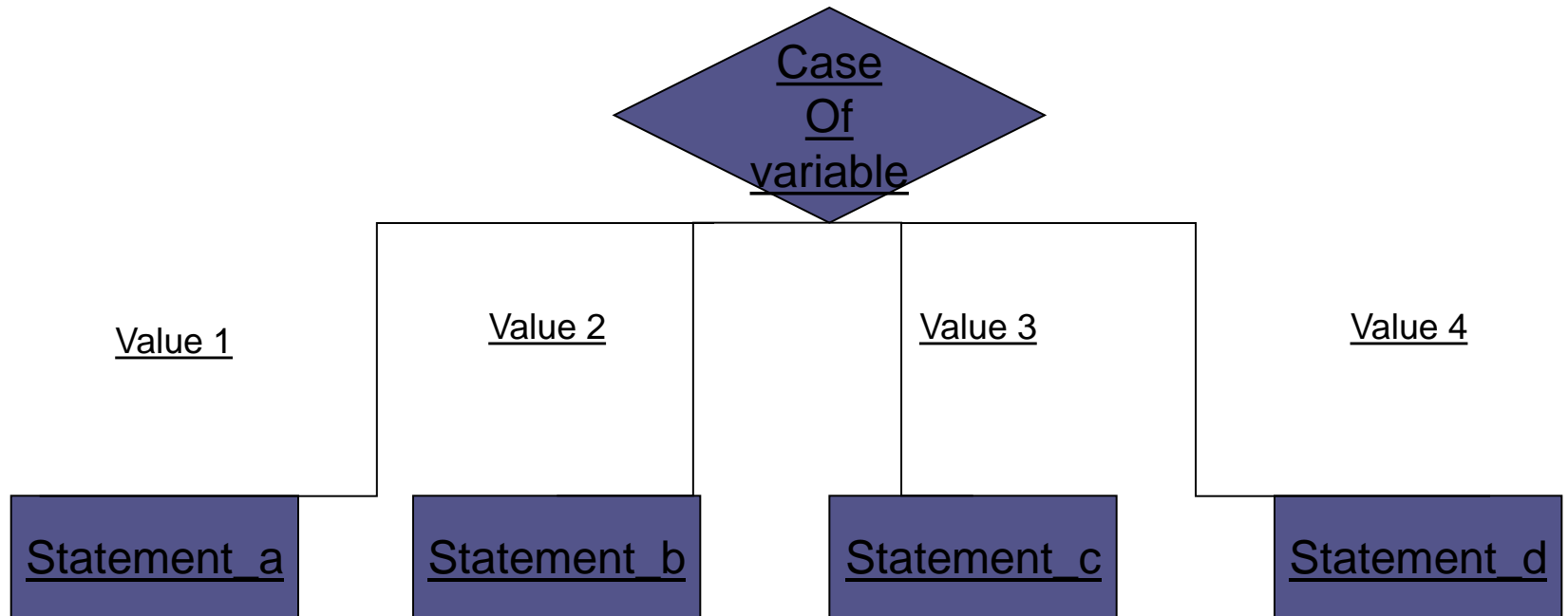
LARGEST OF TWO NUMBERS



NESTED IF STATEMENT

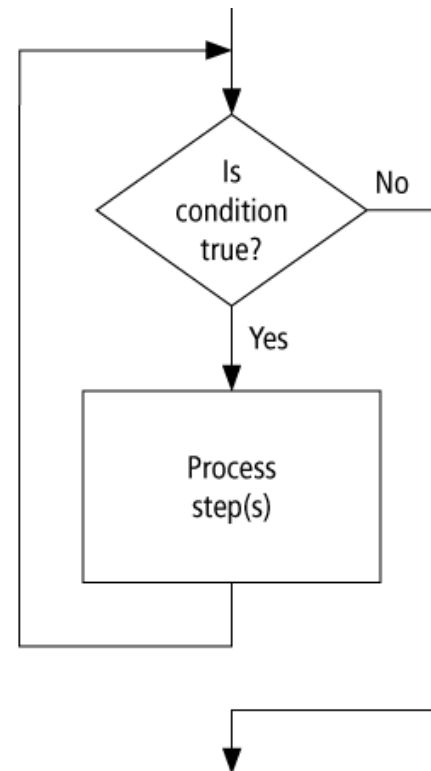


CASE STRUCTURE

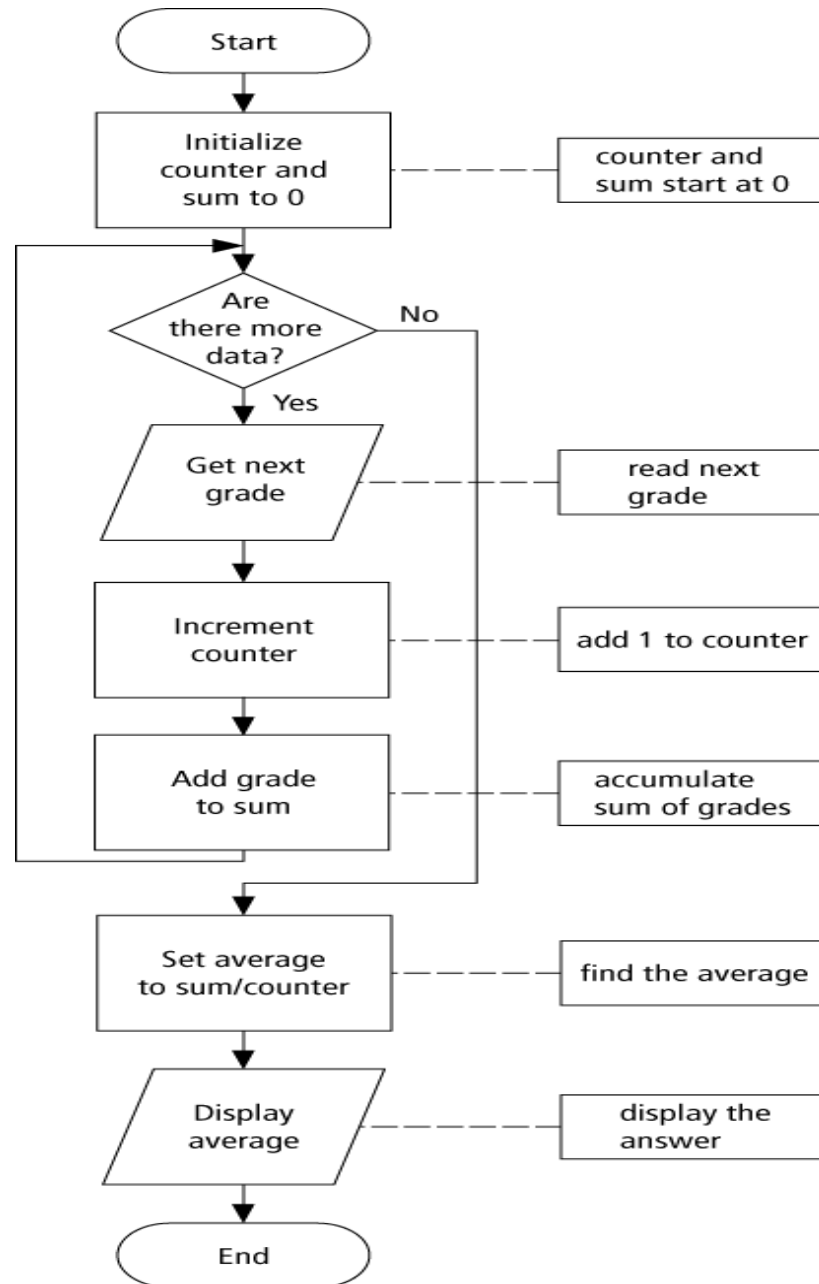


LOOPING FLOW CHART

Do While condition is true
Process step(s)
Loop



FLOWCHART



PRACTICE QUESTION

- 30 Candidates in engg. examination are required to take 5 single subject papers. For passing the exam, a candidate must score at least 35 in each subject and obtain an average of at least 40. The details each candidate i.e. name, rno, m1, m2, m3, m4, m5 is stored in a file. Write a pseudo code to process a file and print the list of successful candidates with their rollno, name and total marks.



Complexity of Algorithms

Prof. Pujashree Vidap



HOW TO ANALYZE ALGORITHMS

Problem statement: Given numbers m, n . Find Gcd

Algorithm GCD(m, n)

{

1. Factorize m such that $m = m_1 * m_2 * \dots$

2. Factorize n such that $n = n_1 * n_2 * \dots$

3. Identify common factors, multiply and return result

}



Algo2:

Euclid(m,n)

1. {
2. while m does not divides n
3. { r= n mod m
4. n=m
5. m=r
6. }
7. return m;
8. }



EXAMPLE:

$$\underline{m=36}$$

$$\underline{n=48}$$

ALGO1:

$$\underline{m=2*2*3*3}$$

$$\underline{n=2*2*2*2*3}$$

$$\underline{\text{Common Factors } 2*2*3}$$

$$\underline{\text{g.c.d } 12}$$

ALGO2:

$$\underline{r = 48 \% 36}$$

$$\underline{= 12}$$

$$\underline{n = 36}$$

$$\underline{m = 12}$$

$$\underline{\text{since now } 12 \text{ divides } 36}$$

$$\underline{\text{iteration ended}}$$

$$\underline{\text{Gcd } 12}$$



WHAT DO YOU CONCLUDE??

- Which algorithm is better?



- Note:

Counting iterations will be useful through out the course.

Judging algorithm have more direct relationship to performance



The algorithms are correct, but which is the best?

- Measure the running time (number of operations needed).
- Measure the amount of memory used.
- Note that the running time of the algorithms increase as the size of the input increases.



ALGORITHM ANALYSIS

An algorithm requires following 2 resources:

1. **Memory Space:** Space occupied by program code and the associated data structures.

1. **CPU Time:** Time spent by the algorithm to solve the problem.

Thus an Algorithm can be analyzed on two accounts *space* and *time*.



Space Complexity: Amount of memory algorithm needs to run to completion.

$S(P) = C + S_p$ (instance characteristics)

C: constant independent of characteristics of i/p and o/p. (e.g. code space, space for constants, fixed size components etc)

S_p : space needed by components whose size is dependent on problem instant

Time Complexity: $T(P)$ is time taken by algorithm P is compile time + runtime

Obtaining exact formula for runtime is impossible task so we have to count program steps.



Factors governing execution time of a program:

1. The speed of the computer system.
2. The structure of the program.
3. Quality of the compiler that has compiled the program.
4. The current load on the computer system.

It is thus better we compare the algorithm based on their relative time instead of actual execution time.

We can obtain total count for operations so we count only number of program steps



FRAMEWORK FOR ALGORITHM ANALYSIS

Basic Idea:

- To study mathematical model of computer
- Mentally Executes algorithm on computer model and evaluate time and not on real computer.



MATHEMATICAL MODEL OF COMPUTER

RAM model(Random Access Machine) :

- model of computation measures the run time of an algorithm by summing up the number of steps needed to execute the algorithm
- Has Processor+ memory
- Memory is a collection of locations
- Has one processor
- Executes one instruction at a time
- Each instruction takes “unit time”/ 1 time step
- Each memory access takes exactly one step
- Takes no notice of whether an item is in cache or on the disk



INSTRUCTION SET OF PROCESSOR

- One step execution
- Arithmetic and logical operations allowed
 $A = b + c$ (1 step instruction)
- Jump and conditional jumps
Goto
If($a > b$) then goto
- Pointer instruction
 $B = *c$
- Array operations
 $A[i] = b$
 $B = c[i]$



- $A = b + c * c - u$ 3 steps in RAM model
- $A[i] = b[i] + c[i]$
 - $x = b[i]$
 - $Y = c[i]$
 - $z = X + y$
 - $A[I] = Z$...4 steps in RAM model



Example

Algo: add(a[],b[])

{

for i:=1 to n

c[i]:=a[i]+b[i]

end for

}

- Lets us convert in RAM model and check the no of instruction required.



Algo: add(a[],b[])

{

1.i=1

2.If(i>n) goto 9

3.x=a[i]

4.y=b[i]

5.z=x+y

6.c[i]=z

7.i=i+1

8.goto 2

9.}

total : $1+n+1+b.n+2n=2+3n+bn=2+n(b+3)$



GENERAL ASSUMPTIONS

- $T_{a1}(n)$ =**Maximum** time taken by our algorithm $a1$ to solve any instance of size n . Measure of goodness of algorithm
- Worst case measure
- Functional form of $T(n)$ is important in both RAM and real computer. “linear”, “quadratic growth”. So functional form is important.
- We will always assume that n is large i.e we will solving larger problems
- We will ignore constants
- $T(n)$ is independent of computer. so we can classify algorithm rather than computer.



LET US ANALYZE MATRIX MULTIPLICATION ALGORITHM

Algorithm matmul(n,n)

```
1. for i=1 to n
2.     for j=1 to n
3.         c[i][j]=0
4.         for k=1 to n
5.             c[i][j] = c[i][j]+a[i][k]*b[k][j]
6.         end
7.     end
8. end
```



ANALYSE THIS IN RAM MODEL

- Order of n^3



IDEAL SOLUTION

- Express running time as a function of the input size n (i.e., $f(n)$).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.



DETERMINE FREQUENCY COUNT

problem1

1. $i=1$
2. While($i \leq n$) {
 1. $x=x+1$;
 2. $i=i+1$;
 3. }

Problem2

1. for ($L=1$; $L \leq n$; $L++$)
 1. for($j=1$; $j \leq L$; $j++$)
 1. $x=x+1$;



DETERMINE FREQUENCY COUNT

Problem 3

1. for (i=1; i<=n; i++)
2. for (j=1; j<=i; j++)
3. for (k=1; k<=n; k++)
4. x=x+1;

Problem 4

1. for (i=1; i<=n; i++)
2. for (j=1; j<=i; j++)
3. for (k=1; k<=j; k++)
4. x=x+1;



THANK YOU



ASYMPTOTIC NOTATIONS

By
Pujashree Vidap



ASYMPTOTIC NOTATIONS

○ Introduction:

- Total step count is useful to represent time complexity of program and it is function of input(no of elements) or output or other simple characteristics.
- But it is not same for many algorithm such as Binary Search algorithm as for the same n , step count varies with position of x (searching element).
- When chosen parameter not adequate to determine step count uniquely, we extract 3 kinds of step count for given parameter n
 - Best step count : Minimum number of step count
 - Average step count: Average number of step count
 - Worst step count: Maximum number of step count
- We can use step count to compare time complexities of algorithms. Eg. $c_1 n^2 + c_2 n$ and $c_3 n$



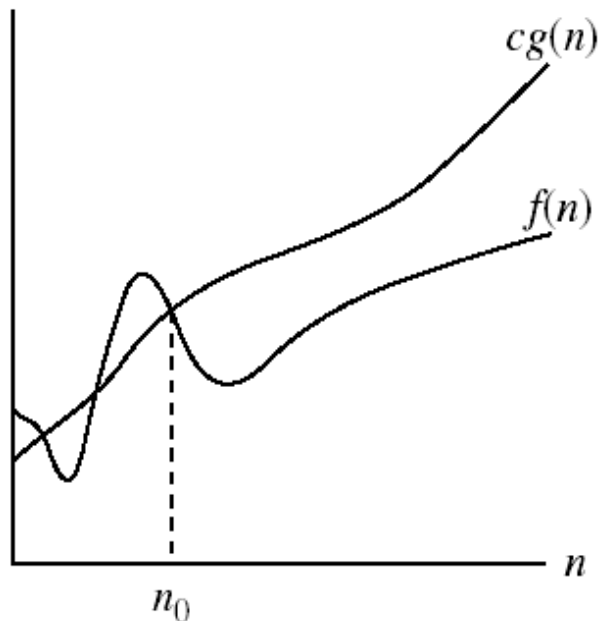
ASYMPTOTIC NOTATIONS

- Notation used to represent/describe complexities of an algorithms
- It treats an algorithm as function.
- Using this we are classifying functions into classes.
- Three notations:
 - Theta notation Θ
 - Big O notation O
 - Omega notation Ω



ASYMPTOTIC NOTATIONS

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\} .$



$g(n)$ is an *asymptotic upper bound* for $f(n)$.



Examples:

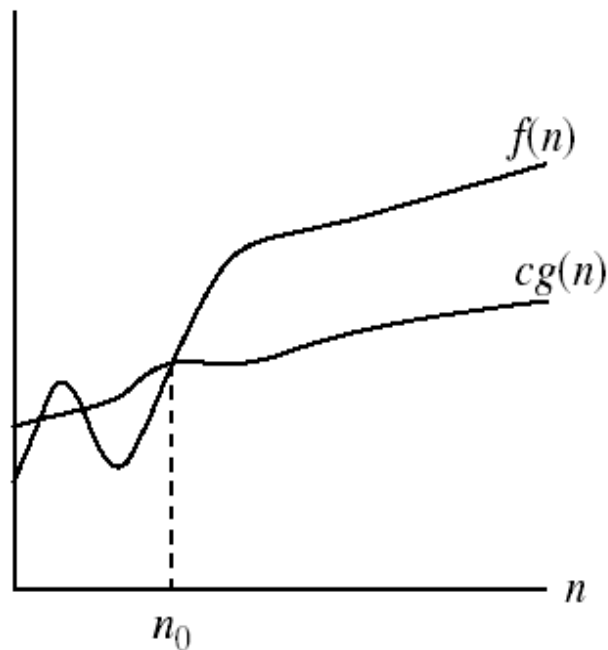
Prove that if $T(n)=10n^2+4n+2$, $T(n)=O(n^2)$.

Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = O(n^3)$.

Prove that if $T(n) = 6*2^n+n^2$, $T(n) = O(2^n)$.



$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\} .$



$g(n)$ is an *asymptotic lower bound* for $f(n)$.



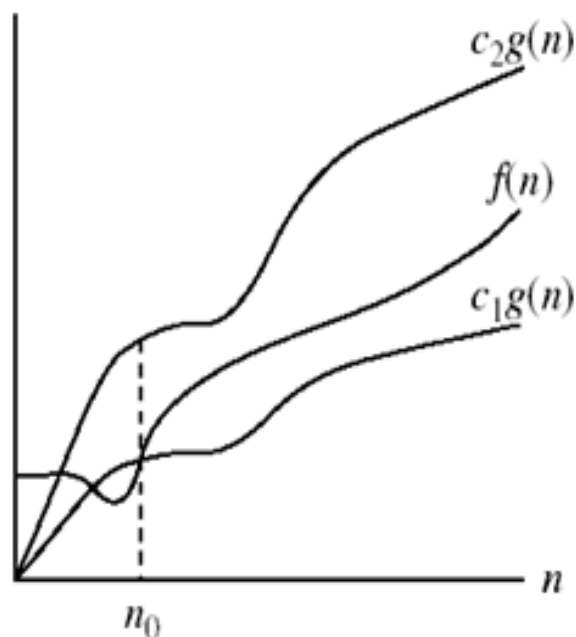
Examples:

- Prove that if $T(n) = 3n + 2$, $T(n) = \Omega(n)$
- Prove that if $T(n) = 10n^2 + 4n + 2$, $T(n) = \Omega(n^2)$
- Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = \Omega(n^3)$



- Θ -notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\} .$



$\Theta(g(n))$ is the set of functions
with the same order of growth
as $g(n)$

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

Examples:

$$T(n) = 10n^3 + 5n^2 + 17$$

$$10n^3 \leq f(n) \leq (10 + 5 + 17)n^3$$

$$c_1 = 10 \quad c_2 = 32$$

For all $n \geq 1$ so $n_0 = 1$

$$T(n) = \Theta(n^3)$$

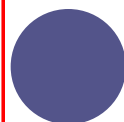
Solve:

$$T(n) = n^3 + n \log n = \Theta(n^3)$$



Summary of the Notation

- $f(n) = O(g(n)) \Rightarrow f \preceq g$
- $f(n) = \Omega(g(n)) \Rightarrow f \succeq g$
- $f(n) = \Theta(g(n)) \Rightarrow f \approx g$
- It is important to remember that a Big-O bound is only an *upper bound*. So an algorithm that is $O(n^2)$ might not ever take that much time. It may actually run in $O(n)$ time.
- Conversely, an Ω bound is only a *lower bound*. So an algorithm that is $\Omega(n \log n)$ might actually be $\Theta(2^n)$.

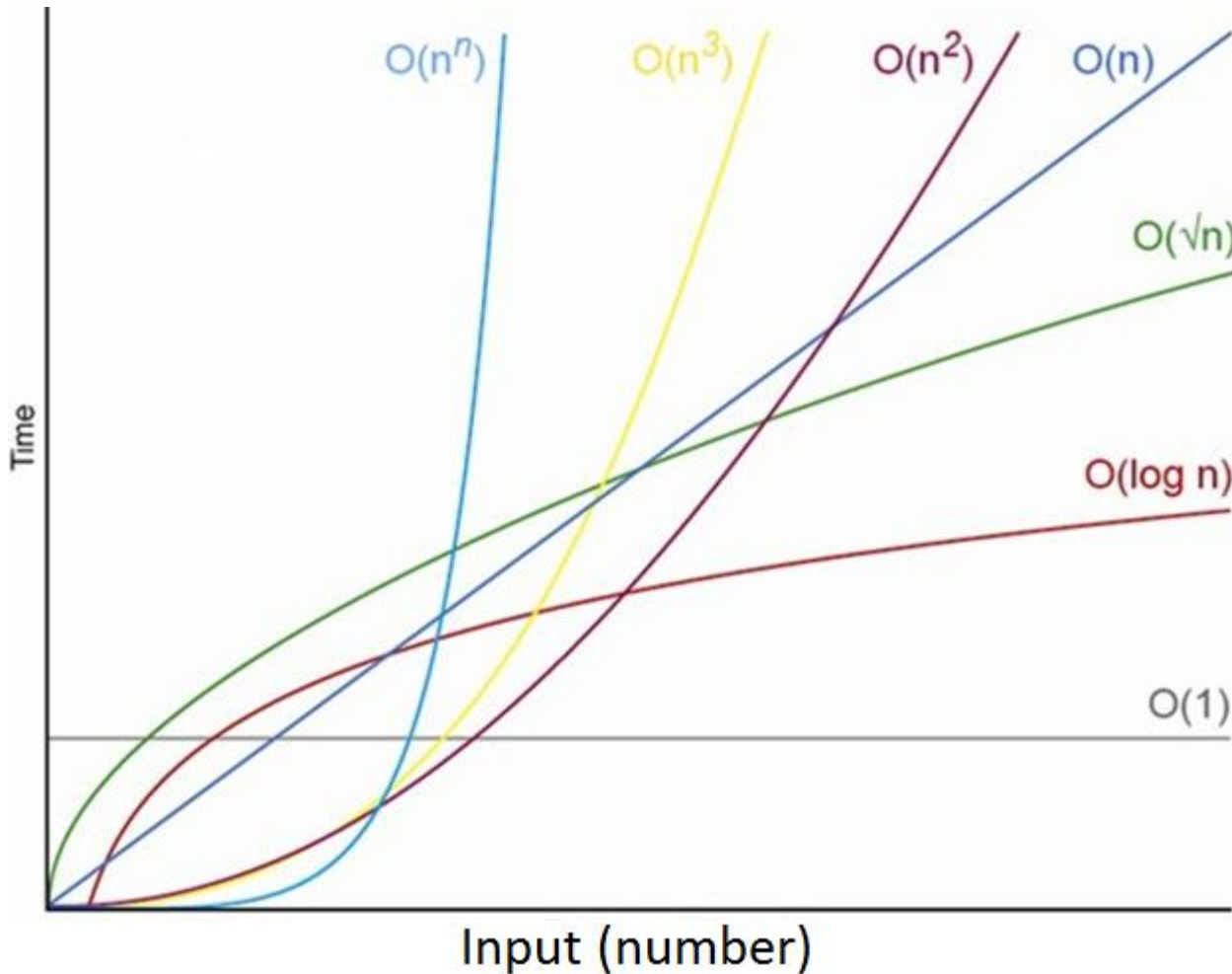


COMMON RATES OF GROWTH

- Constant: $\Theta(k)$, for example $\Theta(1)$
- Linear: $\Theta(n)$
- Logarithmic: $\Theta(\log_k n)$
- $n \log n$: $\Theta(n \log_k n)$
- Quadratic: $\Theta(n^2)$
- Polynomial: $\Theta(n^k)$
- Exponential: $\Theta(k^n)$



Analysis of programming constructs- Linear, Quadratic, Cubic, Logarithmic



	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}



Classification Summary

We have seen that when we analyze functions asymptotically:

- Only the leading term is important.
- Constants don't make a significant difference.
- The following inequalities hold asymptotically:

$$c < \log n < \log^2 n < \sqrt{n} < n < n \log n$$

$$n < n \log n < n^{(1.1)} < n^2 < n^3 < n^4 < 2^n$$

- In other words, an algorithm that is $\Theta(n \log(n))$ is more efficient than an algorithm that is $\Theta(n^3)$.

Example I: Find time complexity of following algorithm in terms on $O(\text{Big } O)$

Algorithm areaTriangle(base, height, area)

{

1.Read base, height

2.area=0.5*base*height

3.Print area

}



- The algorithm takes 3 steps to complete. Thus

$$T(n) = 3$$

- To satisfy the relation $T(n) \leq c * g(n)$,
the above relation can be written as

$$T(n) \leq 3 * 1$$

- Comparing the above two relations we get

$$c = 3, \quad g(n) = 1$$

- Therefore, the above relation can be expressed as:

$$T(n) = O(g(n))$$

$$T(n) = O(1)$$

- We can say that the time complexity of above algorithm is $O(1)$ i.e. of order 1



Example II:

Algorithm sum(a[], n ,sum)

{

1. sum=0

2.for(i=0 to n-1)

 2.1 sum = sum + a[i]

3.Print sum

}



- The algorithm takes $2n+3$ steps to complete. Thus

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

- To satisfy the relation $T(n) \leq c * g(n)$,
the above relation can be written as

$$T(n) \leq 2n + 3$$

$$T(n) \leq 2n + n$$

$$T(n) \leq 3n$$

- Comparing the above two relations we get

$$c = 3, \quad g(n) = n$$

- Therefore, the above relation can be expressed as:

$$T(n) = O(g(n))$$

$$T(n) = O(n)$$

- We can say that the time complexity of above algorithm is $O(n)$ i.e. of order n



Exercise

- $T(n) = 2527$
- $T(n) = 8 * n + 17$
- $T(n) = 5 * n^2 + 6$
- $T(n) = 5 * n^3 + n^2 + 2 * n$



Thank you

Post the queries @ psvidap@pict.edu



SPACE COMPLEXITY

Prof. Pujashree Vidap



SPACE COMPLEXITY

The space complexity is the memory needs by an algorithm to run to completion.

The space requirement depends on

- 1)The **fixed part** that is **independent of** the characteristics of the **input and outputs**. This part typically includes the instruction space(code space), space for simple variable, space for fixed size component variables, space for constant.
- 2)The **variable part**: space needed for component variable whose size is dependent on particular instance, the space needed by **reference variables**, the **recursion stack** space.

$$S(P)=c + S_p(c \text{ is constant})$$

To analyze space complexity we will concentrate on $S_p(\text{instance characteristics})$



EXAMPLES

```
float Abc(float a,float b,float c)
{
Return a+b+b*c+(a+b-c)/(a+b);
}
```

Sp=0

Space complexity : $O(1)$



Algorithm sumarray (a,n)

{

s=0;

for i=0 to n-1

 s=s + a[i];

return s;

}

Space complexity : $O(n)$



```
Float Rsum(float *a,const int n)
{
If(n==0) return 0;
Else return(Rsum(a,n-1)+a[n]))
}
Sp=3(n+1)
```



TYPES OF DATA STRUCTURES

Prof. Pujashree Vidap



TYPES OF DATA STRUCTURES

○ Primitive and Non Primitive DS :

- Primitive(basic type ,machine level support, predefined way of storing and operations , used to make non primitive DS)
- Non primitive (Used to store group of values)

○ Linear and Non linear DS:

- Linear (Items are organized sequentially or linearly)
- Non Linear (Items are not organized sequentially)

○ Static and Dynamic DS:

- Static (Size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.)
- Dynamic (Size of the structure is not fixed and can be modified during the operations performed on it)



PERSISTENT AND EPHEMERAL DS

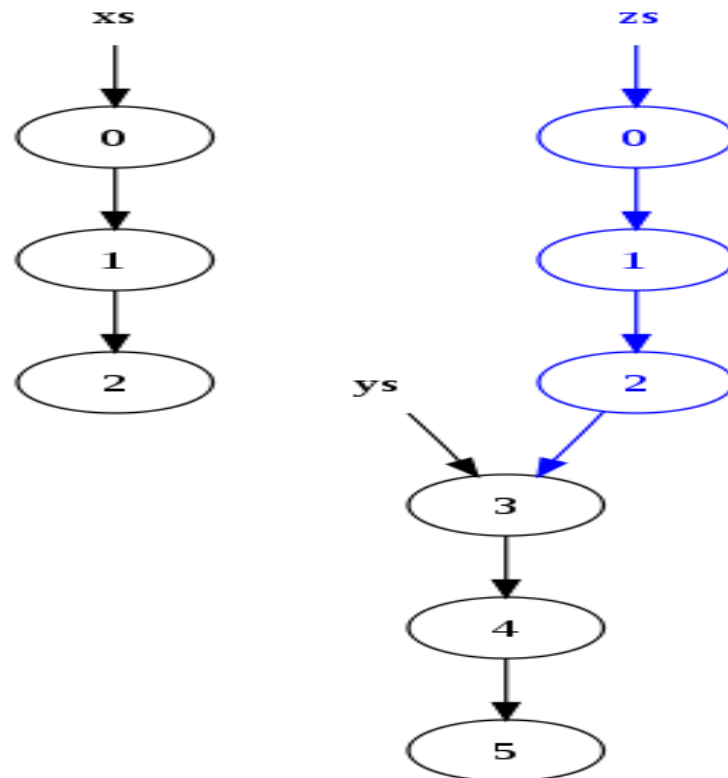
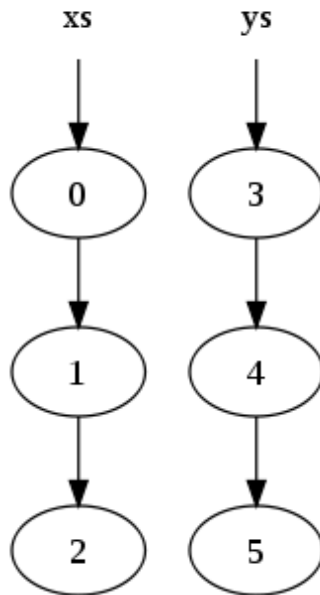
- **Persistent data structure** is a **data structure** that always preserves the previous version of itself when it is modified.
- Such **data structures** are effectively immutable, as their operations do not (visibly) update the **structure** in-place, but instead always yield a new updated **structure**
- A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified
- The data structure is **fully persistent** if every version can be both accessed and modified

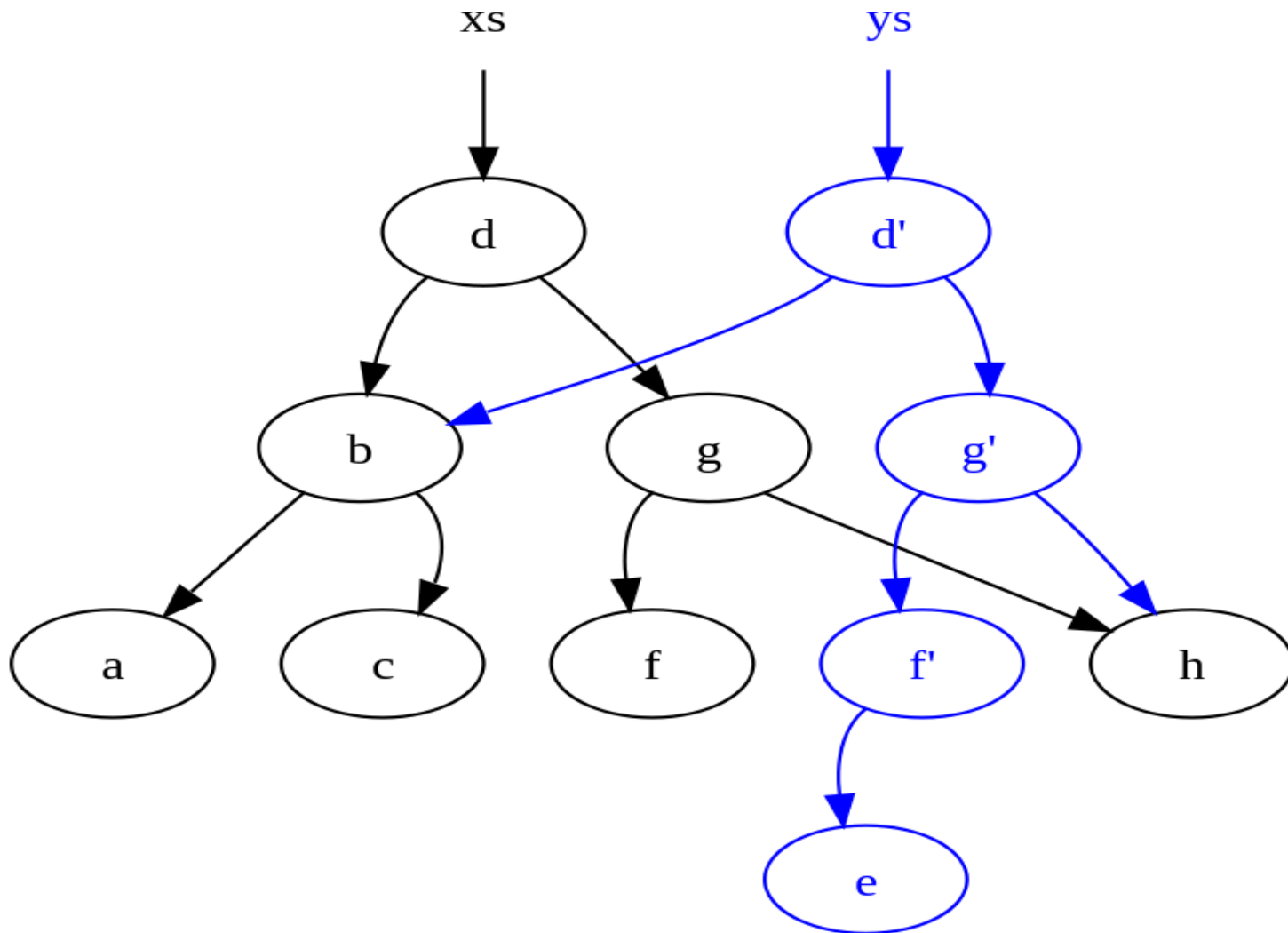


- ML-derived languages and Haskell, they support persistent DS
- For eg: $xs = [0, 1, 2]$ $ys = [3, 4, 5]$. Now concatenating the two lists:
 $zs = xs ++ ys$ results in the following memory structure:



After concating





`ys = insert ("e", xs)`

EPHEMERAL DS

- Data structures that are not persistent are ephemeral
- Modification destroys the version which we modified.
- Changes are destructive so only one version is available



Thank you

Post the queries @ psvidap@pict.edu



ALGORITHM DESIGN STRATEGIES

(DIVIDE AND CONQUER)

By
Pujashree Vidap



ALGORITHMIC STRATEGY

- In problem solving for efficient solution we can use good strategy for algorithm design
 - ◆ DIVIDE AND CONQUER
 - ◆ GREEDY STRATEGY



DIVIDE-AND-CONQUER

- The strategy involves three steps at each level of recursion.
- **Divide:-** Divide the problem into a number of sub problems.
- **Conquer:-** Conquer the sub problems by solving them recursively (Recursively solve these sub problems). If the sub problem sizes are small enough, then just solve the sub problems in a straight forward manner.
- **Combine:-** Combine the solutions to the sub problems to form the solution for the original problem.



CONTROL ABSTRACTION FOR DIVIDE AND CONQUER STRATEGY

Algorithm DAndC (P)

```
{  
  if small(P) then return S(P) //termination condition  
  //small(P) is boolean function so answer can be computed  
  //without splitting if so function S is invoked  
  else  
  {  
    Divide P into smaller instances  $P_1, P_2, P_3 \dots P_k$   $k \geq 1$ ;  
    Apply DAndC to each of these sub problems.  
    return Combine(DAndC( $P_1$ ), DAndC ( $P_2$ ), ... DAndC ( $P_k$ )  
  }  
}
```



COMPUTING TIME OF DAndC

$$\begin{aligned} T(n) &= g(n) \quad \text{when } n \text{ is small} \\ &= T(n_1) + T(n_1) + T(n_2) + \dots + T(n_k) + f(n) \\ &\quad \text{otherwise} \end{aligned}$$

$T(n)$ denotes the time for DAndC on any input of size 'n'.

$g(n)$ is the time to compute the answer directly for small inputs.

$f(n)$ is the time for dividing 'P' and combining the solutions of sub problems.



Binary Search: DIVIDE AND CONQUER STRATEGY EXAMPLE

◆ Binary search algorithm:

- ◆ Is used for searching large lists
- ◆ Searches the element in very few comparisons
- ◆ Can be used only if the list to be searched is sorted



Implementing Binary Search (Contd.)

- ◆ Consider a list of 9 elements in a sorted array.

	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>
<u>arr</u>	<u>9</u>	<u>13</u>	<u>17</u>	<u>19</u>	<u>25</u>	<u>29</u>	<u>39</u>	<u>40</u>	<u>47</u>



Implementing Binary Search (Contd.)

- ◆ You have to search an element 13 in the given list.

	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>
<u>arr</u>	<u>9</u>	<u>13</u>	<u>17</u>	<u>19</u>	<u>25</u>	<u>29</u>	<u>39</u>	<u>40</u>	<u>47</u>



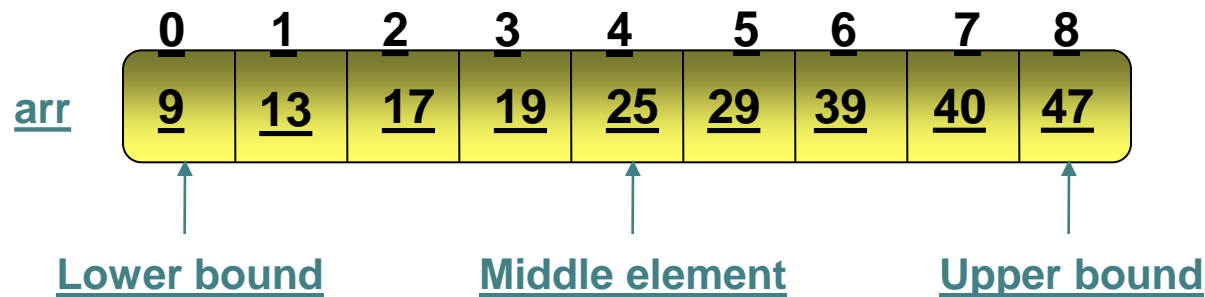
Implementing Binary Search (Contd.)

- ◆ Determine the index of the middlemost element in the list:

$$\text{Mid} = (\text{Lower bound} + \text{Upper bound})/2$$

$$= (0 + 8)/2$$

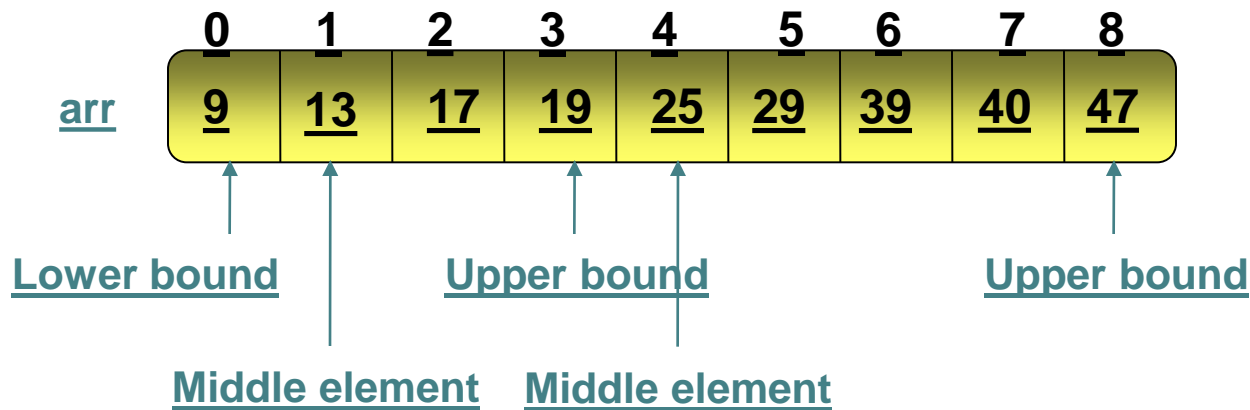
$$= 4$$



Implementing Binary Search (Contd.)

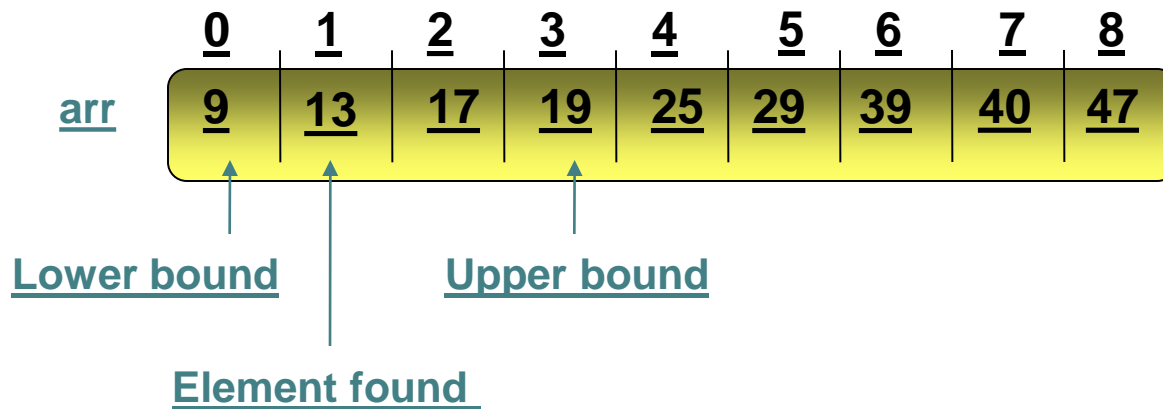
- ◆ 13 is not equal to the middle element, therefore, again divide the list into two halves:

$$\begin{aligned}\text{Mid} &= (\text{Lower bound} + \text{Upper bound})/2 \\ &= (0 + 3)/2 \\ &= 1\end{aligned}$$



Implementing Binary Search (Contd.)

- ◆ 13 is equal to middle element.
- ◆ Element found at index 1.



- ◆ Write an algorithm to implement binary search algorithm.

ALGORITHM BINARYSEARCH(A, N, Data)

{//A is an array of N elements. Data is the element to be searched

1. for i :=0 to N-1 do

1.1 read (A[i])

2. read(Data)

3. low := 0

4. high := N – 1

5. while(low<=high) {

5.1 mid := (low + high)/2

5.2 if (A[mid] = Data){

5.2.1 write “Found”;

5.2.2 exit ;}

5.3. If(Data < A[mid])

5.3.1 high := mid – 1

5.4.If(Data > A[mid])

5.4.1 Set low := mid + 1

}

6. Print “Not Found”

7. Exit

}



BINARY SEARCH USING RECURSION

Algorithm binsearch(a, low, high, Data , N)

{

5.if(low=high) then{

5.1 if(Data=a[low])then return low;

else return -1;

}

else {

5.1 mid := (low + high)/2

5.2 if (a[mid] = Data)

then return mid;

5.3 else if(x<a[mid]) then

return binsearch(a, low, mid-1,Data, N)

else

return binsearch(a, mid+1, high, data, N);

}

}



Efficiency of Binary Search

- ◆ In binary search, with every step, the search area is reduced to half.
- ◆ In the best case scenario, the element to be search is found at the middlemost position of the list:
 - ◆ The number of comparisons in this case is $1.O(1)$
- ◆ In the worst case scenario, the element is not found in the list:
 - ◆ After the first bisection, the search space is reduced to $n/2$ elements, where n is the number of elements in the original list.
 - ◆ After the second bisection, the search space is reduced to $n/4$ elements, that is, $n/2^2$ elements.
 - ◆ After i^{th} bisections, the number of comparisons would be $n/2^i$ elements.
- ◆ After k divisions, the length of array becomes 1. $n/2^k = 1$
 $\Rightarrow n = 2^k \Rightarrow \log_2 (n) = \log_2 (2^k) \Rightarrow \log_2 (n) = k \log_2 (2)$
 $K = O(\log_2 n)$



Just a minute

- ◆ In _____ search algorithm, you begin at one end of the list and scan the list until the desired item is found or the end of the list is reached.

◆ Answer:

◆ linear



Just a minute

◆ To implement _____ search algorithm, the list should be sorted.

◆ Answer:

◆ binary



ALGORITHM DESIGN STRATEGIES

(GREEDY APPROACH)

By
Pujashree Vidap



PRINCIPAL

Basic idea

- Find the optimal **solution** piece by piece.
- Iterative make decision one by one and hope, Everything works out well at the end.
- Once the decision is made then choice should not get changed for the subsequent steps.



OPTIMIZATION PROBLEMS

- For most optimization problems you want to find, not just *a* solution, but the *best* solution.
- A *greedy algorithm* sometimes works well for optimization problems. It works in phases. At each phase:
 - You take the best you can get right now, without regard for future consequences.
 - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.



EXAMPLE: COUNTING MONEY

- Suppose you want to count out a certain amount of money, using the **fewest possible bills and coins**
- A greedy algorithm to do this would be:
At each step, take the largest possible bill or coin that does not overshoot
 - Example: To make \$6.39, you can choose:
 - a \$5 bill
 - a \$1 bill, to make \$6
 - a 25¢ coin, to make \$6.25
 - A 10¢ coin, to make \$6.35
 - four 1¢ coins, to make \$6.39
- For US money, the greedy algorithm always gives the optimum solution

GENERAL PRINCIPAL

- The choice that seems best at the moment is the one we go with.



GREEDY ALGORITHM FAILURE

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
 - A 10 kron piece
 - Five 1 kron pieces, for a total of 15 krons
 - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
 - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

Greedy Algorithms

- The most straight forward design technique; can be applied to a wide range of problems
- Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints
- Any subset that satisfies these constraints is called a **feasible solution**
- we need to find a feasible solution that either maximizes or minimizes a given **objective function**
- A feasible solution that does this is called an **optimal solution**

Greedy Algorithms contd..

- The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time
- At each stage, a decision is made regarding whether a particular input is in an optimal solution
- This is done by considering inputs in an order determined by some selection procedure
- If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution; otherwise it is added
- The selection procedure itself is based on some optimization measure (objective function)

CONTROL ABSTRACTION

Greedy method control abstraction – subset paradigm

```
Algorithm Greedy(a,n) //a[1..n] contains n inputs
  solution =  $\Phi$  ; //initialize the solution
  for i=1 to n do
  {
    x=Select(a); //selecting an input from a
                  and removing it
    if Feasible(solution, x) then
      solution=Union(solution, x); //updates the
```



```
}  
return solution;
```

Feasible is a Boolean value function that determine whether x can be included in solution vector

Union function combines x with solution and updates the objective function

Note terms :-

- Feasible solution
- Optimal solution
- Objective function



KNAPSACK PROBLEM

Problem

Input:

- n objects.
- each object i has a weight w_i and a profit P_i
- Knapsack : M
- Output,
- Fill up the Knapsack Such that the total profit is maximized.
- Feasible solution: (x_1, \dots, x_n)

FORMALLY

Let x_i be the fraction of object i placed in the Knapsack,
 $0 \leq x_i \leq 1$. For $1 \leq i \leq n$.

Then : $P = \text{maximize } \sum p_i x_i$ for $1 \leq i \leq n$

Subject to $\sum w_i x_i \leq m$ for $1 \leq i \leq n$



EXAMPLE

$$n = 3, M = 20,$$

$$(p_1, p_2, p_3) = (25, 24, 15)$$

$$(w_1, w_2, w_3) = (18, 15, 10)$$

Sol:

Largest-profit strategy(Greedy method)

- Pick always the object with largest profit.
- If the weight of the object exceeds the remaining Knapsack capacity, take a fraction of the object to fill up the Knapsack
- The feasible solution is $(1, 2/15, 0)$

$$P = 25 + 2/15 * 24 = 25 + 3.2 = 28.2$$



2nd solution :Smallest-weight strategy

be greedy in capacity: do not want to fill the knapsack quickly.

- Pick the object with the smallest weight.
- If the weight of the object exceeds the remaining knapsack capacity, take a fraction of the object.
- Feasible solution : (0,2/3,1)

$$\begin{aligned} P &= 15 + 2/3 \cdot 24 \\ &= 15 + 16 = 31 \end{aligned}$$



Solution 3: Largest profit-weight ratio strategy

- Order profit-weight ratios of all objects.
- $P_i/w_i \geq (p_{i+1})/(w_{i+1})$ for $1 \leq i \leq n-1$
- Pick the object with the largest p/w
- If the weight of the object exceeds the remaining knapsack capacity, take a fraction of the object.

$$\begin{aligned} \text{➤ } p_1/w_1 &= 25/18 = 1.39 \\ p_2/w_2 &= 24/15 = 1.6 \\ p_3/w_3 &= 15/10 = 1.5 \end{aligned}$$

Optimal solution: $x_1 = 0, x_2 = 1, x_3 = 1/2$

$$\text{total profit} = 24 + 7.5 = 31.5$$



GREEDY-FRACTIONAL-KNAPSACK (M,N)

Greedy-fractional-knapsack (m,n)

**//p[1:n]and w[1:n] contains profits and weights
ordered such //that $p[i]/w[i]>p[i+1]/w[i+1]$,m is
knapsack size and x[1:n] is //solution vector.**

```
{  
for i =1 to n do x[i] =0;  
U=m;  
for i=1 to n do  
{  
if w[i]>U then break;  
X[i]=1.0;U=U-w[i];  
}  
if (i<=n)then x[i]=U/w[i];  
}
```



ANALYSIS

- if the items are already sorted into decreasing order of p_i / w_i
- then the for-loop takes $O(n)$ time
- Therefore, the total time including the sort is in $O(n \log n)$ as quicksort's average time complexity is $O(n \log n)$.



Solve the following example

Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

$$n = 5$$

$$w = 60 \text{ kg}$$

$$(w_1, w_2, w_3, w_4, w_5) = (5, 10, 15, 22, 25)$$

$$(b_1, b_2, b_3, b_4, b_5) = (30, 40, 45, 77, 90)$$

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90



Step-01:

Compute the value / weight ratio for each item-

Items	Weight	Value	Ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step-02:

Sort all the items in decreasing order of their value / weight ratio

<u>I1</u>	<u>I2</u>	<u>I5</u>	<u>I4</u>	<u>I3</u>
(6)	(4)	(3.6)	(3.5)	(3)

Now,

Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.

Since in fractional knapsack problem, even the fraction of any item can be taken.

So, knapsack will contain the following items-

< I1 , I2 , I5 , (20/22) I4 >

- Total cost of the knapsack

$$= 160 + (20/22) \times 77$$

$$= 160 + 70$$

$$= 230 \text{ units}$$

Thank you

Post the queries @ psvidap@pict.edu



ADT-ABSTRACT DATA TYPE

- Tool for specifying logical properties of data type.
- It is a mathematical model for data types.
- It is defined in terms of collection of possible values and set of operations on those values.
- The definition of ADT is not concerned with the implementation details at all.
- Gives an implementation independent view.



Consider INTEGER ADT which: defines the set of objects as numbers (-infinity, ... -2, -1, 0, 1, 2, ..., +infinity);

- specifies the set of operations: integer addition, integer subtraction, integer multiplication, div (divisor), mod (remainder of the divisor), logical operations like <, >, =, etc.

The specification of the INTEGER ADT does not include any indication of how the data type should be implemented.



abstract typedef<integer, integer>Rational;
condition Rational[1]!=0; /*value defination*/

/*operator defination*/

abstract Rational makerational(a,b)

integer a,b;

Precondition b!=0;

Postcondition makerational[0]=a;

makerational[1]=b;

abstract Rational add(a,b)

Rational a,b;

Postcondition add[0]=a[0]*b[1]+b[0]*a[1];

add[1]=a[1]*b[1];



abstract Rational mul(a,b)

Rational a,b;

Postcondition mul[0]=a[0]*b[0];

mul[1]=a[1]*b[1];

abstract Rational equal(a,b)

Rational a,b;

Postcondition equal==(a[0]*b[1]==b[0]*a[1])



ADT for varying length character strings

```
abstract typedef <<char>> STRING;  
abstract length(STRING s);  
postcondition length == len(s);  
abstract STRING concat( STRING s1, STRING s2);  
postcondition concat == s1+s2;  
abstract STRING substr( STRING s1, int i, int j);  
precondition 0<=i<len(s1);  
0<=j<len(s1)-i;  
postcondition substr ==sub( s1, i, j);
```



RECURRENCES AND RUNNING TIME

- An equation or inequality that describes a function in terms of its value on smaller inputs.

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

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
 - Find an explicit formula of the expression
 - Bound the recurrence by an expression that involves n

EXAMPLE RECURRENCES

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
- $T(n) = T(n/2) + c$ $\Theta(\log n)$
- $T(n) = T(n/2) + n$ $\Theta(n)$
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$

SOLVING RECURRENCES

$$\underline{T(n) = c + T(n/2)}$$

$$\underline{T(n) = c + T(n/2)}$$

$$\underline{= c + c + T(n/4)}$$

$$\underline{= c + c + c + T(n/8)}$$

Assume $n = 2^k$

$$\underline{T(n) = c + c + \dots + c + T(1)}$$

$$\underline{= \underbrace{c + c + \dots + c}_{k \text{ times}} + T(1)}$$

$$\underline{= O(\log n)}$$

$$\underline{T(n) = n + 2T(n/2)}$$

$$\underline{T(n) = n + 2T(n/2)}$$

$$\underline{= n + 2(n/2 + 2T(n/4))}$$

$$\underline{= n + n + 4T(n/4)}$$

$$\underline{= n + n + 4(n/4 + 2T(n/8))}$$

$$\underline{= n + n + n + 8T(n/8)}$$

$$\underline{\dots = in + 2^i T(n/2^i)}$$

$$\underline{= kn + 2^k T(1)}$$

$$\underline{= n \log n + n T(1) = O(n \log n)}$$

$$\underline{\text{Assume: } n = 2^k}$$

$$\underline{T(n/2) = n/2 + 2T(n/4)}$$

Exercise

Solve recurrence relation for linear and binary search



Thank You.....
Any Questions!!!!

