Definition An algorithm is an ordered set of unambiguous,executable steps that defines a terminating process.Program Formal representation of an algorithm Process Activity of executing a program**Ordered Set**Steps in an algorithm must have a well-established structure in terms of the order in which its steps are executed Each step must be an executable instruction Example: “Making a list of all the positive integers” is not an executable instruction May involve more than one thread (parallel algorithms)**Unambiguous Steps** During execution of an algorithm, the information in the state of the process must be sufficient to determine uniquely and completely the actions required by each step The execution of each step in an algorithm does not require creative skills. Rather, it requires only the ability to follow directions.**Terminating Process**All execution of an algorithm must lead to an end There are, however, many meaningful application for non-terminating processes Computer science seeks to distinguish both Problems whose answers can be obtained algorithmically Problems whose answers lie beyond the capabilities of algorithmic systems **An algorithm is** an ordered set of unambiguous executable steps that defines a terminating process. **Algorithm and Its Representation** Like a story and a story book Different language translation of a textbook Example: converting temperature readings from Celsius to Fahrenheit 1. F = (9/5)C + 32 2.

Multiply the temperature reading in Celsius by 9/5 and then add 32 to the product 3. Implemented by electronic circuit Underlying algorithm is the same, only the representations differ **Level of Details** May cause problems in communicating algorithms Example: “Convert the Celsius reading to its Fahrenheit equivalent”This might suffice for meteorologists But a layperson would argue that this instruction is ambiguous The problem is that the algorithm is not represented in enough detail for the layperson **Algorithm Representation** Primitive Set of building blocks from which algorithm representations can be constructed Programming language Collection of primitives Collection of rules stating how the primitives can be combined to represent more complex ideas **Levels of Abstraction** Algorithm Procedure to solve the problem Often one of many possibilities Representation Description of algorithm sufficient to communicate it to the desired audience Always one of many possibilities **Machine Instructions as Primitives** Algorithm based on machine instructions is suitable for machine execution However, expressing algorithms at this level is tedious Normally uses a collection of higher level primitives, each being an abstract tool constructed from the low-level primitives provided in the machine’s language **Pseudocode** Less formal, more intuitive than the formal programming languages A notation system in which ideas can be expressed informally during the algorithm development process A consistent, concise notation for representing recurring

semantic structure Comparison with flow chart **The Point of Pseudocode** To communication the algorithm to the readers The algorithm will later turn into program Also help the program maintainer or developer to understand the program **Algorithm Discovery** Development of a program consists of iscovering the underlying algorithm Representing the algorithm as a program Algorithm discovery is usually the more challenging step in the software development process Requires finding a method of solving the problem **Problem Solving Steps** Understand the problem Get an idea Formulate the algorithm and represent it as a program Evaluate the program For accuracy For its potential as a tool for solving other problems **Difficulties** Understanding the problem There are complicated problems and easy problems A complete understanding of the problem before proposing any solutions is somewhat idealistic Get an idea Take the ‘Algorithm’ course Mysterious inspiration **Getting a Foot in the Door** Work the problem backwards Solve for an example and then generalize Solve an easier related problem Relax some of the problem constraints Divide and conquer Stepwise refinement top-down methodology Popular technique because it produces modular programs Solve easy pieces of the problem first bottom up methodology **Work the Problem Backwards** Simplify the problem Build up a scenario for the simplified problem Try to solve this scenario Generalize the special solution to general scenarios Consider a more general problem Repeat the process

Divide and Conquer Concept Not trying to conquer an entire task at once First view the problem at hand in terms of several subproblems Approach the overall solution in terms of steps, each of which is easier to solve than the entire original problem**Divide and Conquer** Steps be decomposed into smaller steps These smaller steps be broken into still smaller ones Until the entire problem has been reduced to a collection of easily solved subproblems Solve from the small subproblems and gradually have it all. **Iterative Structures** Used in describing algorithmic process A collection of instructions is repeated in a looping manner **Search Problem** Search a list for the occurrence of a particular target value If the value is in the list, we consider the search a success; otherwise we consider it a failure Assume that the list is sorted according to some rule for ordering its entries ***while vs. repeat* Structure** In repeat structure the loop's body is always performed at least once (posttest loop) While in while structure, the body is never executed if the termination is satisfied the first time it is tested (pretest loop) **A Sort Problem** Sort a list of names into alphabetical order The constraint is to sort the list "within itself" In other words, sort by shuffling its entries as opposed to moving the list to another location Typical in computer applications to use the storage space efficiently **Recursive Structures** Involves repeating the set of instructions as a subtask of itself An example is in processing incoming telephone calls using the call-waiting feature An incomplete telephone conversation is set aside

while another incoming call is processed Two conversations are performed But not in a one-after-the-other manner as in the loop structure Instead one is performed within the other**Recursion** Execution is performed in which each stage of repetition is as a subtask of the previous stage Example: divide-and-conquer in binary search **Characteristics of Recursion** Existence of multiple copies of itself (or multiple activations of the program) At any given time only one is actively progressing Each of the others waits for another activation to terminate before it can continue **Recursive Control** Also involves Initialization Modification Test for termination (degenerative case) Test for degenerative case Before requesting further activations If not met, assigns another activation to solve a revised problem that is closer to the termination condition Similar to a loop control **Software Efficiency** Measured as number of instructions executed Θ notation for efficiency classes Best, worst, and average case **Big-theta Notation** Identification of the shape of the graph representing the resources required with respect to the size of the input data Normally based on the worst-case analysis Insertion sort: Θ(*n2*) Binary search: Θ(lg *n*) (n2-n)/2 lg2(n+1) - 1 **Formal Definition** Θ(*n2*): complexity is k*n2*+o(*n2*) f(n)/n*2* → k, n →∞ o(*n2*): functions grow slower than *n2* f(n)/n*2* → 0, n → ∞ (n2-n)/2 = ½ n2 - ½ n o(n2) **Problem Solving Steps** Understand the problem Get an idea Formulate the algorithm and represent it as a program Evaluate the program For its potential as a tool for

solving other problems (the faster, the better) For accuracy **Software Verification** Evaluate the accuracy of the solution This is not easy The programmer often does not know whether the solution is accurate (enough) Example: Traveler’s gold chain **Ways to Level the Confidence** For perfect confidence Prove the correctness of a algorithm Application of formal logic to prove the correctness of a program For high confidence Exhaustive tests Application specific test generation For some confidence Program verification Assertions**Program Conditions** Preconditions Conditions satisfied at the beginning of the program execution Propagations The next step is to consider how the consequences of the preconditions propagate through the program **Program Verification** Assertions Statements that can be established at various points in the program To check for rightful precondition & propagation Proof of correctness to some degree Establish a collection of assertions If all assertions pass The program is correct to some degree **Asserting of Insertion Sort** Precondition N == 2, trivial Loop invariant of the outer loop The names preceding the Nth entry form a sorted list Termination condition The value of N is greater than the length of the list The list is sorted

Compilers vs. Interpreters Compilers Compile several machine instructions into short sequences to

simulate the activity requested by a single high-level primitive Produce a machine-language copy of a program that would be executed later **Interpreters** Execute the instructions as they were translated **Imperative Paradigm** Procedural paradigm Develops a sequence of commands that when followed, manipulate data to produce the desired result Approaches a problem by trying to find an algorithm for solving it **Object-Oriented Paradigm** ✹Grouping/classifying entities in the program ✹Entities are the objects ✹Groups are the classes ✹Objects of a class share certain properties✹Properties are the variables or methods

✹Encapsulation of data and procedures✹Lists come with sorting functions ✹Natural modular structure and program reuse ✹Inheriting from mother class definitions ✹Many large-scale software systems are developed in the object oriented fashion **Object vs. Class** ✹Some objects can be categorized into the same class ✹Objects in the same class might share the same property **Declarative Paradigm** ✹Emphasizes ✹"What is the problem?" ✹Rather than "What algorithm is required to solve the problem?" ✹Implemented a general problem-solving algorithm ✹Develops a statement of the problem compatible with the algorithm and then applies the algorithm to solve it **Functional Paradigm** ✹Views the process of program development as connecting predefined "black boxes," each of which accepts inputs and produces

outputs ✹Mathematicians refer to such “boxes” as functions ✹Constructs functions as nested complexes of simpler functions **Advantages of FP** ✹Constructing complex software from predefined primitive functions leads to well-organized systems ✹Provides an environment in which hierarchies of abstraction are easily implemented, enabling new software to be constructed from large predefined components rather than from scratch **Types of Statements**

✹Declarative statements ✹Define customized terminology that is used later in the program ✹Imperative statements ✹Describe steps in the underlying algorithms ✹Comments✹Enhance the readability of a program**Declaration Statements** ✹Data terms ✹Variables ✹Literals ✹Constants ✹Data types✹Declaring data terms with proper types ✹Data structure **Variables, Literals** EffectiveAlt ← Altimeter + 645

✹Variables ✹EffectiveAlt, Altimeter ✹Literals ✹645 **Data Type** ✹Common types ✹Integer, real, character, Boolean ✹Decides ✹Interpretation of data ✹Operations that can be performed on the data **Variable Declarations** ✹Pascal **`Length, width: real; Price, Tax, Total: integer;`** ✹C, C++, Java **`float Length, width; int Price, Tax, Total;`** ✹FORTRAN **`REAL Length, Width INTEGER Price, Tax, Total`** **Data Structure** ✹Conceptual shape of data ✹Common data structure

✹Homogeneous array ✹Heterogeneous array **Declaration of a 2D Array** ✹C **`int Scores[2][9];`**

✹Java `int Scores[][]=new int [2][9];` ✹Pascal `Scores: rray[3..4, 12..20] of integer;`

Operators ✹Operator precedence ✹Operator priority ✹Plus and minus ✹Multiply and divide ✹Add and subtract ✹Operator overloading ✹Exact function depends on the operand data types ✹12 + 43 ✹❻abc' + 'def' **Control Statements**✹Alter the execution sequence of the program ✹`goto` is the simplest control statement

Procedures ✹A procedure ✹A set of instructions for performing a task ✹Used as an abstract tool by other program units ✹Control ✹Transferred to the procedure at the time its services are required ✹Returned to the original program unit (calling unit) after the procedure is finished ✹The process of transferring control to a procedure is often referred to as calling or invoking the procedure **Functions**✹The 6th type of control ✹A program unit similar to procedure unit except that a value is transferred back to the calling unit ✹Example Cost = 2 * TotalCost(Price, TaxRate); **Input/Output Statements** ✹I/O statements are often not primitives of programming languages ✹Not really a control ✹Most programming languages implement I/O operations as procedures or functions ✹Examples `printf( "%d %d\n", value1, value2 ); cout << value << endl;` **Lexical Analyzer** ✹Reads the source program symbol by symbol, identifying which groups of symbols represent single units, and classifying those units ✹As each unit is classified, the lexical analyzer

generates a bit pattern known as a token to represent the unit and hands the token to the parser ✹Like mapping words according to a dictionary, except the dictionary here is much smaller and non-ambiguous **Parsing** ✹Group lexical units (tokens) into statements ✹Identify the grammatical structure of the program ✹Recognize the role of each component **Syntax Diagram** ✹Pictorial representations of a program's grammatical structure ✹Nonterminals (rectangles) ✹Requires further description ✹Terminals (ovals) **Parse Tree** ✹Pictorial form which represents a particular string conforming to a set of syntax diagrams ✹The process of parsing a program is essentially that of constructing a parse tree for the source program ✹A parse tree represents the parser's understanding of the programmer's grammatical composition **Syntax Tree Ambiguity** ✹There could be multiple syntax trees for one statement ✹When the results are the same, it is OK ✹When the results are not the same, we call the statement an **ambiguous statement Code Generation** ✹Given the parse tree, create machine code ✹Z ← X + Y;

✹Load X ✹Load Y ✹ADDI X Y ✹Complication ✹When X is an integer and Y is a floating point number ✹Convert X from integer to floating point number ✹Use ADDF instead

Intertwined Process ✹Lexical analyzer ✹Recognize a token ✹Pass to parser ✹Parser ✹Analyze grammatical structure ✹Might need another token ✹Back to lexical analyzer ✹Recognize a statement

✹Pass to code generator ✹Code generator ✹Generate machine code ✹Might need another statement ✹Back to code generator **Linker** ✹Most programming environments allow the modules of a program to be developed and translated as individual units at different times ✹Linker links several ✹Object programs ✹Operating system routines and utility software ✹#include <xxxx.h> ✹To produce a complete, executable program (load module) that is in turn stored as a file in the mass storage system **Loader** ✹Often part of the operating system's scheduler ✹Places the load module in memory ✹Important in multitasking systems ✹Exact memory area available to the programs is not known until it is time to execute it ✹Loader also makes any final adjustments that might be needed once the exact memory location of the program is known (e.g. dealing with the JUMP instruction) **Software Development Package** ✹Editor ✹Often customized ✹Example ✹Color for reserved words ✹Aligned indentation ✹Translator ✹The compiler/interpreter ✹The most important part ✹Debugger ✹To allow easy tracking of program states **Objects and Classes** ✹**Object** ✹Active program unit containing both data and procedures ✹**Class** ✹A template for all objects of the same type An Object is often called an **instance** of the class. **Components of an object** ✹**Instance variable** ✹Variable within an object ✹**Method** ✹Function or procedure within an object ✹Can manipulate the object's instance variables ✹**Constructor** ✹Special method to initialize a new object instance

Encapsulation ✹**Encapsulation** ✹A way of restricting access to the internal components of an object ✹Private vs. Public **Additional Concepts**✹Inheritance ✹Allows new classes to be defined in terms of previously defined classes ✹Polymorphism ✹Allows method calls to be interpreted by the object that receives the call ✹For example ✹draw() ✹Different for circle vs. square object **Program Concurrent Activities**

✹**Parallel** or **concurrent** processing ✹Simultaneous execution of multiple processes ✹True concurrent processing requires multiple CPUs ✹Can be simulated using time-sharing with a single CPU ✹Examples: Ada task and Java thread **Basic Idea** ✹Creating new process ✹Handling communication between processes ✹Problem accessing shared data ✹Mutually exclusive access over critical regions ✹Mechanism on the program ✹Data accessed by only one process at a time ✹Monitor ✹Mechanism on the data ✹A data item augmented with the ability to control access to itself **Logical Deduction** ✹ Either Kermit is on stage (Q) or Kermit is sick (P) ✹Kermit is not on stage (not Q) ✹Kermit is sick (P) **Resolution** ✹Combining two or more statements to produce a new, logically equivalent statement ✹Resolvent ✹A new statement deduced by resolution **Magic** ✹Deduction computations are implemented in the programming language ✹Resolutions are done automatically ✹All you need to do is to describe the ‘rules’ and ‘facts’ in the logical

forms **Unification** ✹The process of assigning values to variables so that resolution can be performed

Prolog ✹PROgramming in LOGic ✹A Prolog program consists of a collection of initial statements upon which the underlying algorithm bases its deductive reasoning