WORLD'S #1 ACADEMIC OUTLINE

QuickStudy COMPUTER

ALGORITHMS

DEFINITION

Computer algorithms:
- Define how computers process data to achieve desired outcomes
- Are **language** and **platform agnostic** (i.e., free from software and hardware constraints)
- Are expressed or represented and communicated in various forms, depending on the associated problem domain(s) and programming type (a.k.a. program structure)
- Are the starting point of all programming and computer design

EX: Prose, diagram, flow chart, and pseudo code

Formalization

- Turing, Minsky, Savage, and Gurevich created the framework for defining algorithms in rigorous mathematical and logical terms.
- Formalization enables standardized expressions, measurements, and analysis of algorithms.

Turing Machine

- Invented by Alan Turing in 1936
- Simple abstract computational device **intended to investigate** and **determine** the **extent** and **limitations** of **what can be computed by machines**
- Mathematical model of computation
- **Abstract machine** that **manipulates symbols** on a strip of tape **according to** a set of **rules**
- In its simplest form, a Turing machine operates on an infinite memory tape according to a set of user-specified instructions. It:
  - Uses a head to read and write symbols
  - Moves the head to different locations on the tape in order to scan

Note: The fundamental simplicity of the model enables a Turing machine to simulate the logic of any computer algorithm.

Oracle Machine

- A Turing machine with a black box, called an **oracle**, that is able to solve certain decision problems in a single operation
  - The problem can be of any complexity and class.

Turing Complete

- If a Turing reduction (a.k.a. Cook reduction) is possible, an algorithm is said to be **Turing complete**.

- A **Turing/Cook reduction** is:
  - A reduction that solves *A*, assuming the solution to *B* is already known
  - A function computable by an **oracle machine** with an oracle for *B*

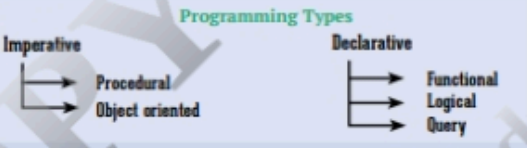
Algorithm

- A **computational process defined by** a **Turing machine**
- Any **sequence of operations** that can be **simulated by** a **Turing-complete system**

PROGRAMMING TYPES

Algorithm design and representation is dictated by the **programming type** or **program structure**.

- **Imperative programming:** Execution of a sequence of statements in linear and/or event-based time (most common type)



  - **Procedural:** Execution of a collection of procedures in linear or parallel time

  EX: Basic, C, Fortran, Java, and Pascal
  - **Object oriented:** Asynchronous interactions of a collection of objects via events and messages; often entails some linear processing

  EX: C#, C++, Java, Python, Ruby, and LUA
- **Declarative programming:** Characterizes computational goals without explicitly stating what steps must be performed
  - **Functional:** Evaluation of mathematical expressions without changing states or mutable data

  EX: F#, JavaScript, and LISP
  - **Logical (a.k.a. rule based):** Resolution of clauses within a logical system with facts, rules, and premises

  EX: Prolog, ASP, and Datalog
  - **Query:** Resolution of queries within a database with predefined structures, rules, constraints, and functionality

  EX: SQL, JPQL, and CQL

Note: Today's computer languages are no longer pure. No language is solely of one programming type; they all incorporate elements from other types as needed.

EX: A pure query language does not have control flow statements, but today's SQL dialects routinely include IF, WHILE, and other imperative programming elements.

FUNDAMENTALS

Algorithm Representation

- Also known as **algorithm expression**
- Algorithms are **represented** or **expressed** in many ways at different levels of specificity and detail, usually in the following order:
  - **Natural language:** Verbose and ambiguous; usually a statement of goals and vision
  - **Flow chart:** More specific and complete than natural language; employs (mostly) standard shape and flow methodologies
  - **Pseudocode:** Exact and detailed; aligns with and often resembles programming code

  Note: Syntax and grammar vary by organization.
  - **Programming language:** Programming code that executes to produce results (the final goal of any computer algorithm)
- Detailed representations make use of a standardized algorithm language (pseudocode) with components specific to the type of programming being employed.
  - **Imperative language set:** Variables, conditionals, control flow, assignments, and loops

  EX: Pseudocode and flowcharts
  - Declarative language sets:
    - **Functional:** Values, conditionals, process flow, and set algebra

    EX: Mathematical formulas and set algebra
    - **Logical:** Facts, statements, conditionals, process flow, and statement resolution

    EX: Logic systems/postulates and state diagrams
    - **Query:** Values, conditionals, relations, process flow, and query resolution

    EX: Queries, table definitions, database rules, and stored procedures

Algorithm Language & Syntax

- Algorithm language and syntax varies based on the standard. The following collection of concepts, terms, and syntax is common to most standards.
- Current computer languages often use components from other programming types.

EX: Today's SQL dialects provide a WHILE statement for easy iteration, whereas a pure functional language would implement iteration via recursion.

Applicable to All Program Types

- **Statement:** An action or step performed by an algorithm
  - Every line in an algorithm is some sort of statement.
  - Every **statement (a.k.a. line)** in an algorithm may be assigned a **line number** and/or **line label (a.k.a. code label)** in order to uniquely distinguish it from all other lines (statements) in the algorithm and enable redirecting execution to that line of the algorithm.
- **Block:** A sequence or collection of statements in a specific order and within a specific location of an algorithm
  - A block of code is indivisible and executes in order and in its entirety under normal conditions.
    - Block execution may be terminated or interrupted when asynchronous conditions arise (e.g., program messages or operating system events).
- **Comment:** Provides documentation for the various components of an algorithm
  - Comments are important for complex or compound algorithms to be self-explanatory.
- **Expression:** A combination of statements, function calls, and/or assignments

EX: (*x* < function(y)) is a Boolean expression composed of a variable reference and a function call.

- **Conditional (Boolean logic):**
  - To enable decisions in processing, Boolean logic is used to represent **logical conditions (a.k.a. conditionals)**, which are used for branching, looping, and conditional processing.

EX: (*x* < 1) is a Boolean expression that tests the value of *x* to be less than 1.

- **Value:** Any kind of data manipulated by an algorithm; usually defined by a data type
  - An *lvalue* refers to a data object that persists beyond a single expression.

- *L* is for left, as historically variables occurred on the left of a statement.
- An *rvalue* is a temporary value that does not persist beyond the expression that uses it.
  - *R* is for right, as historically values occurred on the right of a statement.
- **Imperative languages** require both *lvalue* and *rvalue* expressions.
- **Declarative languages** for the most part employ only *rvalue* expressions.

- **Variable:** A named data entity that contains a value, usually of a specified data type

EX: Int *x* declares the variable *x* of a data type integer.

- **Function (a.k.a. procedure, subroutine, and subprogram):** A named block of code that can be called by other statements, with or without input parameters; a function:
  - Performs known computational work
  - Always produces a result as either direct return values or indirect changes to the algorithm's logical state
- **Parameter:** The inputs of a function; a parameter:
  - Is a named data entity with a value that is usually defined by a data type
  - Is similar to a variable
  - Only exists and has meaning during the lifetime of the function it belongs to
- **Event/message handling:**
  - See **Process Communication**, p. 3.
  - An event (a.k.a. message):
    - Enables asynchronous and nonlinear processing
    - Can be triggered (invoked) by the operating system, the application framework, and even a specific statement

Applicable to Imperative Program Types

Assignment: The action of instantiating a variable with a value that is usually defined by a data type

EX: $x = 1$: Assigns the integer value 1 to the variable *x*

Control flow: The order in which the statements in an algorithm are executed

- Control flow statements alter the flow of execution based on conditional expressions.
- All **imperative languages** must have control flow statements.
- **Declarative languages** do not need control flow statements since they embody control flow as function calls.
  - Some current declarative languages include control flow statements to enable simpler algorithms, testing, and debugging.
- **If-else:** Used to branch execution flow, where a block of code is executed based on a conditional expression

EX: If $(x > 1)$ $x = x + 1$: If the value of *x* is greater than one, add one to the value; otherwise, the value of *x* remains unchanged.

- **Loops:** Allow for repetitive and iterative processing
  - A loop encompasses a code block, called the **loop body**, and repeats execution of that block according to a looping condition. The major types are:
    - **While loop:** Executes the loop body while a specific condition is being satisfied
      - The test for the condition is performed at the start (top) of the loop, so a while-loop's body may never be executed if the condition is not satisfied to start with.
    - **Until loop:** Executes the loop body until a specific condition has been satisfied
      - The test for the condition is performed at the end (bottom) of the loop, so an until-loop's body always executes at least once, regardless of the condition.
    - **For loop:** Iterates through a list of items, or set of numbers, until all list items or numbers have been processed
  - **Break:** A loop control statement that returns execution flow to the testing condition of the loop, bypassing the rest of the loop body
    - Breaks are most often used to detect and react to anomalous conditions.
- **Return:** A statement that immediately terminates execution of a function and returns the execution flow to the caller of that function
  - A return statement may explicitly return a value, variable, or function, or it may have no direct output value.
- **Goto (a.k.a. jump):** Immediately and unconditionally redirects execution to a specific line number or line label within an algorithm
  - Goto statements are deemed taboo in structured programming because algorithms that have them are hard (or impossible) to test, measure, and quantify. However, in rare cases they are absolutely essential to algorithm effectiveness and efficiency.
  - **Rule of thumb:** Only use a goto statement when it reduces complexity, time, and/or space by at least two orders of magnitude. Always profusely document it to ease testing and debugging.

Applicable to Functional Program Types

- As well as the common components on pp. 1–2, functional algorithms and programming languages include basic **set manipulation operations** as atomic statements of the language (e.g., union, intersection, membership, and subset).
- Functions (procedures) in functional algorithms are:
  - **Higher-order functions**, which can take functions as parameters and return functions as output
  - **Pure functions**, which have no side effects on current data variable values and only return a value (*rvalue*)
- Iteration and any looping are accomplished via recursion and set functions instead of iteration statements (e.g., WHILE).
- Functional programs maintain **referential transparency**.
  - There are no assignment statements in functional algorithms (programs).
    - Any variable values remain unchanged during execution and are therefore referentially transparent.
    - An assignment statement is not referentially transparent by its very nature and purpose.

EX: The assignment expression $x = x + 10$ incrementally changes the value of the variable *x*. In contrast, the pure function PlusTen(*x*) never changes the value of *x* while returning a specific new value.

```
x = x + 10      // not referentially transparent; value of x changes
PlusTen ( x )   // referentially transparent; value of x never changes
     return ( x + 10 )
```

Applicable to Logical Program Types

- Logical algorithms and programs are written in a logical language using the syntax of the associated **formal logic system**.
- Implementations of logical algorithms are realized as a collection of **facts**, **conditional statements** (premises), and other known data, together with the ability to process a proposed theorem based on known data.
- The execution of a logical algorithm is the process of proving or disproving a **theorem** via logical inference and deduction.
- Consider a simplified logic system that defines lineage and is able to resolve proposed theorems about lineage via inference.
  - The statements at the top of the figure below define the set of known data (facts and conditional statements), together with explanatory code comments.
  - Proposed theorems are input into the system to be evaluated.

```
Father ( Abe, Sally )                          // fact: Abe is the father of Sally
Mother ( Mary, Sally )                         // fact: Mary is the mother of Sally
Father ( Abe, George )                         // fact: Abe is the father of George
Father ( X, Y ) & Father ( X, Z ) :- Sibling ( Y, Z )   // premise: if X is the father of both Y & Z,
                                                        //   then Y & Z are siblings
Mother ( X, Y ) & Mother ( X, Z ) :- Sibling ( Y, Z )   // premise: if X is the mother of both Y & Z,
                                                        //   then Y & Z are siblings
:- Sibling ( Sally, George )                   // input theorem to be processed
TRUE                                           // output: theorem proven to be true using known data
:- Sibling ( Mary, George )                    // input theorem to be processed
FALSE                                          // output: theorem proven to be false using known data
```

Applicable to Query Program Types

- Query algorithms and programming languages define and process databases by combining elements of both functional and logical languages with database-specific functionality (e.g., the **select**, **join**, and **where** clauses to define conditionals).
- There is a rapidly expanding number of database types/structures and associated languages. Most primitive and foundational among these is the relational database.
- Query algorithms, similar to logical algorithms, first define a set of known data via tables of values with specific relationships. Then a query is processed, similar to a theorem being evaluated, in order to fetch (i.e., locate and return) the desired data.
  - **Relation:** Defines an existing or known relationship between two or more data values
    - Implemented as a database table, where the table represents the relationship, and the columns of the table embody the values being related
  - **Query:** An expression that states the conditions that desired data entities must satisfy
    - Evaluated by searching the referenced tables for data that matches the desired conditions
- Consider a simplified database system that defines lineage and is able to resolve queries about it.
  - The tables to the right show the set of known data.
  - A compound query is input into the system to return all siblings of Sally from the two given tables using database and set manipulation statements.

FatherTable

Father	Child
Abe	Sally
Abe	George
Ted	Ned

MotherTable

Mother	Child
Mary	Sally
Mary	George
Mary	Ned

```
select distinct Child from FatherTable where (FatherTable.Child <> "Sally") and
    (FatherTable.Father = (select distinct Father from FatherTable where FatherTable.Child = "Sally"))
union
select distinct Child from MotherTable where (MotherTable.Child <> "Sally") and
    (MotherTable.Mother = (select distinct Mother from MotherTable where MotherTable.Child = "Sally"))
```

Output:
George
Ned

Process State

- An algorithm's process state is provided and dictated by the operating system (OS).
- All operating systems provide these basic states, although some provide more evolved process states for parallel, query, quantum, and other types of processing:
  - **New:** A process is being created/allocated.
  - **Ready:** A process is ready for execution but has not been started.
  - **Running:** A process is executing/running.
  - **Blocked:** A process is blocked and is unable to continue; a.k.a. **hung** or **frozen.**

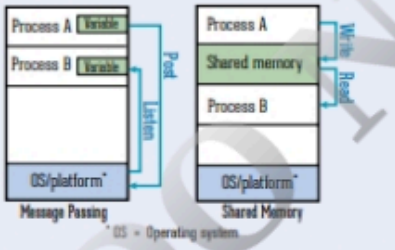
- **Deadlocked:** Two or more processes that are waiting for each other; a type of blocked state
- **Terminated:** A process has terminated or completed.
- **Waiting:** A process is suspended awaiting an event or message.

Logical State

- A logical state is:
  - The collection of all data and values at a given time during the execution of an algorithm
- A snapshot of all data in use by a process during execution
- An algorithm's starting logical state includes any **input** it may require; the ending logical state may include any **output** it produces.
- During execution, the algorithm makes changes to data and their values, thereby changing the logical state.
- It is useful to refer to the logical state of a process when designing test automation, instrumentation, and other metaprograms that operate above the process.

Process Communication

- A.k.a. **interprocess communication**
- Computer algorithms (processes) communicate with each other to cooperate in finding solutions to a given problem and interact with the operating system, hardware, and/or platform on which they run. Current systems and applications routinely employ hundreds of processes that interact with each other during execution. There are two main types:
- **Shared memory:** Refers to data and values accessible to all interacting processes. "Memory" refers to any persistent or nonpersistent data object stored on any medium (e.g., drive, RAM, or ROM).
  - **File and database:** Shared file, database, or record
  - **Pipe and socket:** Unidirectional (pipe) or bidirectional (socket) data channel that uses the physical operating system's input and output channels
  - **Parameter:** Data value or object passed from one process to another
    - **Variable (global):** Global data object accessible by all processes



- **Message passing:** Processes communicate by **posting** and **listening** for specific **messages** (a.k.a. events, although these are not to be mistaken with hardware and operating system events). Each process has one or more associated **message queues** where incoming messages are processed/reacted to.

EX: A process can post a message that includes associated data values rather than calling a process directly and passing that data as parameters.

- Shared memory allows for both *synchronous* and *asynchronous processing*.
- Message passing requires *asynchronous processing*.
- **Synchronous (a.k.a. linear) processing:** A process calls another and waits for the called process to complete.
- **Asynchronous processing:** A process posts a message for another process and immediately continues processing its own algorithm. In some future time, another process listening for that message will process it and react accordingly.

Note: Hardware and operating system events are a separate class of messages. Some of these low-level events can be ignored by a process. However, if a process causes a critical low-level event (e.g., out of memory), the operating system or platform will interrupt a running process. If the process does not handle the critical event, the process is terminated.

ALGORITHM CLASSIFICATION

Beware of conflicting and overlapping taxonomies!
Depending on the school of thought, scientific authority, and even organization-specific standards, the same term may have multiple and sometimes conflicting meanings.

TIP: Learn to understand core algorithm concepts using a single taxonomy, but always be accepting of new labels for these concepts, even if new labels encompass multiple meanings in different contexts.

Design Paradigm

An algorithm may employ (embody) one or more design paradigms.
EX: Iterative, recursive, and geometric
A design paradigm (a.k.a. **algorithm type**, **design approach**, and **algorithm classification**):

- Refers to the basic structure of an algorithm
- Dictates and informs the control flow for an algorithm

Process Type

An algorithm usually implements only one process type.
EX: Logical, parallel, and linear
A process type (a.k.a. **implementation type** and **process classification**):

- Refers to the types and flow of processes required by an algorithm
- Dictates and informs process flow/management for an algorithm

DESIGN PARADIGMS

Brute Force

- Systematically enumerate all possible solutions and linearly check whether each possible solution satisfies the problem's statement.

EX: A linear search of a list for a name starts with the first item and continues to the last item. Each item is examined to see if it's equal to the desired name.

- Brute-force (a.k.a. **exhaustive**) paradigms:
  - Can be useful for small data sets
  - Are simplest to design and implement

Iterative

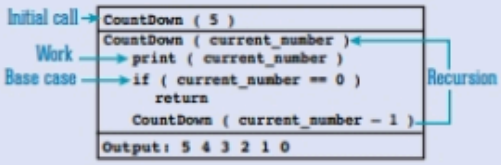
- Iterative paradigms:
  - Execute steps in iterations
    - These paradigms use an initial guess to generate a sequence of improving approximate solutions for a class of problems, in which the *n*th approximation is derived from all previous ones.
  - Are most commonly used in linear programs where large numbers of variables are required

Recursive

TIP: Recursion is counterintuitive and seems unnatural. To fully understand recursion, you must first suspend your disbelief and just accept that it works! As your understanding of computing grows, recursion will make more sense.

- A recursive algorithm calls itself with input of the same data types, where said input is simpler or reduced with each call.
- Recursion is the basis for efficient algorithms and complex problem solving.
- Functional components of recursion:
  - **Initial call:** Starts a recursive process (the recursion); must be made from outside the recursion
  - **Work:** Computational work being done by the algorithm at each stage of recursion
  - **Recursion:** A process's action of calling itself with new input
  - **Base case:** Termination and return condition for the recursion process
- The **direct tail recursion** figure below shows a recursive algorithm that counts backward from a given start number to zero and prints the numbers.
  - **Initial call:** Made from a calling process
  - **Work:** Printing the current number
  - **Recursion:** Calls itself passing the current number minus one
  - **Base case:** Reached when the current number is zero, which terminates the recursive process

Direct Tail Recursion



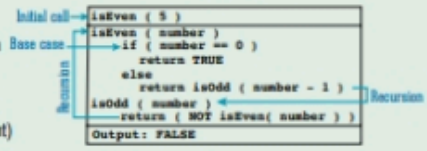
RECURSION TYPES

- A recursive algorithm is:
  - Designed as a combination of one or more **recursion types**
  - Implemented (realized) as one or more **recursion flow types** that must be congruent with the chosen recursion types

Direct vs. Indirect Recursion

- **Direct (simple):** An algorithm that contains an explicit call to itself (e.g., CountDown in the **Direct Tail Recursion** figure above)
- **Indirect (mutual):** An algorithm, *X*, that contains a call to another algorithm that in turn calls *X* (e.g., isEven and isOdd in the **Indirect (Mutual) Recursion** figure to the right)

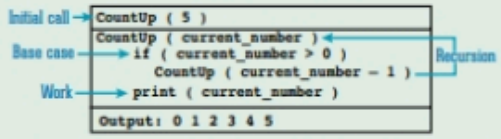
Indirect (Mutual) Recursion



Recursion Flow Types

- **Tail:** The recursive call occurs after the computational work is done (i.e., there is no pending computational work after the recursive call). See the **Direct Tail Recursion** figure, p. 3.
  - Tail recursion is important for functional programming to implement loops.
  - The data size is independent of the number of recursive calls.
- **Head (a.k.a. non-tail):** The recursive call occurs before the computational work (i.e., there is pending computational work after the recursive call).
  - The deepest call must be completed before any values are known. See the **Direct Head Recursion** figure below.
  - Head recursion is important for data-driven/generative algorithms to accumulate data.
  - The data size is directly proportional to the number of recursive calls.

Direct Head Recursion

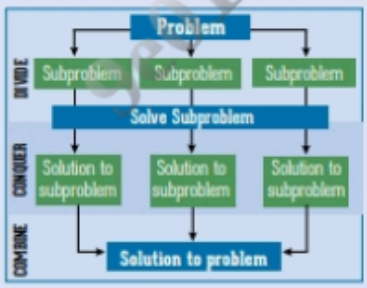


- **Middle:** The recursive call occurs in the middle of the algorithm, so there are work statements before and after the recursive call.
  - In practice, few algorithms have only one middle recursive call; most have two or more calls and are therefore called multibranched.
  - Middle recursion is important for efficient sort, search, and traversal algorithms as well as complex data structure processing.
  - The data size is indirectly proportional to the number of recursive calls.
- **Multibranched (a.k.a. multirecursive):** Two or more recursive calls occur within the algorithm based on one or more base cases.
  - There are work statements before and after the recursive calls.
  - The data size is indirectly proportional to the number of recursive calls multiplied by the number of recursive branches at each call level.
  - Multibranched recursion is:
    - A form of middle recursion
    - The basis for efficient divide-and-conquer algorithms (see the figure under **Divide & Conquer**, below)
    - The foundation for feasible search, sort, traversal, and machine inductive/deductive algorithms

Structural vs. Generative Recursion

- **Structural:** The parameter to each recursive call is a subset of the original input data.
  - Processing always terminates according to base case(s) in finite time.
  - Space usage is independent of the number and depth of recursive calls (call-tree span and depth).
  - CountDown in the **Direct Tail Recursion** figure on p. 3 is structural recursion implemented as tail recursion.
    - Additional data space is not required for each call.
    - The algorithm always terminates on the base case, regardless of input.
  - Structural recursion:
    - Includes tree traversals, search, and math/logic resolution
    - Is deterministically finite in space usage and execution time
    - Is often implemented as tail **recursion**
- **Generative:** The recursive call uses newly generated parameter data constructed from the original input.
  - Space usage grows incrementally with each successive recursive call.
  - CountUp in the **Direct Head Recursion** figure above is generative recursion implemented as head recursion.
    - Work statements (e.g., print(current_number)) are only executed after all recursive calls have completed.
    - Data space grows proportionally to the number of recursive calls.
  - Generative recursion:
    - Includes quicksort, binary search, and mergesort
    - Can be unbounded in space usage and execution time
    - Often requires controls (e.g., space and time limits) and **base case(s)** to terminate processing
    - Is often implemented as **head recursion** or **indirect recursion**

Divide & Conquer



- As shown in the figure above, divide-and-conquer algorithms break down the solutions to subproblems are combined to give a solution to the original problem.
- Divide-and-conquer algorithms are:
  - Often implemented as a multibranched recursion
  - The basis for efficient algorithms, including searching, sorting, mathematics, and machine induction

EX: Binary search and merge sort

Enumeration, Search & Sort

- Enumeration, search, and sort algorithms are among the fundamental building blocks for all algorithms.
  - Their efficiency directly determines an algorithm's feasibility in practical applications.
- **Enumeration:**
  - Is based on input and produces a list of potential answers to a computational problem
  - Does not include determining correct answers or solutions
  - Is combined with other algorithmic methods to determine viable and desired solutions
- **Search:** Locates specific data among an ordered collection of data items
  - The search criteria used to locate data is based on (is a subset of) the sort criteria used to order the list.
- **Sort:** Reorganizes a collection of data items into a specific order based on sort criteria (e.g., alphabetical and low-to-high criteria). There are two basic types:
  - **Comparison sort:** Organizes a list by a comparison operation (e.g., less than or equal to) that determines which of two elements should occur first in the final sorted list
    - The comparison is any abstract operation that results in a true or false, from simple and complex logic to compound function calls.
    - The complexity of the comparison is inversely proportional to the algorithm's efficiency.
  - **Integer sort:** Organizes a list via integer arithmetic performed on key data values in the list
    - This may require preprocessing or continual indexing to ensure all list elements have valid integer keys.

Note: In this context, **"integer key"** includes floating point numbers, rational numbers, and character strings. It is often more efficient than comparison sorts by several orders of magnitude (100x or more).

Other sort algorithms (a.k.a. what not to do): Sort algorithms used for educational purposes to demonstrate and teach algorithm concepts such as efficiency, boundedness, performance, correctness, and completeness

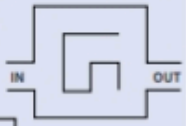
EX: *Bead sort* (can be $O(n)$ for any size dataset; only works with positive integers and requires specialized hardware); *spaghetti sort* (can be $O(n)$ for any size dataset; requires multiple parallel and hardware processors)

Backtracking

- A backtracking algorithm attempts to construct a solution to a computational problem incrementally.
  - At each stage of recursion, solutions that fail are removed. The process continues until all solution alternatives are tested for the constraints of the problem.
- Backtracking:
  - Can only solve computational problems that can have *constraint satisfaction problems* with clear and well-defined constraints for any objective solution
  - **Constraint satisfaction problems (CSPs):**
    - Are mathematical models defined as a set of objects whose state must satisfy a number of constraints or limitations
    - Represent entities as a homogeneous collection of finite constraints over variables
    - Are solved by constraint satisfaction methods

EX: Sudoku, crosswords, kakuro, hidato, and many other logic puzzles

- Backtracking is used in three classes of computational problems:
  - **Decision problem:** To search for a feasible solution
  - **Optimization problem:** To search for the best solution
  - **Enumeration problem:** To find all feasible solutions
    - The figure below shows a high-level algorithm for a decision problem, in this case to find a path through a maze.
      - At each recursive call level, all directions are tested. If all directions fail, the failing locations are removed from the solution.
      - This simple algorithm always finds the upper path through the maze because of the order in which directions are processed.




```
FindPath ( current_location )
    if  ( current_location is Exit ) return true
    if  ( current_location is Blocked ) return false        → Backtrack
    mark current_location as part of solution path          → Build solution
    if ( FindPath ( North of current_location ) == true ) return true
    if ( FindPath ( East of current_location ) == true ) return true
    if ( FindPath ( South of current_location ) == true ) return true
    if ( FindPath ( West of current_location ) == true ) return true
                                                            } Recurse all directions
```

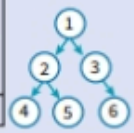
Graph

- A graph algorithm solves graph-related problems, taking one or more graphs as input (e.g., shortest path, minimum spanning tree, and graph traversals).
- **Graph:** Nonlinear data structure consisting of nodes (a.k.a. vertices) and edges, where the nodes are connected by the edges; some fundamental types:
  - **Simple (a.k.a. strict):** An unweighted, undirected graph that doesn't contain any loops and doesn't have multiple edges
  - **Undirected, directed, and mixed:**
    - **Directed:** The edges specify the directions that can be traversed (edge orientation).
    - **Undirected:** The edges can be traversed in both directions.
    - **Mixed:** These graphs contain both directed and undirected edges.
  - **Cyclic and acyclic:**
    - **Cyclic:** Contains a closed loop (a path of edges that ends back at the starting node)
    - **Acyclic:** Doesn't contain any closed loops
      - All tree data structures are some type of acyclic graph.
  - **Labeled:** The nodes and/or edges are assigned labels (e.g., names, numbers, and colors).
    - **Weighted:** A labeled graph whose edges and/or nodes are assigned a number (the weight)
      - Practical weighted graphs are node labeled and edge weighted.

      EX: A road network whose nodes are labeled with city names and whose edges are weighted with distance
- Graphs help in solving and studying real-world problems with road networks, social networks (e.g., Facebook and LinkedIn), power lines, flight plans, communication networks, and molecule chains.
- The most common graph algorithms:
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Shortest path:
    - From the source to all vertices (a.k.a. Dijkstra's algorithm)
    - From every vertex to every other vertex (a.k.a. Floyd-Warshall's algorithm)
  - Minimum spanning tree:
    - Prim's algorithm
    - Kruskal's algorithm
- The figure below shows a high-level depth-first search algorithm to traverse all the nodes in this binary tree.

```
DFS ( Graph, current_node )
   mark current_node as visited
   for all neighbors next_node of current_node in Graph
      if ( next_node is not visited )
         DFS ( Graph, next_node )
```

```
Node traversal order: 1 2 4 5 3 6
```

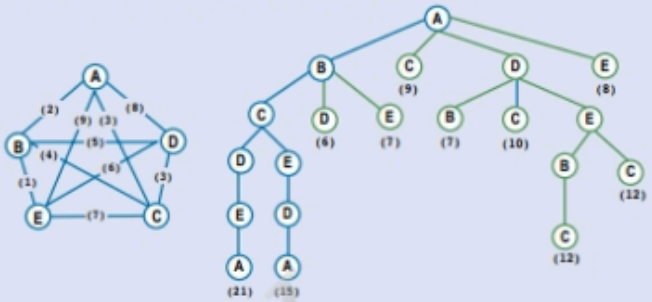


Branch & Bound

- A branch-and-bound algorithm:
  - Constructs a tree of subproblems as it progresses in order to solve an original **root problem**
  - Uses a *bounding method* that determines **upper** and **lower bounds** for each given tree node (subproblem)
    - **Bounding:** Enforcing a bound on a solution quality
    - **If the bounds match:** The solution is deemed feasible for that particular subproblem.
    - **If the bounds do not match:** The current problem is partitioned into two further subproblems, and the infeasible solution is removed (*pruned*). Processing continues until a feasible solution is found or until all nodes (subproblems) have been pruned.
      - **Pruning:** Removing branches in the solution tree whose solution quality is estimated to be poor
- Bounding and pruning are essential concepts of branch-and-bound algorithms, as they enable reduced processing time.

Travelling Salesperson Problem (TSP)

- Branch-and-bound algorithms are used for optimization problems such as the TSP. In a TSP, a traveler has to visit each of *n* cities at least once and wants to minimize the total distance travelled.
  - **Root problem:** Find the shortest route through a set of cities while visiting each city once. To do this:
    - Split the current node (city) into two child problems:
      - Shortest route visiting city A first
      - Shortest route not visiting city A first
    - Continue subdividing similarly as the tree grows.
- In the figure at the top right, the:
  - Numbers (under the solution nodes) indicate the path length for the corresponding potential solution
  - Subtrees (in green) have been pruned as solutions that are not optimal,
  - Algorithm selects the lowest-valued solution among the found feasible solutions as the optimal solution (A→B→C→E→D→A)

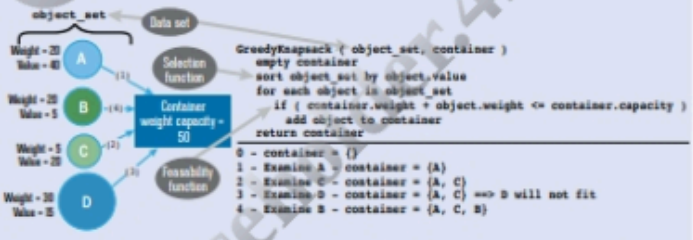


Greedy Method

- A greedy algorithm adopts a **greedy method (a.k.a. greedy strategy)** that makes the next best choice at any given step or stage without regard to the best overall outcome. It:
  - Is easy to implement
  - Is easy to measure for time and space
  - Does not always give the best solution
  - Can be more time consuming than a non-greedy algorithm would be for the same problem

Classic Knapsack Problem

- Given a set of objects with weight and value and a container of a fixed weight capacity, fill the container with the most value possible.
- The iterative algorithm (see below) picks the highest value that will still fit in the container. This continues until all objects are processed or the container is full.



- The greedy strategy requires at least some of these components:
  - A candidate **data set** that needs a solution (required)
  - A **selection function** to choose the best contributor to an overall solution (required)
  - A **feasibility function** to determine if a candidate can be a contributor to a solution (required)
  - An **objective function** to assign a value to a partial solution (optional depending on problem space and algorithm implementation)
  - A **solution function** to determine when the optimal solution has been reached (optional depending on problem space and algorithm implementation)

Heuristic Methods

- A heuristic algorithm:
  - Is inexact and based on a *heuristic* that has a reasonable chance of finding feasible solutions for a given problem
    - **Heuristic:** An approximate but guided attempt at solving a problem

    EX: Educated guess, intuition, current trends, common sense, success measures, and profiling
  - Is practical and not guaranteed to be optimal but sufficient to achieve feasible solutions
  - Uses heuristics (guesses) instead of complex logic to make cognitive decisions, which means it:
    - Is simpler to implement
    - Reaches feasible solutions in faster execution times
    - Sacrifices optimality, completeness, accuracy, and/or precision
  - Is employed in cases where an optimal solution is infeasible or impractical, such as:
    - Artificial intelligence
    - Big data analysis
    - Machine learning
    - Natural language processing
    - Shortest path problems
- Chess programs, strategic games, and battle-theater simulations are cases where:
  - Investigation of all possible moves is impossible in real time
  - Heuristics are used to quickly discard some moves while focusing on key moves that are more likely to lead to victory
- In order to play a game of tic-tac-toe, a computer algorithm can use this

Four ways to win the game | Three ways to win the game | Two ways to win the game

- At each turn, the computer selects an empty tile with the highest number of ways to win (at that time).
- This very simple algorithm can eventually be beaten by a human.

Randomized

- Randomized algorithms are:
  - Algorithms whose behavior and output are determined by both the input and the random values generated
  - Usually simpler and more efficient than deterministic algorithms for the same problem
  - Of two classes:
    - **Randomized Las Vegas algorithms:**
      - **Output:** Always correct
      - **Running time:** Random and must be controlled by boundary limits

    EX: Randomized quicksort algorithm
    - **Randomized Monte Carlo algorithms:**
      - **Output:** May be incorrect with calculatable probability
      - **Running time:** Deterministic

    EX: Randomized algorithm for approximate median
- The figure below shows a randomized quicksort algorithm that sorts the unordered array.
  - **Distribution insensitive:** Time taken does not depend on the initial order of the array, it depends on the random choices of pivot elements.
  - The **partition function** randomly divides the array between the left and right according to a specific randomization scheme (in this case Las Vegas).
  - For a given input:
    - The expected (average) running time is $O(n \log n)$.
    - The worst-case running time is $O(n^2)$.

```
QuickSort ( Array, left, right )
   if ( left < right )
     x =  Array[left]
     i = Partition ( Array, left, right, x )
     QuickSort ( Array, left, i-1 )
     QuickSort ( Array, i+1, right )
```

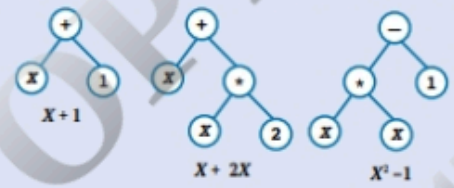
Genetic [Evolutionary Computation]

- A genetic algorithm mimics biological evolution as a problem-solving strategy via these top-level components:
  - **Input:** A set of potential solutions to a given problem
  - **Data:** The evolving set (population) of potential solutions
  - **Fitness:** A metric defining how fit a solution is in solving a problem
  - **Fitness function:** One or more functions that can quantitatively evaluate a solution to produce a fitness metric value
  - **Reproduce function:** One or more functions that imperfectly copy a given solution by introducing random changes in the new copy
    - The randomly changed copy is added to the set (population).
  - **Evolve function (main loop):** The main worker function that controls solution sets and dictates how and when solutions can reproduce

Recursive Implementation of the Evolve Function

```
Evolve ( solution_set )
   next_gen_set = solution_set
   for each solution in solution_set
     if ( IsFit ( solution ) )
        child_solution = Reproduce ( solution )
        add child_solution to next_gen_set
     else
        remove solution from next_gen_set
   Evolve ( next_gen_set )
```

- A typical evolutionary process has these steps, round over round:
  - Each solution in the set is evaluated via the fitness function.
  - Unfit solutions are deleted from the set (general population).
  - Fit solutions are allowed to reproduce.
  - Each generation of solutions is evolved (iteratively and/or recursively) to create new generations of solution sets.
- Expectations from a genetic algorithm are that:
  - The average fitness of the general population will increase with each round (generation) of solutions.
  - A feasible solution to the original problem will eventually be found in some generation of solutions.
- Genetic algorithms are unbounded in execution time and space because of their evolutionary nature; they require thresholds when available processing time and space are limited.
- Consider the problem of discovering the equation for a line or curve.
  - The figure below shows randomly evolving solutions that are equations expressed as binary trees.
  - At each step, a new potential equation (a binary tree) is randomly constructed and its fitness measured.
  - Using the binary trees below, the fitness function maps out a curve using a candidate equation and determines whether it matches the input curve.



Geometric

- Geometric algorithms are designed to solve problems in geometry such as comparing slopes of two lines and finding an equation for a line. They:
  - Require in-depth knowledge of mathematical subjects such as combinatorics, topology, algebra, and differential geometry
  - Solve problems in these general areas:
    - Lines
    - Triangles
    - Rectangles, squares, and circles
    - Polygons
    - 3-D objects
- Applications of computational geometry span robotics, integrated circuit design, computer vision, computer-aided engineering, and geographic information systems (GIS).
- The **contains function** in the figure below implements the mathematical formulas given at the top for discovering the intersection of two lines.
- The notation $x[i]$ indicates a point (coordinates) along line x.

Does the line segment below intersect the ray?


```
contains ( x0, y0 )
 crossings = 0
 for i=0 to N-1
  slope = ( y[i+1] - y[i] ) / ( x[i+1] - x[i] )
  condition1 = ( x[i] <= x0 ) and ( x0 < x[i+1] )
  condition2 = ( x[i+1] <= x0 ) and ( x0 < x[i] )
  above = ( y0 < ( slope * ( x0 - x[i] ) + y[i] ) )
  if ( ( condition1 = TRUE or condition2 = TRUE ) and above = TRUE )
    crossings = crossings + 1
 if ( IsOdd ( crossings ) )
  return TRUE
 else
  return FALSE
```

U.S. $7.95
Author: Babak Ahmadi

Find us on Facebook

6 54614 04216 3

hundreds of titles at
quickstudy.com

ISBN-13: 978-142324216-1
ISBN-10: 142324216-5

9 781423 242161 50795