ECE 217:
Data Structure and Algorithm

Lecture 1: Introduction

Instructor: Shayan (Sean) Taheri, Ph.D.

Assistant Professor

The Department of Electrical and Cyber Engineering (ECE)

The Institute for Health and Cyber Knowledge (I-HACK)

The Gannon University (GU)

Personal Information

- Name: Shayan (Sean) Taheri.
- Date of Birth: July/28/1991.
- Past Position: Postdoctoral Fellow at University of Florida.
- Ph.D. Degree: Electrical Engineering from the University of Central Florida.
- M.S. Degree: Computer Engineering from the Utah State University.
- University Profile: https://www.gannon.edu/FacultyProfiles.aspx?profile=taheri001

Course objectives and understanding

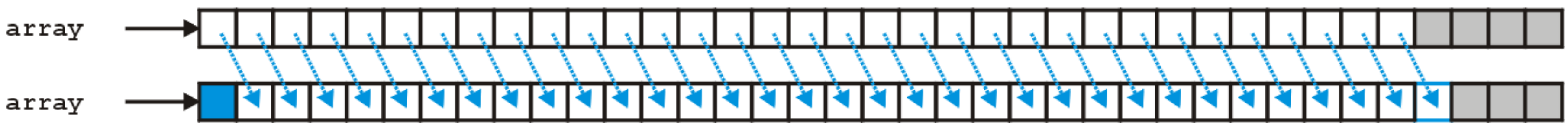
- Learn basic data structures and algorithms
  - *Data structures: Usage for how data is organized*
  - *Data structures for efficiently storing, accessing, and modifying data*
  - *We will see that all data structures have trade-offs*
  - *There is no ultimate data structure*
  - *Algorithms: It can be considered an unambiguous sequence of steps to compute something*
  - *Algorithm analysis: determining how long an algorithm will take to solve a problem*
  - *Algorithms for solving problems efficiently*
  - *The choice of data structures and algorithms depends on our requirements*
- Become a better software developer
  - *"Data Structures + Algorithms = Programs"*
    -- Niklaus Wirth, author of Pascal language

Course objectives and understanding

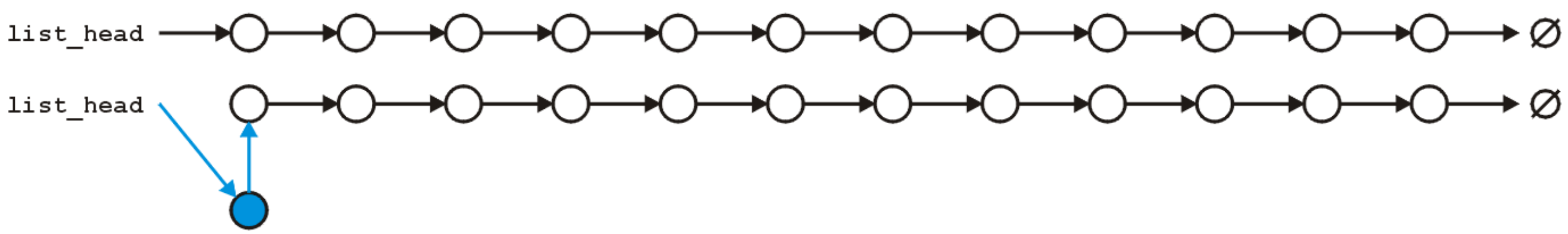
- Become a better software developer
  - *"Data Structures + Algorithms = Programs"*
    -- *Niklaus Wirth, author of Pascal language*
- Ex 1: Consider accessing the kth entry in an array or linked list
  - *In an array, we can access it using an index array[k]*
  - *We must step through the first k – 1 nodes in a linked list*
- Ex 2: Consider searching for an entry in a sorted array or linked list
  - *In a sorted array, we use a fast binary search*
  - *We must step through all entries less than the entry we're looking for*

Course objectives and understanding

- Ex 3: Consider inserting a new entry to the start of an array or a linked list
  - *An array requires that you copy all the elements in the array over*



  - *A linked list allows you to make the insertion very quickly*



- Sample Topics: Storing ordered and sorted objects, Storing an arbitrary collection of data, Sorting objects, Graphs, and Algorithm Design Techniques

Course Specifications

- **Components for evaluation**: You grade is determined based on:
  - *Theoretical Assignments*
  - *Laboratory Assignments*
  - *Exams*
  - *Projects*
- Improve your knowledge and expertise on programming and Unix-based systems
  - *You will be using the C++ programming language in this course*
  - *This course does not teach C++ programming*
  - *Refer to the tutorials that are available online*
  - *Understanding the Unix environment is required*
  - *You will use the G++ compiler*
  - *Codes can be developed in the Windows environment but they should be testable in the Unix environment*
  - *<u>Using a computer to help solve problems</u>: Designing programs (architecture, algorithms), Writing programs, Verifying programs, and Documenting programs*

Definitions

- **Algorithm**: The essence of a computational procedure, step-by-step instructions
- **Program:** An implementation of an algorithm in some programming language
- **Data structure**: Organization of data needed to solve the problem

Importance of Data Structure and Algorithms

Data Structure and Algorithm Design Goals

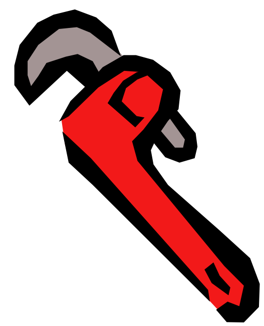
Correctness

Efficiency

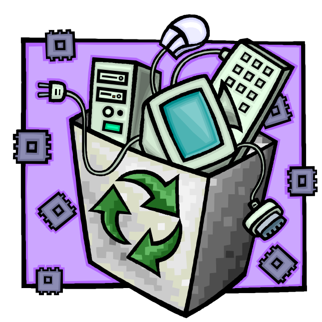
Implementation Goals

Robustness

Adaptability



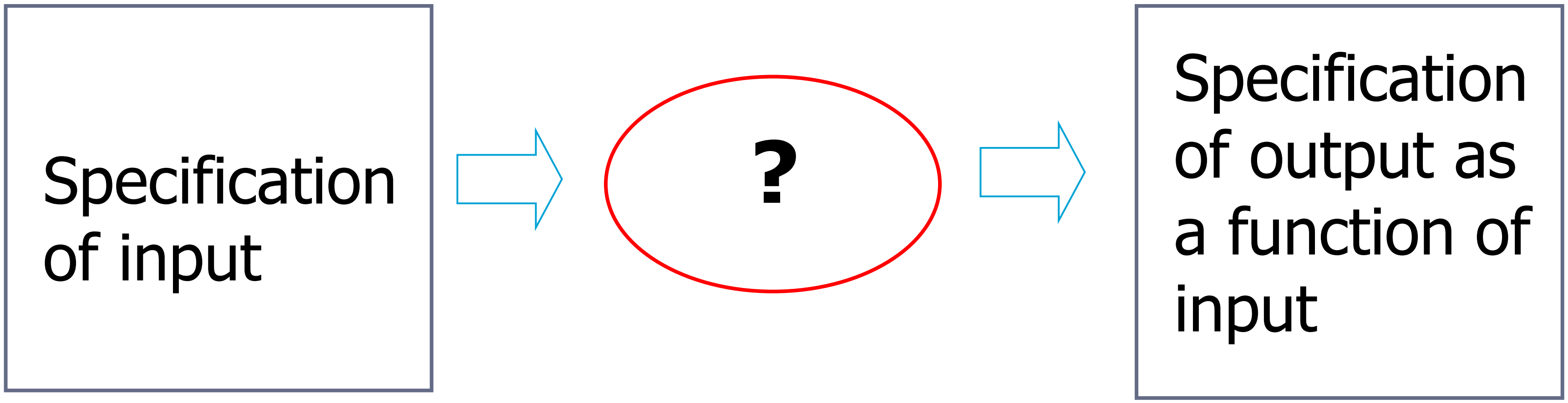
Reusability



History

- **Name:** Persian mathematician Muhammad ibn Musa al-Khwarizmi, in Latin became Algorismus
- **First algorithm:** Euclidean Algorithm, greatest common divisor, 400-300 B.C.
- **In 19th century:** Charles Babbage, Ada Lovelace
- **In 20th century:** Alan Turing, Alonzo Church, John von Neumann

Algorithmic Problem



- Infinite number of input *instances* satisfying the specification. For example:
  - A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000000.
    - 3.

Algorithmic Problem

Input instance, adhering to the specification

Algorithm

Output related to the input as required

- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

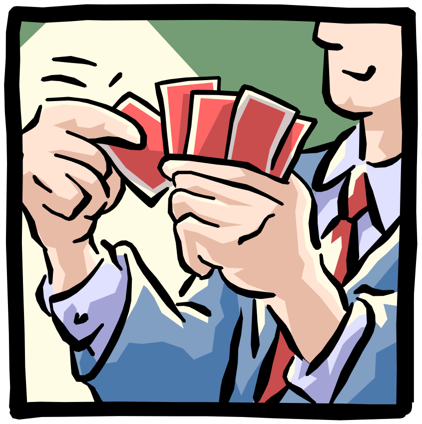
Example Process: *Sorting*

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness

For any given input the algorithm halts with the output:

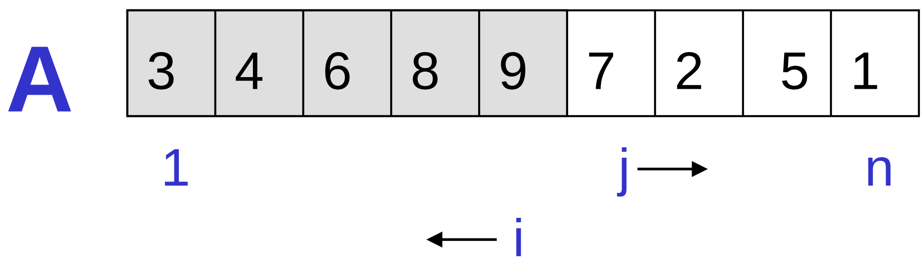
- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

Example: *Sorting and Insertion*



Strategy

- Start “empty handed”
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

```
for j=2 to length(A)
    do key=A[j]
    "insert A[j] into the
    sorted sequence A[1..j-1]"
      i=j-1
      while i>0 and A[i]>key
         do A[i+1]=A[i]
            i--
      A[i+1]:=key
```

Analysis of Algorithms and Memory Usage

- Efficiency:
  - *Running time*
  - *Space used*
- Efficiency as a function of input size:
  - *Number of data elements (numbers, points)*
  - *A number of bits in an input number*
- The RAM model:
  - *Instructions (each taking constant time):*
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
  - *Data types – integers and floats*

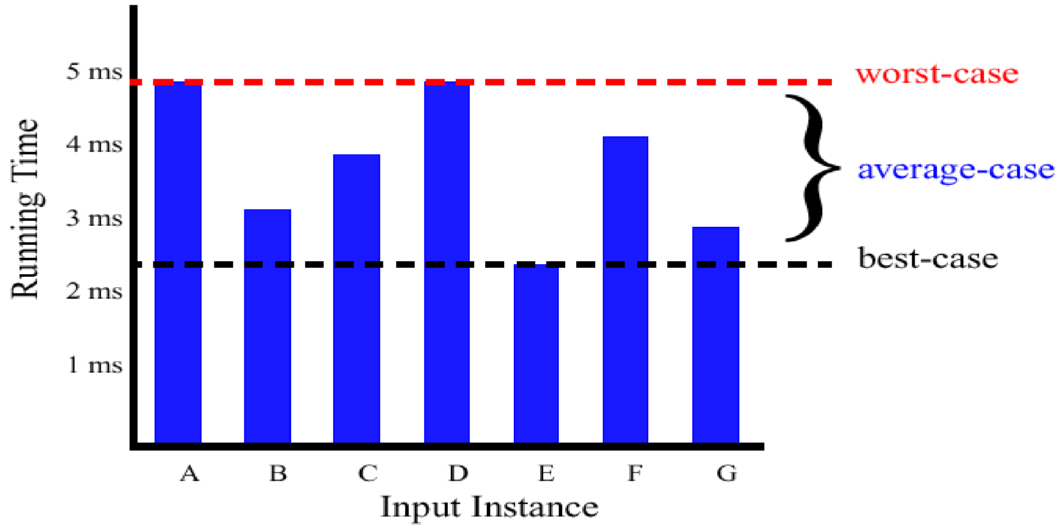
Example: ***Sorting and Insertion Analysis***

- Time to compute the **running time** as a function of the **input size**

	cost	times
for j=2 **to** *length*(A)	c_1	n
do key=A[j]	c_2	n-1
"insert A[j] into the sorted sequence A[1..j-1]"	0	n-1
i=j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]=A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_7	n-1

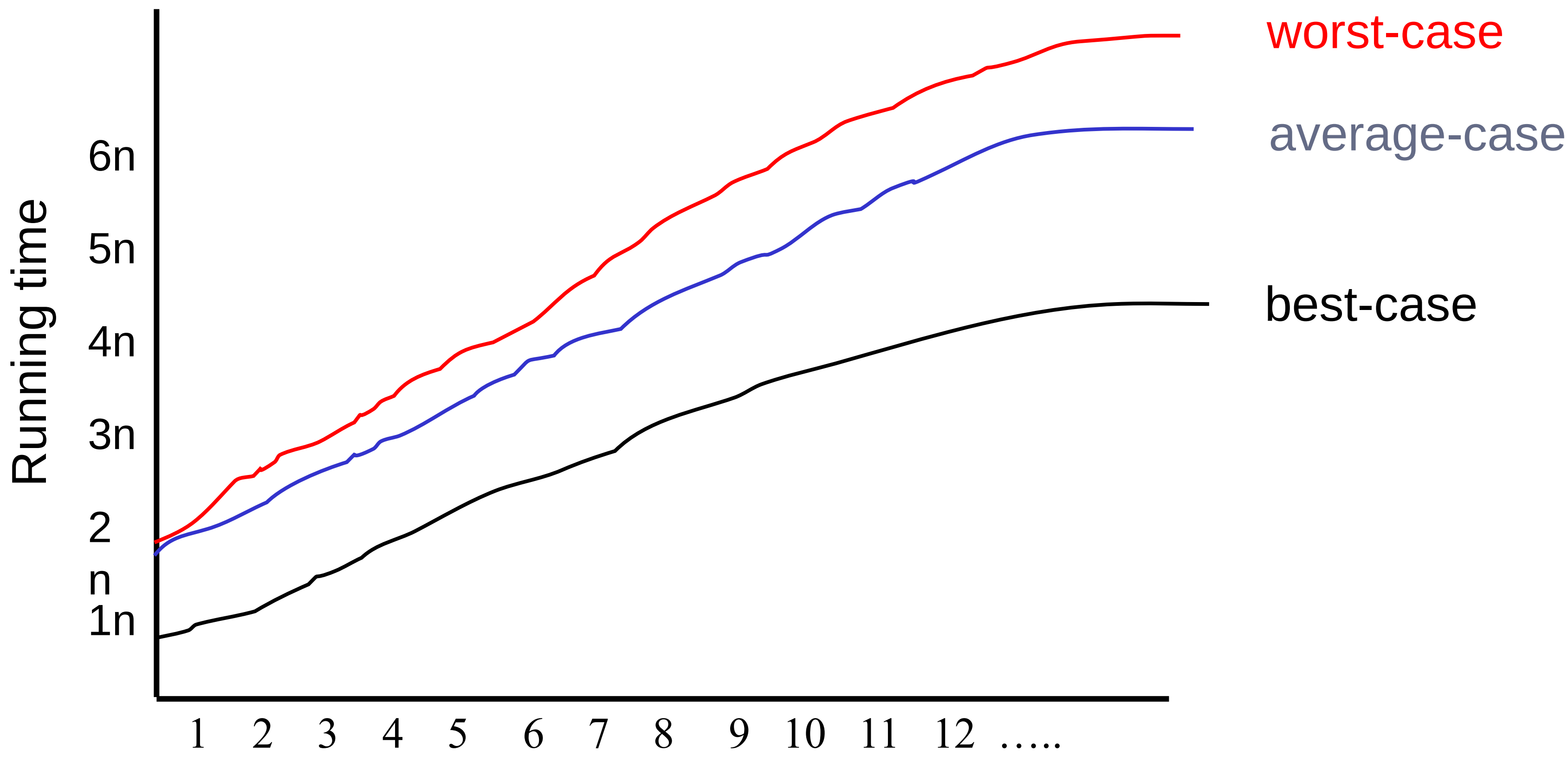
Best/Worst/Average Case

- **Best case**: elements already sorted → $t_j=1$, running time = $f(n)$, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order → $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time
- For a specific size of input n, investigate running times for different input instances:



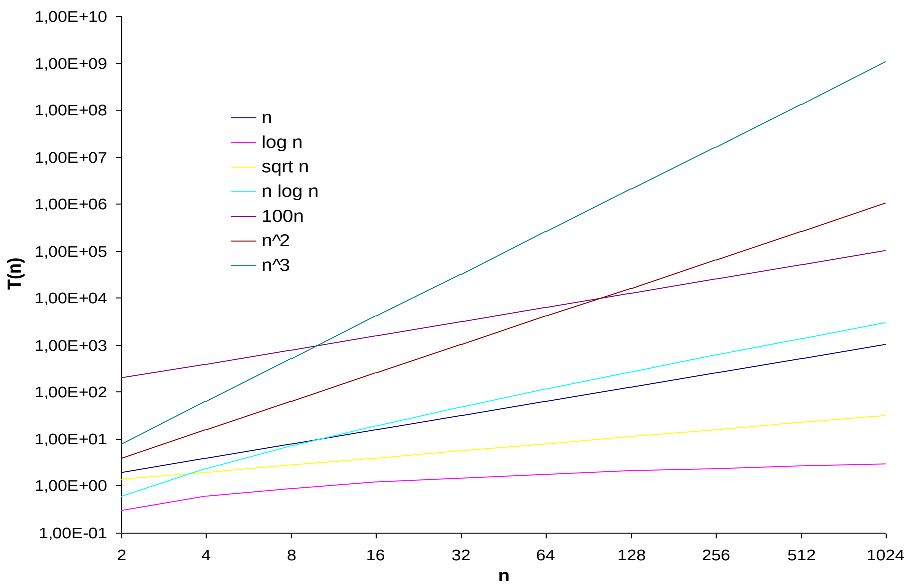
Best/Worst/Average Case

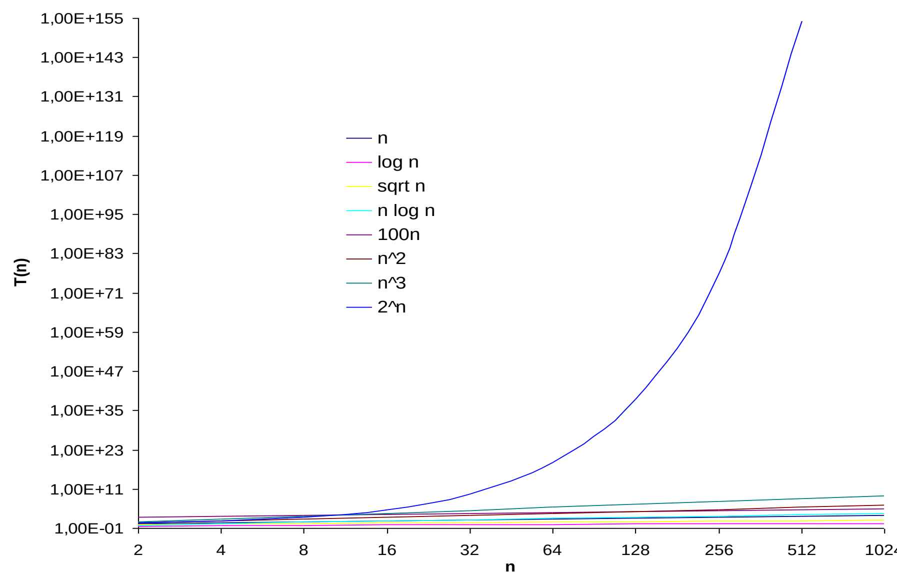
- For inputs of all sizes:



Best/Worst/Average Case

- **Worst case** is usually used:
  - *It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the* ***worst-case*** *time complexity is of crucial importance*
  - *For some algorithms* ***worst case*** *occurs fairly often*
  - *The* ***average case*** *is often as bad as the* ***worst case***
  - *Finding the* ***average case*** *can be very difficult*





More Information on Sorting Process

- Is **insertion sort** the best approach to sorting?
- Alternative strategy based on divide and conquer
- MergeSort
  - *Sorting the numbers <4, 1, 3, 9> is split into*
  - *Sorting <4, 1> and <3, 9> and*
  - *Merging the results*
  - *Running time f(n log n)*

Example Process: ***Searching***

INPUT

- Sequence of numbers (database)
- A single number (query)

$a_1, a_2, a_3, \ldots, a_n; \; q$

2 5 4 10 7; 5

2 5 4 10 7; 9

OUTPUT

- An index of the found number or *NIL*

j

2

NIL

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

Example Process: ***Searching***

- Worst-case running time: *f(n)*, average-case: *f(n/2)*
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

INPUT

- Sorted non-descending sequence of numbers (database)
- A single number (query)

OUTPUT

- An index of the found number or *NIL*

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

2 4 5 7 10; 5 → 2

2 4 5 7 10; 9 → *NIL*

Binary Search

- Idea: Divide and conquer, one of the key design techniques

➢ How many times the loop is executed:

- *With each execution its length is divided to half*
- *How many times do you have to cut n in half to get 1? lg n (log base n)*

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Abstract Data Types

- **Abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - *Describes what a collection does, not how it does it*
  - *Described in Java with interfaces (e.g., `List`, `Map`, `Set`)*
  - *Separate from **implementation***

- ADTs can be implemented in multiple ways by classes:
  - `ArrayList` *and* `LinkedList` *implement* `List`
  - `HashSet` *and* `TreeSet` *implement* `Set`
  - `LinkedList` , `ArrayDeque`, *etc.* *implement* `Queue`
  - *Java messed up on Stack—there's no Stack interface, just a class.*

List ADT

- An ordered collection the form A_0, A_1, ..., A_{N-1}, where N is the size of the list
- Operations described in Java's `List` interface (subset):

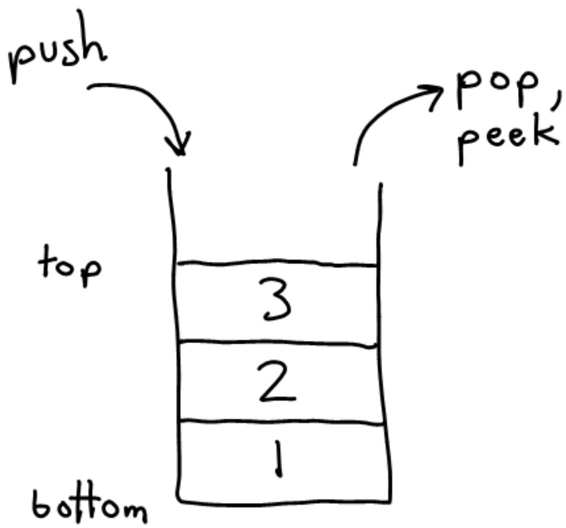
`add(`**elt**`, `**index**`)`	inserts the element at the specified position in the list
`remove(`**index**`)`	removes the element at the specified position
`get(`**index**`)`	returns the element at the specified position
`set(`**index**`, `**elt**`)`	replaces the element at the specified position with the specified element
`contains(`**elt**`)`	returns true if the list contains the element
`size()`	returns the number of elements in the list

- `ArrayList` and `LinkedList` are implementations

Stack ADT

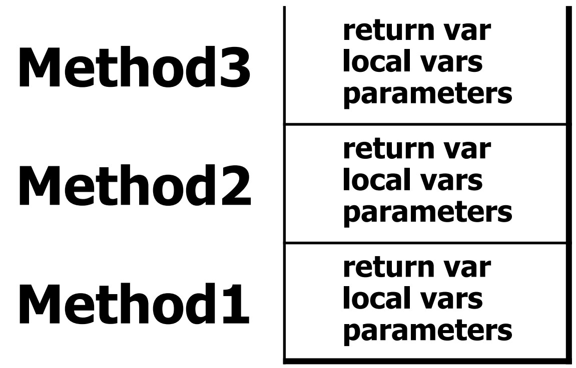
- **Stack**: A list with the restriction that insertions/deletions can only be performed at the top/end of the list
  - *Last-In, First-Out ("LIFO")*
  - *The elements are stored in order of insertion, but we do not think of them as having indexes.*
  - *The client can only add/remove/examine the last element added (the "top").*

- Basic stack operations:
  - ***push**: Add an element to the top.*
  - ***pop**: Remove the top element.*
  - ***peek**: Examine the top element.*



Applications of Stacks

- Programming languages:
  - *Method calls are placed onto a stack (call=push, return=pop)*



- Matching up related pairs of things:
  - *Find out whether a string is a palindrome*
  - *Examine a file to see if its braces { } and other operators match*

- Sophisticated algorithms:
  - *Searching through a maze with "backtracking"*
  - *Many programs use an "undo stack" of previous operations*

Class `Stack`

`Stack<`**E**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);     // bottom [42, -3, 17] top

System.out.println(s.pop()); // 17
```

Stack limitations/idioms

- Remember: You can't loop over a stack like you do a list.

```
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Instead, you pull contents out of the stack to view them.
  - *Idiom: Remove each element until the stack is empty.*

```
while (!s.isEmpty()) {
    do something with s.pop();
}
```

Questions?