# Algorithms Problem Solving

- **Readings: [SG] Ch. 2**
- **Chapter Outline:**
  1. **Chapter Goals**
  2. **What are Algorithms**
  3. **Pseudo-Code to Express Algorithms**
  4. **Some Simple Algorithms [SG] Ch. 2.3**
     1. ***Computing Array-Sum (using Linear Scan)***
     2. ***Structure of Linear Scan Algorithm***
  5. **Examples of Algorithmic Problem Solving**

Last Revised: 30 August 2016.

# Recurring Principles in CS & IT

**RP1: Multiple Levels of Abstraction**
(very high to very low)

**RP2: One Data, Multiple Views**
(thru diff interfaces)

**RP3: Define a (small) set of basic primitives**
(building blocks)

**RP4: Divide & Conquer aka**
(Decomposition)

**RP5: "The Power of Iteration"**
(aka Recursion)

# First…

**A Classic Algorithm (Linear-Scan)**

# A First Simple Algorithm

## Problem: Sum a list of *n* numbers

**First: Store the *n* numbers in an array**
(more convenient, easy to access)

**Example:**
**Input:** [ 2, 5, 10, 3, 12, 24 ]
**Output:** Sum = 56

**PQ:** Try an example…

**More Precise Problem Statement:**
**Input:** A list A[1..$n$] of numbers
**Output:** The sum of these numbers, namely
Sum = (A[1] + A[2] + … + A[$n$])

**PQ:** Restate the problem

# A First Simple Algorithm

**Problem:**

**Input:** A list A[1..$n$] of numbers

**Output:** The sum of these numbers, namely

Sum = (A[1] + A[2] + … + A[$n$])

**PQ:** Can we reuse some algorithm?
(Reuse the result? Reuse the method?)

# Simple iterative algorithm: Array-Sum(*A*,*n*)

**Input:** A list *A*[1..*n*] *of numbers*

**Output:** To compute the <u>sum</u> of the numbers

```
Array-Sum(A, n);
(* Find the sum of A1, A2,…,An. *)
begin
   Sum_SF ← 0;
   k ← 1;
   while (k <= n) do
      Sum_SF ← Sum_SF + A[k];
      k ← k + 1;
   endwhile
   Sum ← Sum_SF;
   Print "Sum is", Sum;   return Sum;
end;
```

Note difference between k and A[k]

Sum_SF represents the sum computed so far


Hon Wai Leong, NUS

# *Exercising* Algorithm Array-Sum(A,n):

**Input:**

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | n=6 |
|---|---|---|---|---|---|---|
| 2 | 5 | 10 | 3 | 12 | 24 | |

Note difference between
k and A[k]

Eg: when k=3,
A[k]=10

**Processing:**

| k | Sum-SF | Sum |
|---|---|---|
| ? | 0 | ? |
| 1 | 2 | ? |
| 2 | 7 | ? |
| 3 | 17 | ? |
| 4 | 20 | ? |
| 5 | 32 | ? |
| 6 | 56 | ? |
| 6 | 56 | 56 |

**Output:** `Sum is 56`

**RP5: "The Power of Iteration"**

# Abstraction: Defining a *new primitive*

**RP1: Defining a new Abstraction**

**Abstraction**:

- Define a new high-level primitive for a common computational task;
- Give primitive a good name,
  specify what inputs it requires, and
  what outputs it will produces;

A good name that suggests what it does

# Defining the Abstraction

*Parameters*: A and n

*Name* of Algorithm

Some *comments* for human understanding

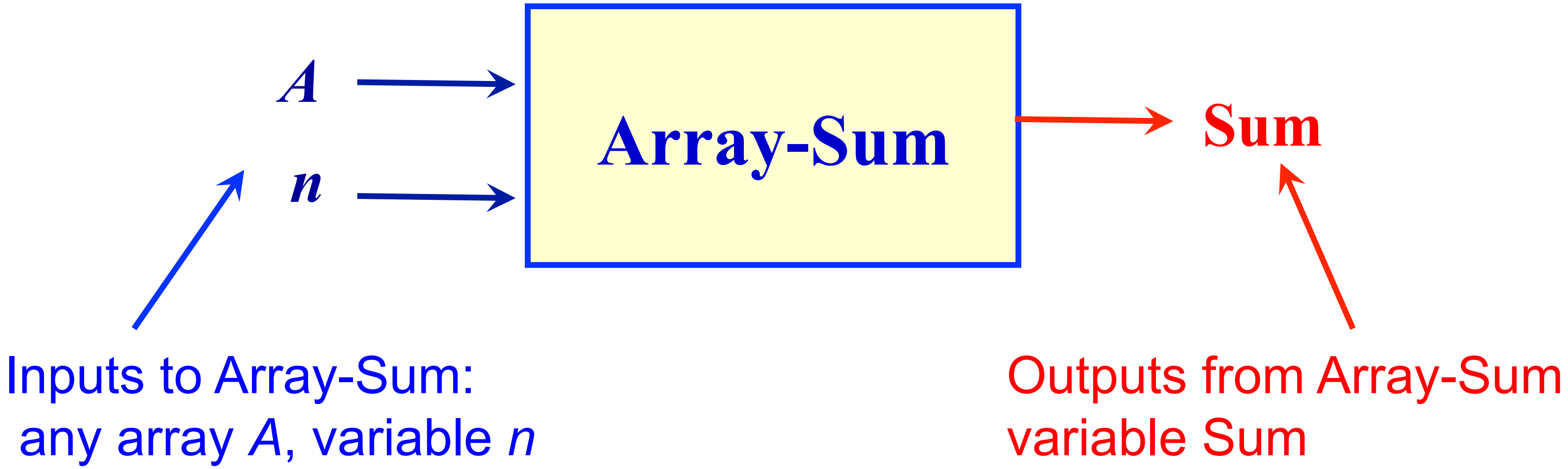
```
Algorithm Array-Sum(A, n);
(* Find the sum of A1, A2,…,An. *)
begin
   Sum_SF ← 0;
   k ← 1;
   while (k <= n) do
      Sum_SF ← Sum_SF + A[k];
      k ← k + 1;
   endwhile
   Sum ← Sum_SF;
   return Sum
end;
```

*Output value returned*: Sum

**RP5: "The Power of Iteration"**
(Linear Scan Algorithm)

# Abstracting a High-level Primitive

- Then Array-Sum becomes a *high- level primitive defined as* Array-Sum ($A$, $n$)




**Definition: Array-Sum ($A$, $n$)**
The high-level primitive Array-Sum takes as input a variable $n$ and an array $A[1..n]$, then it computes and returns the sum of $A[1..n]$, namely, Sum = $(A[1] + A[2] + \ldots + A[n])$.

# Using and re-using the new primitive

**After new primitive is defined**

- We can *call* (*invoke*) the new primitive many times to perform that common task;
- Call new primitive with different inputs
- In this way, we extend the capability of our computational (software) library

# Using a High-level Primitive

**So, we define Array-Sum ($A$, $n$)**
The high-level primitive Array-Sum takes as input a variable $n$ and an array $A[1..n]$, then it computes and returns the sum of $A[1..n]$, namely, Sum = ($A[1] + A[2] + . . . + A[n]$).

**To use the high-level primitive** (or just primitive, in short) we just issue a call to that high-level primitive

**Example 1:** Array-Sum ($A$, 6)
call the primitive Array-Sum to compute the sum of $A$[1 .. 6], and returns the sum as its value

**Example 2:** Top ← Array-Sum ($B$, 8)
"compute the sum of $B$[1 .. 8], and store that in variable Top

**Example 3:** DD ← Array-Sum ($C$, $m$)
"compute the sum of $C$[1 .. $m$], and store that in variable DD

# Using a High-level Primitive

**So, we define Array-Sum ($A$, $n$)**
The high-level primitive Array-Sum takes as input a variable $n$ and an array $A[1..n]$, then it computes and returns the sum of $A[1..n]$, namely, Sum = $(A[1] + A[2] + . . . + A[n])$.

**To use the high-level primitive** (or just primitive, in short) we just issue a call to that high-level primitive

**GOOD POINT #1:**
Can call it many times, no need to rewrite the code

**GOOD POINT #2:**
Can call it to calculate sum of different arrays (sub-arrays) of diff. lengths

# Abstraction: Defining *new primitive*

- The algorithm for Array-Sum ($A$, $n$) becomes a *new high-level primitive*


- Can be used to compute sum of different arrays
- Can be re-used by (shared with) other people

# *Thank you!*

School *of* Computing