CSC 225

Algorithms and Data Structures I

Rich Little
rlittle@uvic.ca
ECS 516

Methodology Requirements

- A language for describing our algorithms
  - *pseudocode*
- A computational model that our algorithms execute within
  - *Random Access Machine (RAM)*
- A metric for measuring algorithm runtime
  - *Counting primitive operations*
- An approach for characterizing those runtimes
  - *Worst-case analysis (big-Oh)*

Pseudo Code

- An algorithm is the procedural or step-by-step solution of a problem
- The procedural solution can be at different levels of abstraction

- Machine code
- Assembly language
- C

- Python
- C++
- Java

- Pseudo code

Problem 1: Find the maximum value in an array of *n* numbers

Pseudo-code vs. Java code

```
Algorithm arrayMax(A,n)
   currentMax ← A[0]
   for k ← 1 to n-1 do
    if currentMax < A[k]
   then
       currentMax ← A[k]
   return currentMax
```

```
                    …
public int arrayMax(int[] A,
   int n) {
   int currentMax = A[0];
   for (int k=1; k<n; k++) {
    if (currentMax < A[k]) {
     currentMax = A[k];
    }
   }
 return currentMax;
}
                    …
```

Problem 1: Find the maximum value in an array of *n* numbers

```
Algorithm arrayMax(A,n):
  Input: An array A storing n ≥ 1
         integers
  Output: The maximum element in A

  currentMax ← A[0]
  for k ← 1 to n-1 do
     if currentMax < A[k] then
          currentMax ← A[k]
  return currentMax
```

Pseudo Code: Expressions

- Assignment operator (←)
  - `currentMax ← A[k]`
- Equality relation (=)
  - `currentMax = A[k]`

  is `true` if *`currentMax`* and `A[k]` have the same value, otherwise `false`.
- Smaller than (<), greater than (>), smaller or equal to (≤), greater or equal to (≥)

Pseudo Code: Method declarations

- **Algorithm** name(param1, param2,…)

```
Algorithm arrayMax(A,n)
```

Pseudo Code: Decision structures

- **if** condition **then**
  true-actions
  [**else**
  false-actions]
  end

- **if** currentMax < A[k] **then**
  currentMax ← A[k]
  end

Pseudo Code: While loops

- **while** condition **do**
  actions
  end

- **while** currentMax < A[k] **do**
  count ← 2*count
  k ← k+1
  end

Pseudo Code: Repeat loops

- **repeat**
  actions
  until condition

- **repeat**
  k ← k+1
  count ← 2*count
  until currentMax < A[k]

Pseudo Code: For loops

- **for** step-definition **do**
  actions
  end

- **for** k ← 1 **to** n-1 **do**
  if currentMax < A[k] **then**
  currentMax ← A[k]
  end
  end

Pseudo Code: Array indexing and record field selection

- A[k] represents the k^{th} cell in array A, indexed from $A[0]$ to $A[n-1]$.
- $r.key$ represents the field *key* in record or struct r

Struct

Pseudo Code: Method calls and return statements

- Method call
  - object.method(args)
  - Example: `arrayMax(X,13)`

- Return statement
  - **return** value

Analysis of Algorithm Efficiency

Not everything that can be counted counts, and not everything that counts can be counted.
—Albert Einstein (1879-1955)

- Time efficiency
  - indicates how fast an algorithm runs
- Space efficiency
  - indicates how much extra space an algorithm requires

Measuring an Algorithm's Running Time

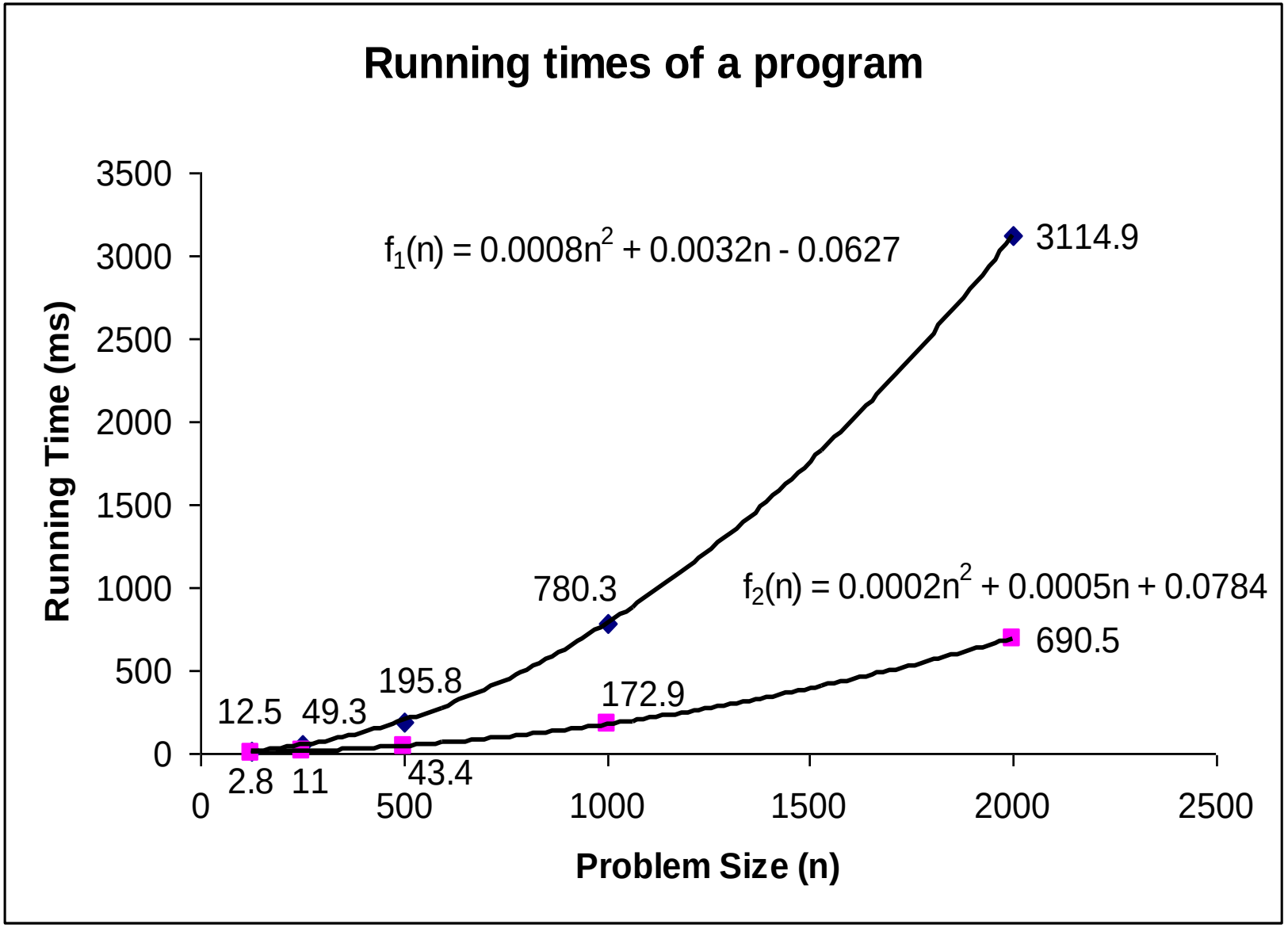
- Two approaches
  - Counting primitive operations and computing upper and lower bounds—topic in CSC 115/116, CSC 225, CSC 226, CSC 320, CSC 423, CSC 425 and many other courses
  - Instrument the code to measure computer clock cycles—topic in SENG 265

What is the running time of this algorithm?

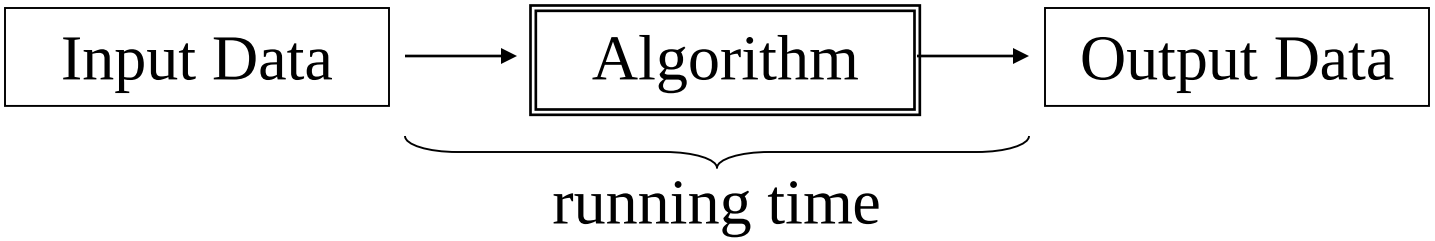
```
Algorithm arrayMax(A,n):
  Input: An array A storing n ≥ 1
     integers.
  Output: The maximum element in A.

  currentMax ← A[0]
  for k ← 1 to n-1 do
     if currentMax < A[k] then
           currentMax ← A[k]
     end
  end
  return currentMax
```

Instrumenting the Code to Measure Clock Cycles

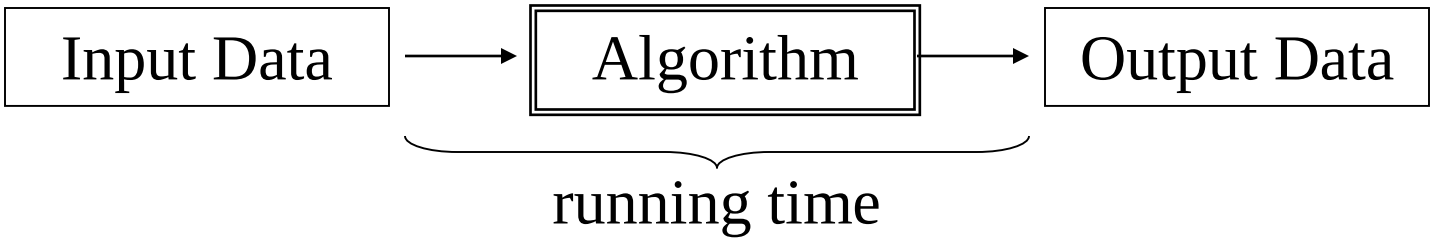


Limitations of Experimental Results



- The algorithm must be implemented
- Experiments can typically be done only on a limited set of test inputs
- When is a set of test inputs representative?
- To compare two algorithms the experiments must run on the same machine

Algorithm Analysis Wish List



- Take all possible inputs into account
- Compare the efficiency of two different algorithms independent from computer and implementation
- Analyze algorithm *before* starting implementation

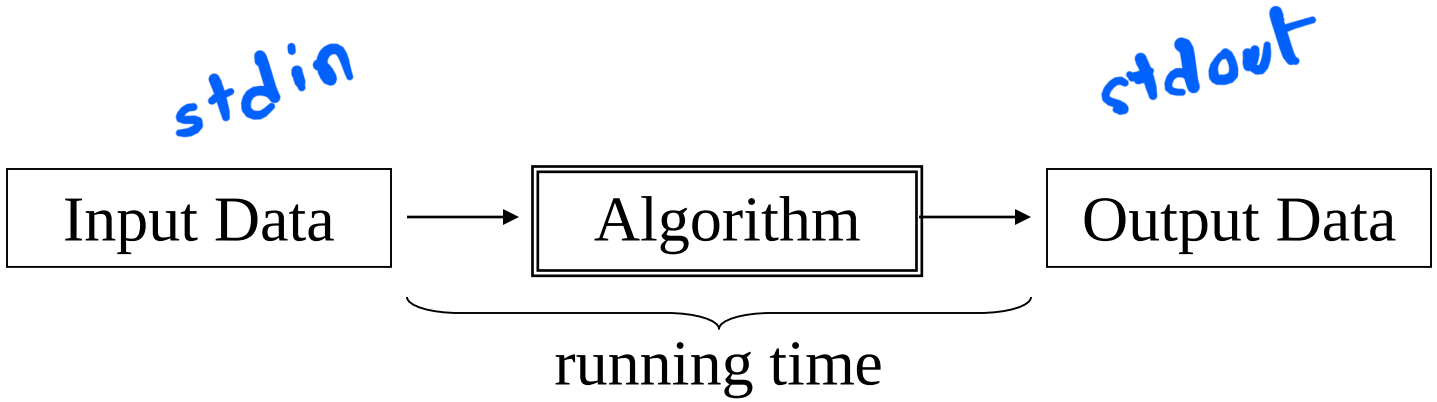
Random Access Machine Model

- RAM – Random Access Machine
  - Computational model
  - CPU and unbounded RAM
- Define a set of high-level primitive operations
  - Independent of programming language
- We simply count how many primitive operations are executed in the RAM model

Primitive Operations

- Assignments (A) $A \leftarrow 2$
- Comparisons (C) $A \leq 2$
- Boolean expressions (B)
- Array indexing (I)
- Record selector or object reference (R)
- Arithmetic operations
  - Add, subtract (S)
  - Multiplication, division (D)
- Trigonometric operations (e.g., sin, cos, tan) (T)
- Calling a method, function, procedure, routine (M)

Algorithm analysis



Categories of algorithm running times

- Best case analysis $T_b(n)$
- Average-case analysis $T_a(n)$
- Worst-case analysis $T(n)$

b = best case
a = average case

Determining the Average-case Time

- Calculate expected running times based on a given input distribution
- Typically the analysis requires heavy math and probability theory
- This will be done in 226
  - Although I may point out average-case times occasionally

Determining the Best-case and Worst-case Running Time

- Worst-case T(n)
  - What is the maximum number of primitive operations (depending on *n*) executed by the algorithm, taken over all inputs of size *n*?
  - Worst-case analysis is most common and may aid in the design of the algorithm
- Best-case $T_b(n)$
  - What is the minimum number of primitive operations (depending on *n*) executed by the algorithm, given the most advantageous or best input configuration of size *n*?

Time Complexity Analysis

- Measuring the size of the algorithm's input
- Computing function for running time (dependent on input size)
- Performing worst-case, best-case, average-case analysis
- Insert function into the order of growth
- Asymptotic notations and efficiency classes
- Analysis of recursive and non-recursive algorithms

Closer Look at the RAM Model

- The RAM model assumes that each primitive operation costs 1 time unit.
  - This includes arithmetic ops, bitwise ops and logical comparisons
- Assume input size is n, and word size is w
  - To store n elements then $w \geq \log n$
  - Why? Indexing up to $n - 1$
- Theoretical outcome of this?
  - Assume $w = c \log n$ for some constant $c \geq 1$

Worst Case Running Time T(n) Counting Primitive Operations

```
currentMax ← A[0]
for k ← 1 to n-1 do
    if currentMax < A[k] then
        currentMax ← A[k]
    end
end
return currentMax
```

Worst case

- 1A + 1I +
- 1A + (N-1)*{
  - 1A + 1S +
  - 1C +
  - 1C + 1I +
  - 1A + 1I +
- }
- 1C (to terminate loop) +
- 1A

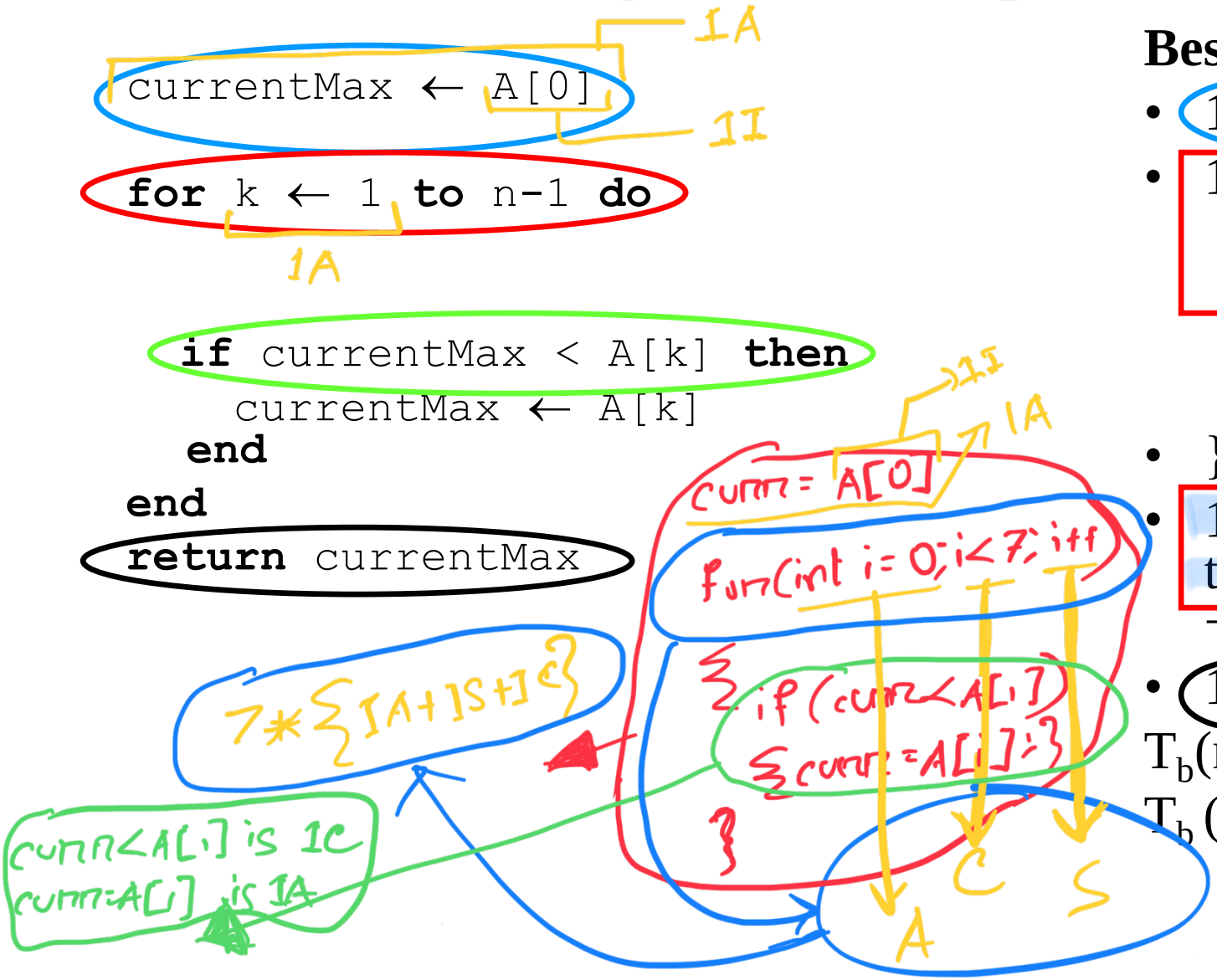
T(n) = 5+(n-1)*7

T(n) = 7n - 2

- How has the input to be arranged to produce the best case and the worst case?

Best Case Running Time T(n)
Counting Primitive Operations

```
currentMax ← A[0]
for k ← 1 to n-1 do
    if currentMax < A[k] then
        currentMax ← A[k]
    end
end
return currentMax
```



Best case

- 1A + 1I +
- 1A + (N-1)*{
  - 1A + 1S +
  - 1C +
  - 1C + 1I +
- }
- 1C (to terminate loop) +
- 1A

$T_b(n) = 5+(n-1)*5$

$T_b(n) = 5n$

Worst Case Running Time T(n)
Counting Assignments & Comparisons

```
currentMax ← A[0]
for k ← 1 to n-1 do
    if currentMax < A[k] then
        currentMax ← A[k]
    end
end
return currentMax
```

Worst case

- 1A +
- 1A + (N-1)*{
  - 1A +
  - 1C +
  - 1C +
  - 1A +
- }
- 1C (to terminate loop)
  +
- 1A

T(n) = 4+(n-1)*4

T(n) = 4n

Examples: Worst-case and Best-case Analysis

Basic units: A & C

```
a ← 3 * n
cnt ← 1
while a > n do
   a ← a - 1
   cnt ← cnt + 1
end
```

T(n) = 2 + 2n(3)+1

T(n) = 3 + 6n

$T_b(n) = T(n)$

Basic units: A & C

```
A: array[0..n-1]
k ← 0
while k < n do
   if A[k] = key then
      return k
   end
   k ← k + 1
end
return "not found"
```

T(n) = 1 + n(3) + 2

T(n) = 3 + 3n

$T_b(n) = 4$

Example: Worst-case Analysis

Basic units: A & C

```
A: array[0..n-1]
Swap: 3A per swap
For: ignore first A and last
   C
for k←0 to n-1 do
   for j←0 to n-1 do
      if A[k]<=A[j] then
         swap(A[k],A[j])
      end
   end
end
```

$$T(n) = (6n + 2)n$$
$$T(n) = 6n^2 + 2n$$

Summations

- In general,

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b)$$

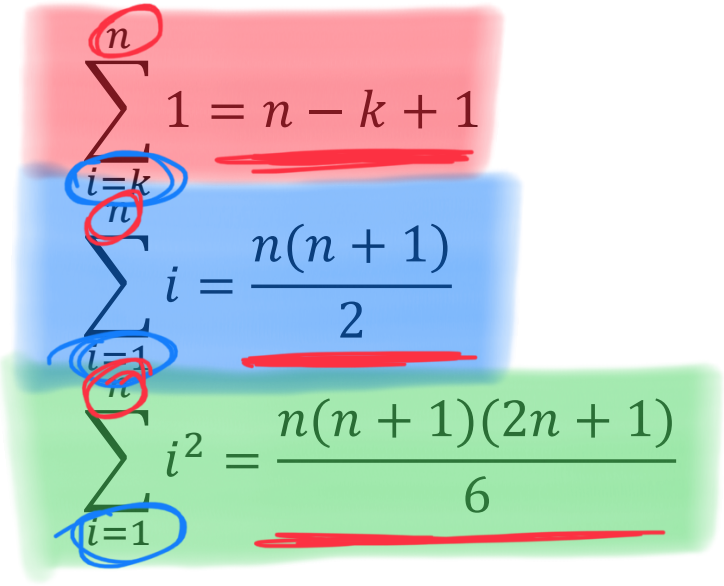
- Identities and formulas,

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} cf(i) = c\sum_{i=1}^{n} f(i)\,, c \text{ independent of } i$$

$$\sum_{i=1}^{n} (f(i) + g(i)) = \sum_{i=1}^{n} f(i) + \sum_{i=1}^{n} g(i)$$

$$\sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a}, \text{where } a \neq 1$$



Example revisited: Worst-case Analysis

Basic units: A & C

```
A: array[0..n-1]
Swap: 3A per swap
Count only the swap
for k←0 to n-1 do
  for j←0 to n-1 do
    if A[k]<=A[j] then
      swap(A[k],A[j])
    end
  end
end
```

$$T(n) = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} 3$$

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