

# Data Structures and Algorithms

## (資料結構與演算法)

### Lecture 1: Algorithm

Hsuan-Tien Lin (林軒田)

`htlin@csie.ntu.edu.tw`

Department of Computer Science  
& Information Engineering

National Taiwan University  
(國立台灣大學資訊工程系)



definition of algorithm

# Name Origin of Algorithm

Muhammad ibn Mūsā al-Ḳwārizmī on a Soviet Union stamp

figure licensed from public domain via

[commons.wikimedia.org/wiki/File:1983\\_CPA\\_5426.jpg](https://commons.wikimedia.org/wiki/File:1983_CPA_5426.jpg)



## algorithm

- named after **al-Ḳwārizmī** (780–850), Persian mathematician and father of algebra
- algebra: **rules** to calculate with symbols
- algorithm: **instructions** to compute with variables

algorithm: **recipe**-like **instructions** for **computing**

a recipe for hamburger on Wikibooks

figure by Gentgeen.

licensed under CC BY-SA 3.0 via Wikimedia Commons

## recipe

Wikipedia: a set of *instructions* that describes how to prepare or make something, especially a dish of prepared food

recipe: instructions to complete a (cooking) task

# Sheet Music for Playing Instrument



first page of the manuscript of Bach's lute suite in G minor

figure licensed as public domain via

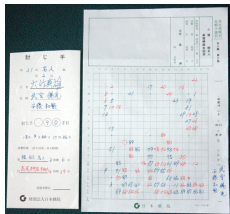
[commons.wikimedia.org/wiki/File:Bachlut1.png](https://commons.wikimedia.org/wiki/File:Bachlut1.png)

## sheet music

Wikipedia: *handwritten or printed form of musical notation ... to indicate the pitches, rhythms or chords of a song*

sheet music: instructions to play instrument (well)

# Kifu for Playing Go



a Japanese kifu

figure by Velobici,

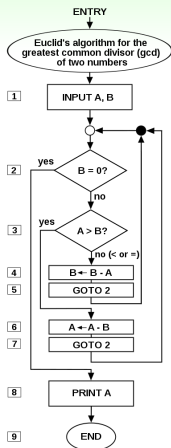
licensed under CC BY-SA 4.0 via Wikimedia Commons

kifu

go game record **of steps** that describe how the game had been played

**kifu**: **instructions** to **mimic/learn to play go** (professionally)

# Algorithm for Computing



flowchart of Euclid's algorithm for calculating the greatest common divisor (g.c.d.) of two numbers

figure by Somepics,

licensed under CC BY-SA 4.0 via Wikimedia Commons

## algorithm

Wikipedia: *algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation*

algorithm ~ computing recipe: (computable) instructions to solve a computing task efficiently/correctly

pseudo code of algorithm



# Pseudo Code for GETMININDEX

## C Version

```
/* return index to min. element
   in arr[0] ... arr[len-1] */
int getMinIndex
    (int arr[], int len){
    int i;
    int m=0;
    for(i=0;i<len;i++){
        if (arr[m] > arr[i]){
            m = i;
        }
    }
    return m;
}
```

## Pseudo Code Version

GET-MIN-INDEX(*A*)

```
1  m = 1
2  for i = 2 to A.length
3      // update if i-th element smaller
4      if A[m] > A[i]
5          m = i
6  return m
```

pseudo code: spoken language of programming

# Bad Pseudo Code: Too Detailed

## Unnecessarily Detailed

GET-MIN-INDEX(*A*)

```
1  m = 1
2  for i = 2 to A.length
3      // update if i-th element smaller
4      Am = A[m]
5      Ai = A[i]
6      if Am > Ai
7          m = i
8      else
9          m = m
10 return m
```

## Concise

GET-MIN-INDEX(*A*)

```
1  m = 1
2  for i = 2 to A.length
3      // update if i-th element smaller
4      if A[m] > A[i]
5          m = i
6  return m
```

goal of pseudo code: communicate efficiently

# Bad Pseudo Code: Too Mysterious

## Unnecessarily Mysterious

GET-MIN-INDEX(*A*)

```
1 x = 1
2 for xx = 2 to A.length
3
4     if A[x] > A[xx]
5         xx = x
6 return xx
```

## Clear

GET-MIN-INDEX(*A*)

```
1 m = 1 // store current min. index
2 for i = 2 to A.length
3     // update if i-th element smaller
4     if A[m] > A[i]
5         m = i
6 return m
```

goal of pseudo code: communicate correctly

# Bad Pseudo Code: Too Abstract

## Unnecessarily Abstract

GET-MIN-INDEX( $A$ )

- 1  $m = 1$  // store current min. index
- 2 **run a loop through  $A$**   
**that updates  $m$  in every iteration**
- 3 **return  $m$**

## Concrete

GET-MIN-INDEX( $A$ )

- 1  $m = 1$  // store current min. index
- 2 **for**  $i = 2$  **to**  $A.length$
- 3     // update if  $i$ -th element smaller
- 4     **if**  $A[m] > A[i]$
- 5          $m = i$
- 6 **return**  $m$

goal of pseudo code: **communicate effectively**

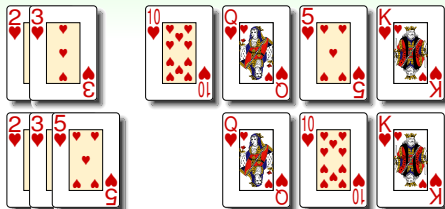
# From GET-MIN-INDEX to SELECTION-SORT

## GET-MIN-INDEX( $A, \ell, r$ )

```

1   $m = \ell$  // store current min. index
2  for  $i = \ell + 1$  to  $r$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 

```



## Good Pseudo Code

- **modularize**, just like coding
- depends on **speaker/listener**
- usually **no formal definition**

## SELECTION-SORT( $A$ )

```

1  for  $i = 1$  to  $A.length$ 
2       $m = \text{GET-MIN-INDEX}(A, i, A.length)$ 
3      SWAP( $A[i], A[m]$ )
4  return  $A$  // which has been sorted in place

```

follow any textbook if you really need a **definition**

criteria of algorithm

# Criteria of Recipe



figure by Larry, licensed under CC BY-NC-ND 2.0 via Flickr

## Cocktail Recipe: Screwdriver

inputs: 5 cl vodka, 10 cl orange juice

- 1 mix inputs in a highball glass with ice
- 2 garnish with orange slice and serve

output: a glass of delicious cocktail

- input:  
ingredients
- definiteness:  
clear instructions
- effectiveness:  
feasible instructions
- finiteness:  
completable instructions
- output:  
delicious drink

algorithm  $\sim$  recipe: same five criteria for algorithm

(Knuth, The Art of Computer Programming)

# Input of Algorithm

*... quantities which are given to it initially before the algorithm begins.  
These inputs are taken from **specified sets of objects**. (Knuth, TAOCP)*

GET-MIN-INDEX(**A**)

```
1   $m = 1$  // store current min. index
2  for  $i = 2$  to  $A.length$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 
```

one algorithm, many uses (on different **legal inputs**)



# Definiteness of Algorithm

*Each step of an algorithm must be precisely defined; the actions to be carried out must be **rigorously & unambiguously specified**. (Knuth, TAOCP)*

## Clear

GET-MIN-INDEX( $A$ )

```
1   $m = 1$  // store current min. index
2  for  $i = 2$  to  $A.length$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 
```

## Ambiguous

GET-ZERO-INDEX( $A$ )

```
1
2  for  $i = 1$  to  $A.length$ 
3
4      if  $A[m]$  is almost zero
5          return  $m$ 
6  // what to return here?
```

definiteness: **clarity** of algorithm

# Effectiveness of Algorithm

... all of the operations to be performed in the algorithm must be sufficiently basic that they can *in principle be done exactly and in a finite length of time* by a man using paper and pencil. (Knuth, TAOCP)

## Effective

GET-MIN-INDEX( $A$ )

```
1   $m = 1$  // store current min. index
2  for  $i = 2$  to  $A.length$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 
```

## Ineffective

GET-SOFT-MIN( $A$ )

```
1   $s = 0$  // sum of exponentiated values
2  for  $i = 1$  to  $A.length$ 
3       $s = s + \exp(-A[i] \cdot 1126)$ 
4
5
6  return  $-\log(s)/1126$ 
```

floating point errors may make some steps ineffective on some computers

# Finiteness of Algorithm

*An algorithm must always terminate after a finite number of steps . . . a very finite number, a reasonable number.*

(Knuth, TAOCP)

GET-MIN-INDEX( $A$ )

```
1   $m = 1$  // store current min. index
2  for  $i = 2$  to  $A.length$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 
```

finiteness (& efficiency): often requiring analysis for sophisticated algorithms (to be taught later)

# Output of Algorithm

... quantities which have a *specified relation* to the inputs (Knuth, TAOCP)

GET-MIN-INDEX( $A$ )

```
1   $m = 1$  // store current min. index
2  for  $i = 2$  to  $A.length$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 
```

output (*correctness*): needs *proving*  
with respect to requirements

correctness proof of algorithm

# Claim



figure by Nick Youngson, licensed CC BY-SA 3.0 via Picpedia.Org

GET-MIN-INDEX( $A$ )

```
1  $m = 1$  // store current min. index
2 for  $i = 2$  to  $A.length$ 
3     // update if  $i$ -th element smaller
4     if  $A[m] > A[i]$ 
5          $m = i$ 
6 return  $m$ 
```

## Correctness of GET-MIN-INDEX

Upon exiting GET-MIN-INDEX( $A$ ),

$$A[m] = \min_{1 \leq j \leq n} A[j]$$

with  $n = A.length$

claim: mathematical statement that **declares correctness**

# Invariant



invariants when constructing fractals  
figures by Johannes Rössel,

licensed from public domain via Wikipedia

## GET-MIN-INDEX(*A*)

```

1  m = 1 // store current min. index
2  for i = 2 to A.length
3      // update if i-th element smaller
4      if A[m] > A[i]
5          m = i
6  return m
  
```

## Correctness of GET-MIN-INDEX

Upon exiting GET-MIN-INDEX(*A*),

$$A[m] = \min_{1 \leq j \leq n} A[j]$$

with  $n = A.length$



## Invariant within GET-MIN-INDEX

Upon finishing the loop with  $i = k$ ,  
denote  $m$  by  $m_k$ ,

$$A[m_k] \leq A[j] \text{ for } j = 1, 2, \dots, k$$

(loop) invariant: property that algorithm maintains

# Proof of Loop Invariant

## Mathematical Induction

### Base

when  $i = 2$ , invariant true because

...

assume invariant true for  $i = t - 1$ ;  
when  $i = t$ ,

- $m_t = t$  if  $A[m_{t-1}] > A[t]$ 
  - $A[m_t] = A[t] \leq A[t]$
  - $A[m_t] < A[m_{t-1}] \leq A[j]$   
for other  $j$
- $m_t = m_{t-1}$  if  $A[m_{t-1}] \leq A[t]$ 
  - $A[m_t] = A[m_{t-1}] \leq A[t]$
  - $A[m_t] = A[m_{t-1}] \leq A[j]$   
for other  $j$

—by mathematical induction,  
invariant true for  $i = 2, 3, \dots, k$

## GET-MIN-INDEX( $A$ )

```

1   $m = 1$  // store current min. index
2  for  $i = 2$  to  $A.length$ 
3      // update if  $i$ -th element smaller
4      if  $A[m] > A[i]$ 
5           $m = i$ 
6  return  $m$ 
  
```

## Correctness of GET-MIN-INDEX



## Invariant within GET-MIN-INDEX

Upon finishing the loop with  $i = k$ ,  
denote  $m$  by  $m_k$ ,

$$A[m_k] \leq A[j] \text{ for } j = 1, 2, \dots, k$$

⇒

proof of (loop) invariants ⇒ correctness claim of algorithm



# Summary

## Lecture 1: Algorithm

- definition of algorithm

**instructions to complete a task by computer**

- pseudo code of algorithm

**communicate alg. efficiently/correctly/effectively**

- criteria of algorithm

**input, definite, effective, finite, output**

- correctness proof of algorithm

**from (loop) invariants to claims**