# COMP108
# Data Structures and Algorithms

## Data structures - Arrays (Part I Searching)

Professor Prudence Wong

pwong@liverpool.ac.uk

2024-25

# Outline

Data structures - arrays

- What are data structures?
- What are arrays?
- Using arrays to look for a number in a sequence of numbers
  - Sequential/Linear search
  - Binary search
  - Finding maximum/minimum

Learning outcome:

- Be able to describe the principles of and apply arrays and their associated algorithms

## What are data structures?

- Dynamic data
  - data may grow, shrink, change over time
- Operations on dynamic data
  - search, insert, delete, minimum, maximum, successor, predecessor
  - some operations actually change the data: insert, delete
- Data structures
  - systematic way of organising and accessing data
- Examples
  - Arrays
  - Queues and Stacks
  - Linked lists
  - Trees and Graphs
  - Hash tables

## Arrays

- Array: numbered collection of data of the same type
- Each cell has an index, containing an element
- Array $A$ of size $n$ with indices 1 to $n$: $A[1..n]$
  - note: in certain programming languages, arrays are indexed as $A[0..(n-1)]$
  - in the lecture notes, we use $1..n$
- Out of bounds: any index not in the range $1..n$
- Use a loop to access the elements of the array

```
// summing all elements of an array
sum ← 0, i ← 1
while i ≤ n do
begin
    sum ← sum + A[i]
    i ← i + 1
end
output sum
```

# Sequential Search . . .

## Sequential/Linear search

- Input: $n$ numbers stored in an array $A[1..n]$, and a target number $key$
- Output: determine if $key$ is in the array or not
- Algorithm (Sequential / Linear search)
  1. From $i \leftarrow 1$, compare $key$ with $A[i]$ one by one as long as $i \leq n$.
  2. Stop and report ``Found!'' when $key$ is the same as $A[i]$.
  3. Repeat and report ``Not Found!'' when $i > n$.

## Sequential search - Example - To find 7

| | | | | | | |
|---|---|---|---|---|---|---|
| **12** | 34 | 2 | 9 | 7 | 5 | ← 6 numbers |
| **7** | | | | | | ← key |
| 12 | **34** | 2 | 9 | 7 | 5 | |
| | **7** | | | | | |
| 12 | 34 | **2** | 9 | 7 | 5 | |
| | | **7** | | | | |
| 12 | 34 | 2 | **9** | 7 | 5 | |
| | | | **7** | | | |
| 12 | 34 | 2 | 9 | **7** | 5 | |
| | | | | **7** | | **FOUND!** |

## Sequential search - Example - To find 10

| | | | | | | |
|---|---|---|---|---|---|---|
| **12** | 34 | 2 | 9 | 7 | 5 | ← 6 numbers |
| **10** | | | | | | ← key |
| 12 | **34** | 2 | 9 | 7 | 5 | |
| | **10** | | | | | |
| 12 | 34 | **2** | 9 | 7 | 5 | |
| | | **10** | | | | |
| 12 | 34 | 2 | **9** | 7 | 5 | |
| | | | **10** | | | |
| 12 | 34 | 2 | 9 | **7** | 5 | |
| | | | | **10** | | |
| 12 | 34 | 2 | 9 | 7 | **5** | |
| | | | | | **10** | **NOT FOUND!** |

## Sequential search: Pseudo code - Ideas

variable $i$ to step through the array
boolean *found* to indicate whether *key* is found

```
i ← ??
found ← ??
while i ≤ ?? AND found == ?? do
begin
    /* check whether the i-th element
        of the array equals key and if so
        set found accordingly */
    i ← i + 1
end
if found == true then
    output ``Found!''
else
    output ``Not found!''
```

## Sequential search: Pseudo code (see `SampleSeqSearch.java` on Canvas)

variable $i$ to step through the array
boolean *found* to indicate whether *key* is found

$i \leftarrow 1$
$found \leftarrow false$
while $i \leq n$ AND $found == false$ do
begin
    if $key == A[i]$ then
        $found \leftarrow true$
    else
        $i \leftarrow i + 1$
end
if $found == true$ then
    output ``Found!''
else
    output ``Not found!''

## Sequential search: Time complexity

$i \leftarrow 1$
$found \leftarrow false$
while $i \leq n$ AND $found == false$ do
begin
    if $key == A[i]$ then
        $found \leftarrow true$
    else
        $i \leftarrow i + 1$
end

How many comparisons this algorithm takes?

Best case: *key* is first number
$\Rightarrow$ 1 comparison
$\Rightarrow O(1)$

Worst case: *key* is last number OR *key* is not found
$\Rightarrow n$ comparison
$\Rightarrow O(n)$

If the data is sorted in ascending/descending order,
can we improve the time complexity?

# Binary Search . . .

## Binary search

- more efficient way of searching when the sequence of numbers is pre-sorted
- Input: a sequence of $n$ sorted numbers $A[1], A[2], \cdots, A[n]$ in ascending order and a target number *key*
- Idea of algorithm:
  1. compare *key* with number in the middle
  2. then focus on only the first half or the second half (depend on whether *key* is smaller or greater than the middle number)
  3. reduce the amount of numbers to be searched by half

## Binary search - Example - To find 24

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 11 | 12 | **15** | 19 | 24 | 33 | 41 | 55 | ← 10 numbers |
| | | | | **24** | | | | | | ← key |
| | | | | | 19 | 24 | **33** | 41 | 55 | ← 5 numbers left |
| | | | | | | | **24** | | | ← key |
| | | | | | **19** | 24 | | | | ← 2 numbers left |
| | | | | | **24** | | | | | ← key |
| | | | | | | **24** | | | | ← 1 number left |
| | | | | | | **24** | | | | **FOUND!** |

## Binary search - Example 2 - To find 30

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 11 | 12 | **15** | 19 | 24 | 33 | 41 | 55 | ← 10 numbers |
| | | | | **30** | | | | | | ← key |
| | | | | | 19 | 24 | **33** | 41 | 55 | ← 5 numbers left |
| | | | | | | | **30** | | | ← key |
| | | | | | **19** | 24 | | | | ← 2 numbers left |
| | | | | | **30** | | | | | ← key |
| | | | | | | **24** | | | | ← 1 number left |
| | | | | | | **30** | | | | **NOT FOUND!** |

## Binary search - Pseudo code

$first \leftarrow 1$
$last \leftarrow n$
$found \leftarrow false$
while $first \leq last$ AND $found == false$ do
begin

    // check with number in the middle

end
if $found == true$ then
    output ``Found!''
else
    output ``Not found!''

$A[first], \cdots, A[mid], \cdots, A[last]$

$key = A[mid]$

$\longleftarrow$ $key < A[mid]$

$key > A[mid]$ $\longrightarrow$

## Binary search - Pseudo code

```
first ← 1
last ← n
found ← false
while first ≤ last AND found == false do
begin

    // check with number in the middle

end
if found == true then
    output ``Found!''
else
    output ``Not found!''
```

$\lfloor\ \rfloor$ is the floor function
truncates the decimal part

```
mid ← ⌊(first + last) / 2⌋
if key == A[mid] then
    found ← true
else
    if key < A[mid] then
        last ← mid − 1
    else
        first ← mid + 1
```

## Binary search - Pseudo code

$first \leftarrow 1$, $last \leftarrow n$, $found \leftarrow false$
while $first \leq last$ AND $found == false$ do
begin
  $mid \leftarrow \lfloor \frac{first+last}{2} \rfloor$
  if $key == A[mid]$ then
    $found \leftarrow true$
  else if $key < A[mid]$ then
      $last \leftarrow mid - 1$
    else $first \leftarrow mid + 1$
end
if $found == true$ then
    output ``Found!''
else
    output ``Not found!''

Why $first \leq last$?

- When there is one number left, both *first* and *last* (and *mid*) point to the same location
- If this number isn't the key, then either *first* becomes $mid + 1$ or *last* becomes $mid - 1$.
- In both cases, *first* becomes larger than *last* and the while condition becomes false; hence the loop terminates.

## Binary search - Time complexity

**Best case:**

- *key* is the number in the middle

$\Rightarrow$ 1 comparison

$\Rightarrow$ $O(1)$

**Worst case:**

- at most $\lceil \log_2 n \rceil + 1$ comparisons

$\Rightarrow$ $O(\log n)$

Why?

Every comparison reduces the amount of numbers by at least half

E.g., $16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$

```
first ← 1, last ← n, found ← false
while first ≤ last AND found == false do
begin
   mid ← ⌊(first+last)/2⌋
   if key == A[mid] then
      found ← true
   else if key < A[mid] then
         last ← mid − 1
      else first ← mid + 1
end
```

Summary: Arrays - linear search and binary search

Next: Arrays - finding maximum/minimum

# For note taking