COMP20007 Workshop Week 8

	Preparation: - open `ws8.pptx` from `github.com/anhvir/c207 and/or` - open `wokshop8.pdf` (from `LMS`), and - download lab files from `LMS`
1	Master Theorem, Merge Sort, problems 1, 3
2	Simple Sorting algorithms, problem 2 Quicksort and Mergesort, problem 3
LAB	Understand the lab's graph module and implement some graph algorithms

The Master Theorem

If

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^d\right)$$

$$T(1) = \Theta(1)$$

where $a \geq 1$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

you can also replace *all* big-θ by big-O

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^d\right)$$

$$T(1) = \Theta(1)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

T1: Find time complexity (big- or big-O) for:

(a) $T(n) = 9T\left(\frac{n}{3}\right) + n^3,\ T(1) = 1$

(b) $T(n) = 64T\left(\frac{n}{4}\right) + n + \log n,\ T(1) = 1$

(c) $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(1) = 1$

(d) $T(n) = 2T\left(\frac{n}{2}\right),\ T(1) = 1$

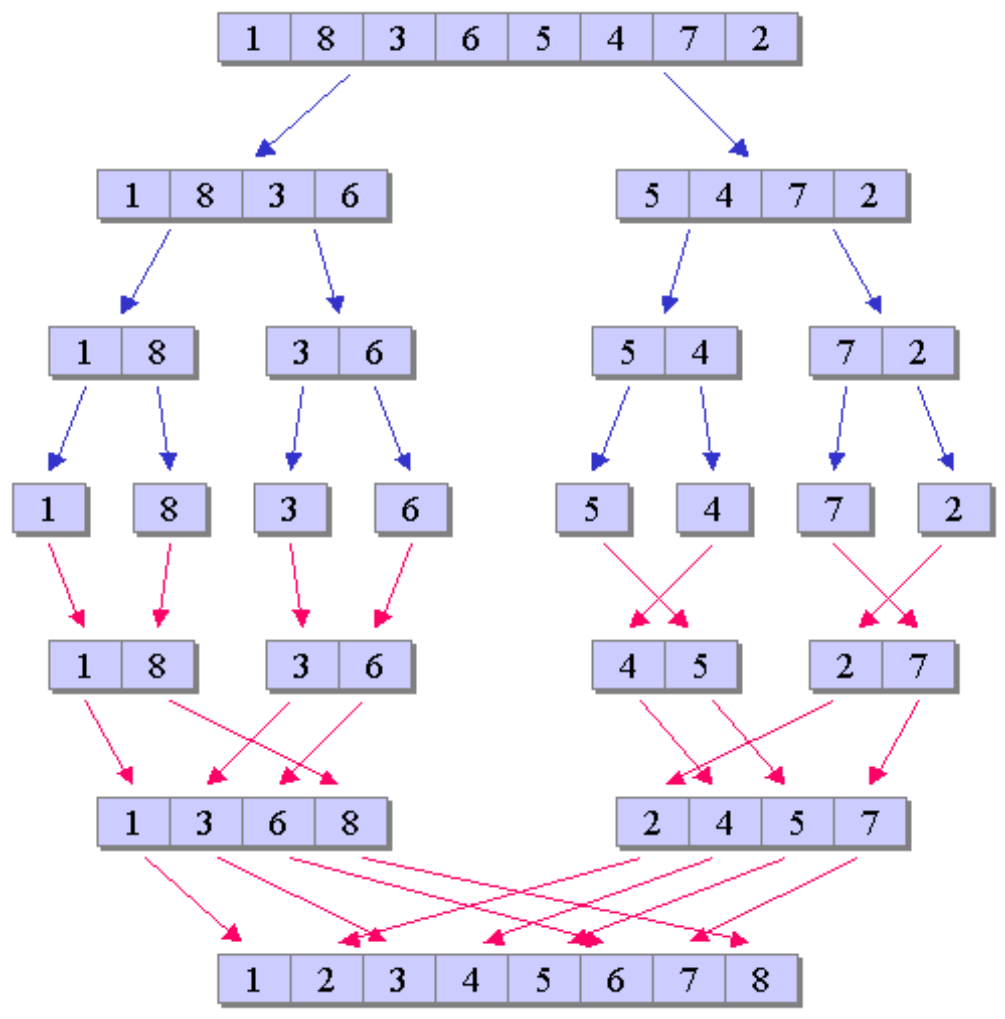
(e) T(n) = 2T(n-1) + 1

T3: Mergesort Time Complexity

Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

- Construct a recurrence relation to describe the runtime of mergesort Explain where each term in the recurrence relation comes from.
- Use the Master Theorem to find the time complexity of Mergesort in Big Theta terms.

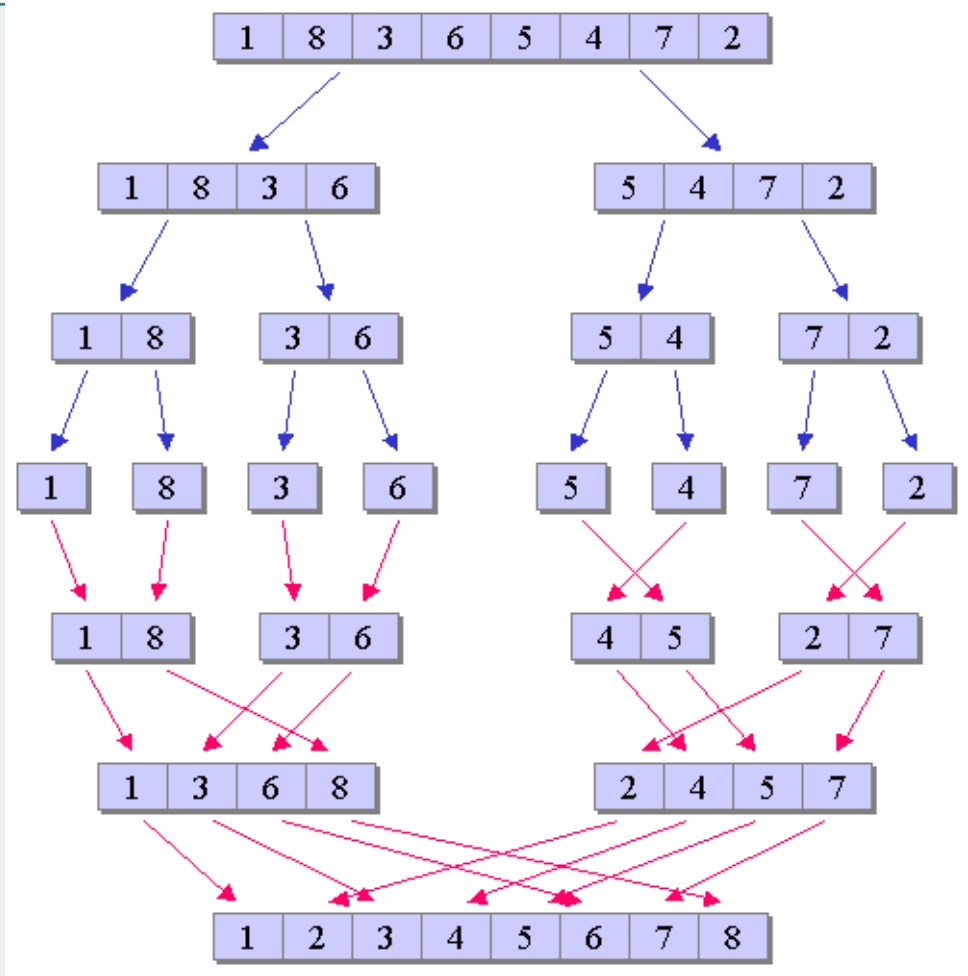


T3: Mergesort Time Complexity

- Construct a recurrence relation to describe the runtime of mergesort . Explain where each term in the recurrence relation comes from.

  T(n)=
  T(1)=

- Use the Master Theorem to find the time complexity of Mergesort in Big-Theta terms.



$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^d\right)$$

$$T(1) = \Theta(1)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Simple Sorting Algorithms

- Any problem with Insertion Sort and Selection Sort?

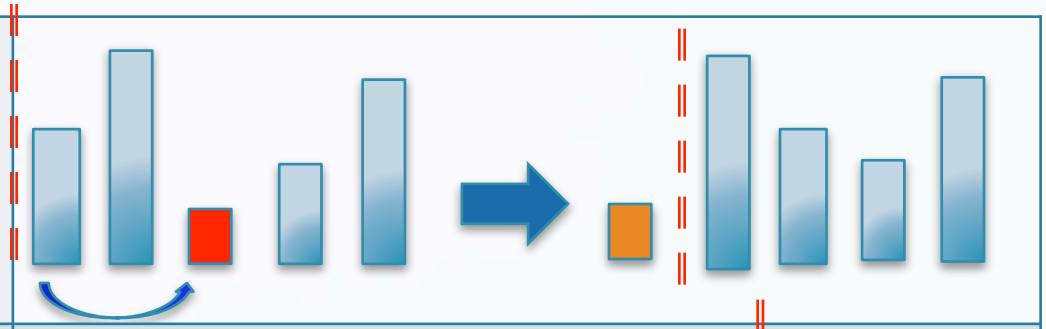
Selection Sort = ?

In each round, finalise the position of 1 element for the sorted array:

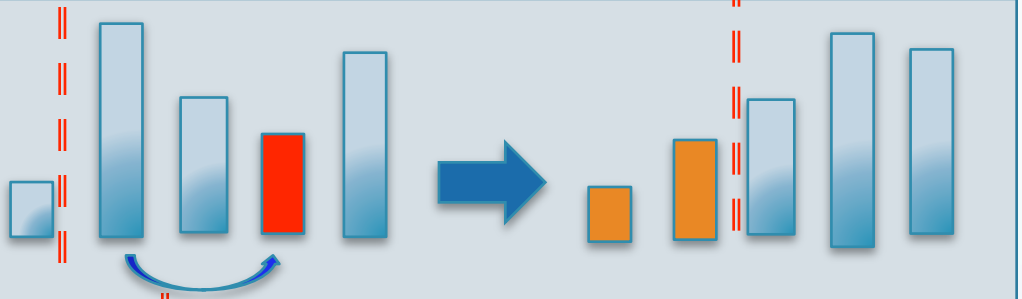
find the location of the smallest (or largest) element

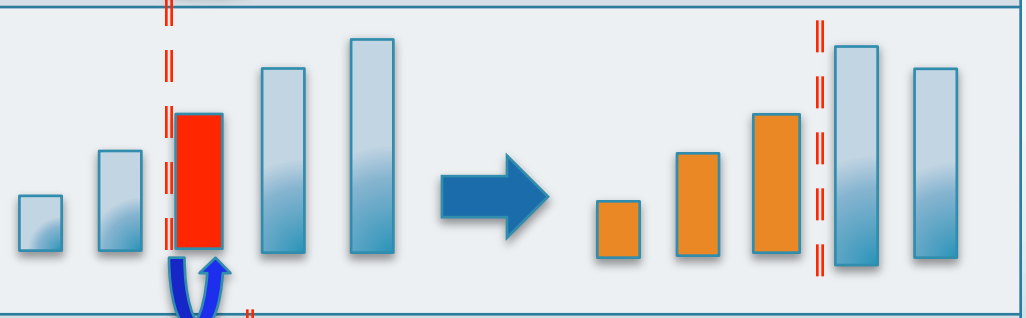
Selection Sort: n=5, increasing order, select the smallest

Round 1: consider `A[0..4]`
- *determine position of the largest*
- *swap with the last, ie.* `A[4]`

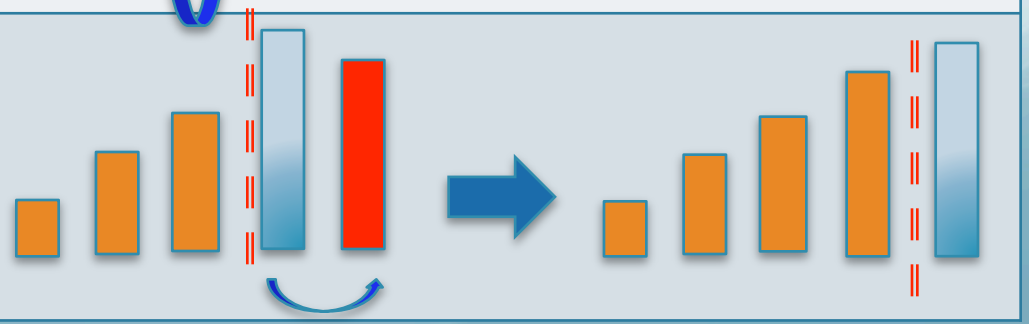


Round 2: consider `A[0..3]`
- *determine position of the largest*
- *swap with the last, ie.* `A[3]`





Round 4: consider `A[0..1]`
- *determine position of the largest*
- *swap with the last, ie.* `A[1]`



COMP20007.Worshop

Selection Sort

```
function SelectionSort(A[0..n-1]
```

1	**A** N A L Y S I S
2	

Run the algorithm on the input array: `[A N A L Y S I S]` , Suppose: **Sort in increasing order.**

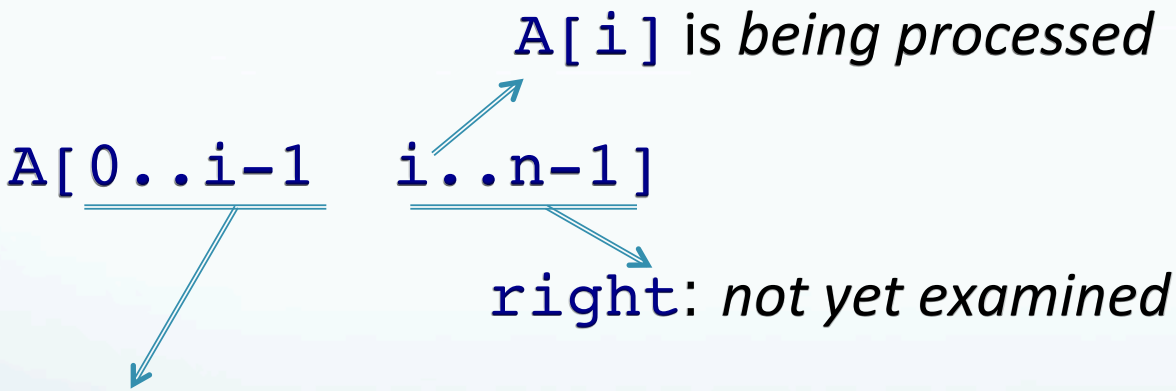
i. What is the time complexity of the algorithm?
ii. Is the sorting algorithm stable?
iii. Does the algorithm sort in-place?
iv. Is the algorithm input sensitive?

Insertion Sort

- Insertion Sort for array A[0..n-1]:

  - process one input at a time

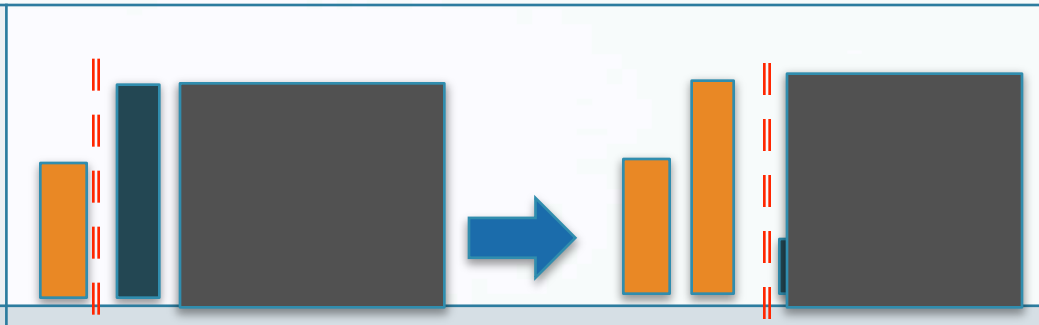
  - keeping *processed elements* sorted by inserting `A[i]` to the left



Insertion Sort: understanding (n=5)

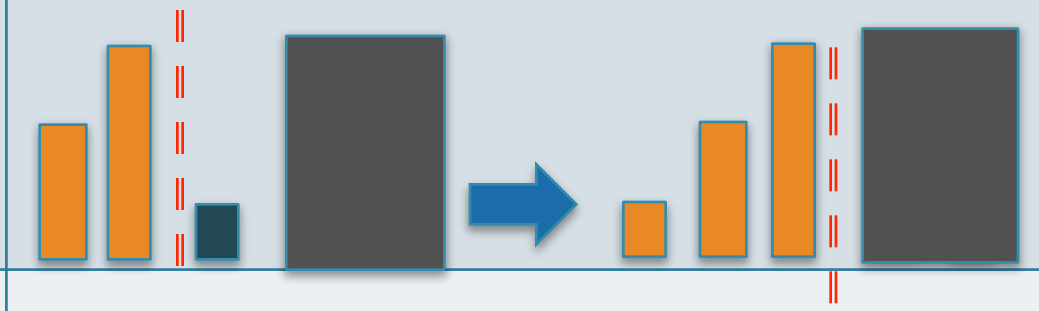
Round 1: consider `A[1]`

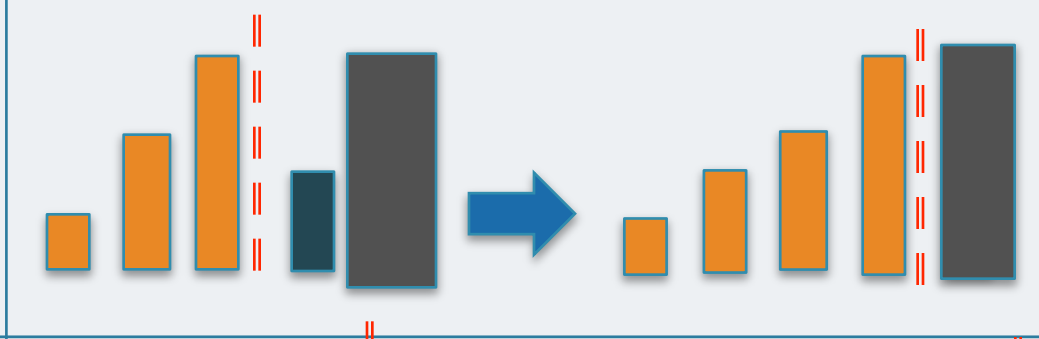
- `A[0..0]` *is sorted*
- *insert* `A[1]` *to the left so that* `A[0..1]` *is sorted*



Round 2: consider `A[2]`

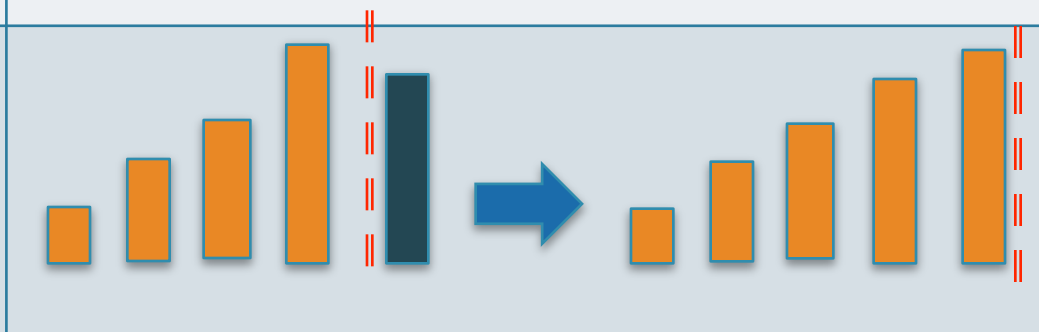
- `A[0..1]` *is sorted*
- *insert* `A[2]` *to the left so that* `A[0..2]` *is sorted*





Round i: consider `A[4]`

- `A[0..4]` *is sorted*
- *insert* `A[i]` *to the left so that* `A[0..4]` *is sorted*



Insertion Sort

```
function InsertionSort(A[0..n − 1])
    for i ← 1 to n − 1 do
        j ← i − 1
        while j ≥ 0 and A[j + 1] < A[j] do
            SWAP(A[j + 1], A[j])
            j ← j − 1
```

1	A N A L Y S I S
2	

i. Run the algorithm on the input array: `[A N A L Y S I S]` ,
   Suppose: **Sort in increasing order.**
ii. What is the time complexity of the algorithm?

iii. Is the sorting algorithm stable?
iv. Does the algorithm sort in-place?
v. Is the algorithm input sensitive?

Quicksort (usage: `Quicksort(A[0..n-1])`

function QUICKSORT($A[l..r]$)
  if $l < r$ **then**
    $s \leftarrow$ PARTITION($A[l..r]$)
    QUICKSORT($A[l..s-1]$)
    QUICKSORT($A[s+1..r]$)

```
Partition(A[l..r])
      <=A[s]          >=A[s]
    A[l..s-1   s    s+1..r   ]

   return s
```

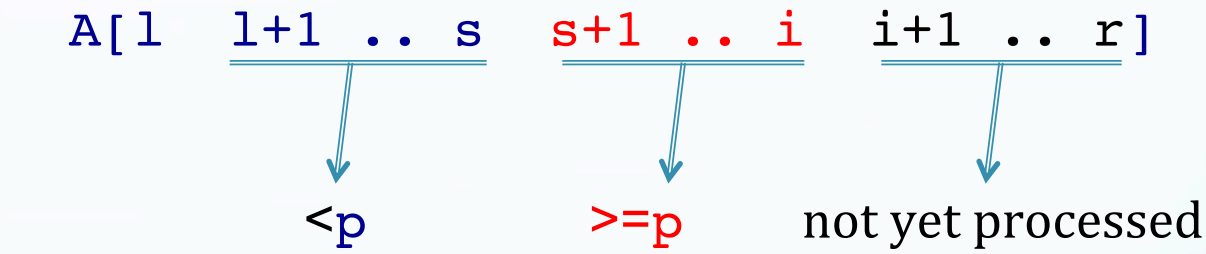
Quicksort: Lomuto Partitioning

```
function LomutoPartition(A[l..r])
    p ← A[l]
    s ← l
    for i ← l + 1 to r do
        if A[i] < p then
            s ← s + 1
            Swap(A[s], A[i])
    Swap(A[l], A[s])
    return s
```

Using `p= A[l]` as the *pivot*

Loop: processing `A[i] with i=l+1,...,r`

During the loop, maintaining *loop invariants*:



After the loop: swap A[l] with A[s]

Example: partitioning for 4 2 5 1 6

`s=0, i=1` $4_s\ 2_i\ 5\ 1\ 6$

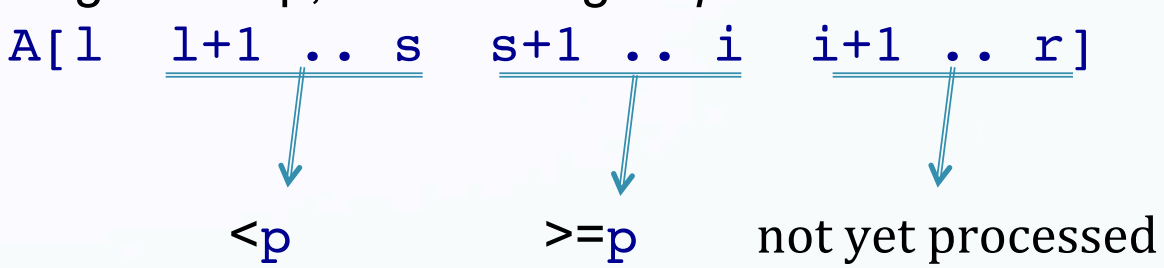
Quicksort: Lomuto Partitioning

```
function LomutoPartition(A[l..r])
    p ← A[l]
    s ← l
    for i ← l + 1 to r do
        if A[i] < p then
            s ← s + 1
            Swap(A[s], A[i])
    Swap(A[l], A[s])
    return s
```

Using `p= A[l]` as the *pivot*

Loop: processing `A[i]`

During the loop, maintaining *loop invariants*:



After the loop: swap A[l] with A[s]

Exercise: quicksort for ANALYSIS

A N A L Y S I S	

Quicksort (usage: `Quicksort(A[0..n-1])`)

function QUICKSORT($A[l..r]$)
  if $l < r$ **then**
    $s \leftarrow$ PARTITION($A[l..r]$)
    QUICKSORT($A[l..s-1]$)
    QUICKSORT($A[s+1..r]$)

input-sensitive?
complexity=?

stable?
in-place?

```
Partition(A[l..r])
      <=A[s]          >=A[s]

    A[l..s-1   s    s+1..r   ]

  return s
```

Quicksort: Hoare Partitioning

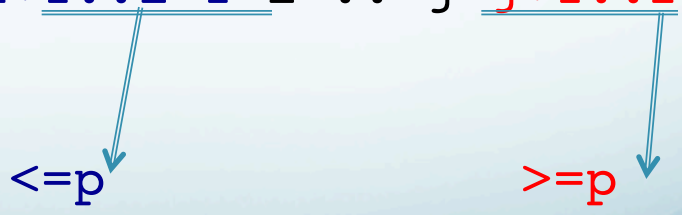
```
function HoarePartition(A[l..r])
    p ← A[l]
    i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        Swap(A[i], A[j])
    until i ≥ j
    Swap(A[i], A[j])
    Swap(A[l], A[j])
    return j
```

moving `i` *forward,* `j` *backward so that:*

```
A[l+1..i-1]<=p, A[i]>=p
A[j+1..r]>=p,   A[j]<=p
```

```
A[l..j .. i ..r]
undo last swap (why?)
```

```
------during the loop------
A[l l+1..i-1 i .. j j+1..r]
```



Example:

4 2 5 1 6

Quicksort: Hoare Partitioning

```
function HoarePartition(A[l..r])
    p ← A[l]
    i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        Swap(A[i], A[j])
    until i ≥ j
    Swap(A[i], A[j])
    Swap(A[l], A[j])
    return j
```

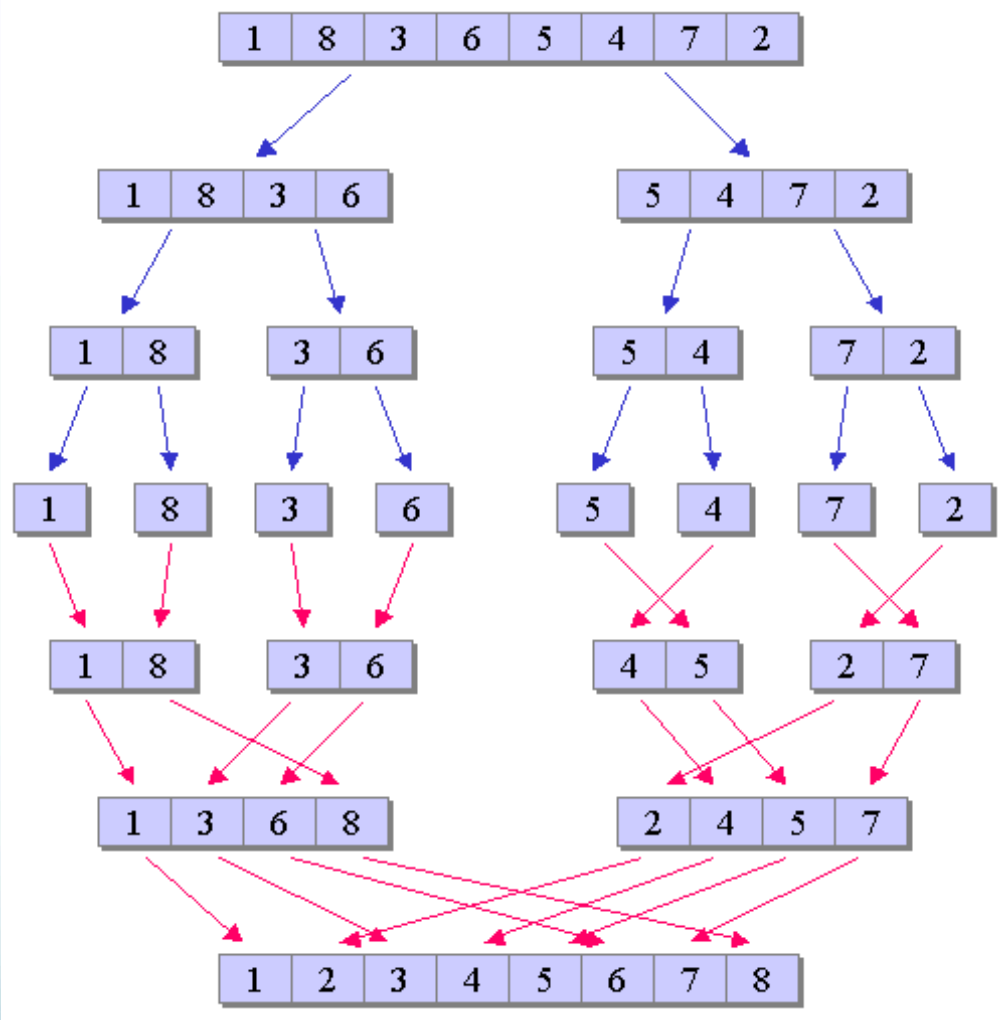
Exercise: run Quicksort

A N A L Y S I S p=**A**

T3: Mergesort: Top-Down & Bottom-Up Algorithms

function MERGESORT($A[0..n-1]$)
  if $n > 1$ **then**
    $B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1]$
    $C[0..\lfloor n/2 \rfloor - 1] \leftarrow A[\lfloor n/2 \rfloor..n-1]$
    MERGESORT($B[0..\lfloor n/2 \rfloor - 1]$)
    MERGESORT($C[0..\lfloor n/2 \rfloor - 1]$)
    MERGE(B, C, A)

```
function BottomUpMS(A[0..n-1])
  create_empty_queue
  enqueue singleton array A[0],
    A[1],..., A[n-1] into Q
  while queue.size>1 do
    dequeue arrays A and B
    merge A and B to C
    enqueue C
  dequeue A
```



T3: Mergesort: Top-Down & Bottom-Up Algorithms

function MERGESORT($A[0..n-1]$)
  if $n > 1$ **then**
    $B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1]$
    $C[0..\lfloor n/2 \rfloor - 1] \leftarrow A[\lfloor n/2 \rfloor..n-1]$
    MERGESORT($B[0..\lfloor n/2 \rfloor - 1]$)
    MERGESORT($C[0..\lfloor n/2 \rfloor - 1]$)
    MERGE(B, C, A)

Exercise: run bottum-up:

```
   ANALYSIS
[A] [N] [A] [L] [Y] [S] [I] [S]
```

```
function BottomUpMS(A[0..n-1])
  create_empty_queue
  enqueue singleton array A[0],
    A[1],..., A[n-1] into Q
  while queue.size>1 do
    dequeue arrays A and B
    merge A and B to C
    enqueue C
  dequeue A
```

T3: Closest-pair and element-distinction

4. Lower bound for the Closest Pairs problem The closest pairs problem takes n points in the plane and computes the Euclidean distance between the closest pair of points.

The algorithm provided in lectures to solve the closest pairs problem applies the divide and conquer strategy and has a time complexity of $O(n \log n)$.

The *element distinction problem* takes as input a collection of n elements and determines whether or not all elements are distinct. It has been proved that if we disallow the usage of a hash table[4] then this problem cannot be solved in less than $n \log n$ time (*i.e.*, this class of problems is $\Omega(n \log n)$).

Describe how we could use the closest pair algorithm from class to solve the element distinction problem (where the input is a collection of floating point numbers), and hence explain why this proves that the closest pair problem must not be able to be solved in less that $n \log n$ time (and is thus $\Omega(n \log n)$).