COMP20007 Workshop Week 7

1	The Master Theorem: Q7.3
2	Binary Heap as a concrete DS for Priority Queues, Heap Sort, Q7.1
3	Quick Sort
4	Sorting Algorithms, Q 7.2 (extended)
	The k-smallest problem, Q 7.5, 7.6
	Lower Bound for the Closest Pair Problem: Q7.4
LAB	Play with some sorting algorithm, especially different choices of pivot for qsort
	Using the provided code, understand how to do timing

The Master Theorem (for complexity of divide-and-conquer algorithms)

If

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

$$T(1) = \mathrm{Q}(1)$$

where $a \geq 1,\ b > 1,$ then

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

Note:

- Similar results hold for O and **Ω**
- Also OK if T(1)= 0

Q7.3: Find time complexity

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

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

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

we can also compare $\log_b a$ & d

(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, T(1)=1

(a)

(b)

(c)

(d)

(e)

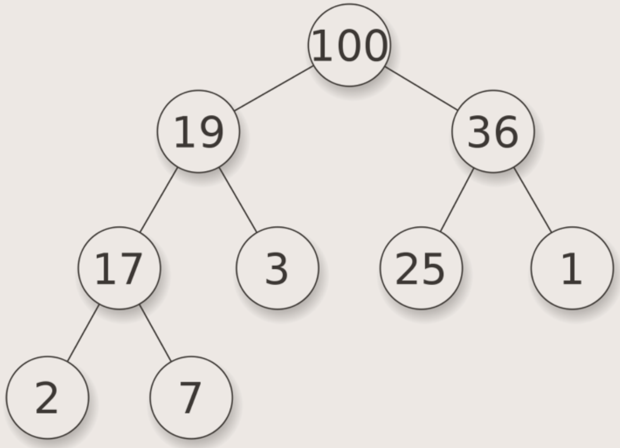
ADT Priority Queue and one of its implementation: Binary Heap

'Can I borrow your baby?...'

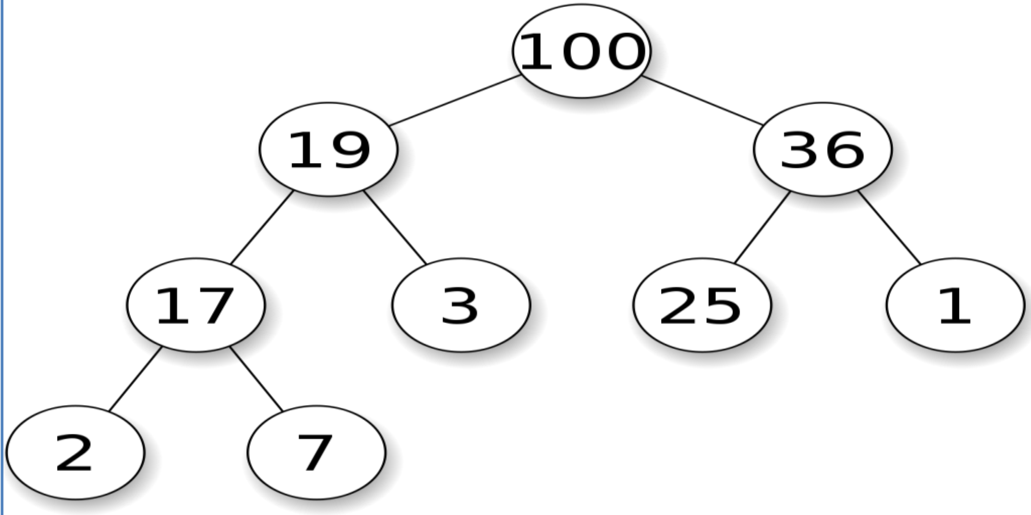
Binary Heap as a *concrete data type* (implementation) for PQ.

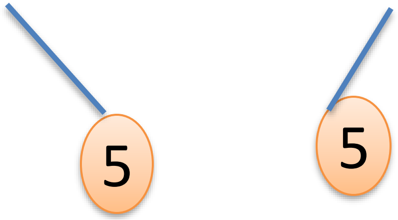
Simple priority: higher (max heap)
lower (min heap)

What is a, say, max heap?
How is it implemented?



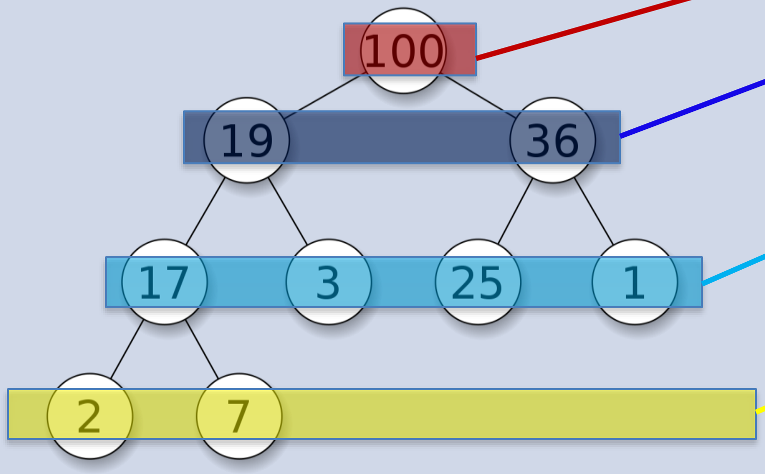
Binary Heap: conceptually, is a binary tree satisfying 2 conditions

Example	Conditions
	1. The tree is *complete*: • all levels, except for the last, are full • the last level is filled from left to right 2. The *heap property*: each node has a higher priority (here, is larger) than any of its descendants (or equivalently, just its children).



Binary Heap: is implemented as an array!

Visualisation: as a complete binary tree



Implementation: using arrays

idx	0	1	2	3	4	5	6	7	8	9
val	×	**100**	**19**	**36**	**17**	**3**	**25**	**1**	**2**	**7**

note: for convenience:
`H[1]` is for the root
`H[0]` not used (or used for a sentinel)

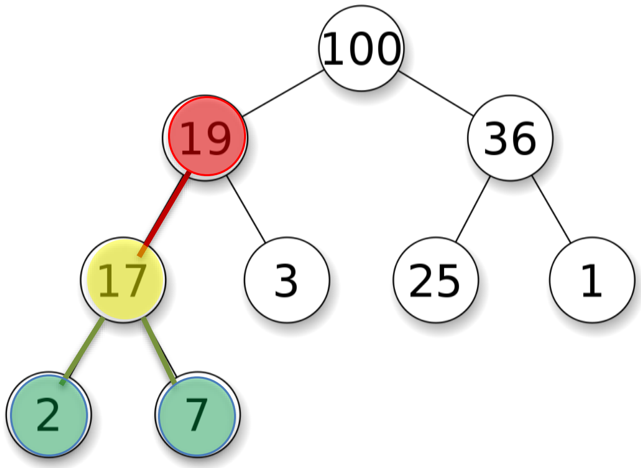
Heap is `H[1..n]`

- level `i` occupies 2^{i} cells in array `H[1..n]` (except for the last level)
- if root is level 1, then level `i` starts from `H[`2^{i-1}`]`

Binary Heap: parent and children of a node

- left child of H[i] is H[2*i]
- right child of H[i] is H[2*i+1]

parent of H[i] is H[i/2]



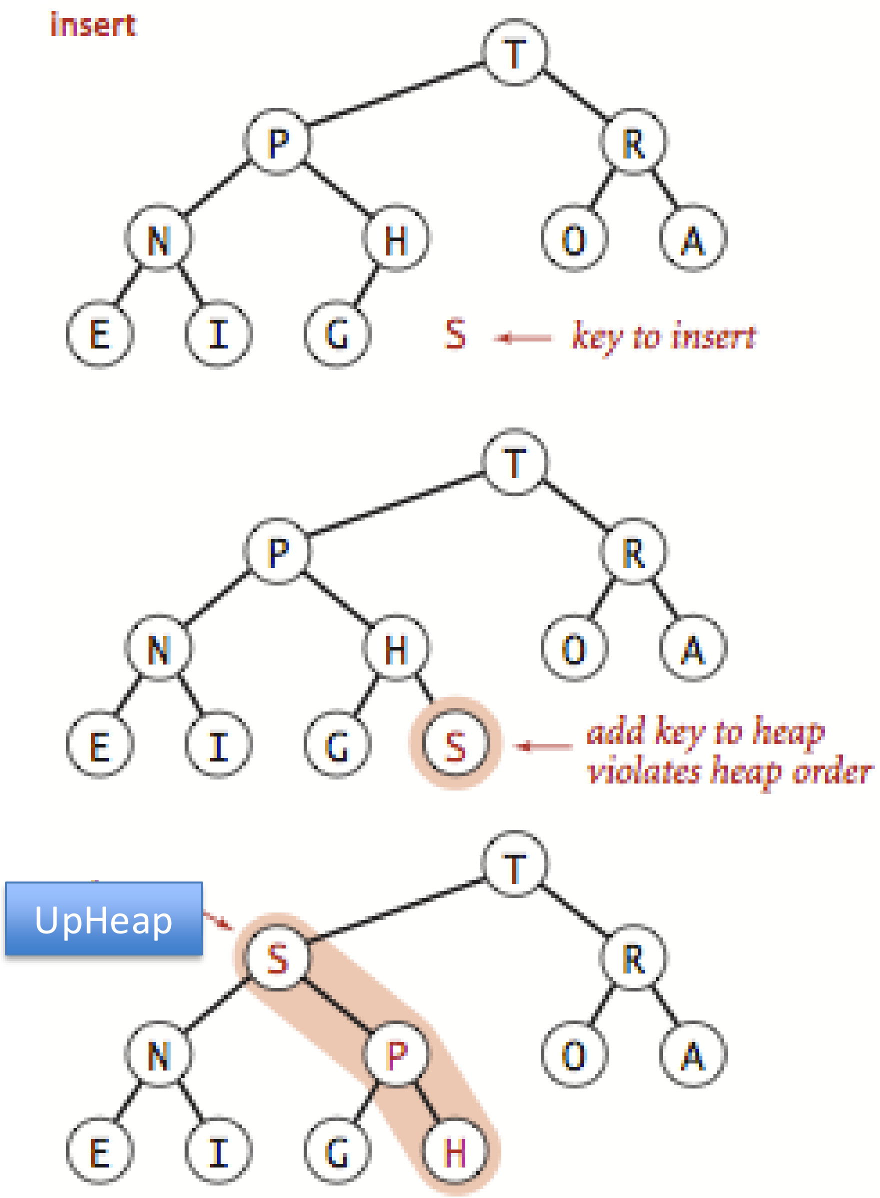
4/2 i=4 4*2 4*2+1

idx	0	1	2	3	4	5	6	7	8	9
val	×	100	19	36	17	3	25	1	2	7

Note:

- The use of indices 1..n is not a rule
- index 0..n-1 can also be used for heaps, but the formula for parent/children are not so nice

inject = enPQ = Insert a new elem into a heap using UpHeap



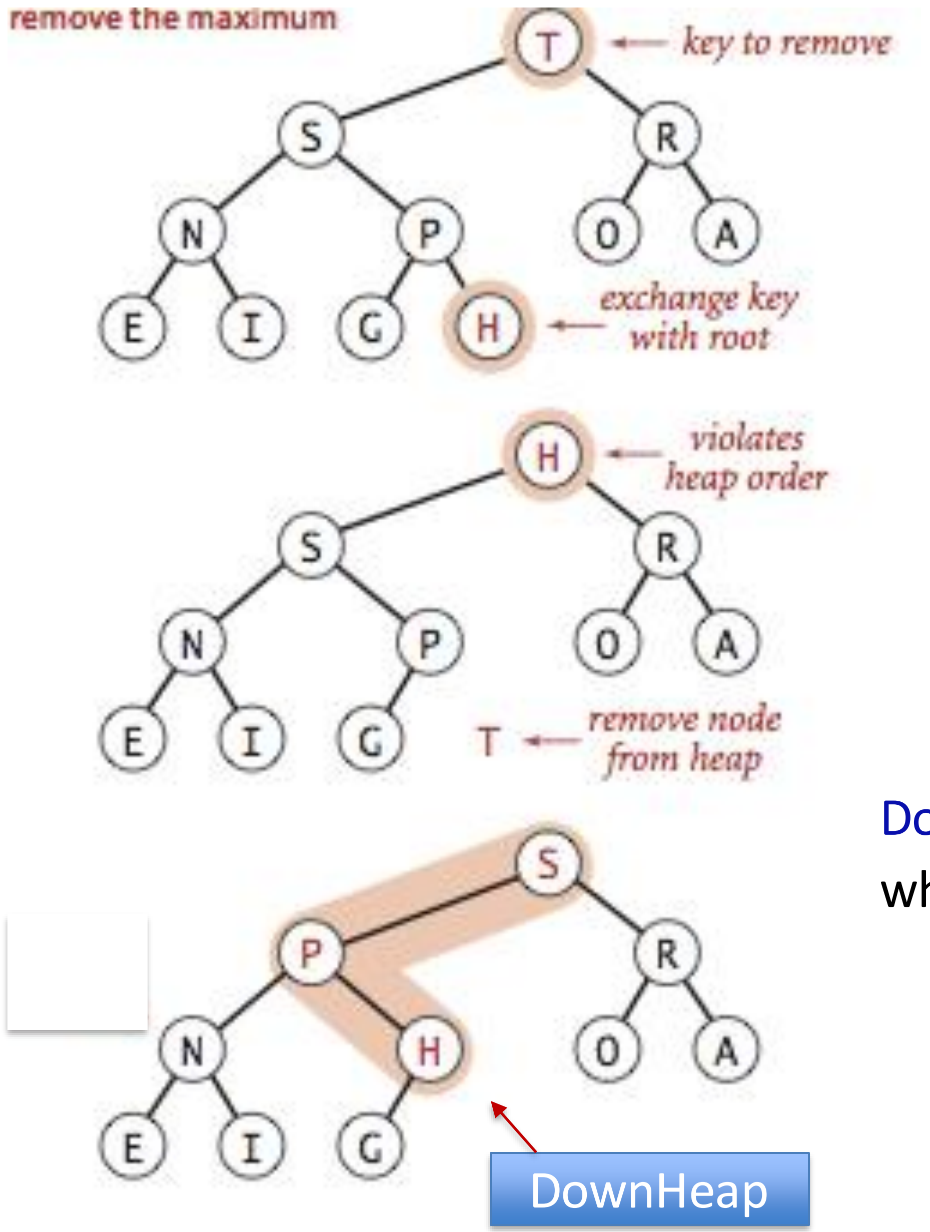
Notes:
Procedure:

Complexity:

UpHeap (aka. SiftUp)

while (has parent and parent has lower priority):
    swap up with the parent

eject = delete (and returns the heaviest) and repair using DownHeap



Notes:

Procedure:

Complexity:

DownHeap (aka. SiftDown):

while (has children & the heavier child has higher priority):

swap down with the *heavier* child

Heapify: Turning an array H[1..n] into a heap

function Heapify(H[1..n])

  for i ← n/2 downto 1 do

    downheap(H, i)

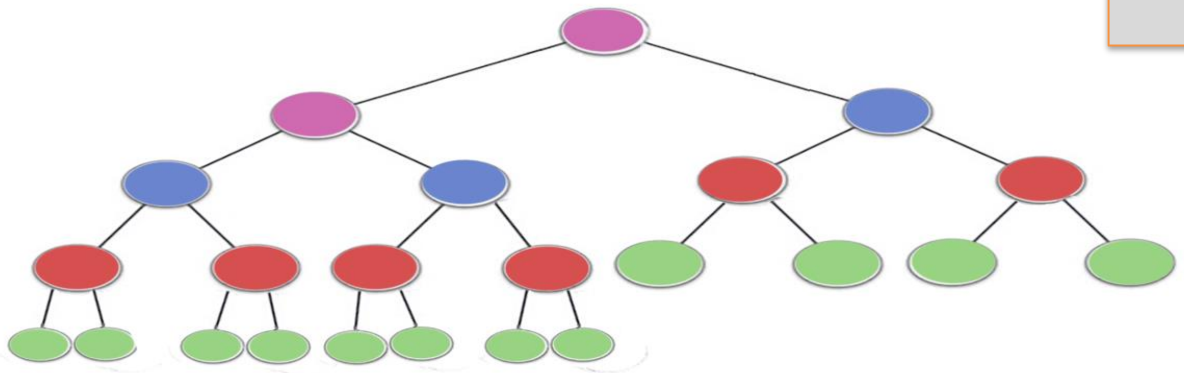
= **Θ(n)** (see lectures and/or ask Google for a proof)

The operation is aka. **Heapify**/Makeheap/ Bottom-Up Heap Construction

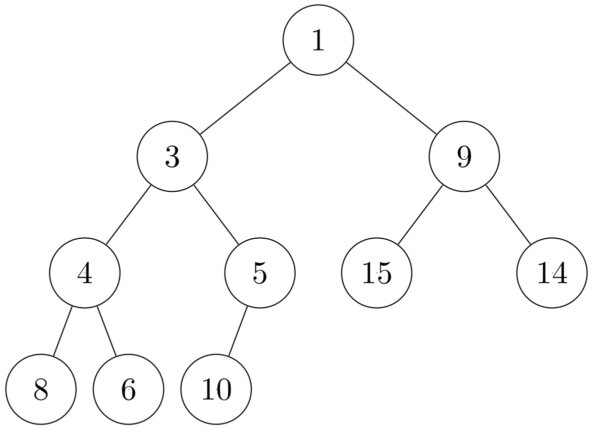
Example: build maxheap for keys E X A M P

Notes:

Complexity:



Q7.1



a)Show how this heap would be stored in an array as discussed in lectures (root is at index `1`; node at index `i` has children at indices `2i` and `2i+1`)

b)Run the `RemoveRootFromHeap (eject)` algorithm from lectures on this heap by hand

c)Run the InsertIntoHeap `(inject)` algorithm and insert the value 2 into the heap

Your answers:

a) array is: [???]

b) Run the RemoveRootFromHeap:

c) Run the InsertIntoHeap(2):

Heap Sort

Describe an algorithm of using a heap for:

- Sorting an array in increasing order
- Sorting an array in decreasing order

Then:

- Do Complexity Analysis, use correct O or $\boldsymbol{\Theta}$
- Compare Heap Sort with Selection Sort
- Compare Heap Sort with fast sorting algorithms

Priority Queues: A Summary

A Priority Queue (PQ) is an Abstract Data Type (ADT) that enables the efficient removal or examination of the highest-priority element. Main operations:

- inject, or enPQ, or insert
- eject, or dePQ, or deleteMin (deleteMax)

Priority Queues are particularly useful in scenarios requiring:

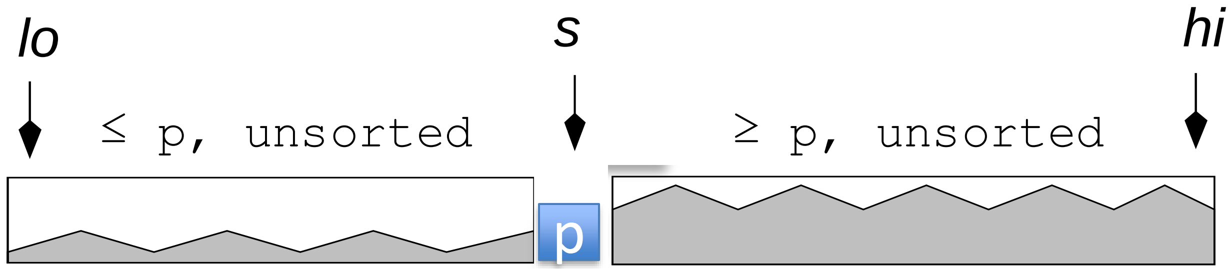
- Repeated removal of the smallest or largest element.
- Processing a set of elements in ascending or descending order of priority.

Priority Queues can be efficiently implemented using a Binary Heap, providing the following time complexities:

- $O(\log n)$ for insertion, deletion, and priority change operations.
- $\Theta(n)$ for building a priority queue from an unsorted collection

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

```
function QuickSort(A[lo..hi])
  if lo < hi then
    s := Partition(A[lo..hi])
    QuickSort(A[lo..s-1])
    QuickSort(A[s+1..hi]
```



p = A[s] is called the ***pivot*** of this partitioning

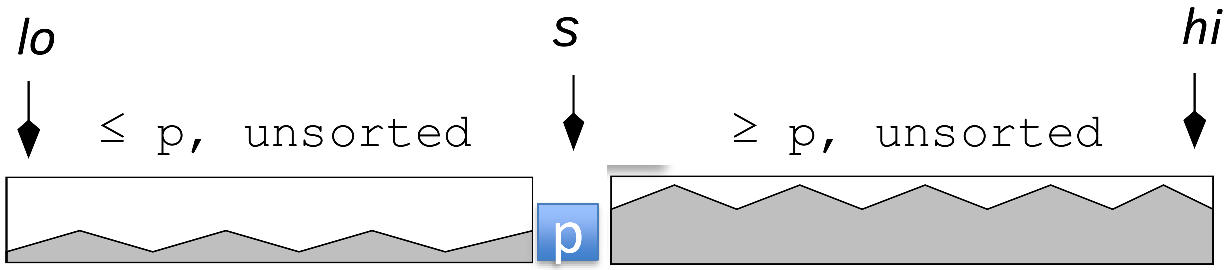
Note: a Partition of n elements has the complexity of $\theta(n)$

Questions:

- What is the (additional) space complexity of Quicksort?
- What is the time complexity?
- Is it input-sensitive?
- Is it in-place?
- Is it stable?

Quicksort Properties - Check your answers

```
function QuickSort(A[lo..hi])
  if lo < hi then
    s := Partition(A[lo..hi])
    QuickSort(A[lo..s-1]
    QuickSort(A[s+1..hi]
```



Quicksort complexity depends on the relative lengths of the left and the right parts in partitioning.

BEST: always balanced: $\theta(n \log n)$

n

2 x (n/2)

4 x (n/4)

$\log_2 n$ levels

AVERAGE: $\boldsymbol{\theta(n \log n)}$

WORST: one half always empty: $\theta(n^2)$

0:n-1

0:n-2

0:n-3

0:n-4

...

n-1 levels

What is the (additional) space complexity of **Quicksort**? for ecursion:
BEST/AVERAGE O(log n) WORST O(n)

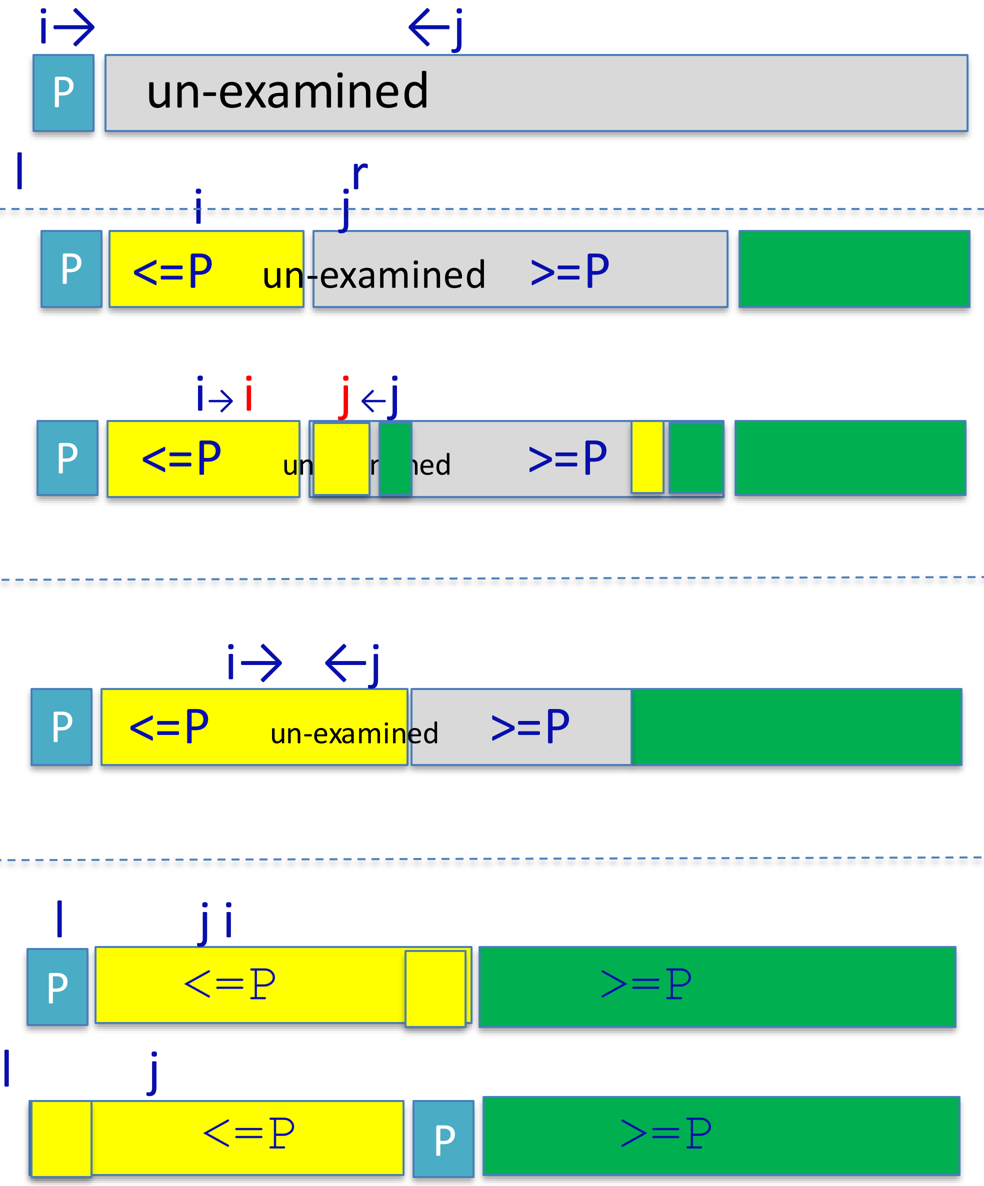
- Is it input-sensitive? Y
- Is it in-place? Y
- Is it stable? N – but we need to check with Partitioning

loop invariant	l i j h A[l+1..i-1]≤P A[i..j] un-examined A[j+1..h]≥P

```
function Partition(A[l..h])
 i ← l; j ← h
 P ← A[l]          # loop init
   repeat
      # move i forward until A[i]>P
      while i<h and A[i]≤P do i ← i+1
      # move j backward until A[j]<=P
      while A[j]>P do j ← j-1

      # extend yellow and green area
      #   at the same time by swapping ----→
    if (i<j) then Swap(A[i], A[j])
until  i ≥ j

# at loop's exit: i and j crossed
Swap(A[l], A[j])
return j
`
```



Note	This slide shows that the algorithm presented here and the one in the lectures are basically the same. They are just one variation of implementation of Hoare's Partitioning

```
function Partition(A[l..h])

 P ← A[l]; i ← l; j ← h

   repeat
      # move i forward until A[i]>P
      while i<h and A[i]≤P do i ← i+1
      # move j backward until A[j]<=P
      while A[j]>P do j ← j-1
      # extend yellow and green area
    if (i<j) then Swap(A[i], A[j])
 until  i ≥ j

  # at loop's exit: i and j crossed
 Swap(A[l], A[j])

 return j
`
```

function $\text{Partition}(A[lo..hi])$

$p \leftarrow A[lo];\ i \leftarrow lo;\ j \leftarrow hi$

repeat

while $i < hi$ **and** $A[i] \leq p$ **do** $i \leftarrow i + 1$

while $j \geq lo$ **and** $A[j] > p$ **do** $j \leftarrow j - 1$

$swap(A[i], A[j])$

until $i \geq j$

$swap(A[i], A[j])$ — undo the last swap

$swap(A[lo], A[j])$ — bring pivot to its correct

return j

end function

Example: Run quicksort with Hoare's for `[ 2 4 1 3]`

```
function HoarePartition(A[l..h]) \

 P ← A[l]; i ← l+1; j ← h

   repeat
      # move i forward until A[i]>P
      while i<h and A[i]≤P do i ← i+1
      # move j backward until A[j]<=P
      while A[j]>P do j ← j-1
      # extend yellow and green area
     if (i<j) then Swap(A[i], A[j])
 until  i ≥ j

   # at loop's exit: i and j crossed
 Swap(A[l], A[j])

 return j
`
```

Start with:

$2_i \; 4 \; 1 \; 3_j$

To simplify, we will write out the sequence only:

- at the beginning and end of the loop, and
- after a swap.

Question 7.1: Group Work

Know how to run by hand the following algorithms:

a) Selection Sort
b) Merge Sort
c) Quick Sort with Hoare's Partitioning

For each algorithm: Run the algorithm on the array:

- [A N A L Y S I S]

Question 9.1

For each sorting algorithm:

i. Run the algorithm on the following input array: [A N A L Y S I S]

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?

- skip stuffs that, after a short discussion, your group agrees that it's easy!
- be careful with letter ordering, perhaps write down:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Sorting Algorithms

	Insertion Sort	Selection Sort	Heap Sort	Quick Sort	Merge Sort
Basic Idea					
Complexity					
Expected Complexity					
In-place?					
Stable?					
Input-Sensitive?					
The Good					
The Bad					

Work with Your Classmates

As part of (extended) Question 7.2:

- fill in the table,
- imagine about scenarios when each sorting method would be the choice (e.g. better than other), that is
- be able to choose the best sorting algorithm for a particular application.

Also... Mergesort

Make sure you can run (by hand) Merge Sort

`[4 1 3 5 2]`

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

And, review the lectures for the remaining questions of problem 1. For each sorting algorithm, think:
which is the best situations when we want to employ that algorithm?
in which situations when we definitely don't want that algorithm?

Group Work on Algorithm Design: The k-th smallest problem

Problem: Given an array, find its k-th smallest value.

Task: Design and compare the complexity of multiple (say, 5) different algorithms.

A reasonable plan:

- Discuss and briefly describe algorithms with complexity analysis in class.
- Choose two distinct optimal methods for pseudocode implementation in class.
- Complete remaining work at home, if desired.

Note: Do Not read questions 7.5 and 7.6. This discussion covers these questions anyway.

Q7.4: Closest-pair and element-distinction

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 then the element distinction 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

```
function ElemDistinction(?)
```

- explain why this proves that the closest pair problem must not be able to be solved in less that nlogn time (and is thus $\Omega(n \log n)$).

```
?
```

LAB

- Play with some sorting algorithm, especially different choices of pivot for qsort
- Discuss: What’s the best method of choosing qsort?
- Using the provided code, understand how to do timing in C

Additional Slides

CHECK: qsort with Hoare's Partitioning

Q9.1: Run quicksort with Hoare's for `[ A N A L Y S I S].`

Partitioning of
`A N A L Y S I S`

A_i N A L Y S I S_j
A N_i A_j L Y S I S
A A_j N_i L Y S I S

[A] A_j [N L Y S I S]

Partitioning of
`N L Y S I S`

N_i L Y S I S_j
N L I_j S_i Y S
N L I_j S_i Y S

[I L] N_j [S Y S]

Partitioning of
`I L`

I_i L_j
I_j L_i

I_j [L]

Partitioning of
`S Y S`

S_i Y S_j
S Y_i S_j
S S_j Y_i

[S] S_j [Y]

Final sorted sequence
`A I L N S S Y`