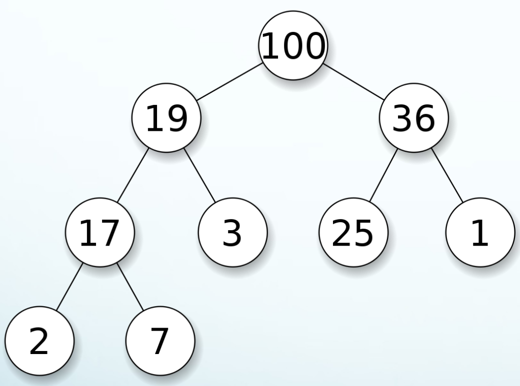
COMP20007 Workshop Week 8

	Preparation: - use the powerpoint or PDF slides for note taking if you like, but - ***have papers and pens (or drawing tools) ready***, *and/or* - *ready to work on whiteboards* - ***open Ed.Week 8 Workshop***
1	Binary Heap: Operations, Heapsort, Questions 8.2, 8.3
2	Sorting Algorithms - quicksort: Question 8.1
	Quickselect (Question 8.4)
LAB	Lab: Sorting algs, follow the given instructions

A Priority Queue: Binary Heap = ?

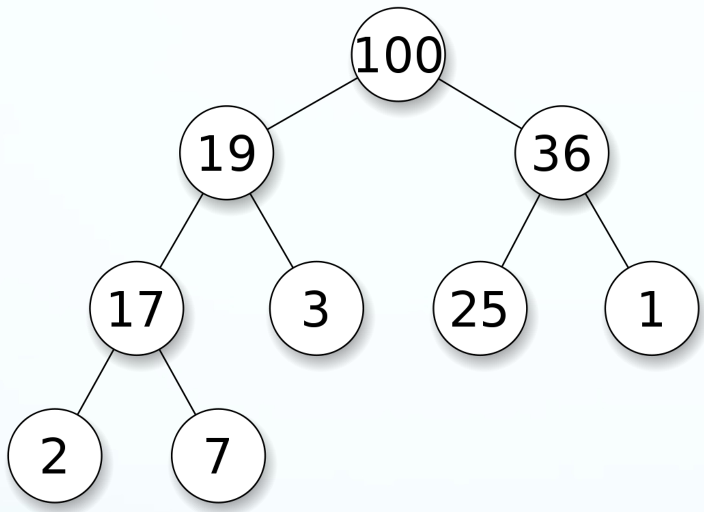
Binary Heap as a concrete data type (implementation) for PQ.
min heap, max heap = ?

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



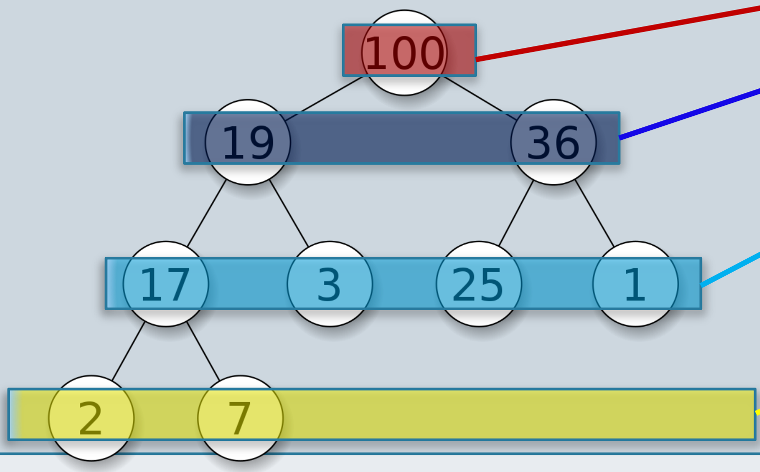
‘Can I borrow your baby?...’

Binary Heap: conceptually, is a binary tree

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

Heap is `H[1..n]`

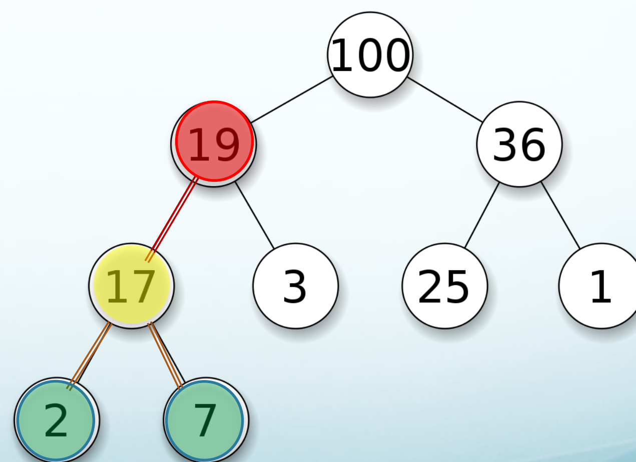
- level i occupies 2^i cells in array `H[1..n]` (except for the last level)
- if root is level 1, then level i starts from `H[i]`

Binary Heap: parent and children of a node

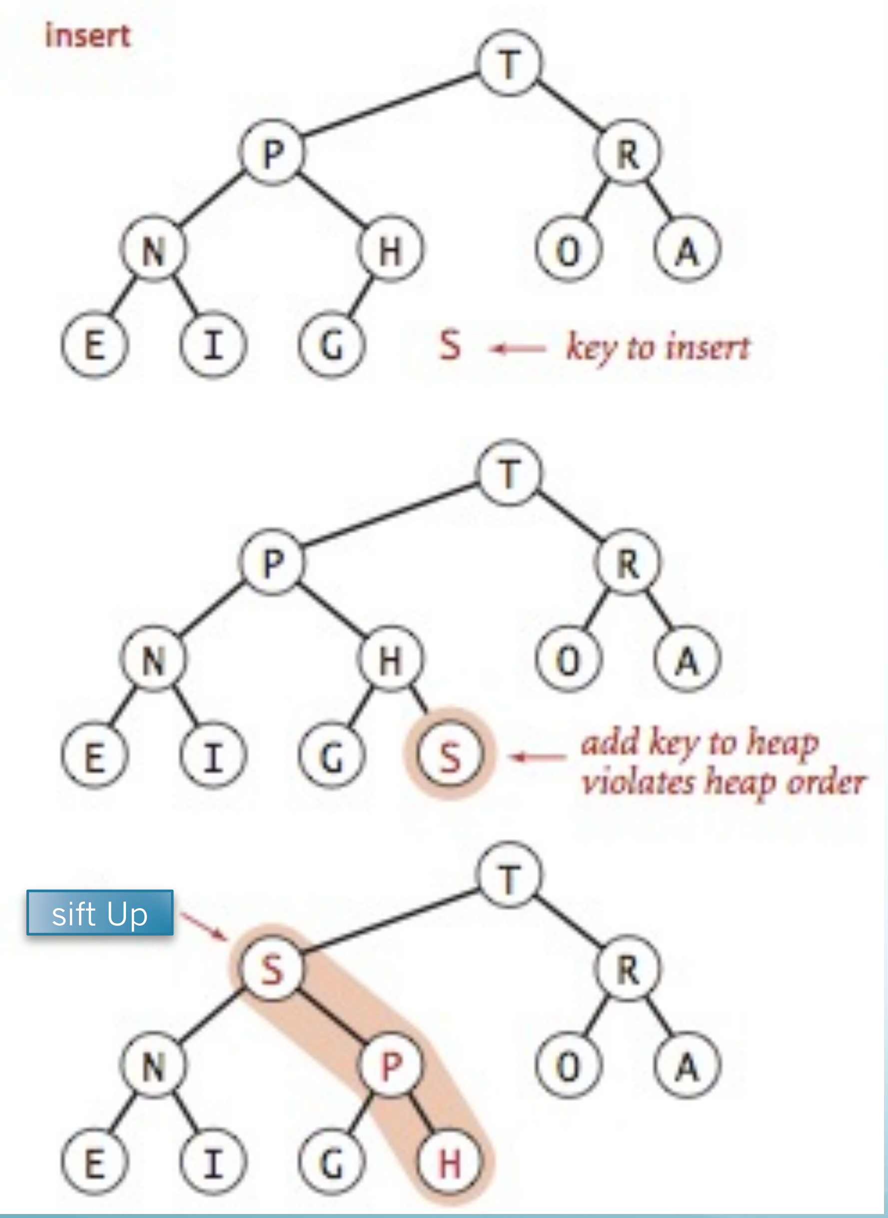
- left child of `H[i]` is `H[2*i]`
- right child of `H[i]` is `H[2*i+1]`

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



inject = enPQ = Insert a new elem into a heap

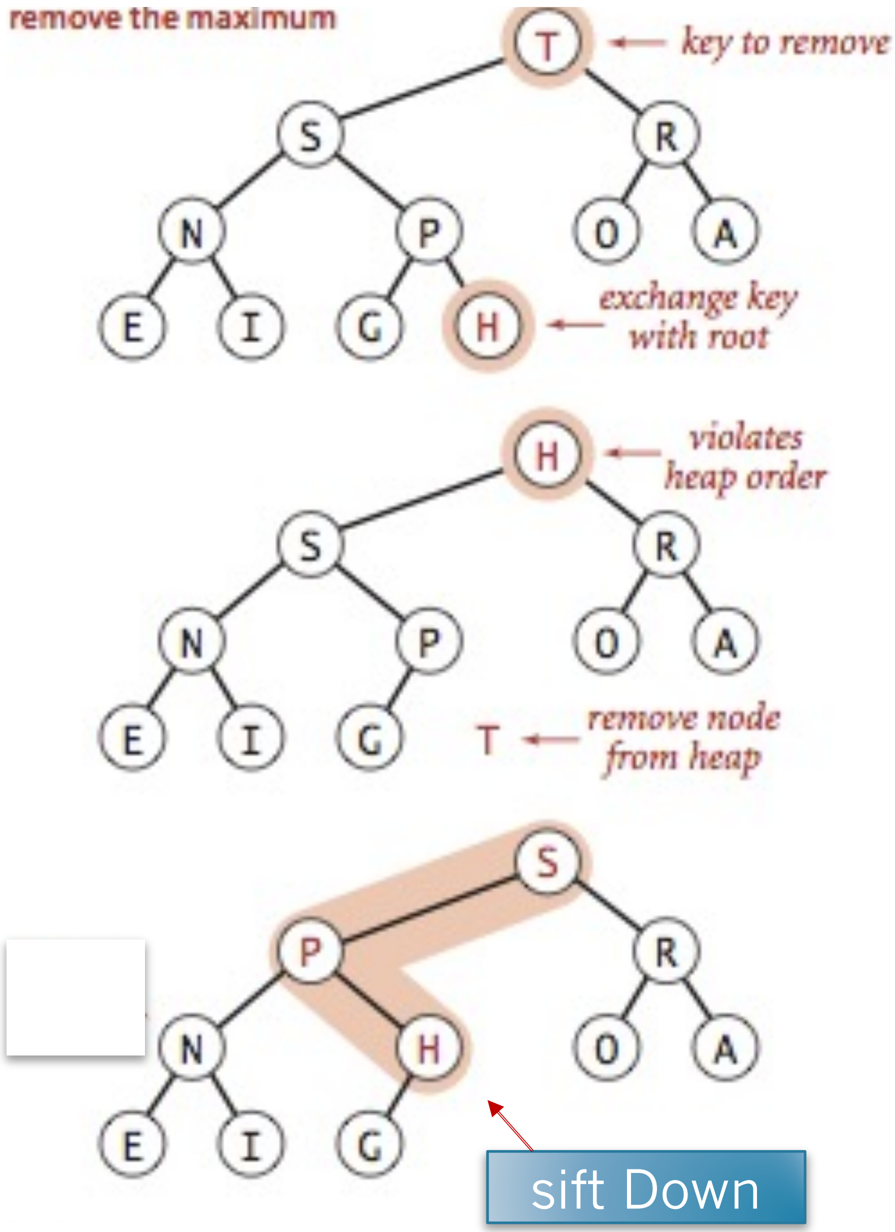


Notes:

Sift Up

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

eject = delete (and returns) the heaviest



Notes:

`Sift Down`:

while (has children and the heavier child has higher priority): swap down with the *heavier* child

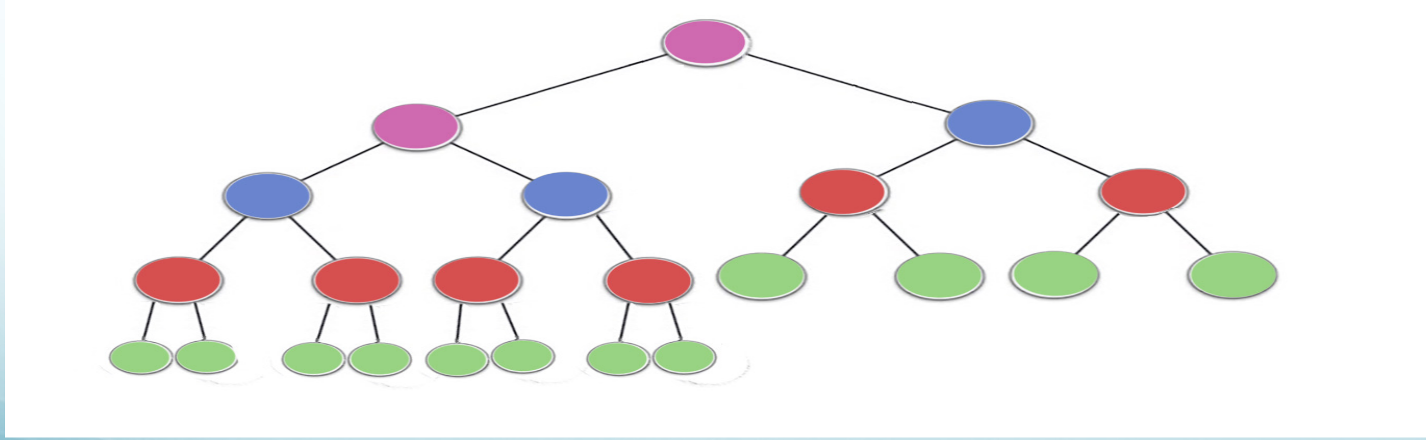
Heapify: Turning an array H[1..n] into a heap

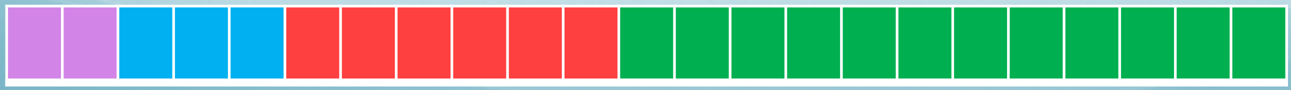
```
function Heapify(H[1..n])
  for i ← n/2 downto 1 do
     downheap(H, i)
```

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

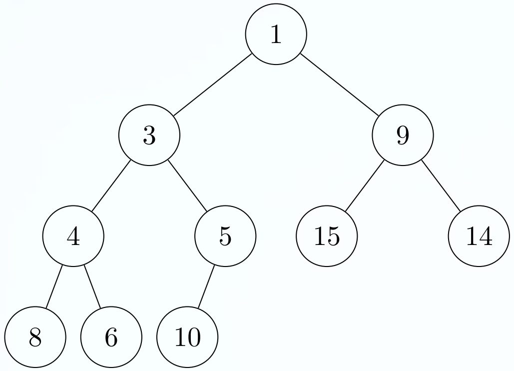
The operation is aka. **Heapify**/Makeheap/ Bottom-Up Heap Construction

Example: build maxheap for keys `E X A M P`





Notes:



Q 8.2

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 (i.e., swap the root and the "last"element and remove it. To maintain the heap property we then SiftDown from the root).

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):

NOTES ON HEAPSORT:

Q 8.3 [opt]: k-smallest using min-heap

- The *k-th smallest* problem:
  - Given an array `A[]` of `n` elements, and an integer `k`
  - Find the `k`-th smallest value (suppose that `k` is zero-origin, that is, `k` can be any of 0, 1, 2, ..., n-1)

- How can we use a min-heap data structure to solve the kth-smallest element problem? What is the time-complexity of this algorithm?

Your answer:

Algorithm	Complexity
`function HeapkthSmallest(A[0..n-1],k)`	

Basic Sorting Algorithms

Know how to run by hand the following algorithms:

- Selection Sort
- Insertion Sort
- Quick Sort with Lomuto's Partitioning
- Quick Sort with Hoare's Partitioning
- Merge Sort
- Heap Sort

Run example with keys:

E X A M P

LAB: follow instructions in workshop sheet

Question 8.1

We saw the following sorting algorithms,

(a) Selection Sort

(b) Insertion Sort

(c) Quicksort (with Lomuto partitioning)

For each algorithm:

(i) Run the algorithm on the array:

[A N A L Y S I S]

(ii) time complexity of the algorithm=?

(iii) Is the sorting algorithm stable?

(iv) Does the algorithm sort in-place?

(v) Is the algorithm input sensitive?

(vi) What is the strongest point of the algorithm (when should it be used)?

If you get time, try to answer these questions for

(d) Quicksort (with Hoare partitioning), and

(e) Merge Sort

(f) Heap Sort

Quicksort for A[l..r]

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

```
function quicksort(A[l..r])
```

`if l >= r then return`

`s ←` do partitioning A[l..r], ie: # Lomuto or Hoare or ...

rearrange `A[l..r]` into `A[l..s-1]` `A[s]` `A[s+1..r]` so that

where ***pivot*** `P` is any element in `A`, we always take `P= A[l]`

if we want some `A[i]` be the pivot: just swap with `A[l]` in advance

```
quicksort(A[l..s-1))
quicksort(A[s+1..r)
```

Notes:

- the left `A[l..s-1]` or the right array `A[s+1..r]` could be empty
- we will show that the partitioning is **θ**(n)
- qsort complexity depends on the relative lengths of the left and the right
  - Best case: they always have about the same length → **θ** (n logn)
  - Worst case: one of them always empty → O(n^2)

```
function LOMUTOPARTITION(A[l..r])
```

```
# loop init:
  P ← A[l]
  s ← l,  i ← l+1
# loop invariant:
# A[l+1..s]<P    A[s+1..i-1]≥P
# A[i..r] not yet examined

  while i ≤ r do
    # move i++ until A[i]<P
    if A[i] < P then

      #extend the yellow area
      s ← s + 1
      SWAP(A[s], A[i])

    i ← i+1
  # at loop exit (i=r+1) ----------------→
  # A[s] is the last yellow (OR s=l)


  SWAP(A[l], A[s])
  return s
```

function LOMUTOPARTITION$(A[l..r])$

$p \leftarrow A[l]$

$s \leftarrow l$

for $i \leftarrow l + 1$ **to** r **do**

  if $A[i] < p$ **then**

    $s \leftarrow s + 1$

    SWAP$(A[s], A[i])$

SWAP$(A[l], A[s])$

return s

l i r

function LomutoPartition(A[l..r])

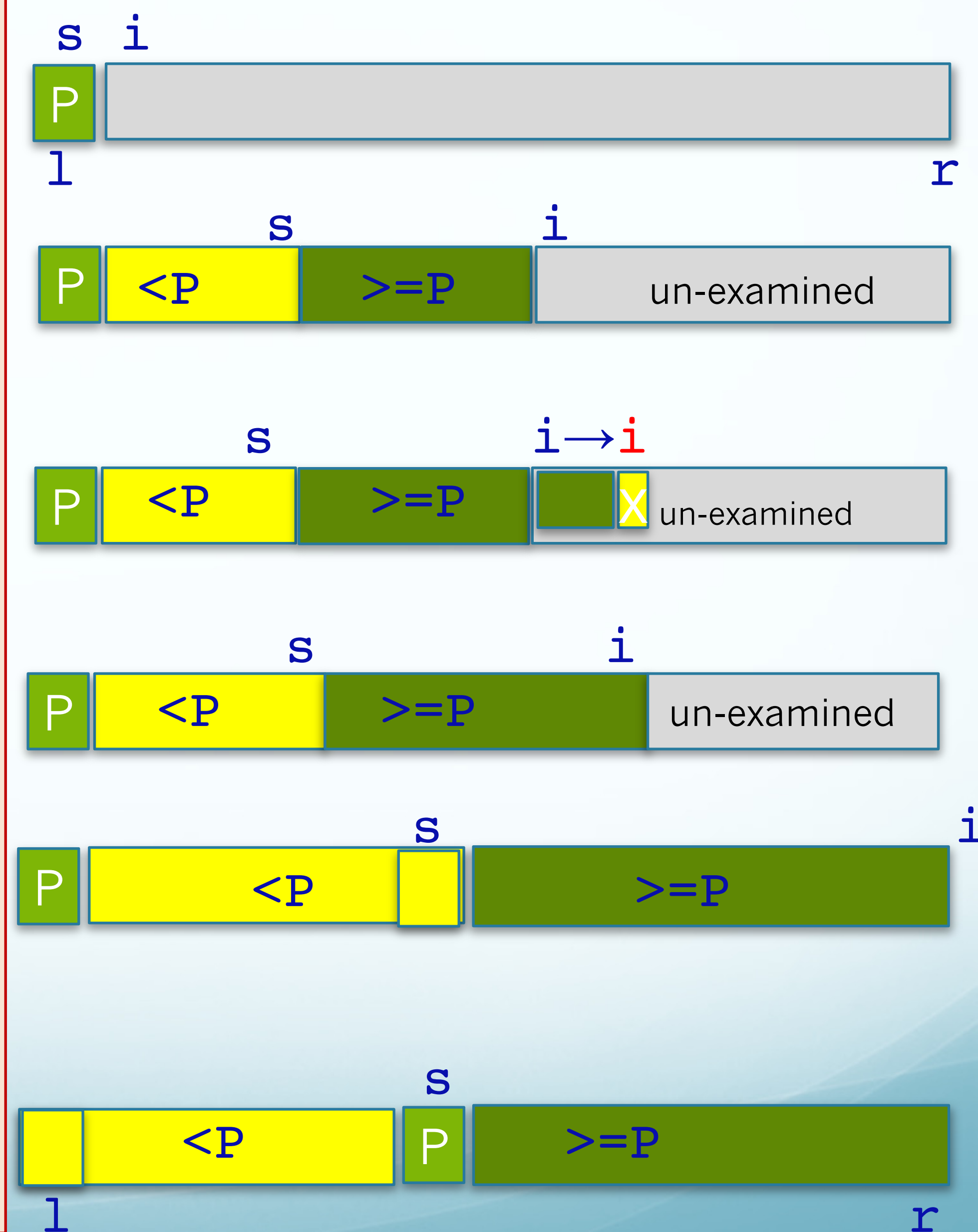
```
# loop init:
  P ← A[l]
  s ← l,  i ← l+1
# loop invariant:
# A[l+1..s]<P     A[s+1..i-1]≥P
# A[i..r] not yet examined

  while i ≤ r do
    # move i++ until A[i]<P
    if A[i] < P then

      #extend the yellow area
      s ← s + 1
      Swap(A[s], A[i])

    i ← i+1
  # at loop exit (i=r+1) ----------------→
  # A[s] is the last yellow (OR s=l)


  Swap(A[l], A[s])
  return s
```



Example: Run quicksort with Lomuto's for `[ E X A M P]`

```
# loop init:
  P ← A[l]
  s ← l,  i ← l+1
# loop invariant:
# A[l+1..s]<P    A[s+1..i-1]≥P
# A[i..r] not yet examined

  while i ≤ r do
     # move i++ until A[i]<P
     if A[i] < P then

        #extend the yellow area
        s ← s + 1
        SWAP(A[s], A[i])

     i ← i+1
  # at loop exit (i=r+1) ----------------→
  # A[s] is the last yellow (OR s=l)


  SWAP(A[l], A[s])
  return s
```

Partitioning `[ E X A M P]`

E_s X_i A M P

E_s X A_i M P

E A_s X_i M P

E A_s X M P $_i$

[A] E_s [X M P]

Partitioning `[ X M P]`

X_s M_i P

X M_{si} P

X M_s P_i

X M P_{si}

X M P_s $_i$

[P M] X_s

Partitioning `[ P M]`

P_s M_i

P M_{si}

P M_s $_i$

[M] P_s

To simplify, we will write out the sequence only: at the beginning and end of the loop, and after a swap.

The algorithm is quite slow on small arrays!

Final sorted: `[ A E M P X]`

qsort with Lomuto's Partitioning

Q8.1c): Run quicksort with Lomuto's for `[ A N A L Y S I S].` Here we rewrite the sequence after each swap and at the start/end of the loop.

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

A_s N_i A L Y S I S
???

2. Partitioning of
???

???

3. Partitioning of
???

???

4. Partitioning of
???

???

5. Partitioning of
???

???

Final sorted sequence
???

CHECK: qsort with Lomuto's Partitioning

Q8.1c): Run quicksort with Lomuto's for `[ A N A L Y S I S].` Here we rewrite the sequence after each swap and at the start/end of the loop.

Partitioning of
A N A L Y S I S

A_s N_i A L Y S I S
A_s N A L Y S I S $_i$
A_s [N A L Y S I S]

Partitioning of
N A L Y S I S

N_s A_i L Y S I S
N A_{si} L Y S I S
N A L_{si} Y S I S
N A L I_s S Y_i S
N A L I_s S Y S $_i$

[I A L] N_s [S Y S]

Partitioning of
I A L

I_s A_i L
I A_{si} L
I A_s L $_i$

[A] I_s [L]

Partitioning of
S Y S

S_s Y_i S
S_s Y S $_i$

S_s [Y S]

Partitioning of
Y S

Y_s S_i
Y S_{si}
Y S_s $_i$

[S] Y_s

Final sorted sequence
A I L N S S Y

```
function HOAREPARTITION(A[l..r])
```

```
# loop init:
P ← A[l]
i ← l; j ← r + 1
# loop invariant:
# A[l+1..i]≤P     A[j..r]≥P
# A[i+1..j-1]  not yet examined
repeat
   # move i forward until A[i]>=P
   repeat i←i+1 until A[i] ≥ P
   #  same as do i←i+1 while A[i]<P
   # move j backward until A[j]<=P
   repeat j←j-1 until A[j] ≤ P

   # 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:



SWAP(A[l], A[j])
return j
```

function HOAREPARTITION$(A[l..r])$

$p \leftarrow A[l]$

$i \leftarrow l; j \leftarrow r + 1$

repeat

repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$

repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$

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

until $i \geq j$

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

SWAP$(A[l], A[j])$

return j

function HOAREPARTITION(A[l..r])

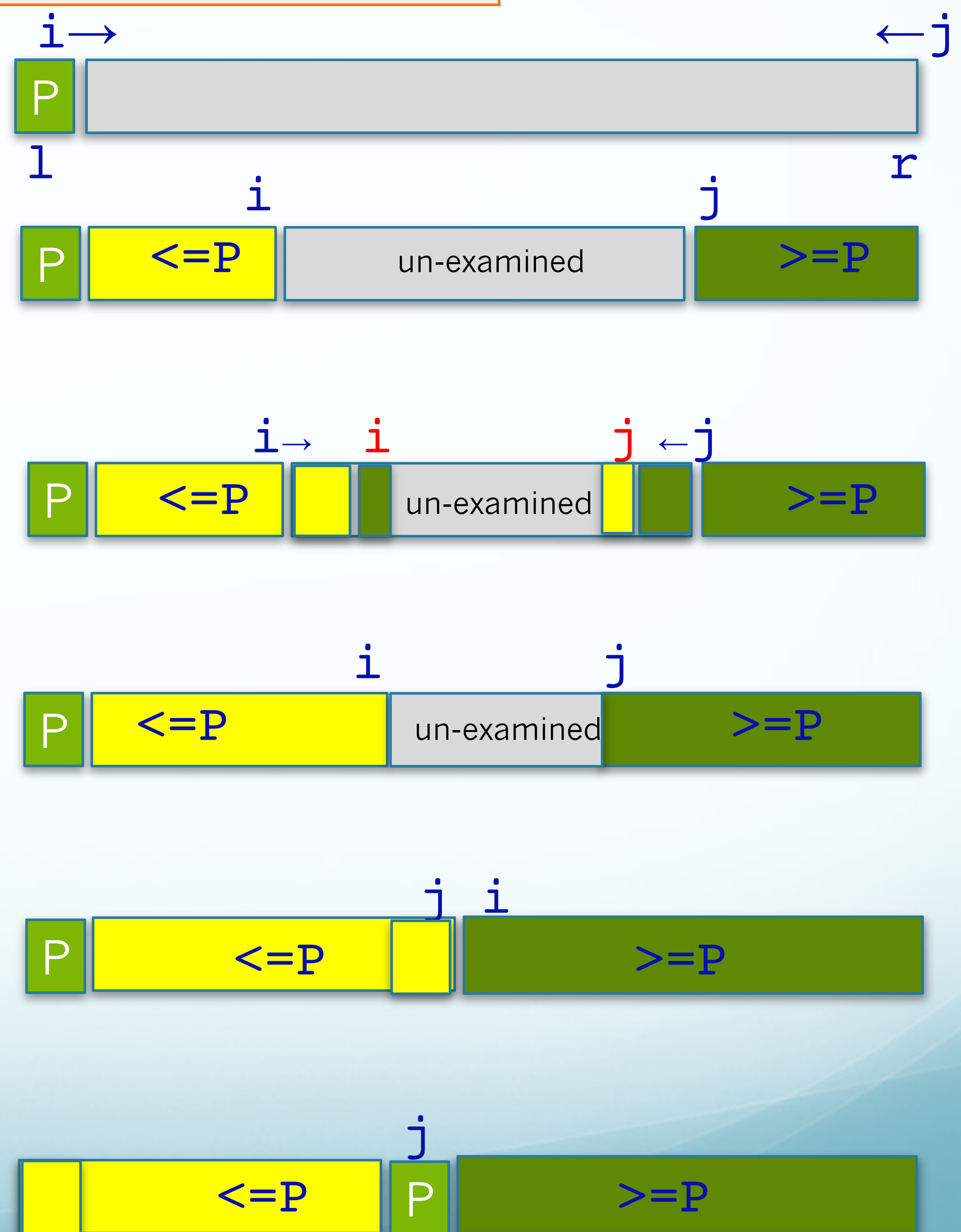
```
# loop init:
P ← A[l]
i ← l; j ← r + 1
# loop invariant:
#  A[l+1..i]≤P     A[j..r]≥P
#  A[i+1..j-1]  not yet examined
repeat
   # move i forward until A[i]>=P
   repeat i←i+1 until A[i] ≥ P
   #  same as do i←i+1 while A[i]<P
   # move j backward until A[j]<=P
   repeat j←j-1 until A[j] ≤ P

   # 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:



SWAP(A[l], A[j])
return j
```



Example: Run quicksort with Hoare's for [E X A M P]

```
# loop init:
P ← A[l]
i ← l; j ← r + 1
# loop invariant:
#  A[l+1..i]≤P      A[j..r]≥P
#  A[i+1..j-1]  not yet examined
repeat
   # move i forward until A[i]>=P
   repeat i←i+1 until A[i] ≥ P
   #  same as do i←i+1 while A[i]<P
   # move j backward until A[j]<=P
   repeat j←j-1 until A[j] ≤ P

   # 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:



SWAP(A[l], A[j])
return j
```

Partitioning [E X A M P]

E_i X A M P $_j$

E X_i A_j M P

E A_i X_j M P

E A_j X_i M P

[A] E_j [X M P]

Partitioning [X M P]

X_i M P $_j$

X M P_j i

[M P] X_j

Partitioning [M P]

M_i P $_j$

M_j P_i

M_j [P]

To simplify, we will write out the sequence only: at the beginning and end of the loop, and after a swap.

The algorithm is still slow on small arrays!

Final sorted: [A E M P X]

qsort with Hoare's Partitioning

Q8.1d): Run quicksort with Hoare's for `[ A N A L Y S I S].` Here we rewrite the sequence after each swap and at the start/end of the loop.

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

A_i `N A L Y S I S` $_j$
`???`

2. Partitioning of
`???`

`???`

3. Partitioning of
`???`

`???`

4. Partitioning of
`???`

`???`

Final sorted sequence
`???`

CHECK: qsort with Hoare's Partitioning

Q8.1d): Run quicksort with Hoare's for `[ A N A L Y S I S].` Here we rewrite the sequence after each swap and at the start/end of the loop.

Partitioning of
A N A L Y S I S

A_i N A L Y S I S $_j$
A A_i N_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_i S Y_j 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 S_i Y_j
S S_{ij} Y_j

[S] S_j [Y]

Final sorted sequence
A I L N S S Y

Also

Make sure you can run (by hand) Merge Sort and HeapSort for

- `[E X A M P]`
- `[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?

Q 7.4 [opt]

a) Design an algorithm Quickselect based on Quicksort which uses the `Partition` algorithm to find the `k`-th smallest element in an array `A`.

b) Show how you can run your algorithm to find the `k`-th smallest element where `k = 4` and `A = [9,3,2,15,10,29,7]`.

c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

e) What is the expected-case (i.e., average) time-complexity of your algorithm?

f) When would we use this algorithm instead of the heap based algorithm from Question 3

g)

partitionning & qselect

```
partition( A[lo..hi]):
   ...
   return m
```

```
function qselect( A[lo..hi], k)
  m= partition(A[lo..hi])
```

```
function ksmallest(A[0..n-1], k)
  if (k>=0 && k<n)
    return qselect(A[0..n-1], k)
```

Check: partitionning & qselect

```
partition( A[lo..hi]):
   ...
   return m

function qselect( A[lo..hi], k)
   m= partition(A[lo..hi])
   if (k==m) then return A[m]
   if (k<m) then
      qselect(A[lo..m-1], k)
   else
      qselect(A[m+1..hi], k)

function ksmallest(A[0..n-1], k)
   if (k>=0 && k<n)
      return qselect(A[0..n-1], k)
```

Problem 4

a) Design an algorithm based on Quicksort which uses the Partition algorithm to find the `k`-th smallest element in an array `A`.

b) Show how you can run your algorithm to find the `k`-th smallest element where `k = 4` and `A = {9,3,2,15,10,29,7}`.

c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

e) What is the expected-case (i.e., average) time-complexity of your algorithm?

LAB