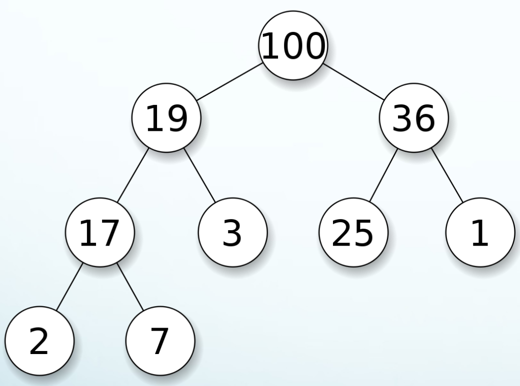
COMP20007 Workshop Week 8

	Preparation:
	- open workshop8.pdf, download & unzip lab_files.pdf
	- *have draft papers and pen ready, and/or*
	- *ready to work on whiteboards*
1	Binary Heap: Operations, Heapsort, Problems 2, 3
2	Sorting Algorithms, and ... quicksort: Problem 1
	Quickselect (Problem 4)
LAB	Lab: Sorting algs, follow the given instructions

A Priority Queue: Binary Heap = ?

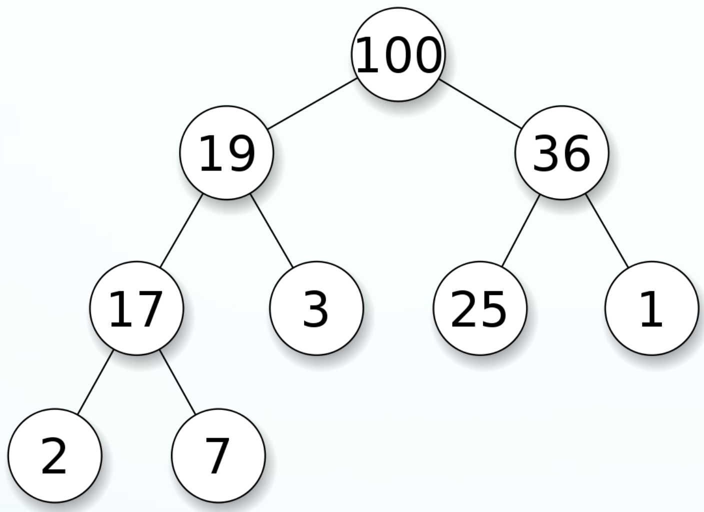
Binary Heap as a concrete data type (implementation) of PQ.
min heap, max heap = ?

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



'Can I borrow your baby?...'

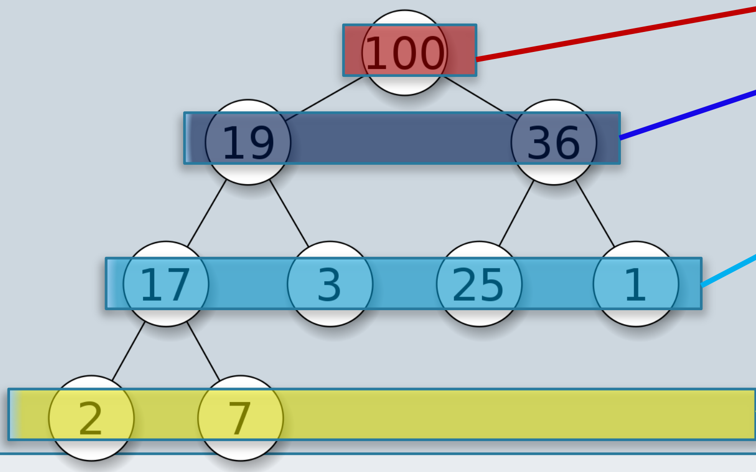
Binary Max/Min Heap

<table>
<tr><th>Example</th><th>Conditions</th></tr>
<tr><td>
</td><td>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).</td></tr>
</table>

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)

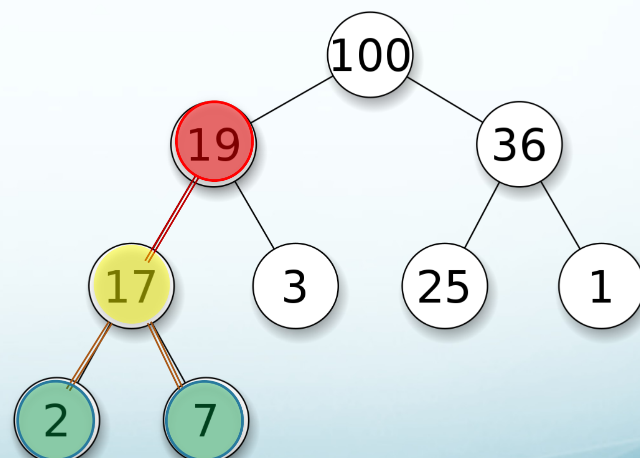
Binary Heap is implemented as an array!

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

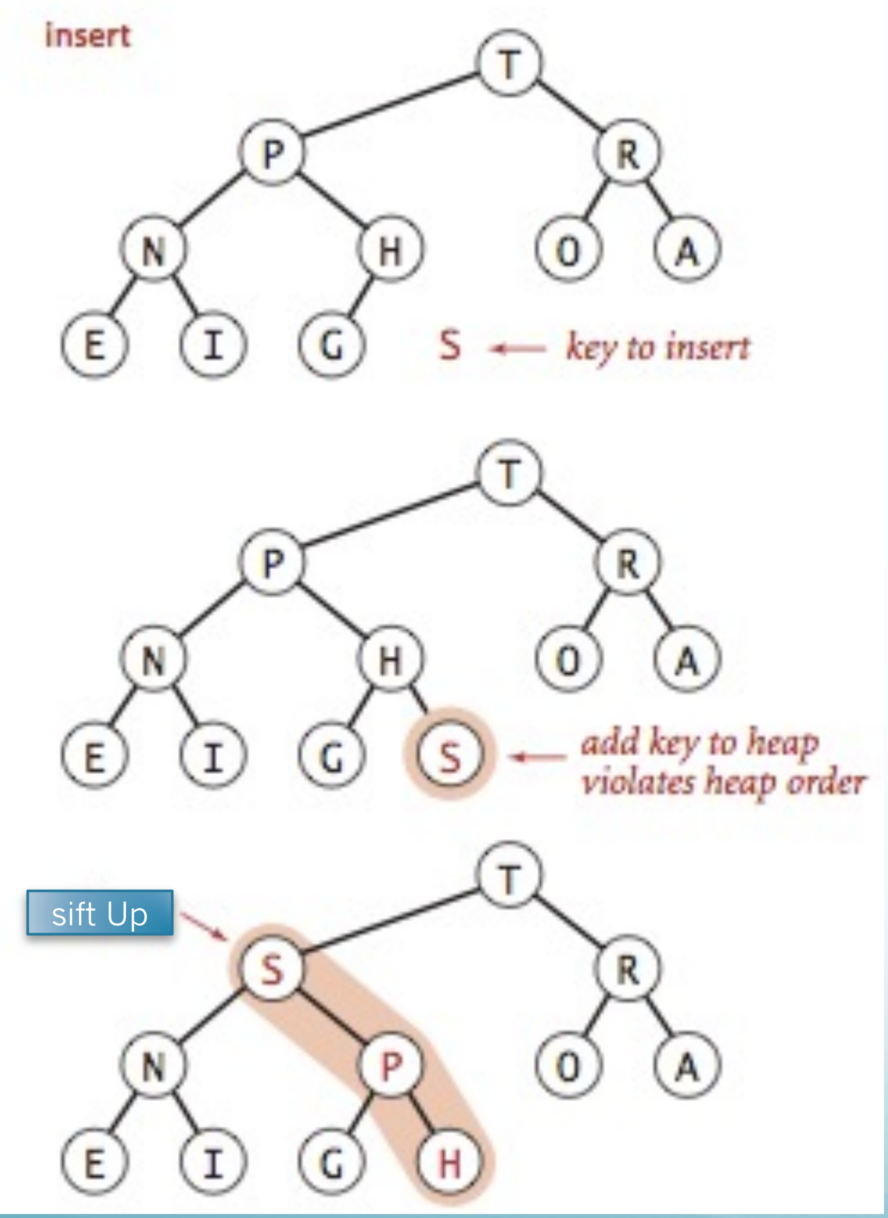
parent of `H[i]` is `H[i/2]`

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**



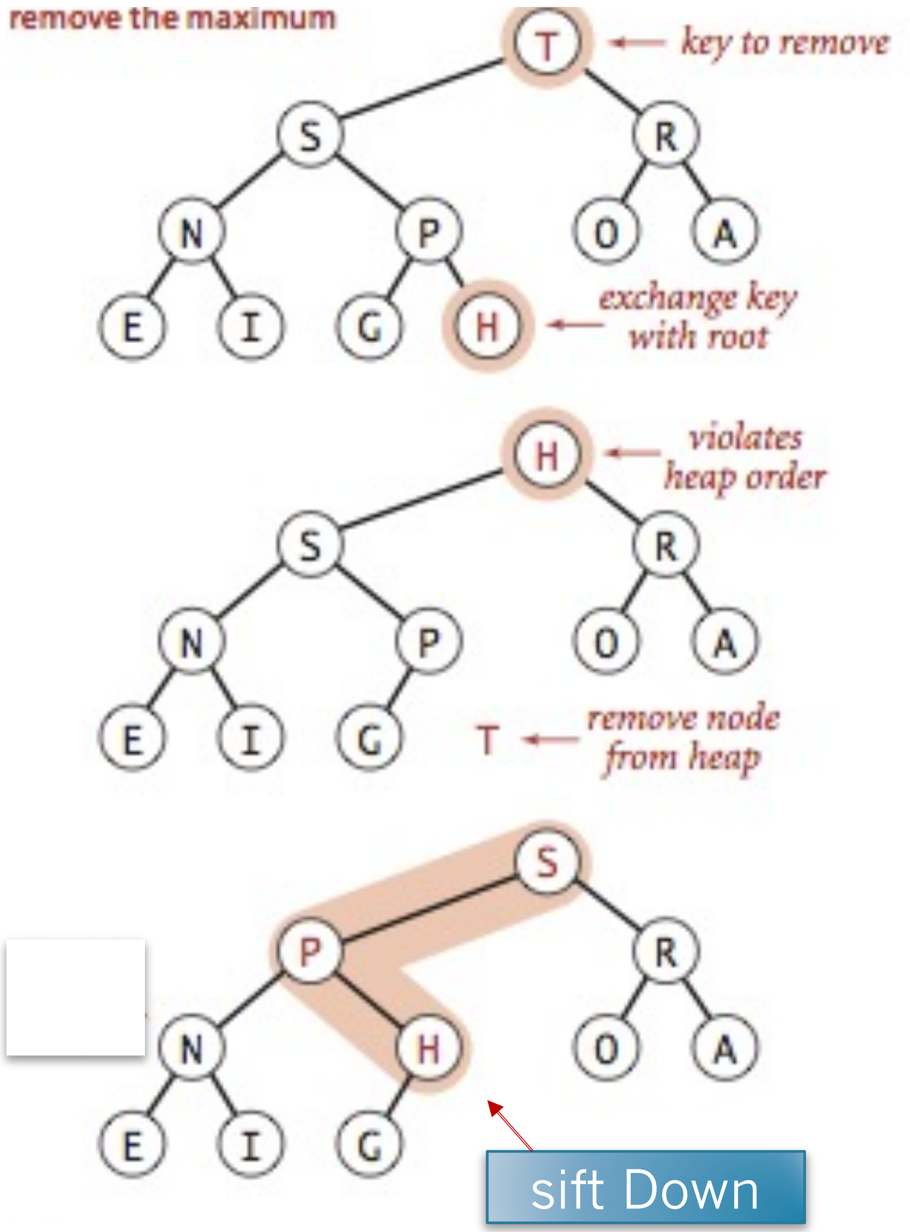
inject = Insert a new elem into a heap



Sift Up

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

eject = delete (and returns) the heaviest



`Sift Down`:

while (has children and at least one children has higher priority): swap down with the *highest-priority* child

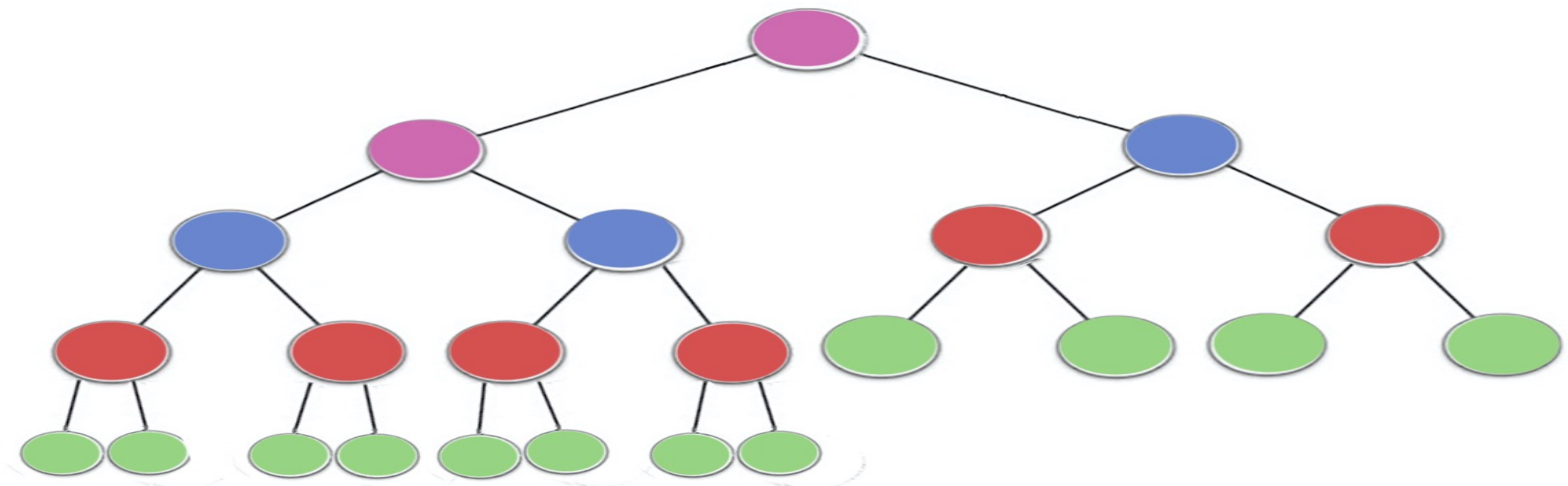
Heapify: Turning an array H[1..n] into a heap

```
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)

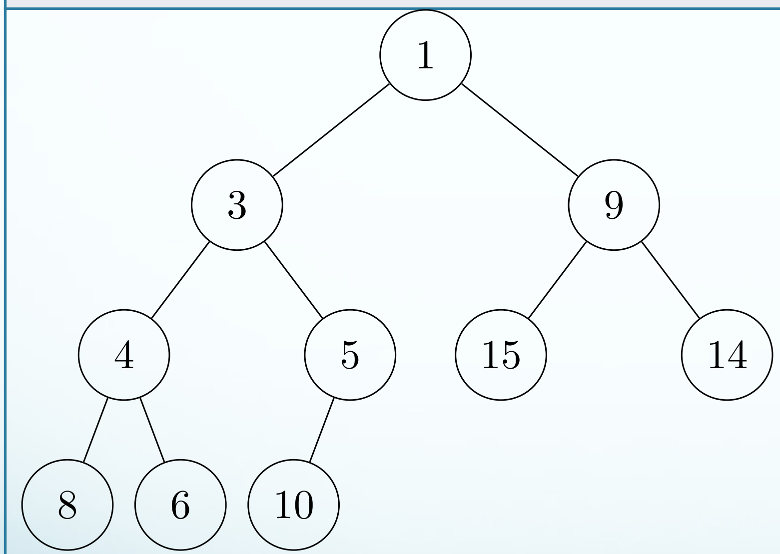
The operation is aka. **Heapify**/Makeheap/ Bottom-Up Heap Construction

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



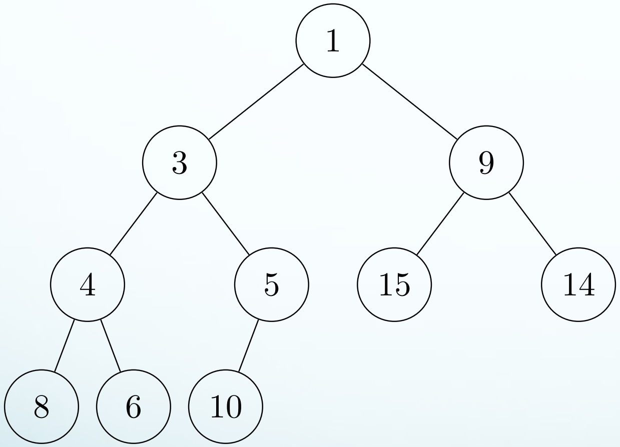
Problem 2: Binary Min Heap

2a) 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`)

Heap Visualisation	
	*2a)* *Array is:* [ ]

Problem 2b & 2c

- **2b)** 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).
- **2c)** Run the `InsertIntoHeap (inject)` algorithm and insert the value 2 into the heap



Problem 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)
- Using min-heap for the k-th smallest:
  - How
  - What the complexity?

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

Problem 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.

Quicksort for A[l..r]

```
function quicksort(A[l..r])
   if l >= r then return
   s ← do partitioning, ie:                 # Lomuto or Hoare or ...
      pivot= A[l]
      rearrange A[] into A[l..s-1] A[s] A[s+1..r] so that
      A[s]= pivot
      all elements in A[l..s-1] are <= pivot
      all elements in A[s+1..r] are >= pivot
   quicksort(A[l..s-1])
   quicksort(A[s+1..r])
```

Notes:

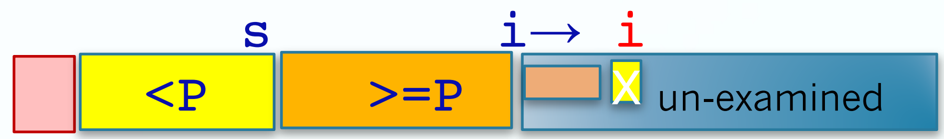
- We can choose any element in `A[l..r]` as `pivot`: just swap it with `A[l]` at the start of partitioning
- `A[l..s-1]` or `A[s+1..r]` could be empty
- Complexity the relative lengths of `A[l..s-1]` and `A[s+1..r]`

Lomuto's Partitioning

- Set P← A[l], and leave A[l] aside, then run a loop

Start:

During loop:

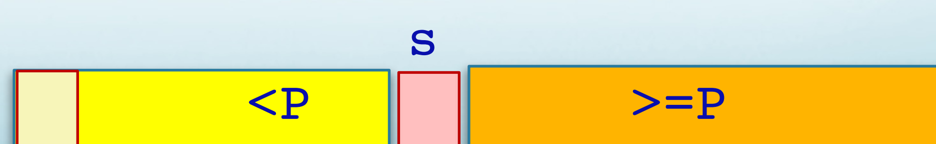


Loop iteration:

- move i forward to the first position i that A[i]<P
- if that exists (ie. i<=r):
    - advance s← s+1 (extending the yellow area)
    - swap (A[s], A[i])
    - continue the loop

At the end:

- Then, swap A[l] with the last yellow A[s], and return s



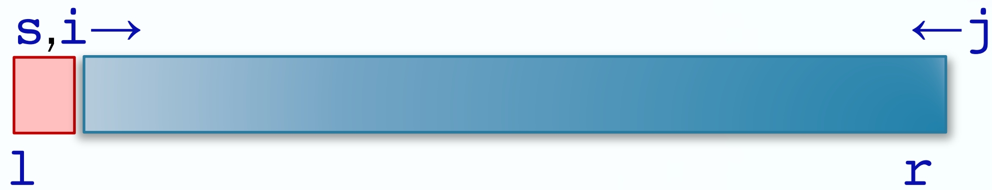
Lomuto's Partitioning

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

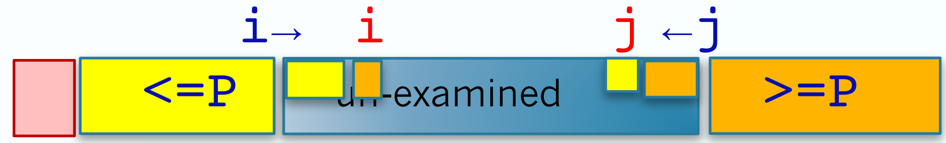
Hoare's Partitioning

- Likre Lomuto, set `P`← `A[l]`, and leave `A[l]` aside, then run a loop

Start:

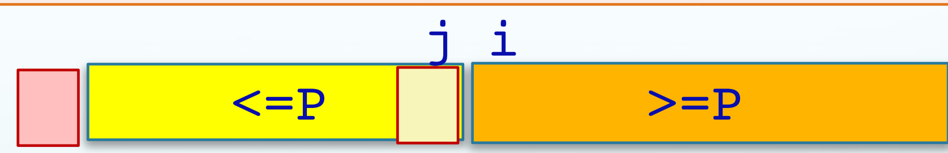


During loop:



- move `i` forward to the first position that `A[i]>=P`
- move `j` backward to the first position that `A[j]<=P`
- if I and j not crossed (ie. `i<j`):
  - `swap (A[i], A[j])`
  - hence, extended both yellow and orange area
  - continue the loop

At the end:



- Then, swap `A[l]` with the last yellow `A[j]`,and `return j`

Hoare's Partitioning

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

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?

Problem 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])
  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

Download `lab_files.zip` and follow the instructions in the workshop sheet of this week.