Software 2 (Theory)

Lecture 5: Sorting

365447

THE UNIVERSITY of York

Today's Topics

- Study some more sorting algorithms
- Understand the design principles behind them
- Look at the application of sorting to improving other algorithms

Selection sort

- Recall selection sort from lecture 3.
- Our analysis showed that its complexity was $O(n^2)$, poor for a sorting algorithm

Selection sort

```
SelectionSort(l : list)

for k=0…l.length-1
      i=IndexOfMin(l(k…end))
      swap(l(i),l(j))
end for
```

Selection sort

- Recall selection sort from lecture 3.
- Our analysis showed that its complexity was $O(n^2)$, poor for a sorting algorithm.
- In lecture 4, we looked at the heap data structure.
- These two operations were $O(\log n)$
- Note we can execute '**`IndexOfMin`**' for a heap in $O(\log n)$

Selection sort

```
SelectionSort(l : list)

for k=0…l.length-1
      i=IndexOfMin(l(k…end))
      swap(l(i),l(j))
end for
```

Heap Data Structure

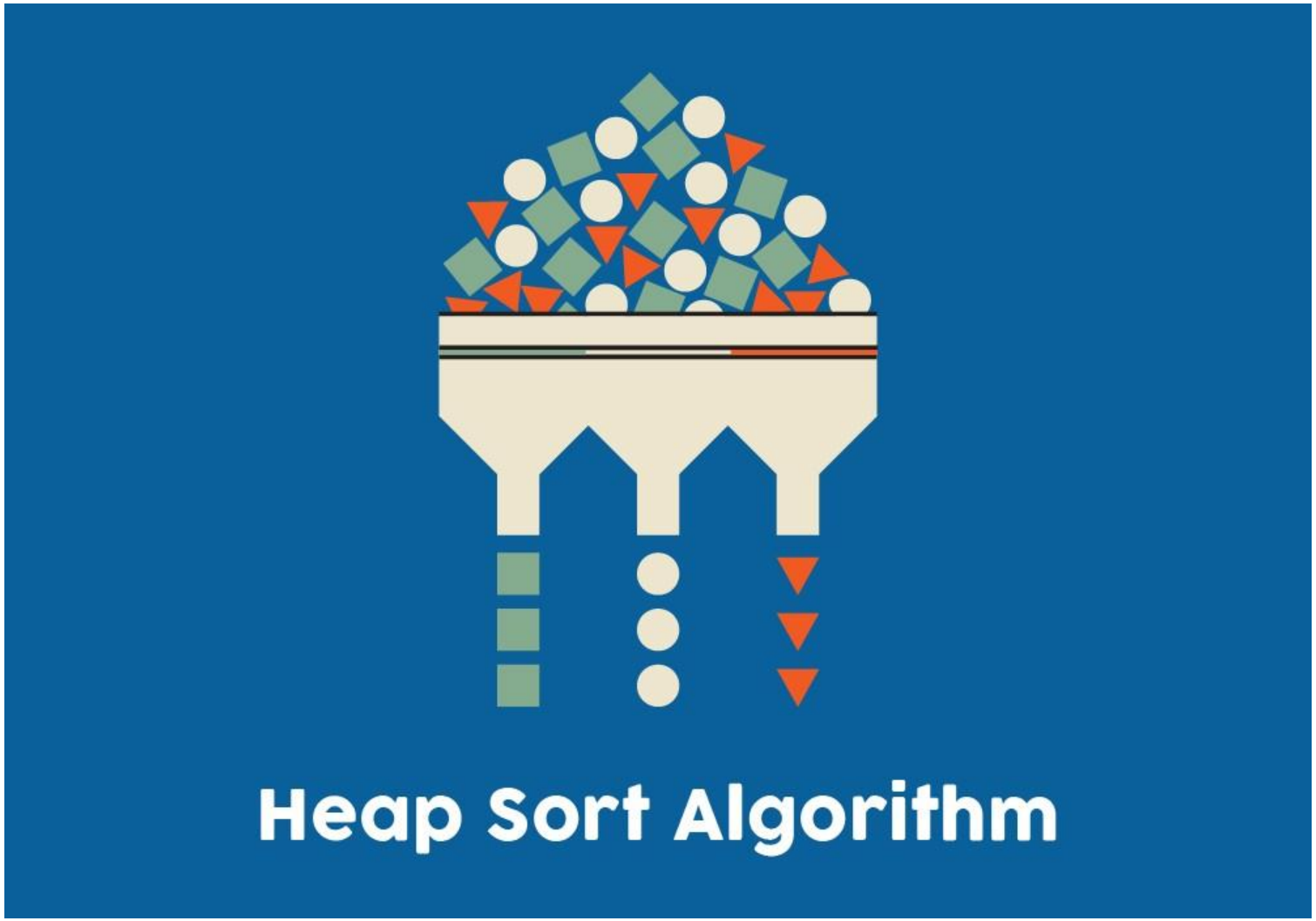
Organization

Binary Tree

Common operations

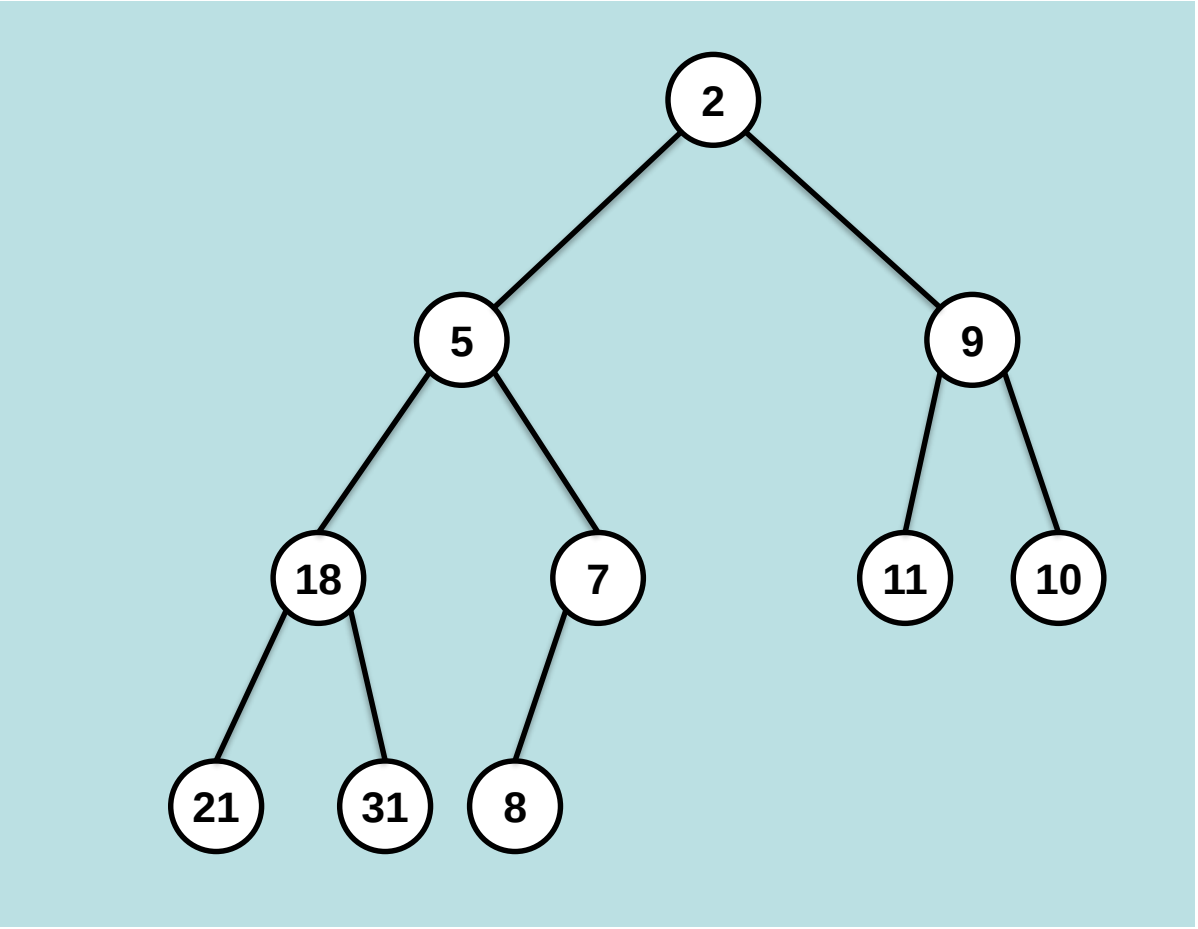
`Insert(v)`	Insert element v
`ExtractMin()`	Remove and return the minimum element

Heap Sort



Heaps Sort Algorithm

- Convert the array (list) into heap data structure.
  - Build a heap from the given input array (list).
- One-by-one, delete the root node of the heap.
  - Swap the root element with the last heap element
  - Restore the heap property
- The sorted array is obtained from the deleted root nodes of the Min-heap.
- Example: sort the list [12, 18, 11, 13, 9]



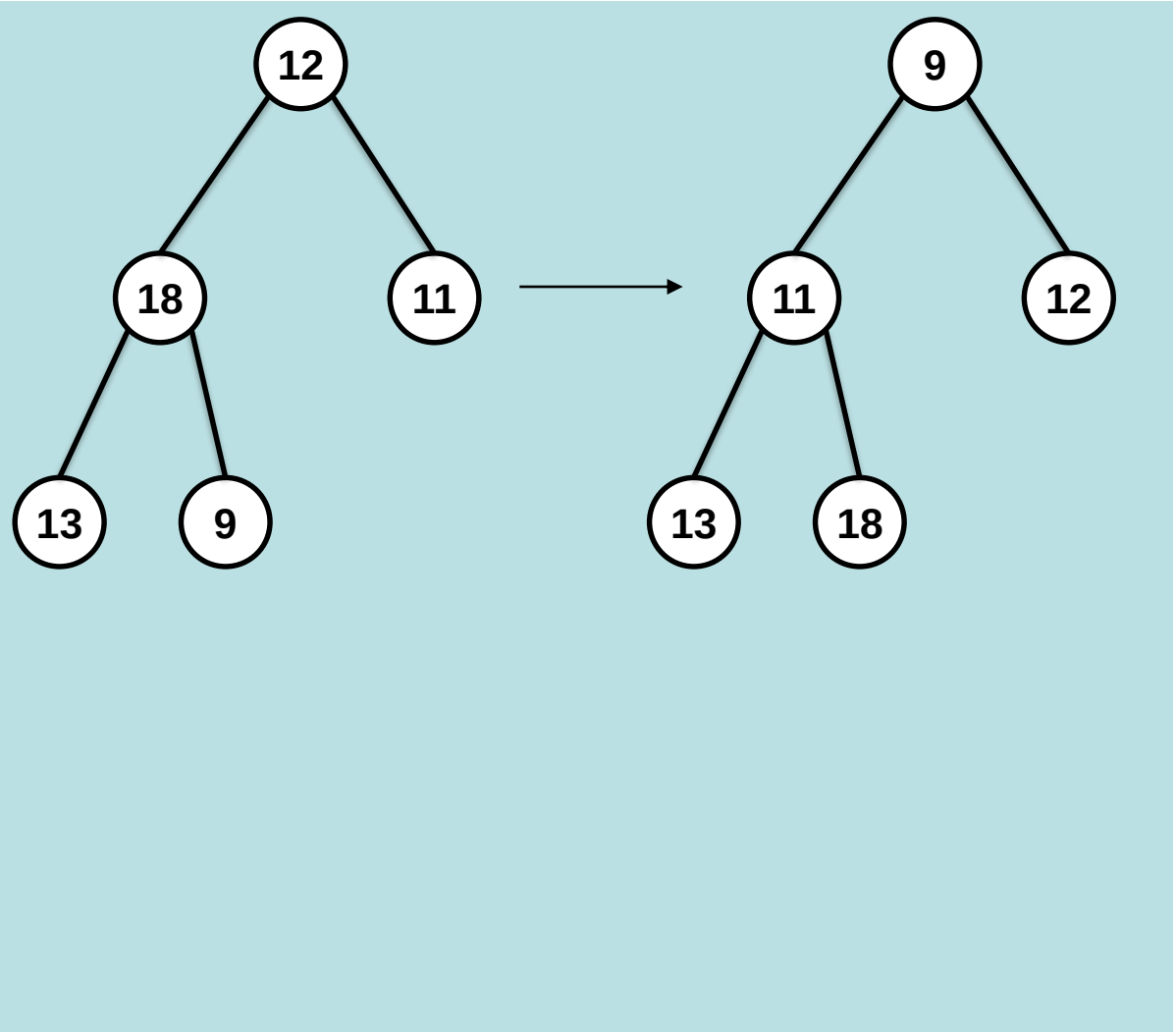
Heaps Sort Example

- Example: sort the list [12, 18, 11, 13, 9]

- Build a binary tree from the array (list).
- Transform the binary tree into a Min-heap

- Perform Min-heap sort
  1. Remove the minimum value into the sorted-array
  2. Restore the heap property
  3. Repeat steps 1 and 2 till the heap is empty

- The sorted list is [9, 11, 12, 13, 18]



Heap Sort Pseudocode

- HeapSort is an implementation of SelectionSort, but using a heap to get the smallest element.
- **`Insert(.)`** in the first loop takes $O(\log n)$, hence this loop is $O(n \log n)$
- **`ExtractMin()`** in the second loop takes $O(\log n)$ and so the second loop is $O(n \log n)$
- Overall complexity of HeapSort is $O(n \log n)$
- Just choosing the right data structure makes SelectionSort competitive

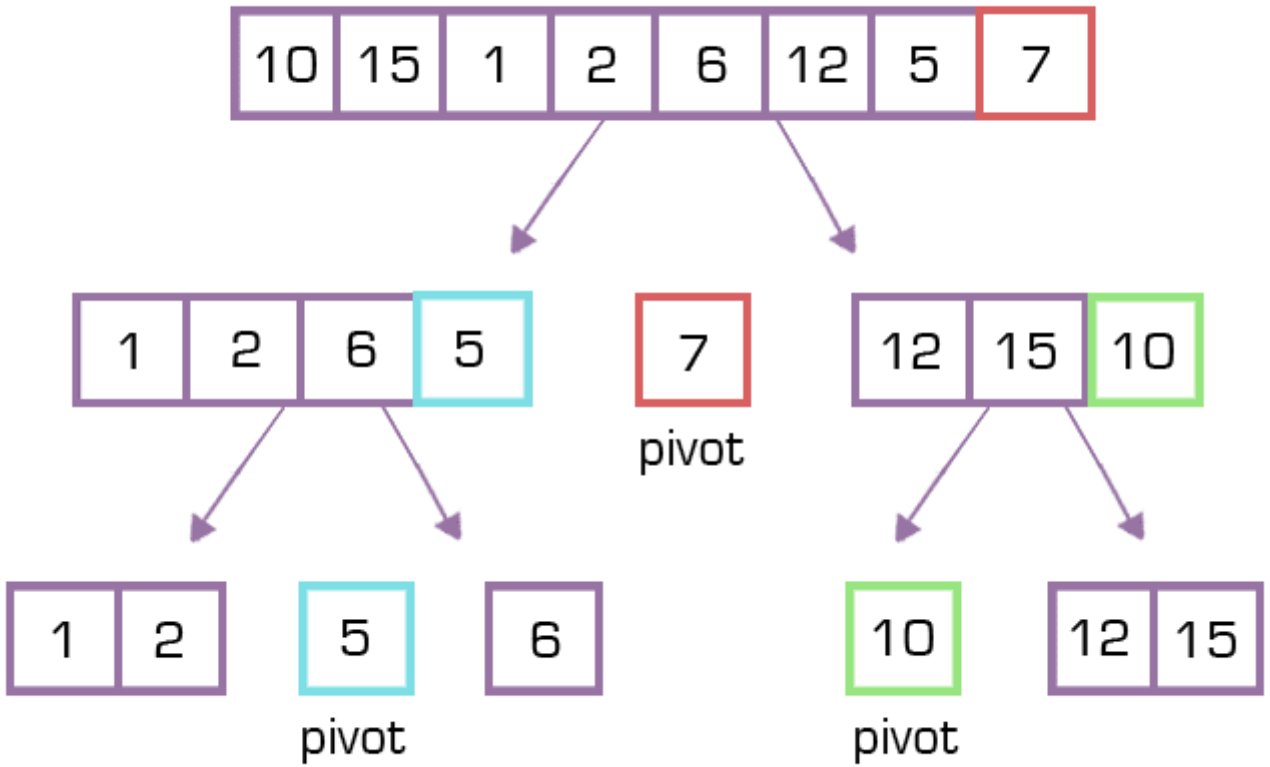
Heap sort

```
HeapSort(l : list)

h = new heap

for k=0…l.length-1
      h.Insert(l(k))
end for
for k=0…l.length-1
      l(k)=ExtractMin()
end for
```

Quick Sort – Partition in Place.

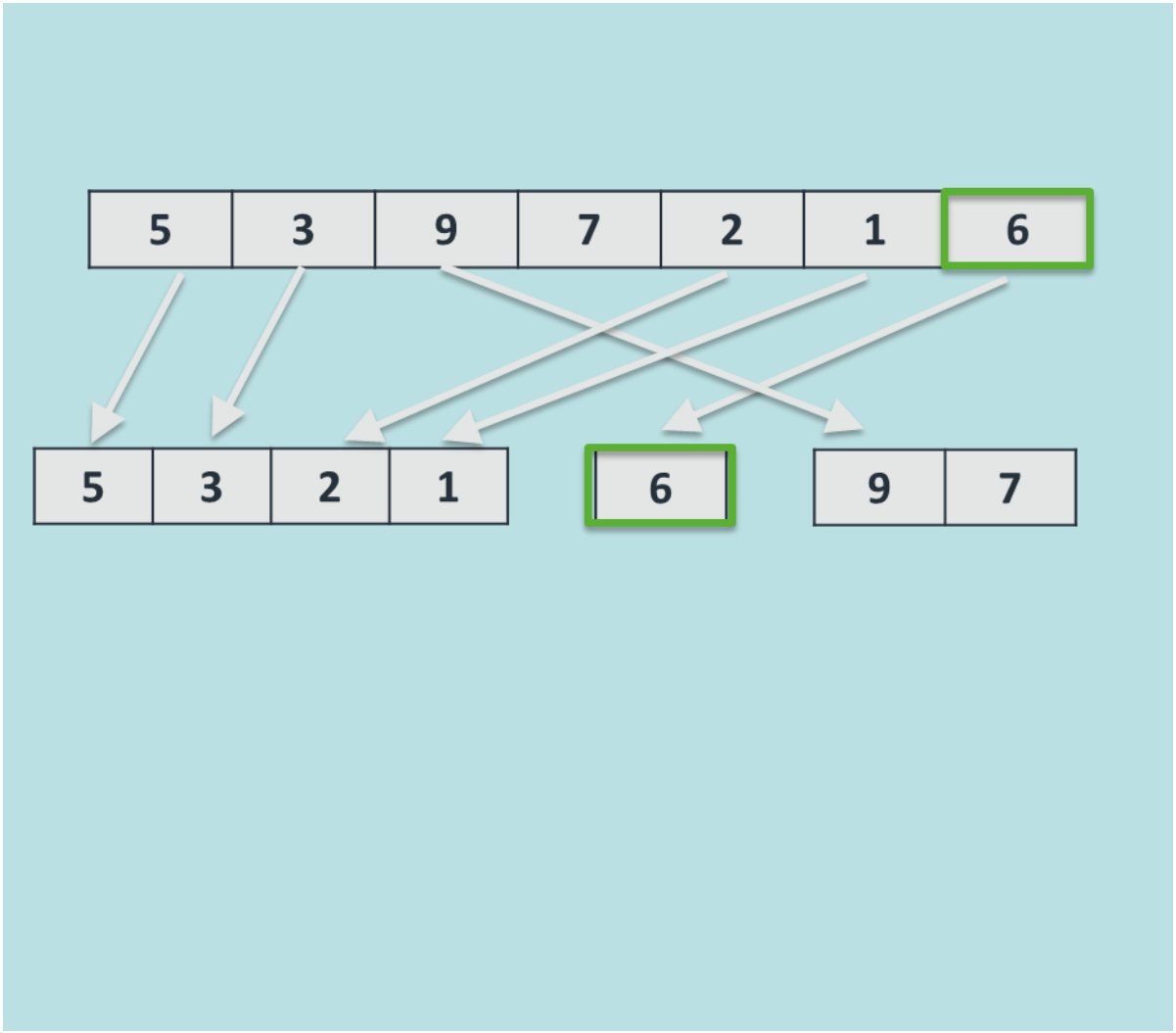


QuickSort - Partitioning

QuickSort is an example of the divide-and-conquer algorithm strategy

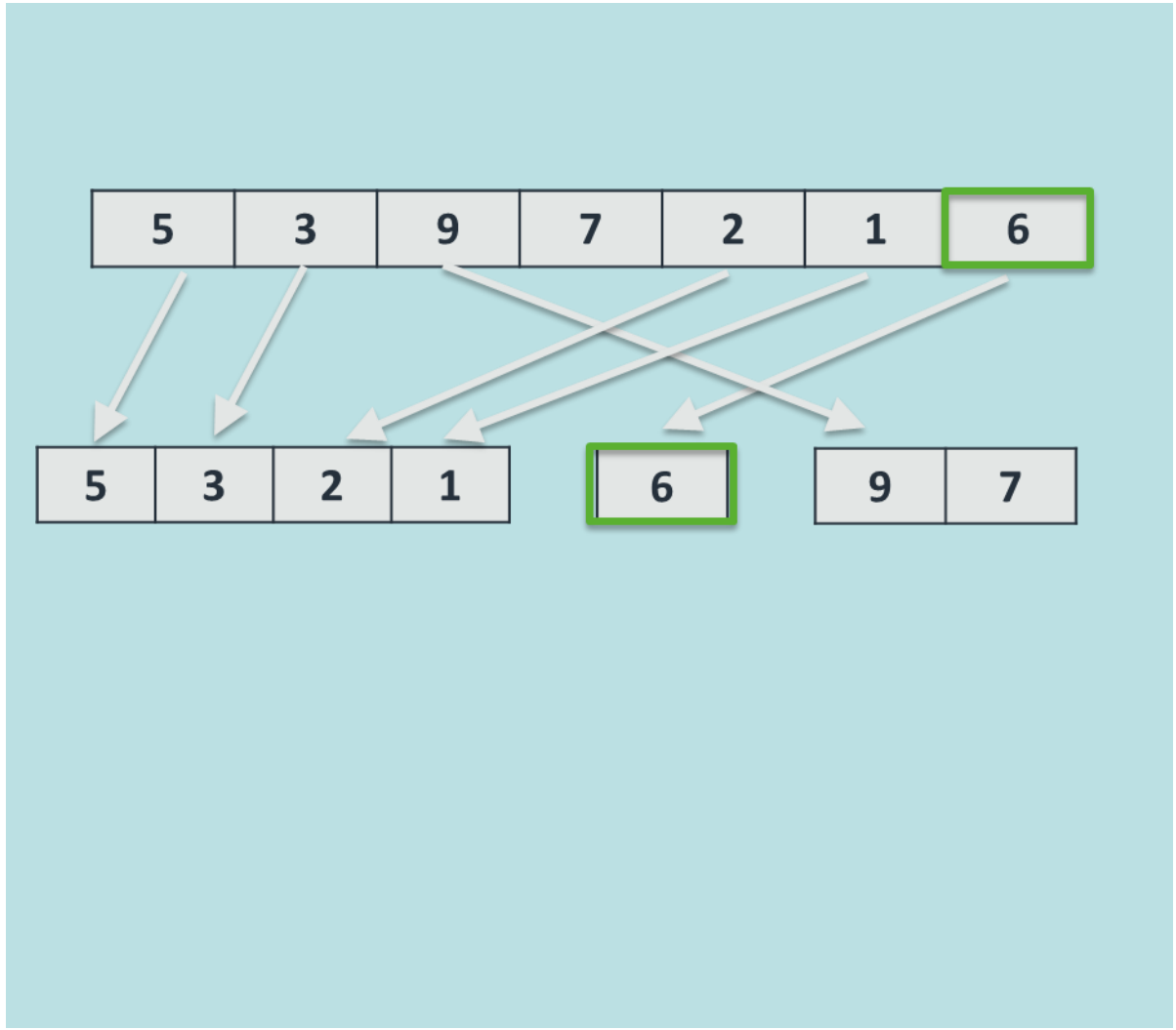
Idea:

- Choose a pivot
- Anything less than or equal to the pivot goes in a lower list
- Anything greater than the pivot goes in an upper list



QuickSort - Recursion

- Now the pivot (6) is in the correct place.
- Sort the lower and upper lists individually.
- If we chose the ideal pivot, these would be half the size (compare MergeSort)



QuickSort – Pseudocode

- **`partition`** does the main work. It chooses a pivot and divides the list into the two parts.
- We could implement this by creating two new lists at each partition (lower and upper).
- But with good choice of pivot and using minimum swaps, we can work in the original list

QuickSort

```
QuickSort(s : list, l : int, u : int)
// l,u indicate which part of the list to
// sort. Initially l=0, u=n-1
if l<u
        p = partition(s,l,u)
        QuickSort(s,l,p-1)
        QuickSort(s,p+1,u)
end if
```

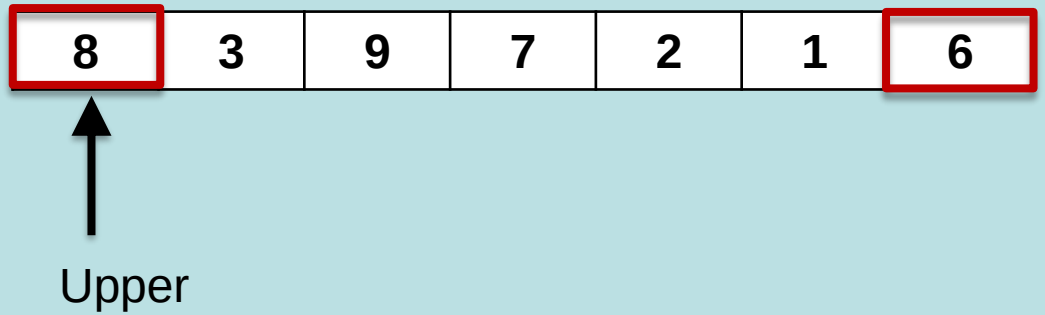
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Compare each element with the pivot
If lower, swap with first upper element, then move up the upper index



8≥6, so no swap

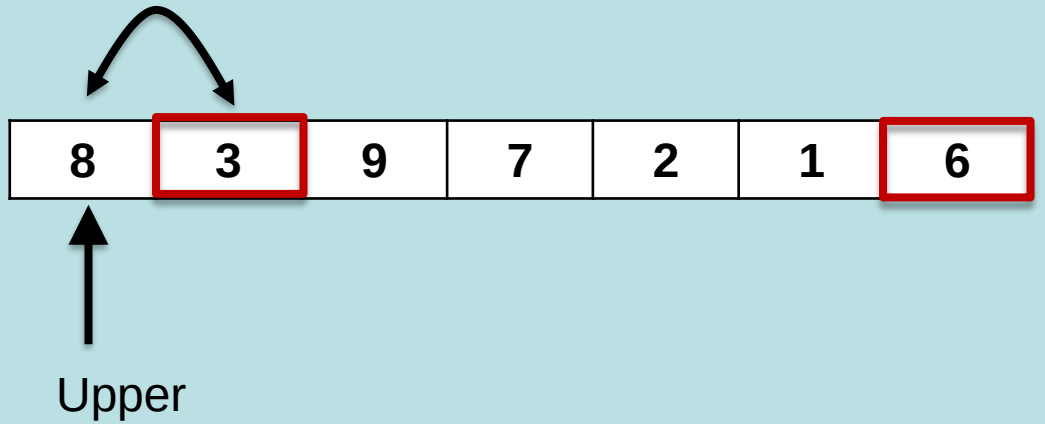
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Compare each element with the pivot
If lower, swap with first upper element, then move up the upper index



3<6, so swap and move upper index up

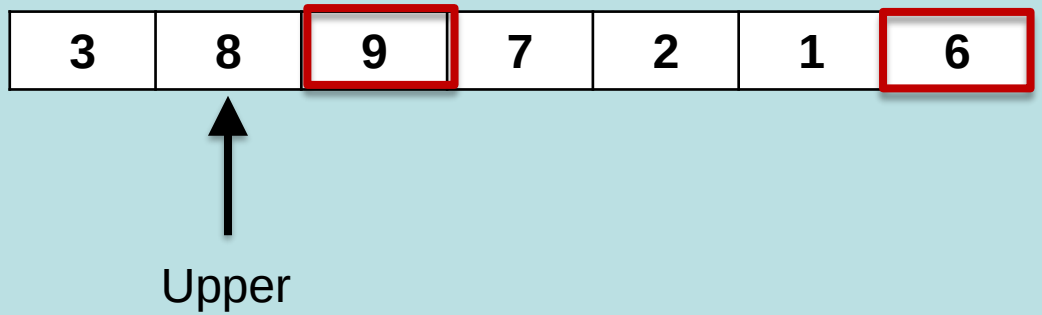
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Compare each element with the pivot
If lower, swap with first upper element, then move up the upper index



9≥6, so no swap

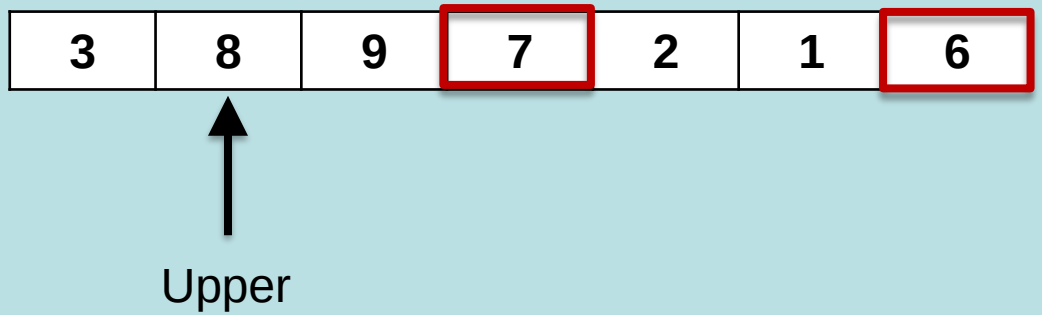
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Compare each element with the pivot
If lower, swap with first upper element, then move up the upper index



7≥6, so no swap

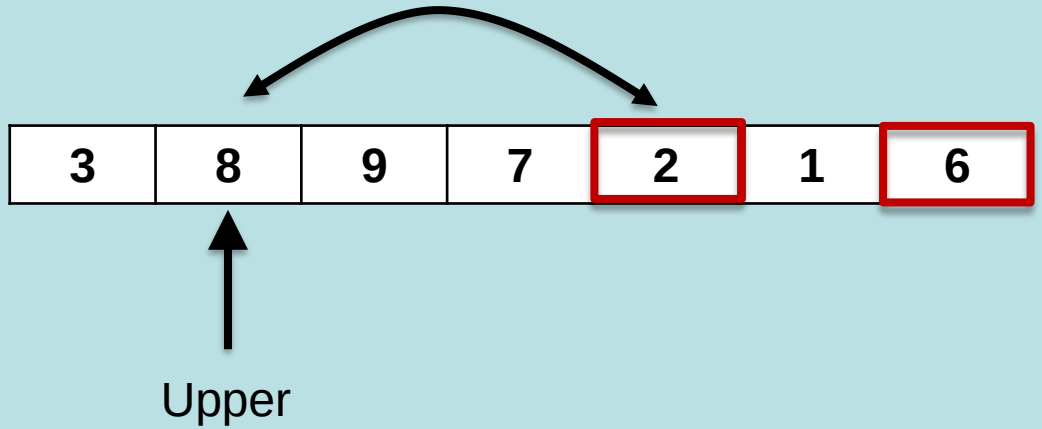
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Compare each element with the pivot
If lower, swap with first upper element, then move up the upper index



2<6, so swap and move upper index up

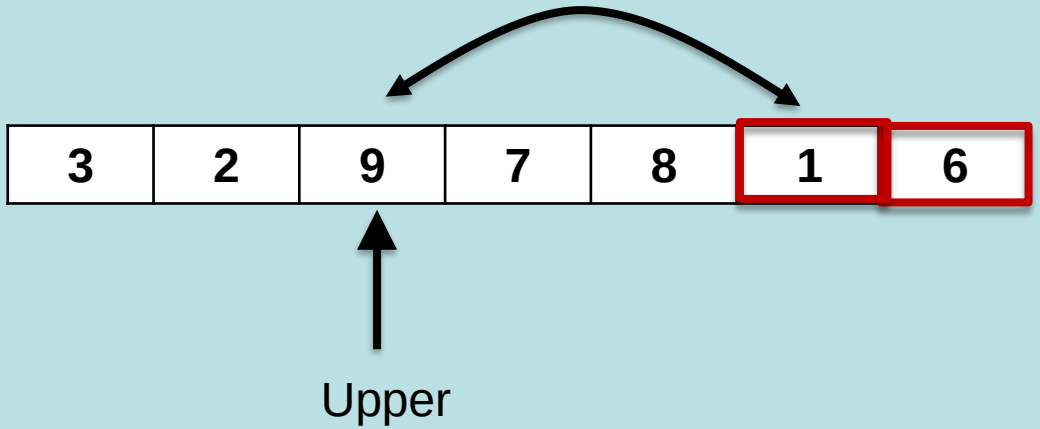
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Compare each element with the pivot
If lower, swap with first upper element, then move up the upper index



1<6, so swap and move upper index up

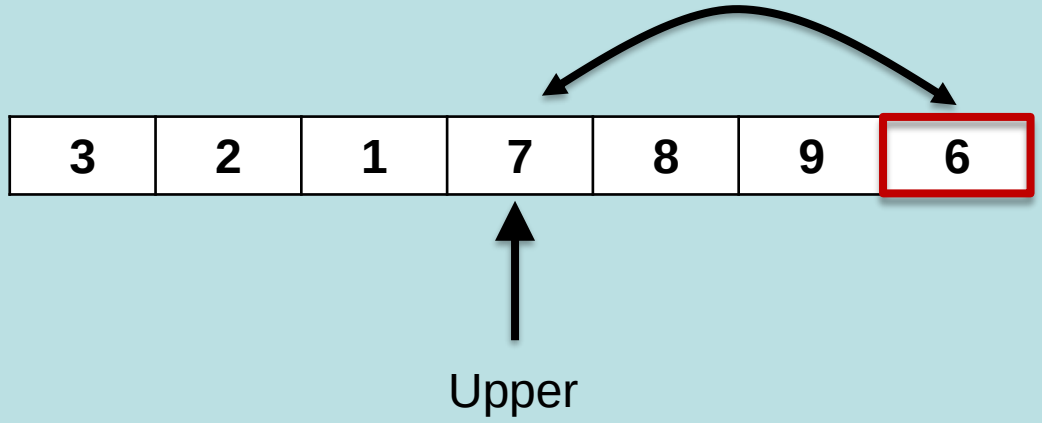
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Finally, swap pivot and move up upper index



Swap pivot

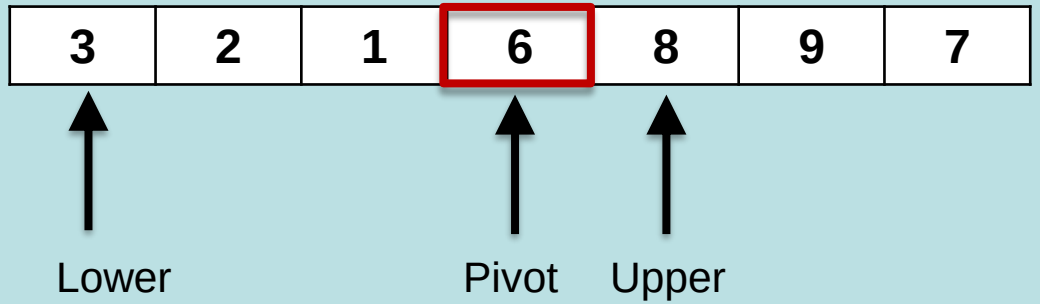
Partitioning in place - Example

To partition in place, we choose the last item as the pivot.

We put the lower elements at the start.

An index marks the start of the upper list

Finally, swap pivot and move up upper index



Partitioning Algorithm

Partition

```
Partition(s : list, l : int, u : int)
// l,u indicate which part of the list to
// partition.
p = u
upperindex = l
for i=l…u-1
       if s(i)<s(p)
          swap(i,upperindex)
          upperindex = upperindex + 1
       end if
end for
swap(p,upperindex)
return upperindex    // This is the new
                     // pivot position
```

`partition` does one pass through the list; hence it is $O(n)$.

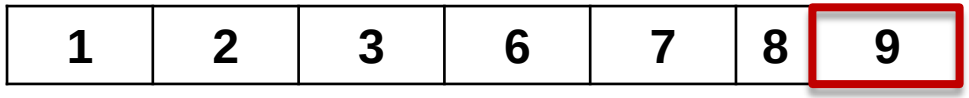
QuickSort Complexity(1)

What is the complexity of QuickSort?

Remember this is the worst-case time taken.

If the pivot splits the list exactly in the middle, each sub-list is around half the size (like MergeSort), so we would expect $O(n \log n)$ complexity.

The pivot could be right at the end, (e.g. if the list were sorted)



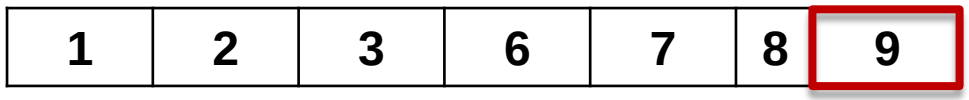
QuickSort Complexity(2)

The lower list is size $n - 1$, and this must now be sorted.

It takes n recursive calls to QuickSort to finish the sorting, each calling Partition, which itself is $O(n)$

Hence QuickSort is $O(n^2)$, worse than MergeSort and HeapSort!

Why is it called QuickSort, and why is it the standard library sorting routine?



QuickSort Expected Complexity(1)

To study expected complexity, we need to make assumptions about the expected input.

Assumption

The input is in a uniformly random order. Any element has equal chance to appear at any position.

Then our choice of the last element as pivot is equivalent to choosing a random element.

QuickSort Expected Complexity(4)

In summary, QuickSort recurses to expected depth $O(\log n)$, and executes $O(n)$ operations (in Partition) at each level, so the expected complexity is $O(n \log n)$

The same as MergeSort, HeapSort, but it has an inferior worst case complexity. So, which is best?

Depends on the exact situation, but the partition operations of QuickSort are in-place with minimum swaps, so very efficient. Typically, it is several times quicker than the others.

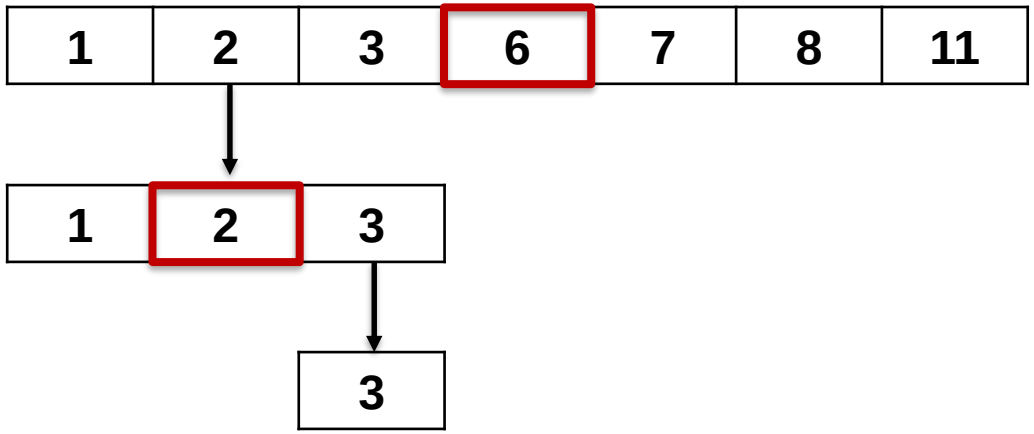
This expected complexity analysis is only an estimation, we should really compute the probability of every configuration, then average the runtime.

Applications of Sorting – Improved search

Sorted list
Compare the query with the middle item in the list.
If it is less, search the first half
If it is greater, search the second half

This is Binary Search $O(\log n)$

Query 3:



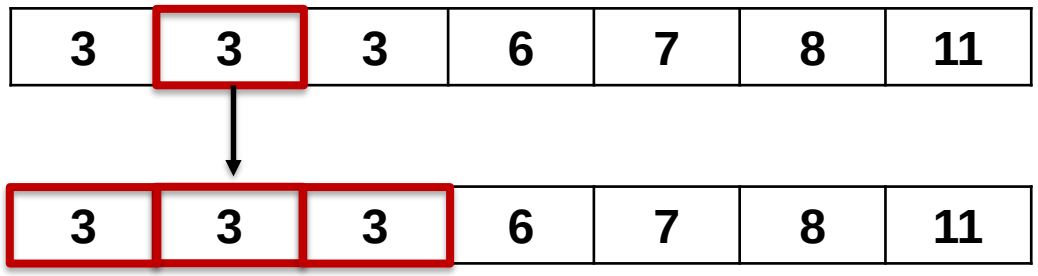
Unsorted list
Search every item, return if equal to query $O(n)$

Since sorting is $O(n \log n)$, we gain if we intend to make more than $O(\log n)$ queries.

Applications of Sorting – Improved search

Sorted list
To find the number of occurrences of an item x, first search for the item in $O(\log n)$ time.
Then scan the list locally – all occurrences of x will be next to each other.
This takes $O(\log n + c)$ where c is the number of occurrences.

Query 3:



Unsorted list
Scan the list, keeping a count of the item
$O(n)$

Since sorting is $O(n \log n)$, we gain if we intend to make more than $O(\log n)$ queries.

Summary

- Understand:
  - HeapSort and QuickSort
  - The difference between worse-case and expected-time
  - Simple expected-time analysis
  - How sorting can speed up other algorithms
- Read
  - Skiena, Sections 3.5, 4.1, 4.2, 4.6
  - Attempt exercises 4-2, 4-20, 4-21

- Next
  - Algorithms for trees