School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Quick Sort and Merge Sort

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort
- Quick Sort
- Merge Sort
- Bucket sort
- Heap sort

Quick Sort

- Quick sort is a divide-and-conquer sorting algorithm.
- We divide the larger list into two smaller lists, the low elements and the high elements.
- We then recursively sort the sub-lists until we reach the base case, lists of length 0 or 1.
- Compared to what, do we determine if something is lower or higher?
  - We pick a value for comparison - the pivot.

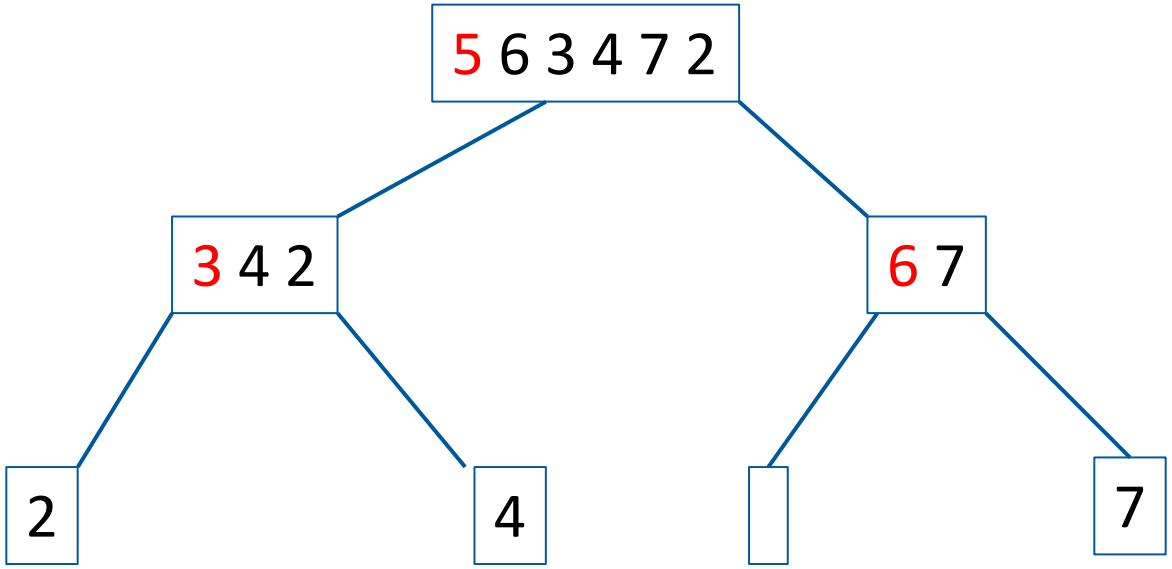
Quick Sort

- Steps for quick sort, starting with a list of values:
  - Pick an element, the pivot, from the list.
  - Reorder the list so that everything smaller than the pivot comes before it, and everything greater comes after it. (The partitioning step)
  - Recursively sort the ‘smaller’ list and the ‘greater’ list.

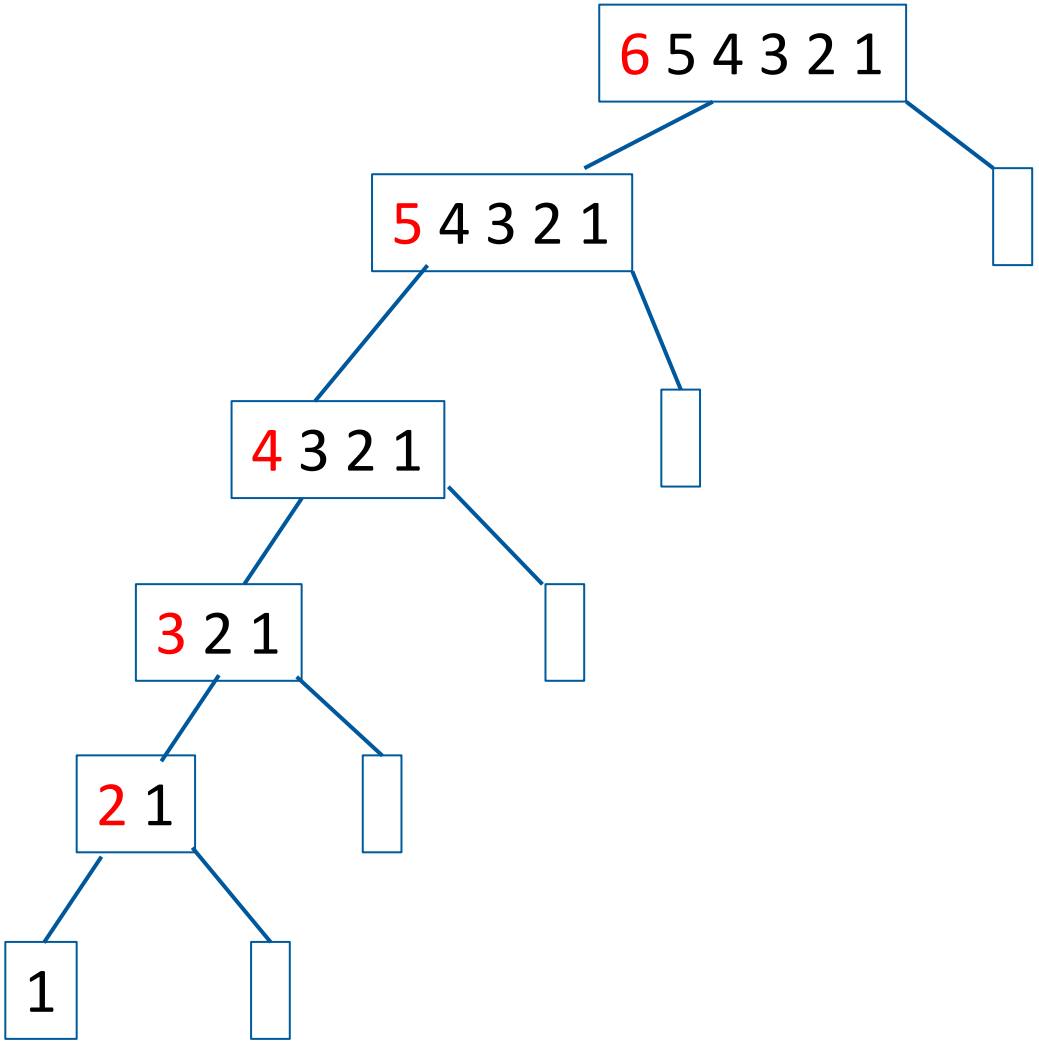
Quick Sort

```
function quicksort(mylist) {
  if length(mylist) <= 1
    return mylist                                    // sorted.

  list less, greater = empty list

  select and remove 'pivot' from mylist

  for each value in array {
    if ( value <= pivot )
      add value to end of less
    if ( value > pivot )
      add value to end of greater
  }

  return(append(quicksort(less),pivot,quicksort(greater))
}
```

Quick Sort Example



Quick Sort Worst-Case Example



Quick Sort Worst-Case Example

Input: [n, n-1, n-2, …, 1], assume that we always take the first element of given subproblem as pivot element

- Pivot element: n
  We get [n-1, …, 1] + [n]+ [] leading to
  [n-1, …,1, n]
- Pivot element n-1 for [n-1, …, 1]:
  We get [n-2, …, 1] [n-1]+[] leading to
  [n-2, …, 1, n-1, n]
- Pivot element n-2 for [n-2, …, 1]:
  We get [n-3, …, 1]+[n-2]+ [] leading to
  [n-3, …, 1, n-2, n-1, n]

…….

Quick Sort

- The choice of pivot is essential.
- To discuss this, let's look at the best and worst case performance for quicksort.
  - What is the worst case?
    - The pivot choice can make all the difference.
  - What is best case?
    - $T(n)=2T(n/2)+cn$
  - What is the average case? O(n log n)
    - Idea: average over n! permutations of n elements
- If the list is small, insertion sort may even take less time!
  - Good to combine quick and insertion sort

Merge Sort

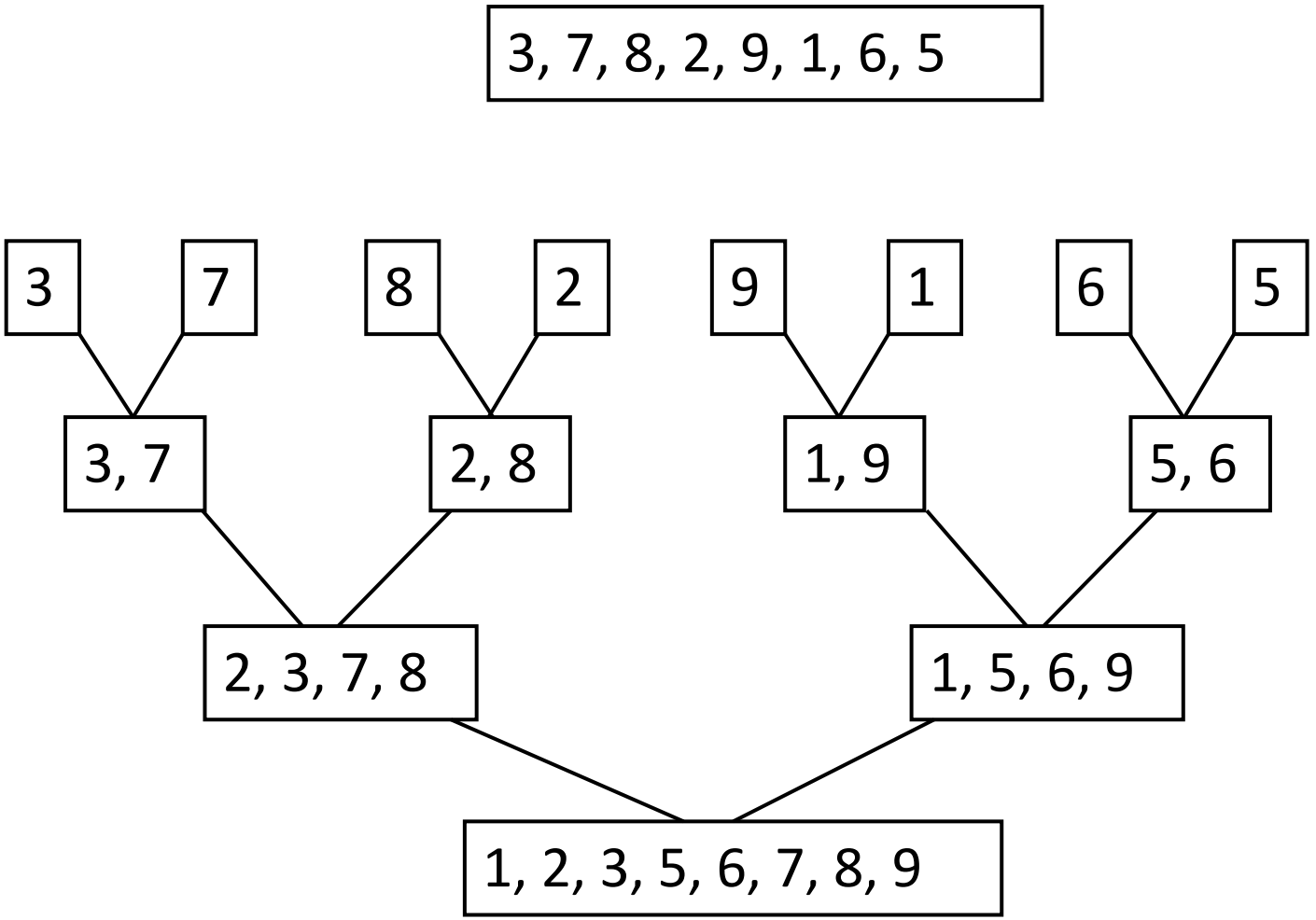
- Merge sort uses a divide-and-conquer approach.
- What would happen if we had two (smaller) sorted lists?
- The final sort would require us to merge the two sorted lists
  - Compare the head of the lists
  - Place the smaller element in the resulting list
  - Remove that from the given list that it belongs to
  - Go back to comparing the heads until one list is empty
  - Place all elements of the remaining list in the resulting list

- Base case: lists of size one

Merge Sort – recursive version

- We start with a list of length *n* and split it into two, which will be merged once they are sorted.
- To sort each sub-list, divide into two that can be merged once they are sorted.
- Continue recursively until we hit the single element base condition.
- Then we unwind and merge everything.

Merge Sort Example



Merging two sorted lists

Use zipper mode

2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9
2, 3, 7, 8	1, 5, 6, 9

1

1, 2

1, 2, 3

1, 2, 3, 5

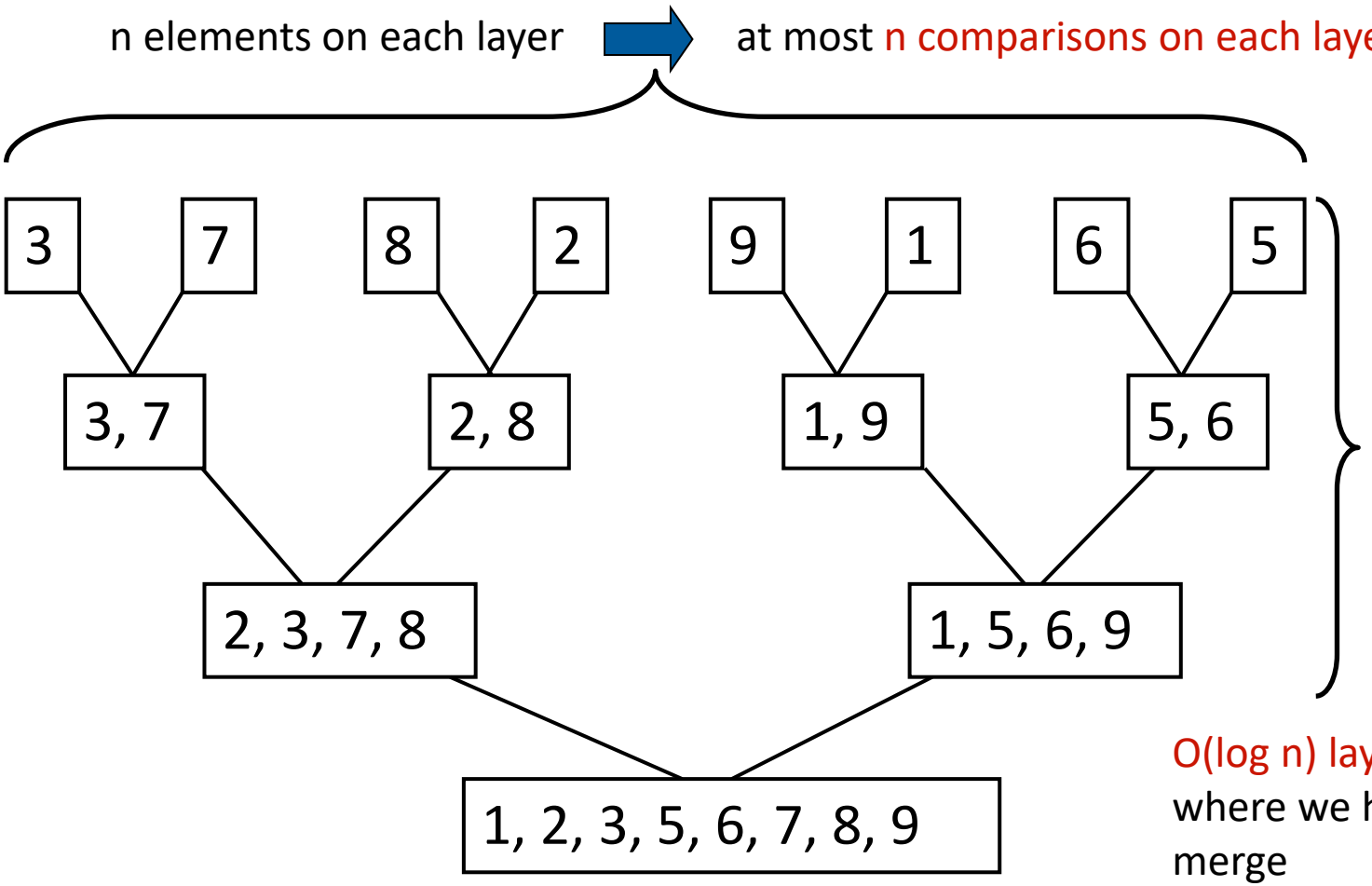
1, 2, 3, 5, 6

1, 2, 3, 5, 6, 7

1, 2, 3, 5, 6, 7, 8

1, 2, 3, 5, 6, 7, 8, 9

Runtime



In total: O(n log n) comparisons

Merge Sort

```
function mergesort(list m){

  if length(m) <= 1                          // for a single element
     return m                                // regard as sorted

  list left, right, result
  middle = length(m)/2                       // find the middle point

  for each x in m up to middle               // add to the left list
     add x to left
  for each x in m from middle to the end     // add to the right list
     add x to right

  left = mergesort(left)                     // recursive sort
  right = mergesort(right)

  result = merge(left, right)                // compare and merge

  return result
}
```

Merge Sort

```
function merge(left, right){
    result = empty list

    while left is not empty and right is not empty do
        if first(left) <= first(right) then
            append first(left) to result and remove
        else
            append first(right) to result and remove

    // Either left or right list may have elements left
    while left is not empty do
        append first(left) to result and remove

    while right is not empty do
        append first(right) to result and remove

    return result

}
```

Merge Sort

- Performance
  - Worst case
  - Average case
  - Best case

$O(n \log n)$

Quick Sort vs Merge Sort

- Compared with quick sort
  - Quick sort has a worse complexity in worst case
    - But O(n^2) happens with very low probability
  - Quick sort does not need to keep an extra memory block
  - Quick sort is not stable!

Sorting Algorithms

- Selection Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Insertion Sort (stable)
  - Complexity
    - Worst and average-case $O(n^2)$
- Bubble Sort (stable)
  - Complexity
    - Worst and average-case $O(n^2)$
- Quicksort
  - Complexity
    - Worst-case $O(n^2)$
    - Average-case $O(n \log n)$
- Merge Sort (stable)
  - Complexity
    - Worst and average-case $O(n \log n)$

Comparison Sort

- Most of what we've been looking at have been comparison sorts
- The fundamental engine of the sort is comparing two values.
- Comparison sorts include:
  - quicksort
  - merge sort
  - bubble sort
  - insertion sort
  - selection sort
  - heapsort (we will cover it at the end of the semester)
- How good can the comparison sort be?

Typical causes of complexity

- There are some typical algorithmic structures that lead to common growth rates and data structures/algorithms with well known upper bounds on their growth rates.
- O(1) - a number of statements independent of input size (hash tables, with perfect hash function)
- O(n) - loop over size *n* input (linear search)
- $O(n^c)$ - nested loops over size *n* input (selection, insertion sort)
- O(log n) -repeatedly divide input in half (binary search)
- O(n log n) - "divide and conquer" split input in half and recursively call function until O(1) function can be performed, combine returned values in O(n) (mergesort)
- $O(c^n)$ - multiple possible combinations of input (or permutations of a string)


THE UNIVERSITY
of ADELAIDE