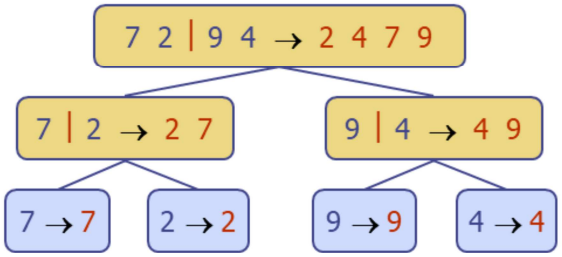
COMP2054-ADE

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Mergesort



1

Many will already have seen merge sort, and so will do it quickly. But the main point is to do the analysis of the complexity.

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1 (or "small enough to be done directly")
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

2

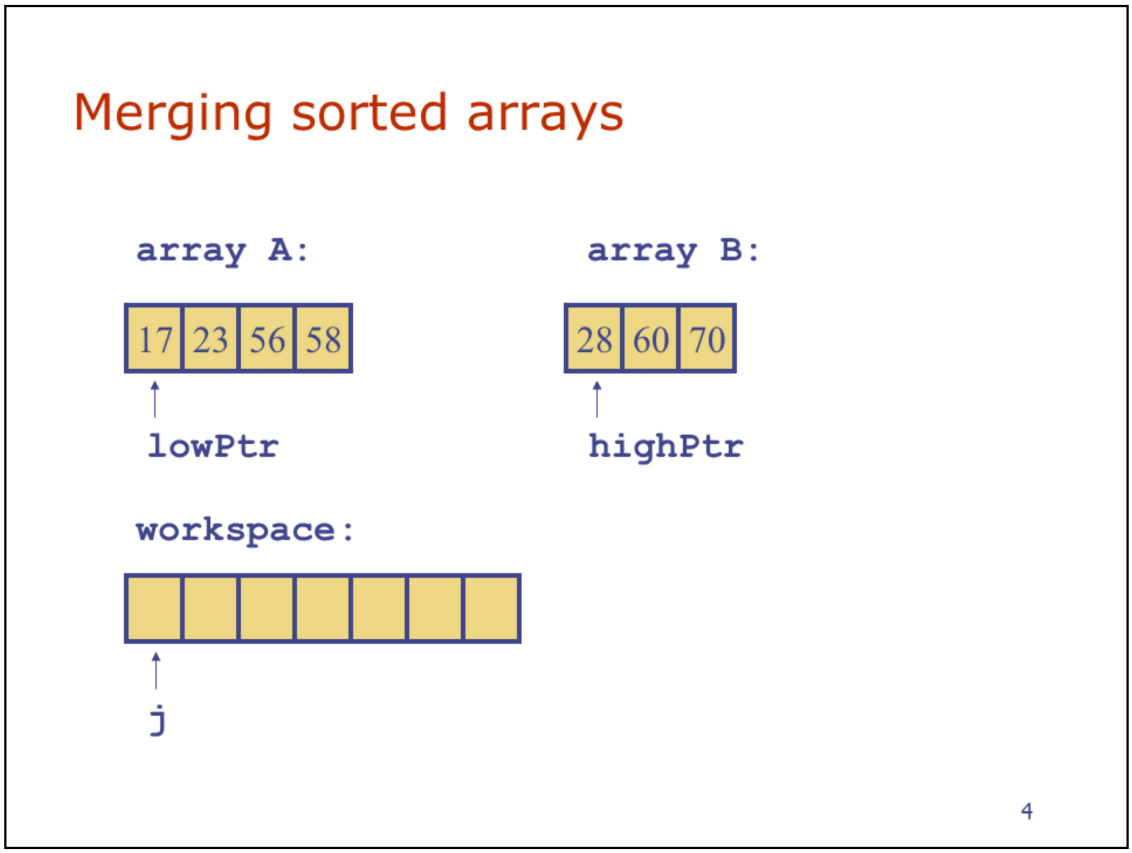
Firstly the general class of the algorithm is that relies on what is called "divide and conquer".
We split problems up, "divide"
solve the subproblems and then conquer by merging the results.

Merge-Sort

- Merge-sort on an input sequence (array/list) S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence
- First questions:
  Is the merge easy?
  What is the big-Oh of the merge?

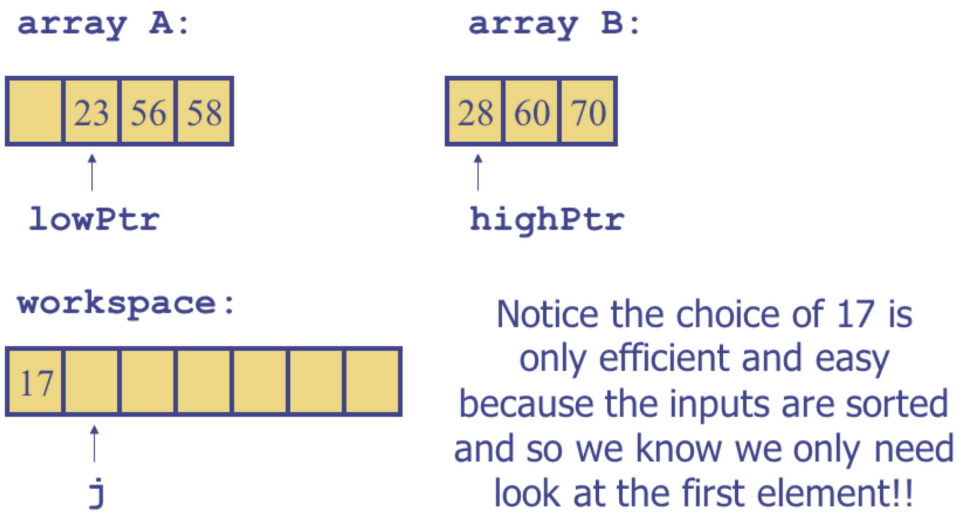
3

The general pattern is just to split into two separate sets.
Then do recursion and sort each one.
Then merge back together the sorted list.
First lets consider the complexity of the merge step.



Just do a simple example – we want to merge two sorted arrays and place into a new different array space.
Keep pointers to the tops of each one, and work down.
Firstly we know the smallest must be at the left of one of the sorted arrays.
We will get back to why they are "lowPtr" and "highPtr" and "workspace"

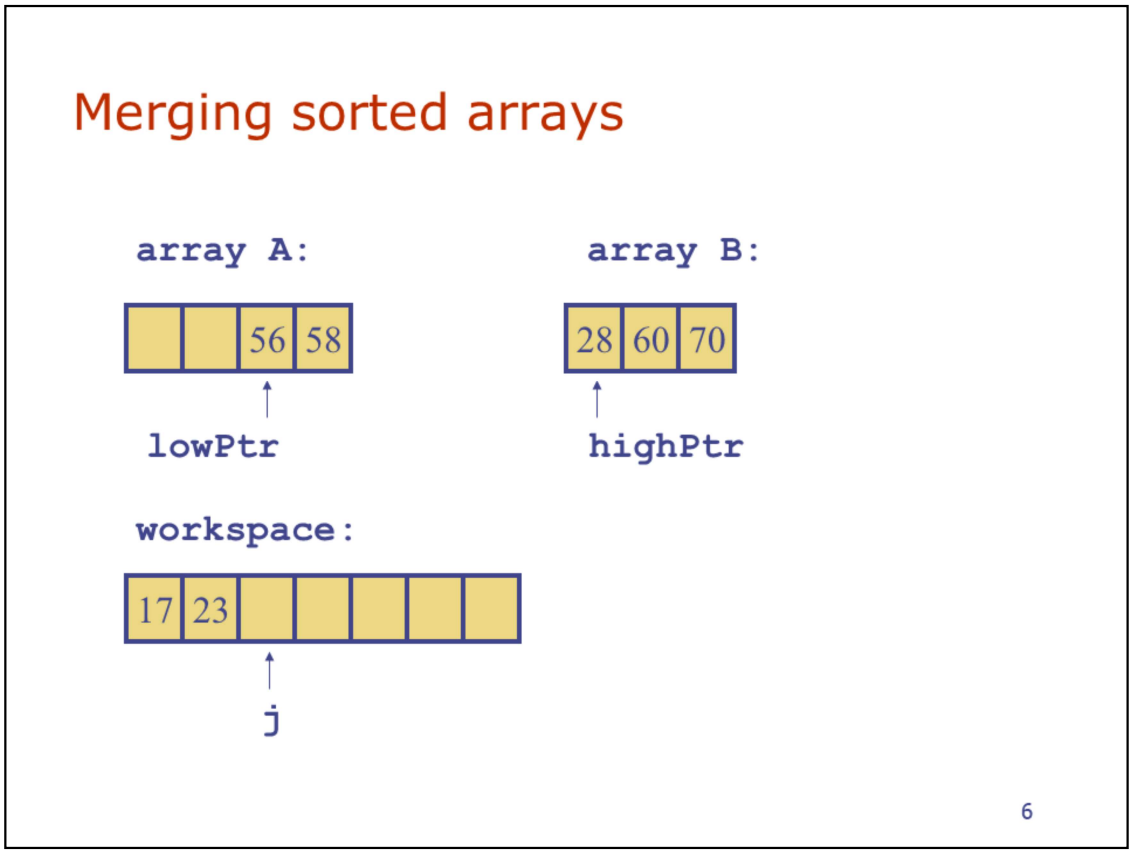
Merging sorted arrays



Notice the choice of 17 is only efficient and easy because the inputs are sorted and so we know we only need look at the first element!!

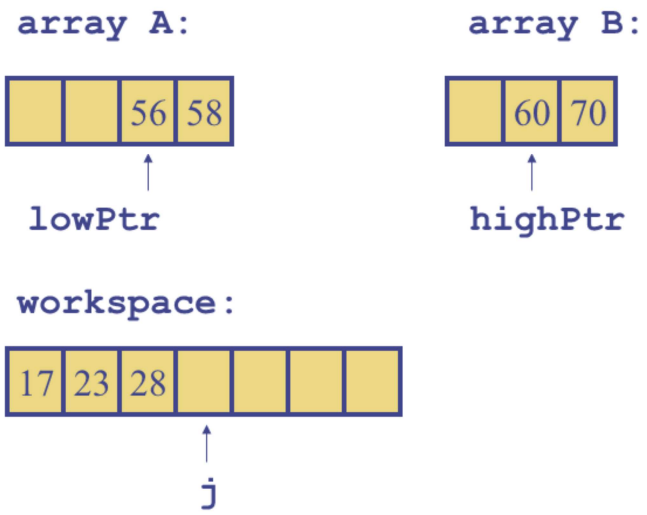
5

Notice that any formal proof of correctness would need to include something to prove that the minimum is indeed at one of the pointers.



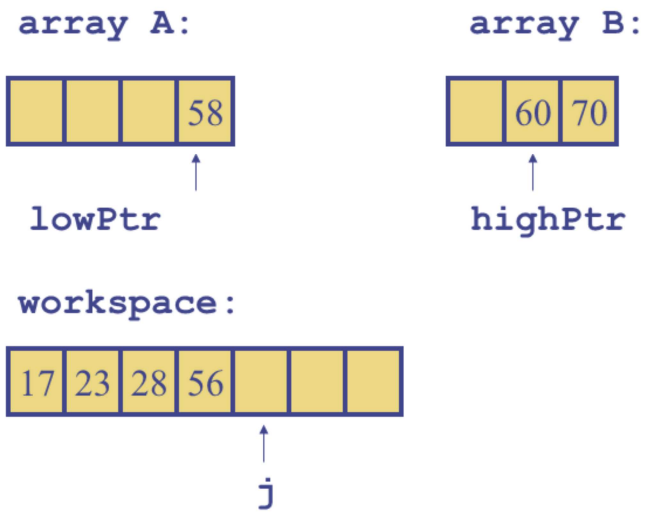
The process then just continues in the same fashion

Merging sorted arrays



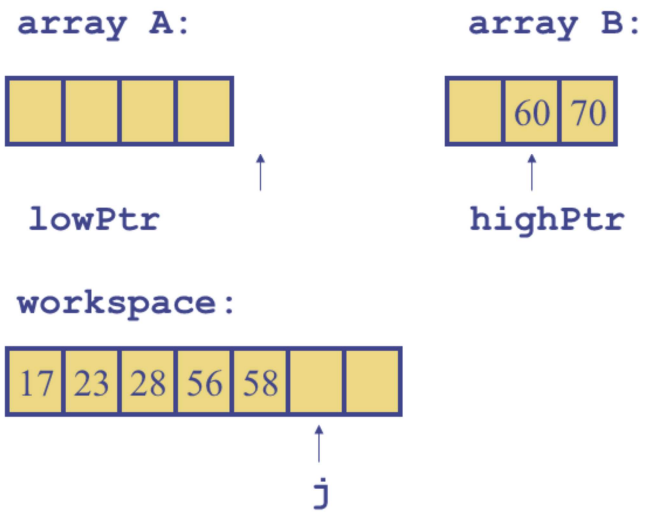
7

Merging sorted arrays



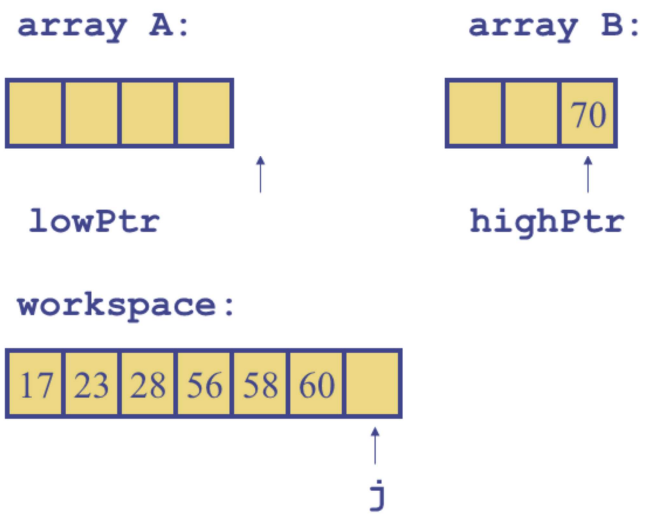
8

Merging sorted arrays

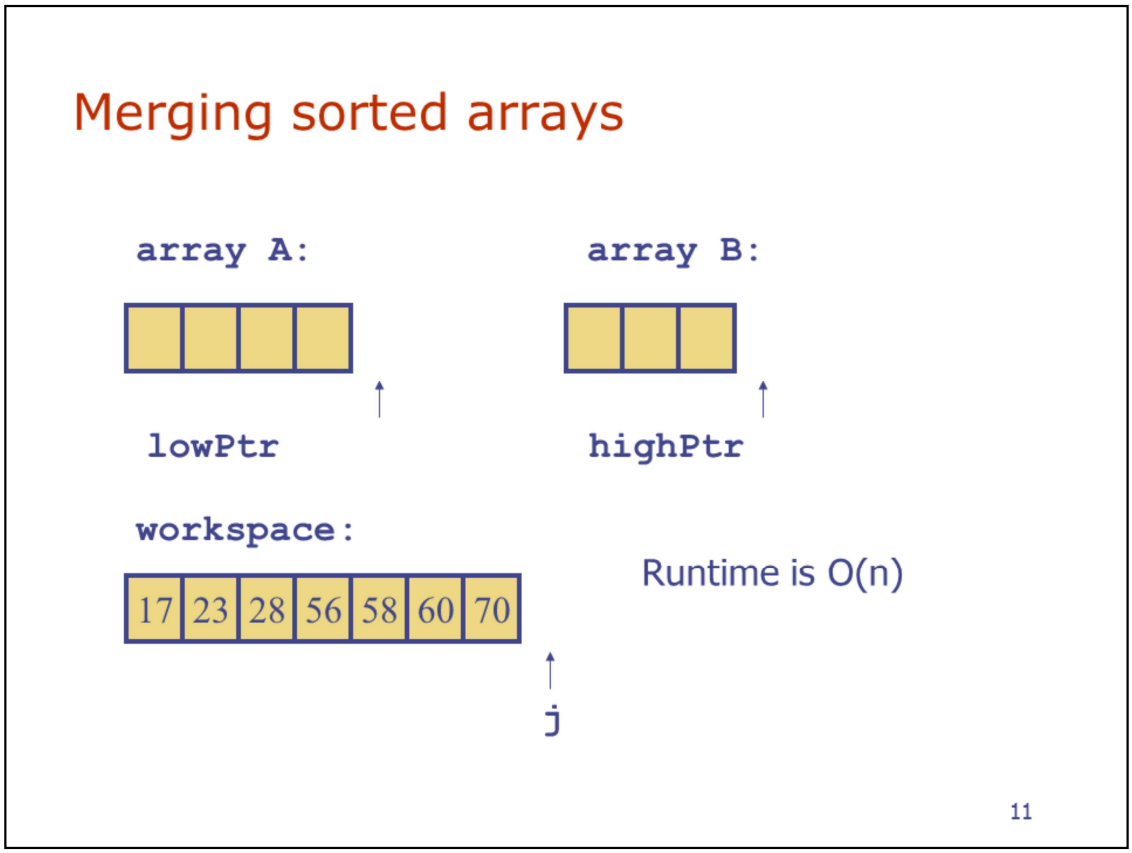


9

Merging sorted arrays



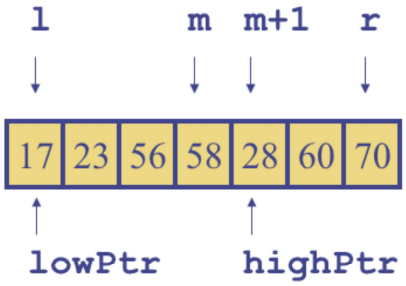
10



At each iteration we just have to select between two numbers, and copy one of them across, so each iteration is O(1).
We need to do this n times.
Hence overall runtime is O(n).

Merging halves of an array

- Pass boundaries of sub-arrays to the algorithm instead of new arrays, and merge into a "workspace":
- After merge, copy the workspace back to the original array



12

The actual implementation can work using sub-arrays of a larger array.

Implementation

```
public static void recMergeSort
                    (int[] arr, int[] workSpace, int l, int r) {
    if (l == r) {
        return;
    } else {
        int m = (l+r) / 2;
        recMergeSort(arr, workSpace, l, m);
        recMergeSort(arr, workSpace, m+1, r);
        merge(arr, workSpace, l, m+1, r);
    }
}
```

Initial call is with l=0 and r to be end of the array

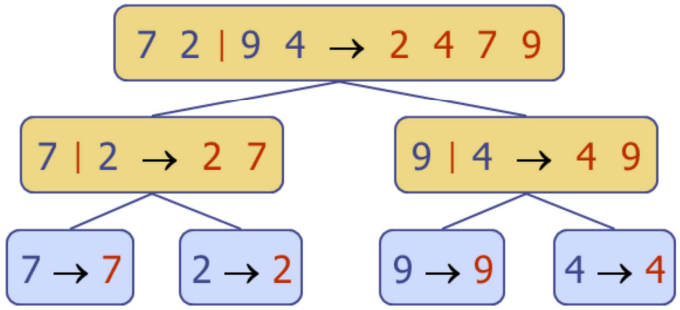
13

This means the actual code structure we think of is given here.
Note that it does a standard trick of generalising so as to work on sorting a range, specified by l – left, and r right, in a given array.
Then this makes it much easier to implement.
It is rather counter-intuitive by starting by making the problem "more difficult" – but is a useful general trick.
Of course we just start with l and r pointing the start and end.
The split is just to pick a middle point m, then we sort each half, and merge them together.
The workspace is carried through the recursions – but is just used for the merge step.

Merge-Sort Tree

Not actually implemented this way! A `node' here is conceptual not real

- An execution of merge-sort is **depicted** by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



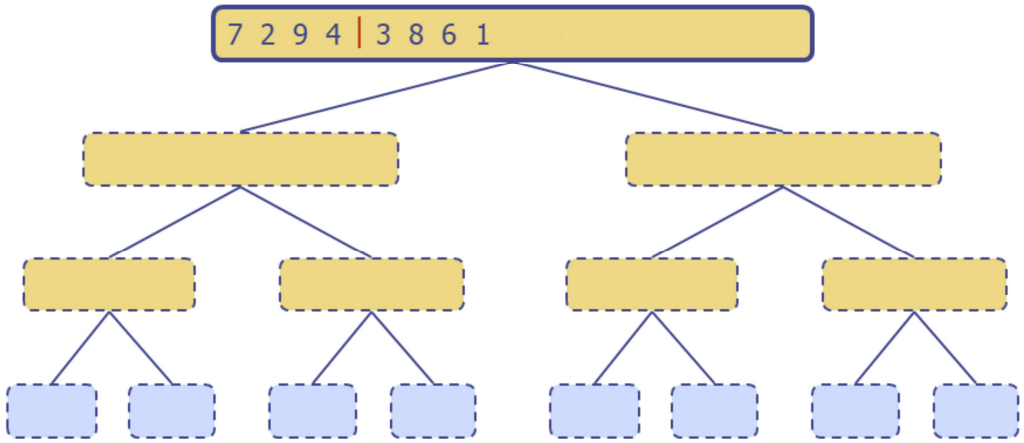
14

A question is then how we think about analysing this algorithm.
The technique is to view the execution as a tree.
Corresponding the recursive calls.
The most vital thing here is to realise that this is just a depiction for the analysis.
The actual implementation does not have any binary trees at all!

Let's do a an example.

Execution Example

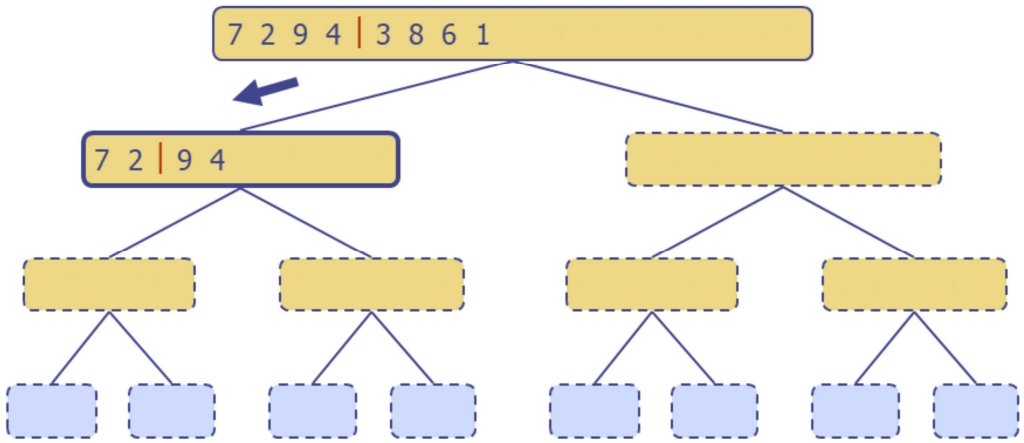
- Partition



15

Execution Example (cont.)

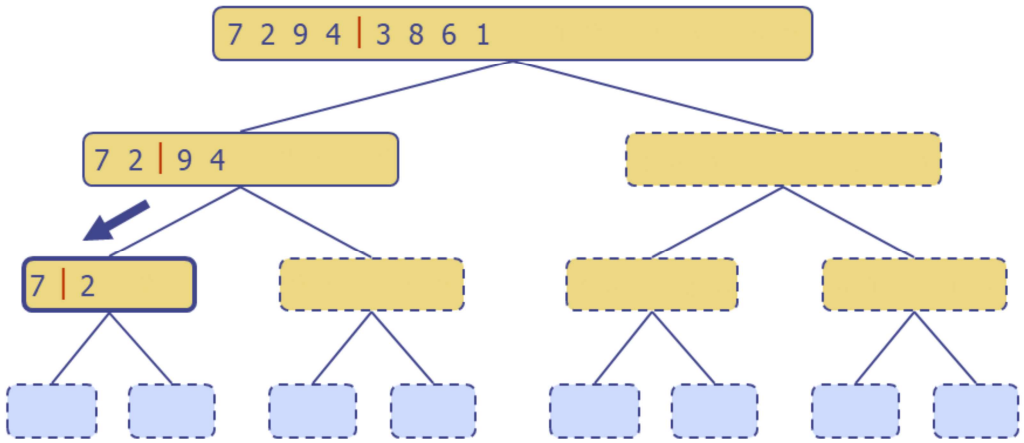
- Recursive call, partition



16

Execution Example (cont.)

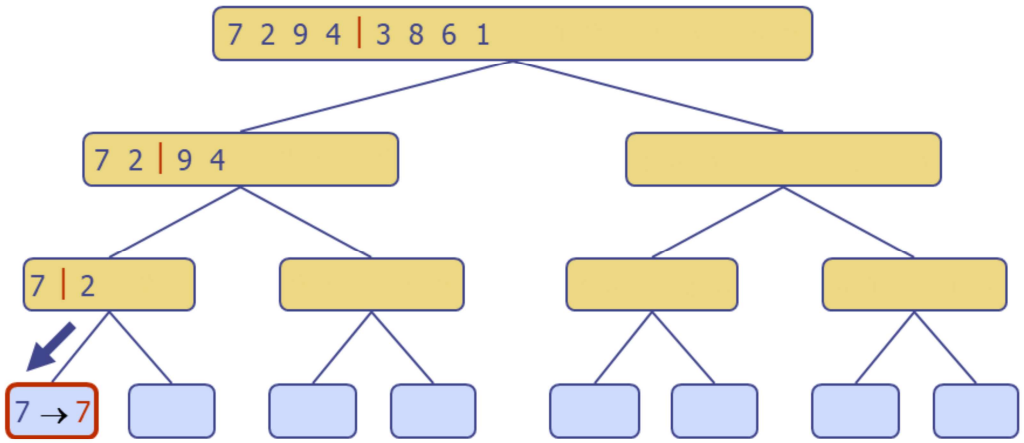
- Recursive call, partition



17

Execution Example (cont.)

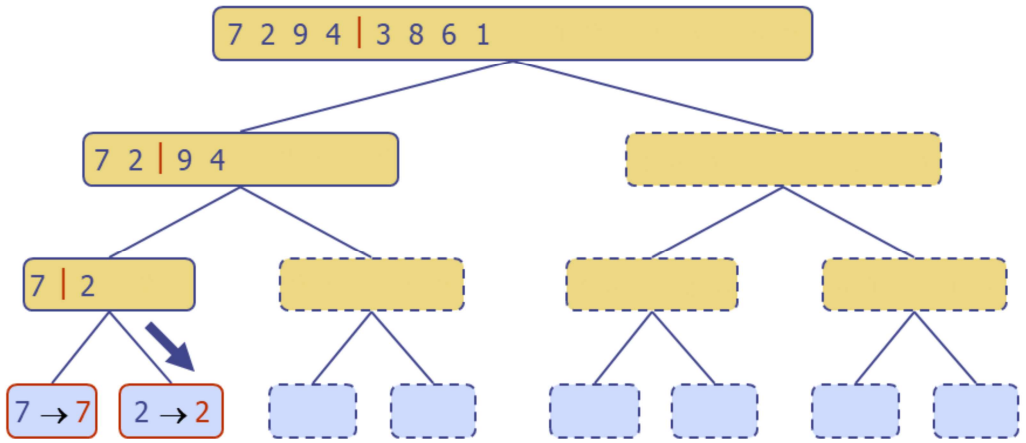
- Recursive call, base case



18

Execution Example (cont.)

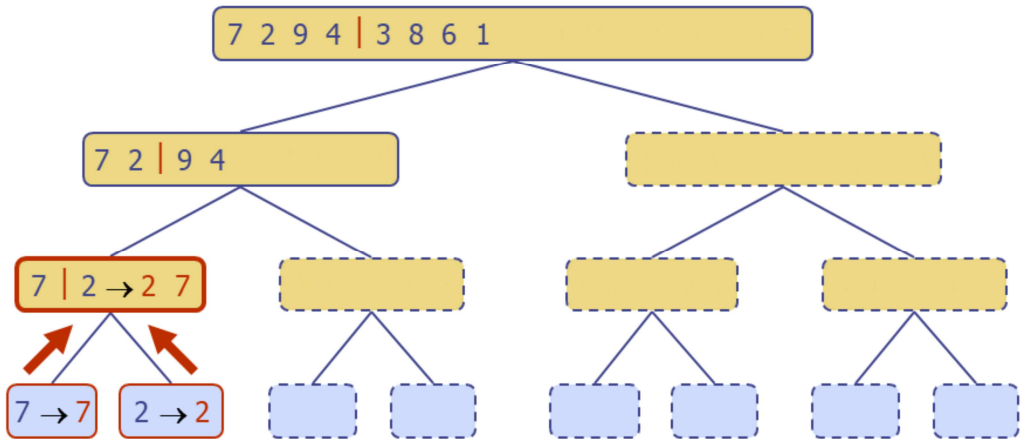
- Recursive call, base case



19

Execution Example (cont.)

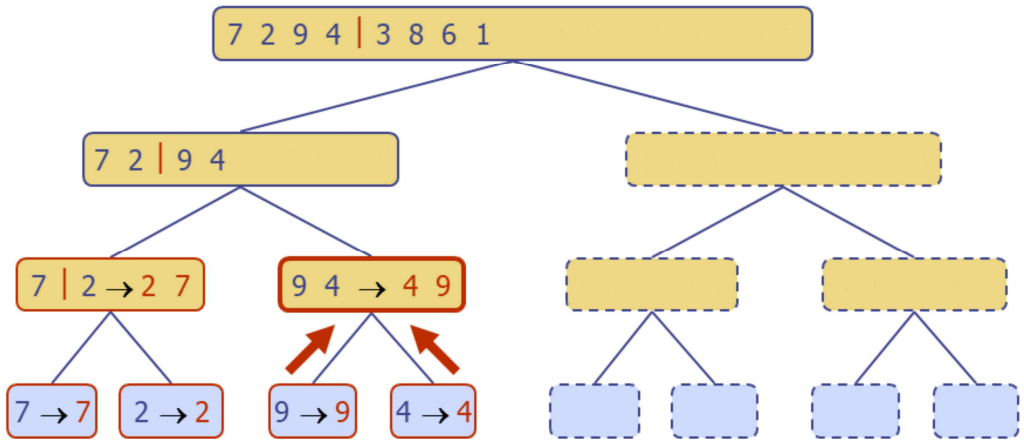
- Merge



20

Execution Example (cont.)

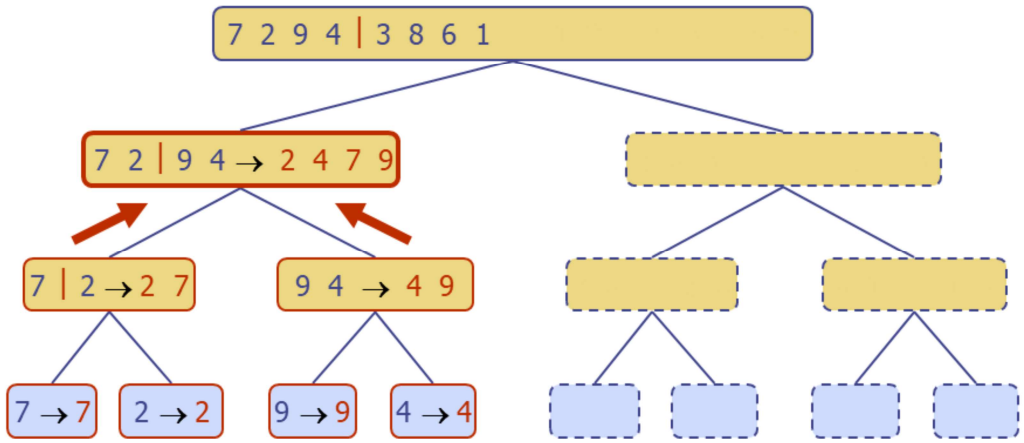
- Recursive call, …, base case, merge



21

Execution Example (cont.)

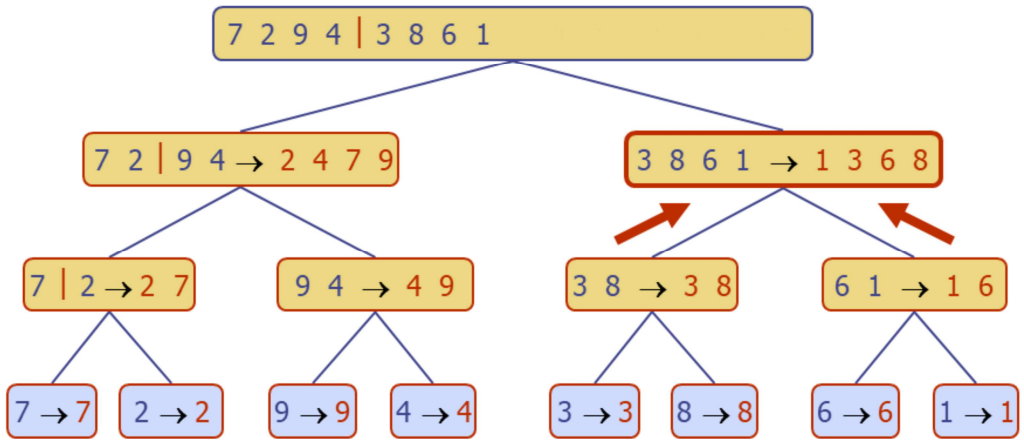
- Merge



22

Execution Example (cont.)

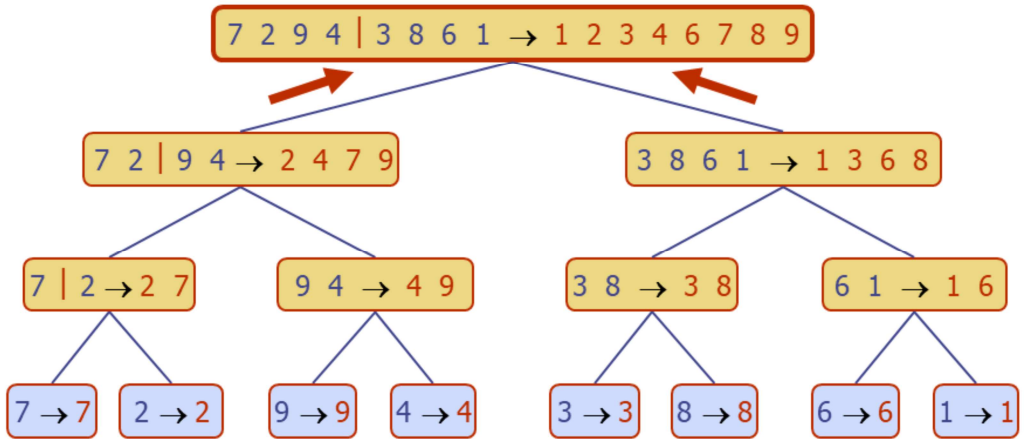
- Recursive call, …, merge, merge



23

Execution Example (cont.)

- Merge

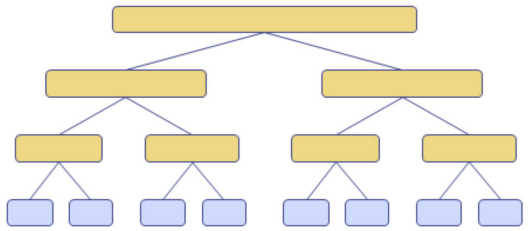


24

Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at all the nodes at depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
  - the numbers all occur and are all "used" at each depth
    - So, each depth uses O(n) work
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



25

Now we get to the analysis, and we can use the depiction by the binary tree.
This is the trickiest part of this lecture.
We need to take the tree and use it to figure out how much work is done.
One approach is to consider the amount of work done at each level (depth).
So firstly, the easy part – what is the height of the tree? Dividing by two until reach 1, is just log_2, but do not care about the base in O() notation.
The work at each level is O(n) – basically each original number appears once, and then is copied to the workspace, and back, so it is O(1) work for each of the n numbers at a level.
Hence, is O(n * log n)

Using merge sort

- Fast sorting method for arrays
- Good for sorting data in external memory – because works with adjacent indices in the array (data access is sequential)
  - It accesses data in a sequential manner (suitable for sorting data on a disk)
- Not so good with lists: relies on constant time access to the middle of the sequence

26

Questions to ask about sorting algorithms

- Big-Oh complexity (both time and space)?
  - Best case inputs? Worst case inputs?
- Extra workspace needed?
  Or is it `in-place'?
- Stable sorts?
- "Dynamic sorting" – how well does it do if the data is already "nearly sorted"
- Data access patterns?
  - Sequential? Random Access?
- Relevant and appropriate assertions

Aim to understand these issues for various sorting algorithms.

27

Minimum Expectations

- Know the Mergesort algorithm and how it works on examples
- Know and be able to justify/prove the big-Oh family of behaviours of Mergesort

28