COMP2054-ADE

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

Quicksort

1

Today we do a second very common sorting algorithm. Many might have seen it before, but we may bring out some new aspects, and also we need to do some analysis of the runtime – for the "E" in "ACE".

Motivations

- In merge sort the `divide' is simple, and the `merge' (relatively) complicated
- Can we swap this and make the 'merge' simple?
  - Answer: make the `divide' more complicated so that the merge becomes `concatenate'
- Analogy: sort a pack of cards by
  1. divide into 'red' and 'black' cards
  2. divide by suit (red into hearts and diamonds,…)
  3. divide by value …

2

So let's have a high level view of the mergesort algorithm.
The "divide" was to just pick a point in the middle of the array – this is just an O(1) computation of a midpoint. Hence very fast.

In contrast the merge was relatively complicated – and the merge step was O(n).
Can we swap this around? Can we make the "merge" trivial, though at the cost of a more complicated "divide"?
When is the merge simple? Think of a pack of cards. A natural way to sort is to divide by "type" – split into red and black, etc. Suppose we have a sorted stack of red cards and a sorted stack of black cards, then the "merge" is just to put one pack on top of the other.

Because all of one set are before all of those in the other pack.

When is `merge' simple?

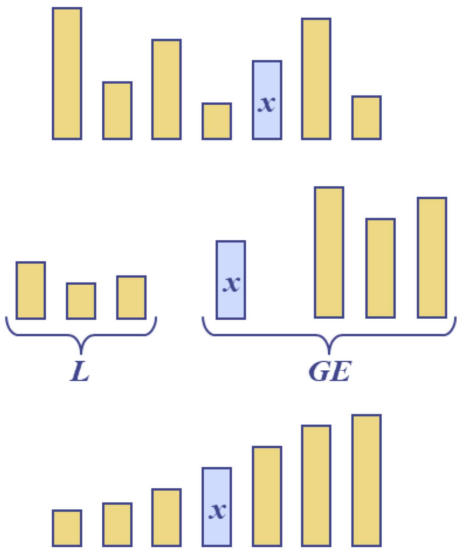
- When the lists A and B are sorted and known to be in disjoint ordered ranges
  - all of elements of A are smaller than all those of B
- If A and B are stored as consecutive sub-arrays, then merge needs no work at all:
  - Just "forget the boundary"

3

So merge is simple when it reduces to concatenate, because the first subarray is entirely before the second array.
If the subarrays are consecutive and contiguous, then the "merge" is actually not even an operation, and has no cost at all.
This is the key idea in quicksort.

Quick-Sort

- Quick-sort is a (randomized) sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick an element x (called pivot) and partition S into
    - L: elements less than x
      - *Have to be careful it is not empty*
    - GE: elements greater than or equal to x
    - *Pivot is often picked as a random element*
  - Recur: sort L and GE
  - Conquer: join L, GE



4

Put into sets for less than and greater than or equal.
There is a danger that the set L might end up as empty, and this needs to be avoided.
Then just recurse and merge as before.

So now the big question is how to do the "divide" or "partition" and the complexity of it.

Partition of lists (using extra workspace)

- Suppose store L, EG as separate structures (e.g. as arrays, vectors or lists)
- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning (or end) of the sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

5

If data is stored as linked lists then it is easy – we just walk through moving each element to new lists.
Handling each element is O(1) and there are n elements so overall is O(n).

But what about arrays,

"In-place" or "extra workspace"?

- For sorting algorithms (and algorithms in general) an important issue can be how much extra working space they need besides the space to store the input
- "**In-place**" means they only a "little" extra space (e.g. O(1)) is used to store data elements.
  - The input array is also used for output, and only need a few temporary variables
  - Exercise: check that bubble-sort is "in-place"
  - Previous "merge" used extra O(n) array (can be made in-place, but messy and so we ignore this option)

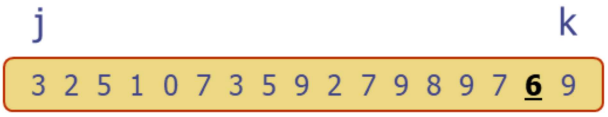
6

Firstly, a new concept of "in-place".
We will say a sorting algorithm is "in-place" if it only uses a small amount of extra space – meaning some amount that is O(1) and in particular is not as bad as n.
We already have this is the simple sorting algorithms such as bubble sort.
There only use a few extra variables, e.g. two for the index into the array, and then one for the swap.

However, the merge-sort used an extra workspace, which had to be linear size. If sorting very large arrays then this can be a problem.
Avoiding this in merge sort is possible but complex, and we will not cover it.
But in quicksort there is a nice way to do this.

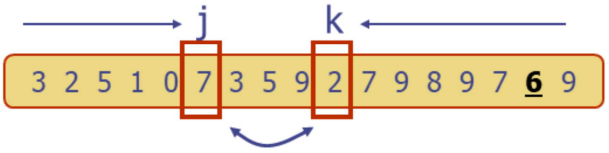
Partitioning arrays "in-place"

- Perform the partition using two indices to do a "2-way split" of S into L and E+G.



(pivot = 6)

- Repeat until j and k cross (or meet, until j >= k):
  - Scan j to the right until finding an element ≥ pivot.
  - Scan k to the left until finding an element < pivot.
  - Swap elements at indices j and k



The scans are not done in 'lock step' but independently work inwards. Do some examples and make sure you understand how and why this works!!

7

The core of quicksort is hence the in-place splitting or partition of an array. We consider the 2-way split, and so given a pivot value we want to end up with a subarray on the left that is all less than.
With the subarray on the right all being at least the size of the pivot. Not that it is fine and usually necessary for the pivot to move.
The trick is to maintain two indices or pointers into the array – starting at the outside and moving them towards the middle until they meet, and doing swaps as needed.

Partitioning

- Partitioning is actually more subtle
- Have to make sure that we make progress – specifically that neither of the partitions are empty
  - GE always contains the pivot, so is okay.
  - But L could be empty
    - So the pivot should not be a minimum element
- Or simpler is to do a "3-way" split:
  - check and move the pivot to the left and do a 3-way split into L, {pivot}, E+G
  - And so always ensure a one copy of the pivot (there could be many) is moved to the middle.

8

Exercise (offline)

- Write Java code to do the partition and check that it works on some examples
  - (It is only 10-20 lines of code, but will greatly help clarify the algorithm.)
- Investigate and explore some versions of how to do it effectively and efficiently

9

Quicksort Overall Implementation

With the previous (2-way) split:

```
public static void recQuickSort(int[] arr, int left, int right) {
  if (right - left <= 0) return;
  else {
    int border = partition(arr, left, right); // "crossing position"
    recQuickSort(arr, left, border);
    recQuickSort(arr, border+1, right);
  }
}
```

10

The "crossing position" is where the 'j' and 'k' meet and so is the position of the border between the L and "E+G" sets, so the recursive call acts on these sets.

Quicksort Overall Implementation

With a 3-way split (assuming just one copy of the pivot):

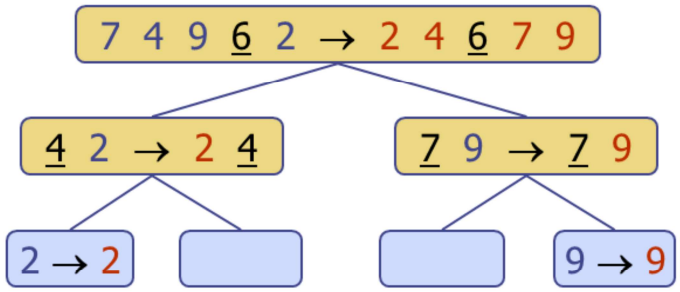
```
public static void recQuickSort(int[] arr, int left, int right) {
  if (right - left <= 0) return;
  else {
    int border = partition(arr, left, right); // pivot position
    recQuickSort(arr, left, border-1);
    recQuickSort(arr, border+1, right);
  }
}
```

11

IF a different modified partition were used to guarantee that the "Pivot ends in the middle", and we have a 3-way split into L, E, and G sets, then we can exclude the middle "E" from the further sorting.
This is mostly for conceptual completeness – the partition before gives a 2 way split.

Quick-Sort Tree

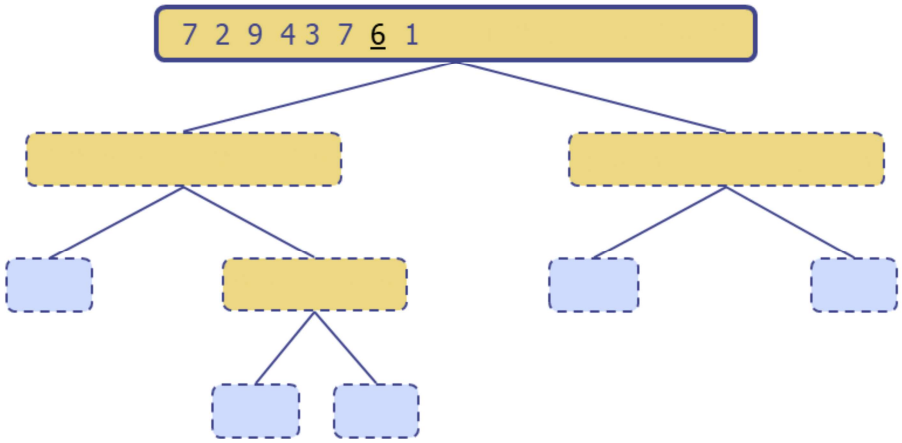
- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1
  - Example shows 3-way split (one copy of the pivot is `removed' on each partition).



12

Execution Example

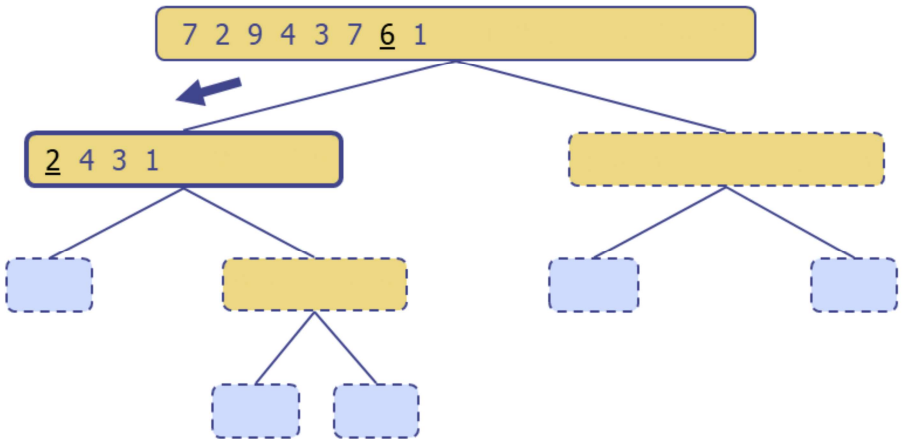
- Pivot selection



13

Execution Example (cont.)

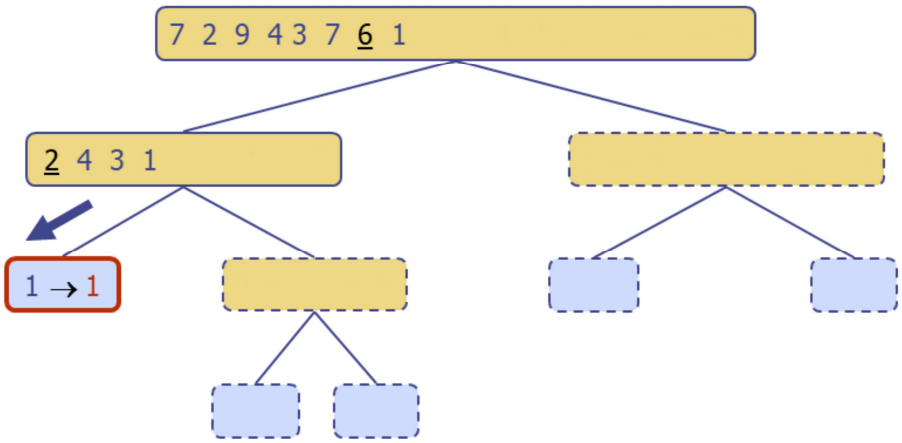
- Partition, recursive call, pivot selection



14

Execution Example (cont.)

- Partition, recursive call, base case

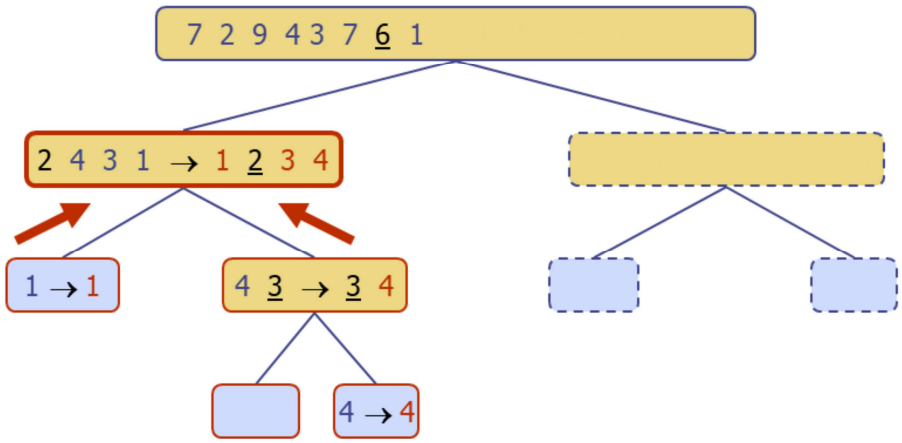


15

NOTE: this is assuming the partition is by copy to a separate array – as the in-place version on the start array would put '1' at the start of the array.

Execution Example (cont.)

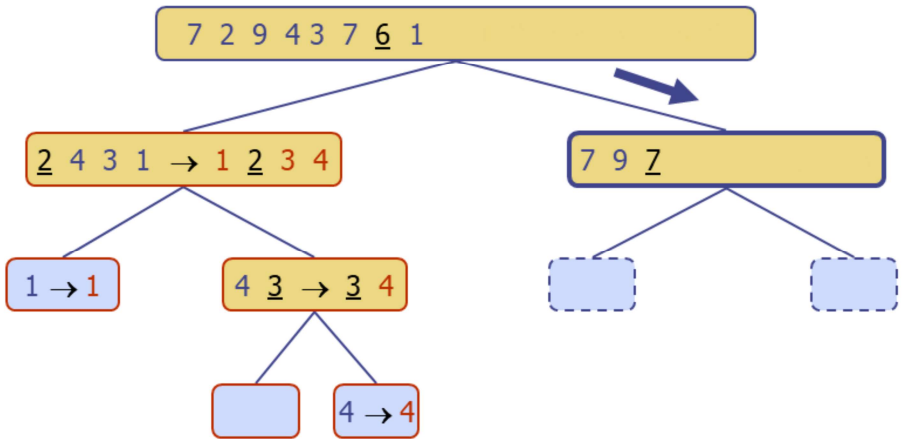
- Recursive call, …, base case, join



16

Execution Example (cont.)

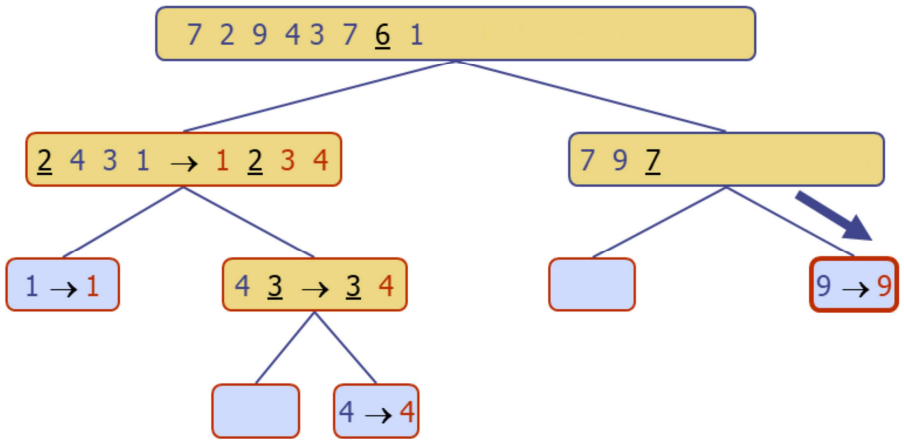
- Recursive call, pivot selection



17

Execution Example (cont.)

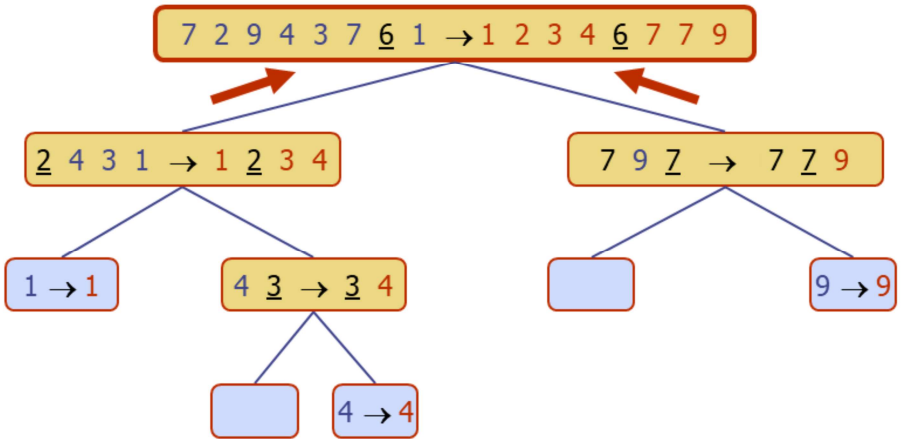
- Partition, …, recursive call, base case



18

Execution Example (cont.)

- Join, join



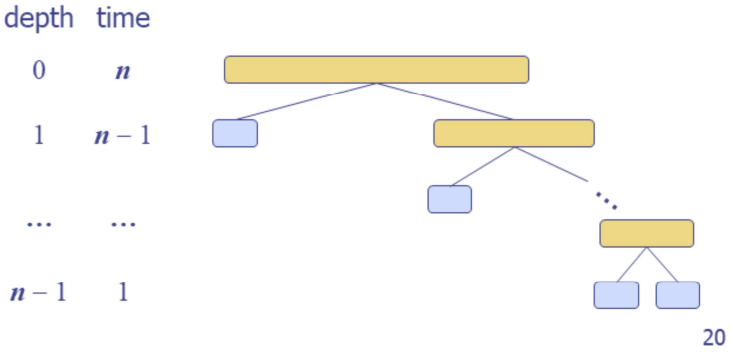
19

Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and E+G has size $n - 1$ and the other has size 1
  - (3-way split, so the pivot is always "removed", so that some progress is made)
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$



20

Note that it basically ends up somewhat similar to bubble-sort or selection sort

Best-case Running Time

- The best case for quick-sort occurs when the pivot is the median element
- The L and G parts are equal – the sequence is split in halves, like in merge sort
- Thus, the best-case running time of quick-sort is $O(n \log n)$

21

Average-case Running Time

- The average case for quick-sort: half of the times, the pivot is roughly in the middle
- Thus, the average-case running time of quick-sort is $O(n \log n)$ again
  - Detailed proof in textbook
- Basic idea: suppose that the pivot is always in the middle third
  - The both L and E+G are size at least n/3, and at most 2n/3 = n / (3/2)
  - So height of the call tree is the number of times we can divide n by (3/2)

22

Motivations for quicksort

- Why do we select a pivot? I.e. what advantages might quicksort ever have over mergesort?
  - Because it can be done "in-place"
    - Uses a small amount of workspace
    - Because the "merge" step is now a lot easier!!
  - The "split" is more complicated, and the merge "much" easier – but turns out that the quick-sort split is easier to do in-place than the merge-sort merge

23

Minimum Expectations

- For both merge- and quicksort:
  - know the algorithm and how it works on examples
  - know and be able to justify/prove their big-Oh behaviours

24