CS211FZ: Data Structures and Algorithms II

2- Divide and Conquer – Quicksort

LECTURER: CHRIS ROADKNIGHT

CHRIS.ROADKNIGHT@MU.IE

Introduction

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. (Quicksort's worst-case scenario occurs when the algorithm consistently picks the worst possible pivot at each recursive step)

Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average:

Its expected running time is $\Theta(n \log n)$, and the constant factors hidden in the $\Theta(n \log n)$ notation are quite small.*

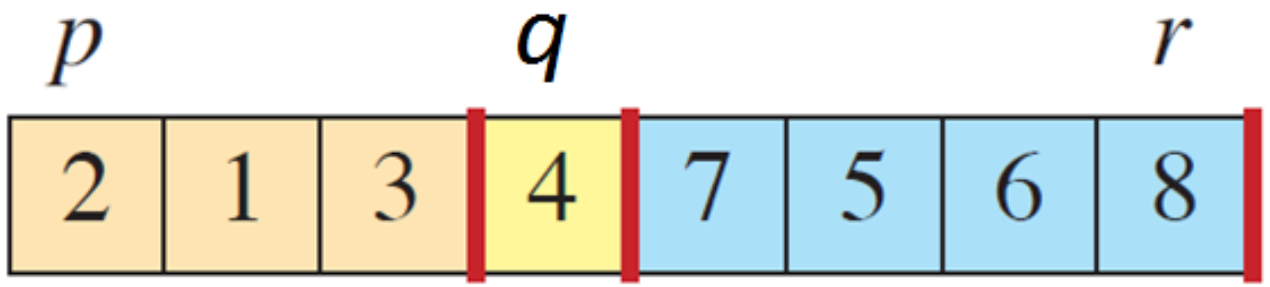
Quicksort, like merge sort, applies the divide-and-conquer method.

*when multiple solutions have the same growth functions constant factors matter

Description of Quicksort (1)

Here is the three-step divide-and-conquer process for sorting a subarray $A[p:r]$:

Divide by partitioning (rearranging) the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ (the *low side*) and $A[q+1:r]$ (the *high side*) such that each element in the low side of the partition is less than or equal to the *pivot* $A[q]$, which is, in turn, less than or equal to each element in the high side. Compute the index q of the pivot as part of this partitioning procedure.



Description of Quicksort (2)

Conquer by calling quicksort recursively to sort each of the subarrays

$$A[p: q-1] \quad \text{and} \quad A[q+1: r].$$

Combine by doing nothing: because the two subarrays are already sorted, no work is needed to combine them. All elements in $A[p: q-1]$ are sorted and less than or equal to $A[q]$, and all elements in $A[q+1: r]$ are sorted and greater than or equal to the pivot $A[q]$. Thus, the entire subarray $A[p: r]$ is sorted!

Implementation (1)

The QUICKSORT procedure implements quicksort. To sort an entire n-element array $A[1:n]$, the initial call is QUICKSORT(A, 1, n).

```
QUICKSORT(A, p, r)
1  if p < r
2        // Partition the subarray around the pivot, which ends up in A[q].
3        q = PARTITION(A, p, r)
4        QUICKSORT(A, p, q − 1)  // recursively sort the low side
5        QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Implementation (2)

```
PARTITION(A, p, r)
1  x = A[r]                                  // the pivot
2  i = p − 1                                 // highest index into the low side
3  for j = p to r − 1                        // process each element other than the pivot
4      if A[j] ≤ x                           // does this element belong on the low side?
5          i = i + 1                         // index of a new slot in the low side
6          exchange A[i] with A[j]           // put this element there
7  exchange A[i + 1] with A[r]               // pivot goes just to the right of the low side
8  return i + 1                              // new index of the pivot
```

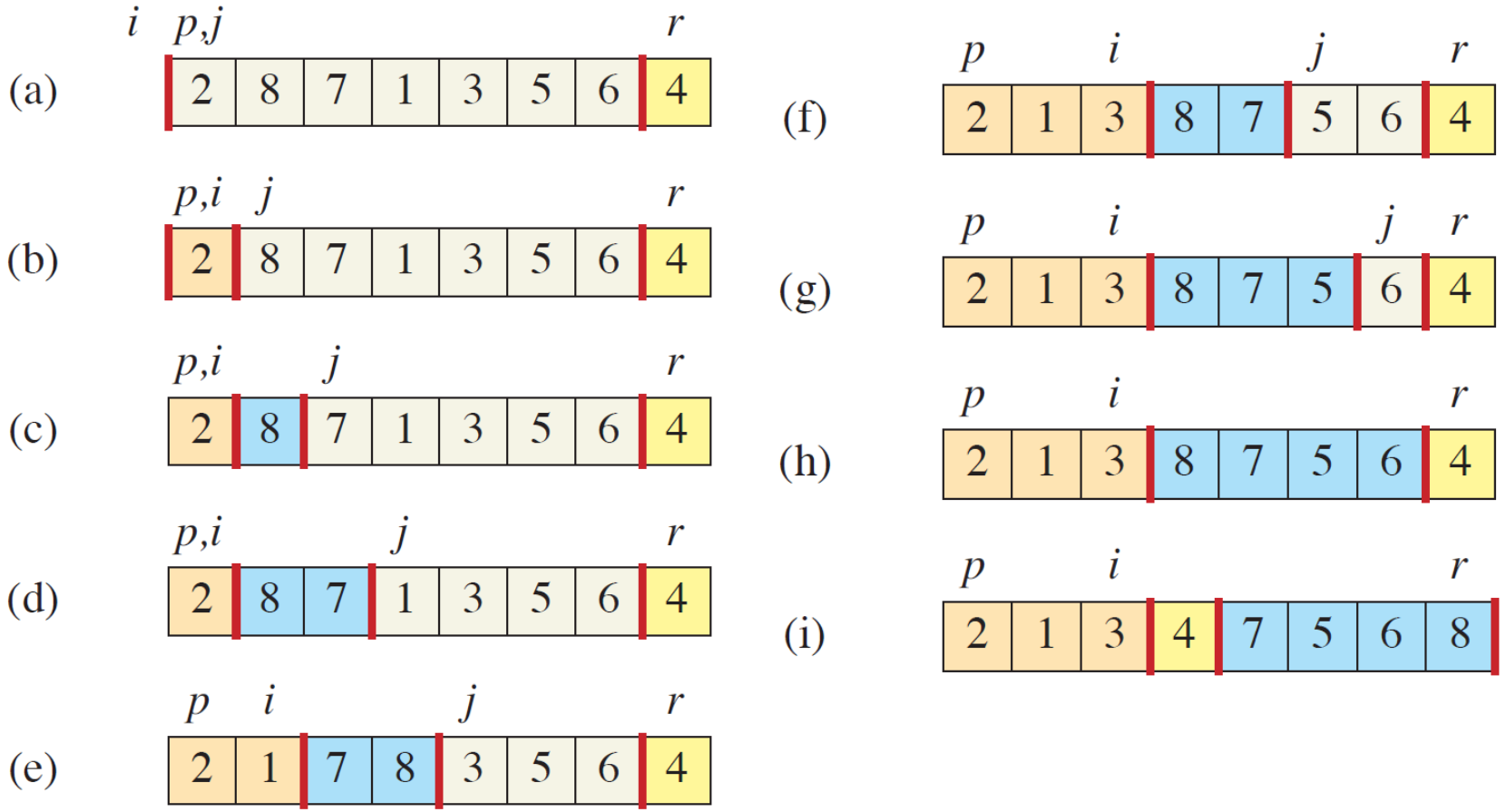
The operation of PARTITION on a sample array

Array entry $A[r]$ becomes the pivot element x.

Tan array elements all belong to the low side of the partition, with values at most x.

Blue elements belong to the high side, with values greater than x.

White elements have not yet been put into either side of the partition, and the yellow element is the pivot x.



An explanation of the previous slide

(**a**) The initial array and variable settings. None of the elements have been placed into either side of the partition.

(**b**) The value 2 is “swapped with itself” and put into the low side.

(**c**)–(**d**) The values 8 and 7 are placed into to high side.

(**e**) The values 1 and 8 are swapped, and the low side grows.

(**f**) The values 3 and 7 are swapped, and the low side grows.

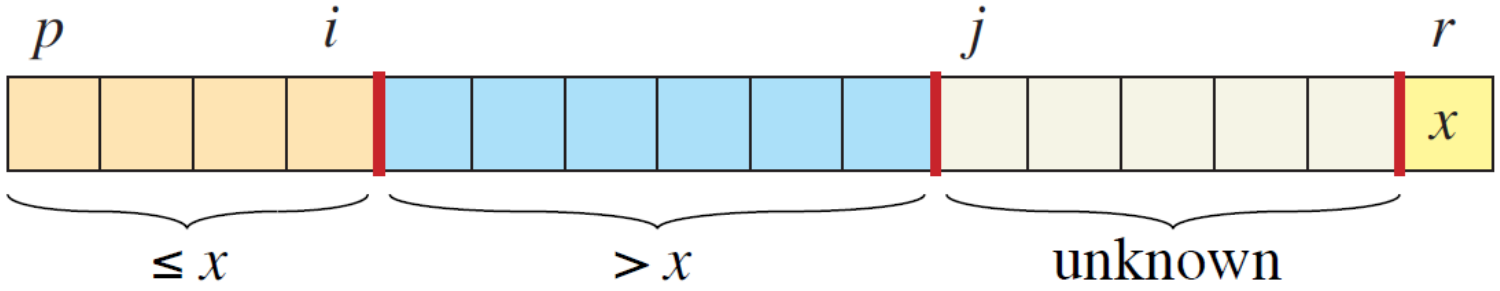
(**g**)–(**h**) The high side of the partition grows to include 5 and 6, and the loop terminates.

(**i**) Line 7 swaps the pivot element so that it lies between the two sides of the partition, and line 8 returns the pivot’s new index.

Loop Invariant for procedure PARTITION

At the beginning of each iteration of the loop of lines 3–6, for any index k,

1. If $p \leq k \leq i$, then $A[k] \leq x$ (the tan region of the following figure);
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$ (the blue region);
3. If $k = r$, then $A[k] = x$ (the yellow region).



Initialization

Prior to the first iteration of the loop, we have $i = p - 1$ and $j = p$.

Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied.

The assignment in line 1 satisfies the third condition.

Maintenance

As Figure in the next slide shows, we consider two cases, depending on the outcome of the test in line 4.

Part (**a**) shows what happens when $A[j] > x$: the only action in the loop is to increment j. After j has been incremented, the **second condition** holds for $A[j-1]$ and all other entries remain unchanged.

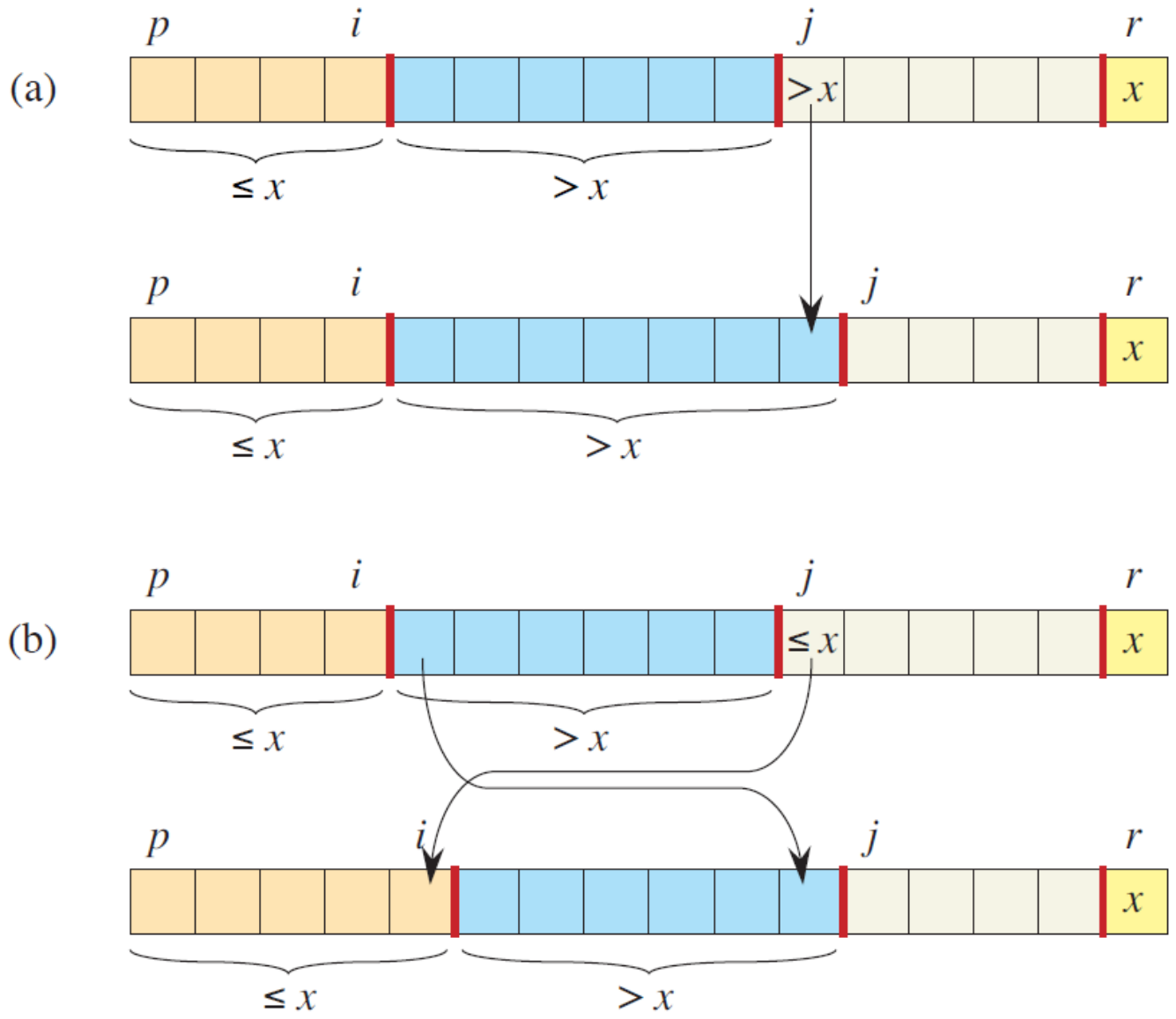
The part (**b**) in the next slide shows what happens when $A[j] \leq x$: the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[i] \leq x$, and **condition 1** is satisfied.

Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

The two cases for one iteration of procedure PARTITION.

(**a**) If $A[j] > x$, the only action is to increment j, which maintains the loop invariant.

(**b**) If $A[j] \leq x$, index i is incremented, $A[i]$and $A[j]$are swapped, and then j is incremented. Again, the loop invariant is maintained.



Termination (1)

Since the loop makes exactly $r - p$ iterations, it terminates, whereupon $j = r$. At that point, the unexamined subarray $A[j : r - 1]$ is empty, and every entry in the array belongs to one of the other three sets described by the invariant. Thus, the values in the array have been partitioned into three sets:

1. those less than or equal to x (the low side),
2. those greater than x (the high side),
3. and a singleton set containing x (the pivot).

Termination (2)

The final two lines of PARTITION finish up by swapping the pivot with the leftmost element greater than x , thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index.

The output of PARTITION now satisfies the specifications given for the divide step.

In fact, it satisfies a slightly stronger condition: after line 3 of QUICKSORT, $A[q]$ is strictly less than every element of $A[q+1:r]$.

Performance of quicksort

The running time of quicksort depends on how balanced each partitioning is, which in turn depends on which elements are used as pivots.

If the two sides of a partition are about the same size - the partitioning is balanced - then the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

Worst-case partitioning (1)

The worst-case behaviour for quicksort occurs when the partitioning produces one subproblem with $n - 1$ elements and one with 0 elements.

Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\theta(n)$ time. Since the recursive call on an array of size 0 just returns without doing anything, $T(0) = \theta(1)$, and the recurrence for the running time is

$$T(n) = T(n-1) + T(0) + \theta(n) \ = T(n-1) + \theta(n)$$

Intuitively, if we sum the costs incurred at each level of the recursion, we get $\Theta(n^2)$.

Worst-case partitioning (2)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$.

Therefore, the worst-case running time of quicksort is no better than that of insertion sort.

Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted — a common situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor (n-1)/2 \rfloor \leq n/2$ and one of size $\lceil (n-1)/2 \rceil - 1 \leq n/2$.

In this case, quicksort runs much faster.

The recurrence for the running time is then

$$T(n) = 2T(n/2) + \Theta(n),$$

this recurrence has the solution $T(n) = \Theta(n \lg n)$.

Balanced partitioning (average-case) (1)

The average-case running time of quicksort is much closer to the best case than to the worst case.

By appreciating how the balance of the partitioning affects the recurrence describing the running time, we can gain an understanding of why.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first look seems quite unbalanced. We then obtain the recurrence

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \theta(n)$$

on the running time of quicksort.

Sorting Summary

Merge Sort: Divide-and-conquer algorithm; splits array into halves, recursively sorts, then merges. Stable, consistent O(n log n) time complexity, but requires O(n) extra space. Best for large, linked-list data.

Quicksort: Divide-and-conquer; picks a pivot, partitions array, recursively sorts subarrays. Unstable, average O(n log n) time, worst $O(n^2)$ if pivot is poorly chosen. In-place, fast in practice.

Insertion Sort: Iterative; builds sorted array one element at a time by inserting into correct position. Stable, $O(n^2)$ time, but O(1) space. Excellent for small or nearly sorted datasets.