BM267 - Introduction to Data Structures

7. Quicksort

Ankara University

Computer Engineering Department

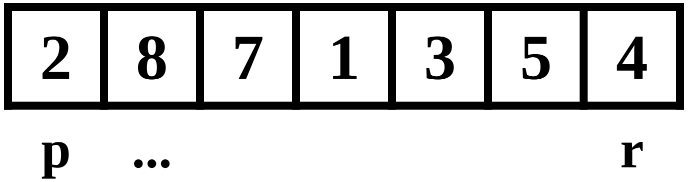
Bulent Tugrul

Quicksort

- Quicksort uses a **divide-and-conquer** strategy
    - A recursive approach
    - The original problem partitioned into simpler subproblems.
    - Each sub problem considered independently.
- Subdivision continues until sub problems are simple enough to be solved directly.

Quicksort - Example 1

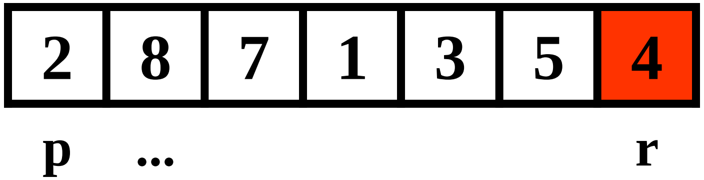
How to partition an array A[p,r]:



2	**8**	**7**	**1**	**3**	**5**	**4**
p	**...**					**r**

Choose some element called a **pivot**

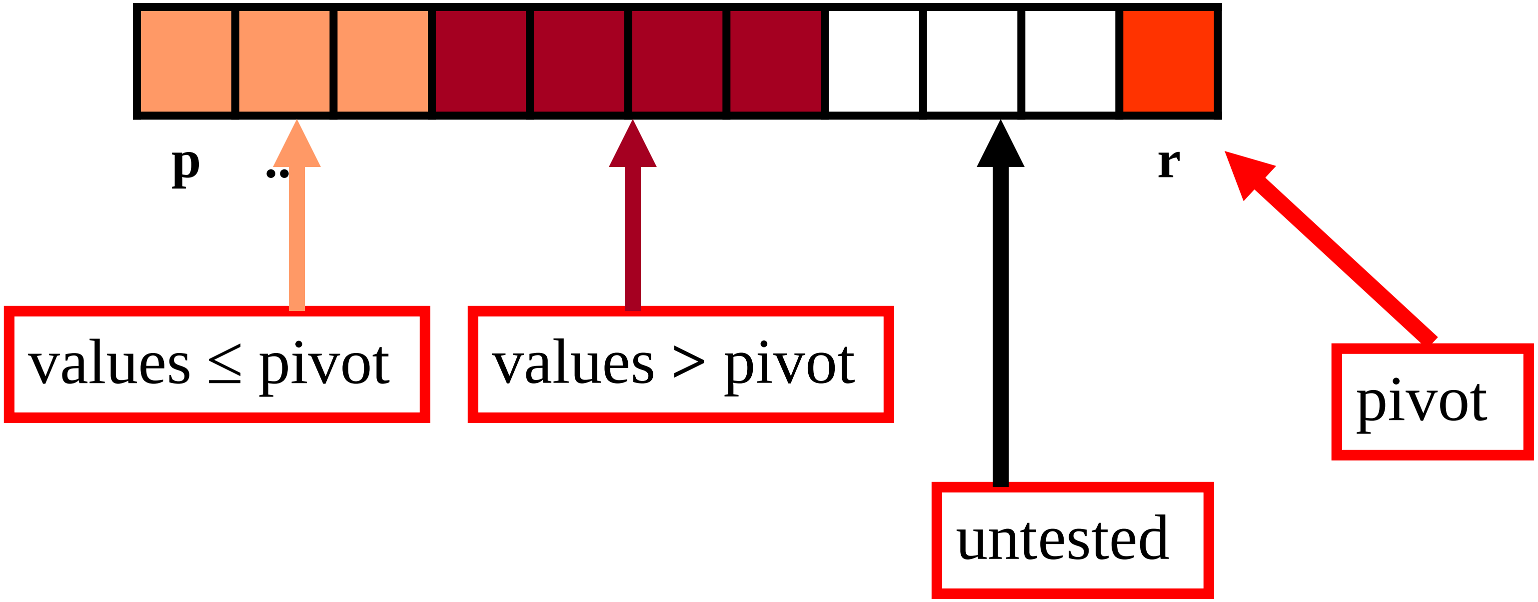
(Usually the rightmost or leftmost element)



2	**8**	**7**	**1**	**3**	**5**	🟥 **4**
p	**...**					**r**

Quicksort - Example 1

The array will have three sections, plus the pivot element



values ≤ pivot

values > pivot

untested

pivot

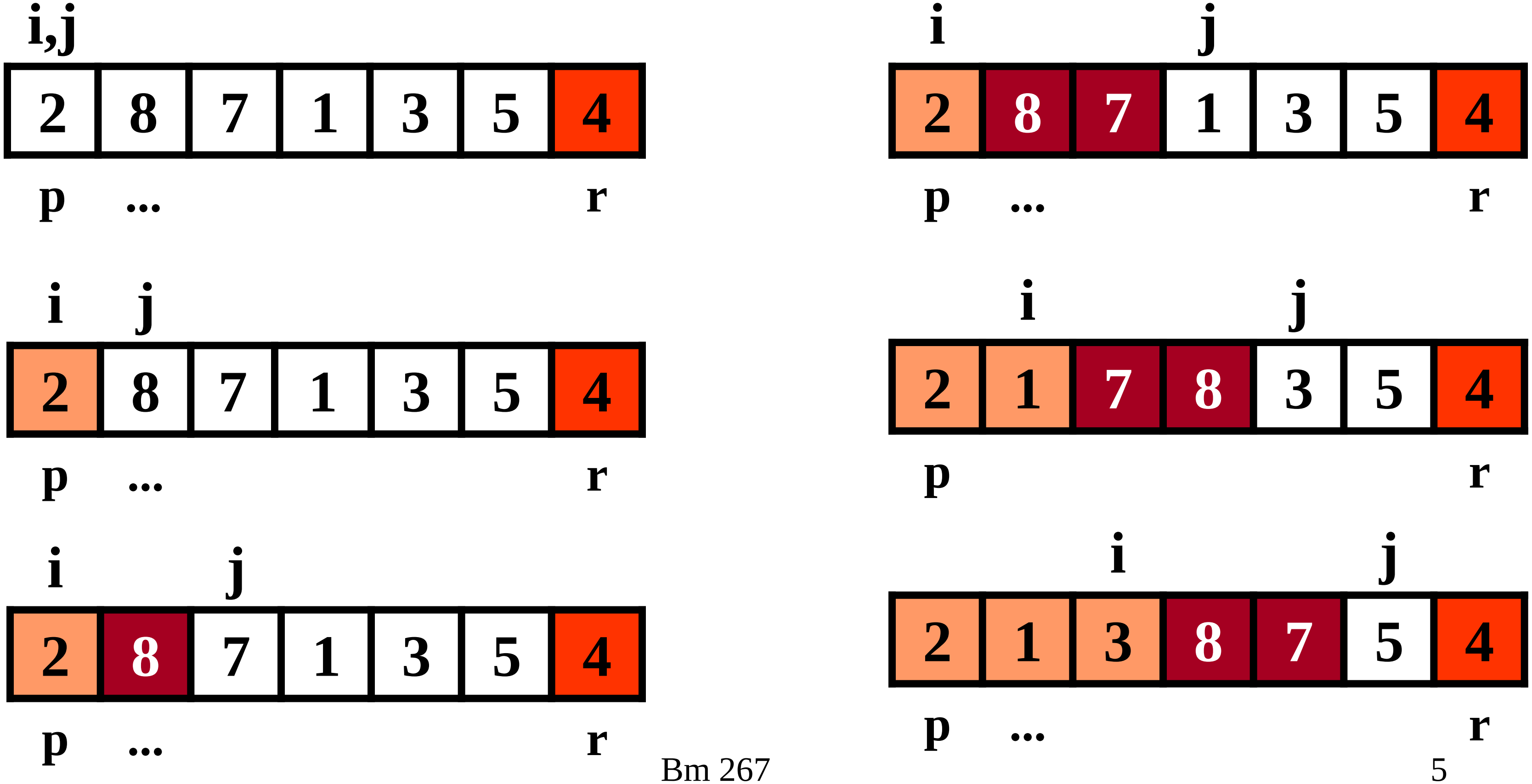
i will point to the high end of the 'smaller' sublist

j will point to the high end of the 'larger' sublist

Quicksort - Example 1

Perform a sequence of exchanges so that

- All elements that are less than pivot go to left and
- All elements that are greater than the pivot go to right.



i,j
2 8 7 1 3 5 4
p ... r

i j
2 8 7 1 3 5 4
p ... r

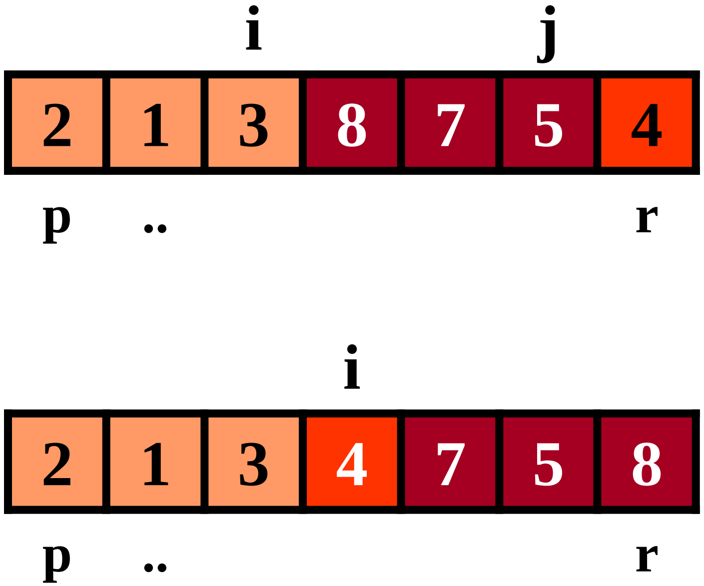
i j
2 8 7 1 3 5 4
p ... r

i j
2 8 7 1 3 5 4
p ... r

i j
2 1 7 8 3 5 4
p r

i j
2 1 3 8 7 5 4
p ... r

Quicksort - Example 1



(exchange element i+1
with the pivot)

- This operation divides the array into two smaller sub arrays,
- Each of which may then be sorted independently in the *same* way.

Quicksort

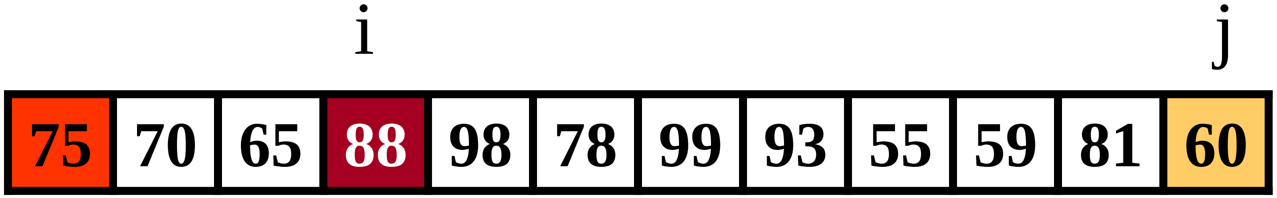
```
Quicksort (A[p..q])
If the array has 0 or 1 elements,
        then return. // the array is sorted
else do:
        Pick an element in the array to use as the pivot.
        Split the remaining elements into two disjoint groups:
        –    "Smaller"  elements  not greater than the pivot,     A[p...m-1]
        –    "Larger"   elements  greater than  pivot,            A[m+1… r]

 Return the array rearranged as:
   Quicksort(A[p...m-1]),
   pivot,
   Quicksort(A[m+1… r]).
```

Quicksort- Example 2

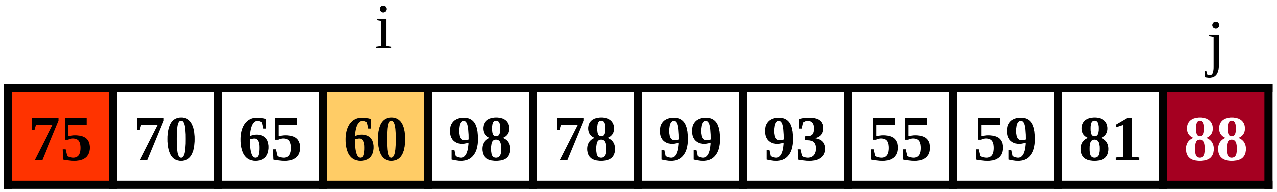
Here is a slightly different partitioning algorithm:

- Select, arbitrarily, the first element, 75, as pivot.
- Search from right for the first element ≤ 75, (which is 60)
- Search from left for the first element > 75, (which is 88)



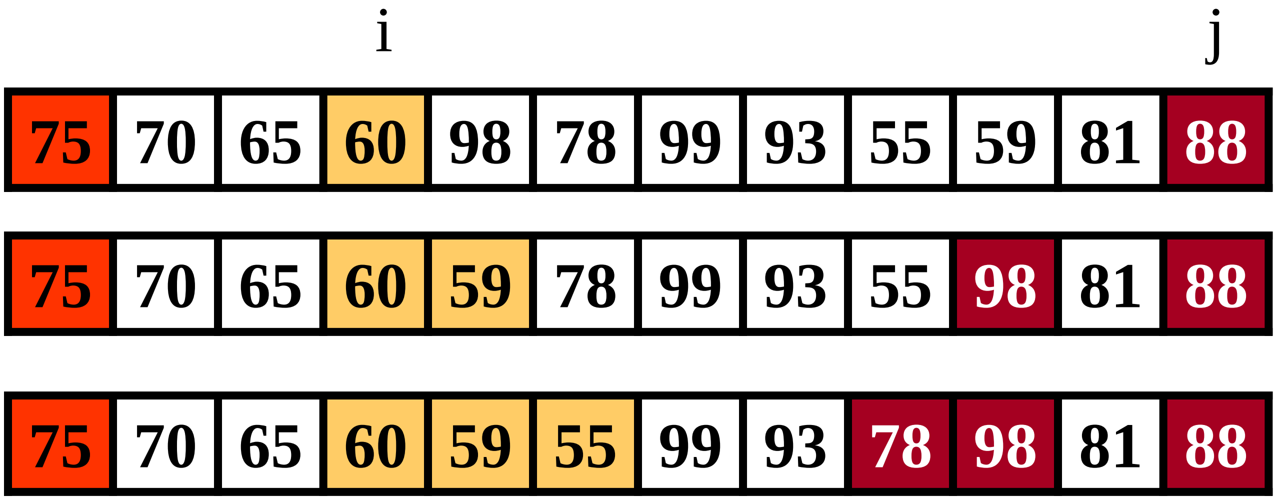
75	**70**	**65**	**88**	**98**	**78**	**99**	**93**	**55**	**59**	**81**	**60**

- Swap these two elements, and then repeat this process



75	**70**	**65**	**60**	**98**	**78**	**99**	**93**	**55**	**59**	**81**	**88**

Quicksort- Example 2



75	70	65	60	98	78	99	93	55	59	81	88
75	70	65	60	59	78	99	93	55	98	81	88
75	70	65	60	59	55	99	93	78	98	81	88

When done, exchange the rightmost element in group "Smaller" with the pivot

55	70	65	60	59	75	99	93	78	98	81	88

75 is now placed appropriately.

Need to sort sublists on either side of 75.

Quicksort

```
int partition(Item a[], int l, int r);
void quicksort(Item a[], int l, int r)
  { int m;
    if (r <= l) return;
    m = partition(a, l, r);
    quicksort(a, l, m-1);
    quicksort(a, m+1, r);
  }
```

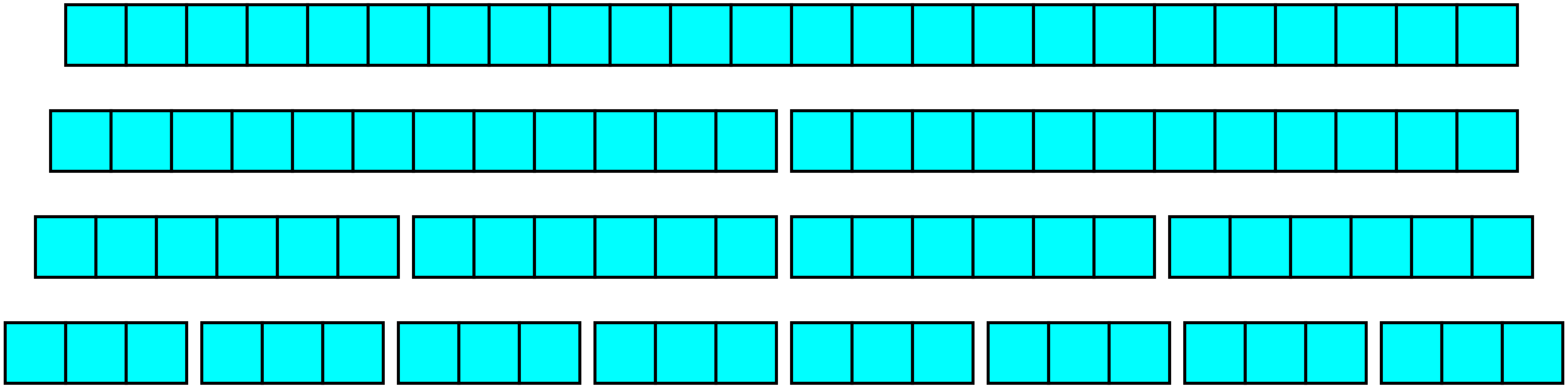
```
int partition(Item a[], int l, int r){
      int i = l-1, j = r; Item v = a[r];
      for (;;){
            while (less(a[++i], v)) ;
            while (less(v, a[--j])) if (j == l) break;
            if (i >= j) break;
            exch(a[i], a[j]);
      }
      exch(a[i], a[r]);
      return i;  }
```

Quicksort - Analysis

Best Case

- If the pivot results in sub arrays of approximately the same size.
- $T(n) = 2T(n/2) + n - 1$

  $= n \log_2 n$



Quicksort - Analysis

Best case $O(n \log_2 n)$

- We cut the array size in half each time
- So the depth of the recursion in **$\log_2 n$**
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the best and average cases, quicksort has time complexity **$O(n \log_2 n)$**

Quicksort - Analysis

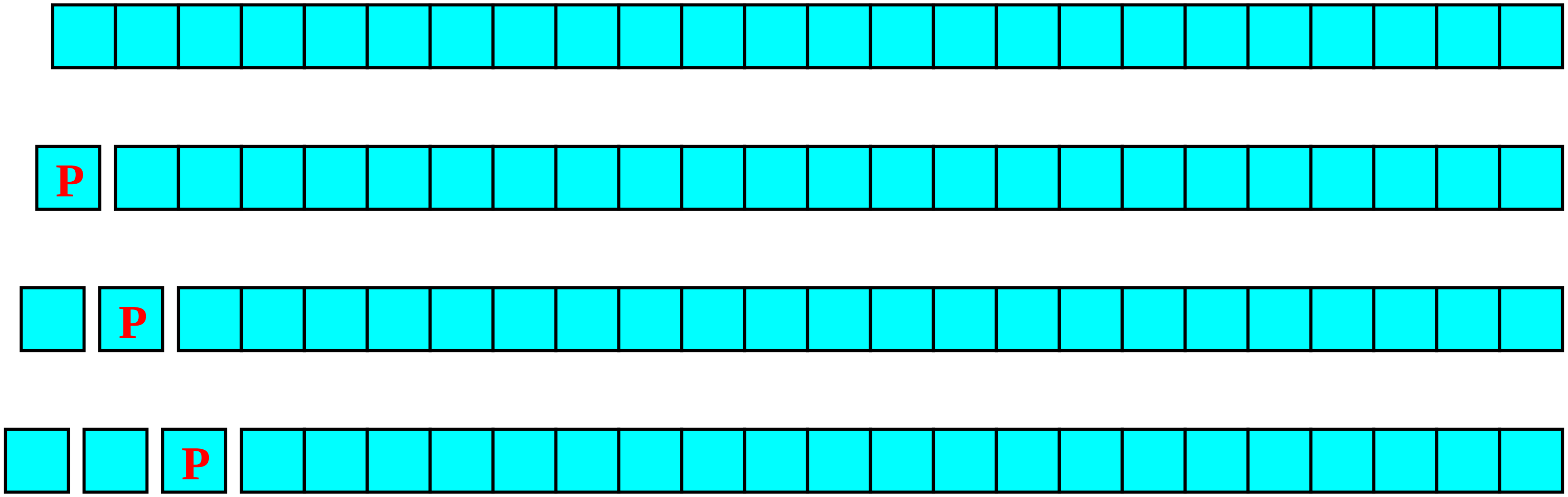
O(n^2) worst-case

- List already ordered (either way)
- Then the pivot element is the largest or smallest element: one of the sublists is almost always empty.
- Partitioning always divides the size n array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length **n-1** part, containing everything else

Quicksort - Analysis

Worst-case

P = Pivot element



- We don’t recur on the zero-length part
- Recurring on the length **n-1** part requires (in the worst case) recurring to depth **n-1**

Quicksort - Analysis

- If the array is already sorted, Quicksort is terrible: **$O(n^2)$**
- However, Quicksort is *on the average* **$O(n \log_2 n)$**
- The constants are so good that Quicksort is generally the fastest algorithm known
- Most real-world sorting is done by Quicksort

Quicksort - Possible Improvements

- Almost anything you can try to “improve” Quicksort will actually slow it down
- One *good* idea is to switch to a different sorting method when the subarrays get small (say, 10 or 12)
  - Quicksort has too much overhead for small array sizes
- For large arrays, it *might* be a good idea to check beforehand if the array is already sorted

Quicksort - Possible Improvements

- Often the list to be sorted is already partially ordered.
- An *arbitrary pivot* gives a poor partition for nearly sorted lists
- In these cases, virtually all the elements either go into the group "Smaller" or to the "Larger", all through the recursive calls.
- Quicksort takes quadratic time to do essentially nothing at all.
- There are better methods for selecting the pivot, such as the **median-of-three rule:**

  Select the median of the first, middle, and last elements in each sublist as the pivot.
- Median-of-three rule will select a pivot closer to the middle of the sublist than will the “first-element” rule.

Quicksort - Possible Improvements

```
#define M 10
void quicksort(Item a[], int l, int r)
  { int i;
    if (r-l <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    compexch(a[l], a[r-1]);
      compexch(a[l], a[r]);
        compexch(a[r-1], a[r]);
    i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
```

Quicksort - Possible Improvements

```
void sort(Item a[], int l, int r)
{
    quicksort(a, l, r);
    insertion(a, l, r);
}
```