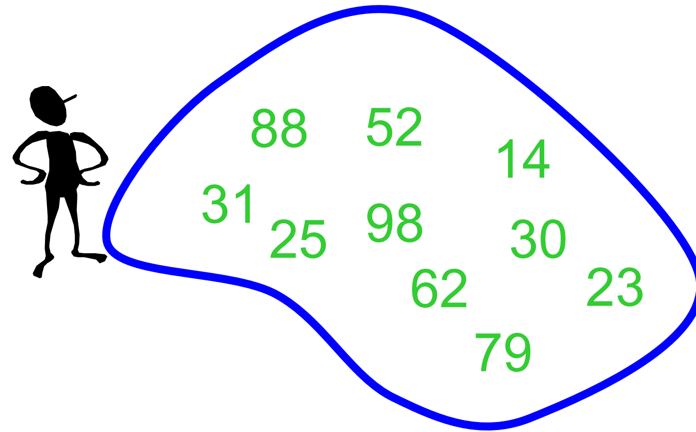
Algorithms Analysis

Quick sort

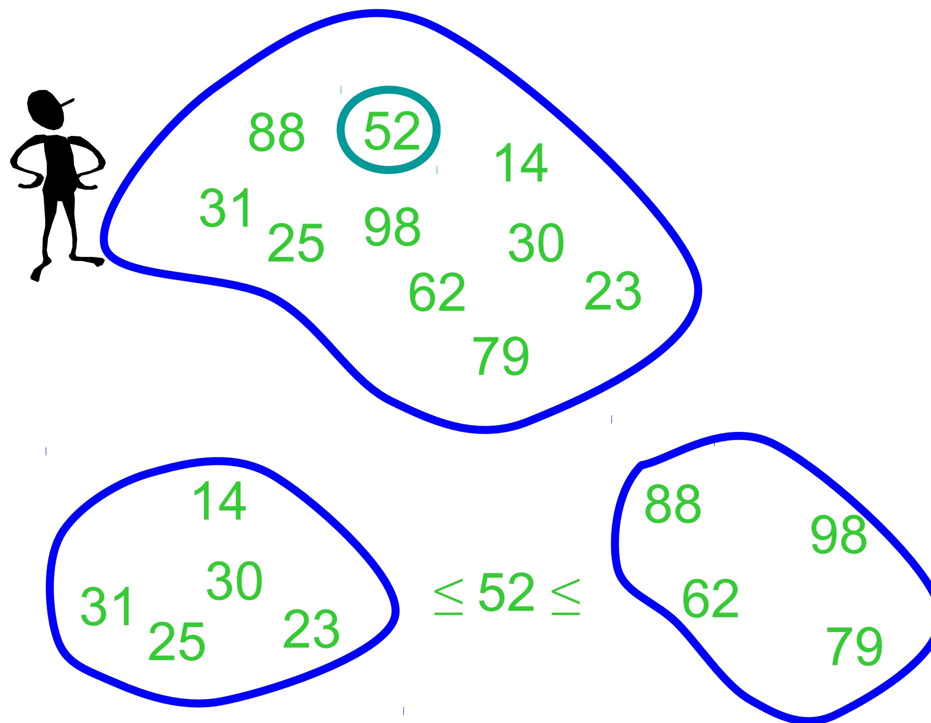
Quick Sort



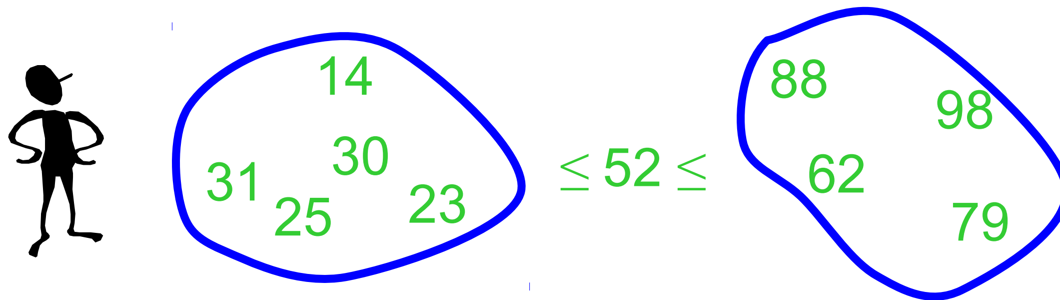
Divide and Conquer

Quick Sort

Partition set into two using randomly chosen pivot



Quick Sort



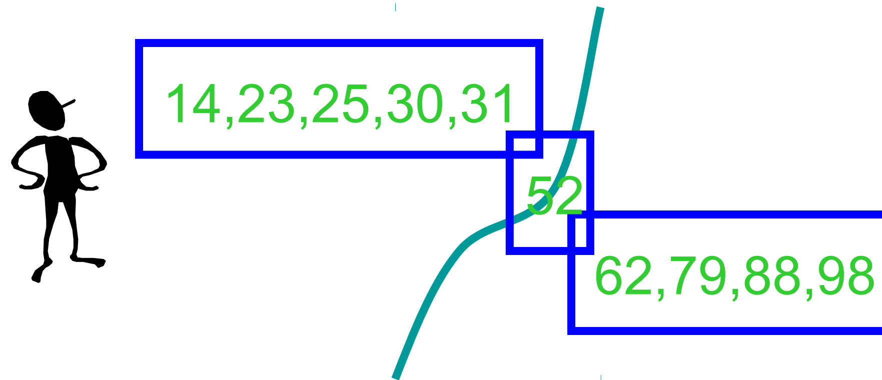
sort the first half.

14,23,25,30,31

sort the second half.

62,79,98,88

Quick Sort



Glue pieces together.

14,23,25,30,31,52,62,79,88,98

Quicksort

- Quicksort pros [advantage]:
  - Sorts in place
  - Sorts $O(n \lg n)$ in the average case
  - Very efficient in practice , it's quick

- Quicksort cons [disadvantage]:
  - Sorts $O(n^2)$ in the worst case
  - And the worst case doesn't happen often … sorted

Quicksort

- Another divide-and-conquer algorithm:
- *Divide*: $A[p \ldots r]$ is partitioned (rearranged) into two nonempty subarrays $A[p \ldots q\text{-}1]$ and $A[q\text{+}1 \ldots r]$ s.t. each element of $A[p \ldots q\text{-}1]$ is less than or equal to each element of $A[q\text{+}1 \ldots r]$. Index q is computed here, called **pivot**.
- *Conquer*: two subarrays are sorted by recursive calls to quicksort.
- *Combine*: unlike merge sort, no work needed since the subarrays are sorted in place already.

Quicksort

- The basic algorithm to sort an array A consists of the following four easy steps:
  - If the number of elements in A is 0 or 1, then return
  - Pick any element ***v*** in A. This is called the ***pivot***
  - Partition A-{***v***} (the remaining elements in A) into two disjoint groups:
    - $A_1 = \{\boldsymbol{x} \in A-\{\boldsymbol{v}\} \mid \boldsymbol{x} \leq \boldsymbol{v}\}$, and
    - $A_2 = \{\boldsymbol{x} \in A-\{\boldsymbol{v}\} \mid \boldsymbol{x} \geq \boldsymbol{v}\}$
  - return
    - { quicksort(A_1) followed by ***v*** followed by quicksort(A_2)}

Quicksort

- Small instance has $n \le 1$
  - Every small instance is a sorted instance
- To sort a large instance:
  - select a pivot element from out of the $\boldsymbol{n}$ elements
- Partition the $\boldsymbol{n}$ elements into 3 groups left, middle and right
  - The middle group contains only the pivot element
  - All elements in the left group are ≤ pivot
  - All elements in the right group are ≥ pivot
- Sort left and right groups recursively
- Answer is sorted left group, followed by middle group followed by sorted right group

Quicksort Code

```
P: first element
r: last element
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r)
        Quicksort(A, p  , q-1)
        Quicksort(A, q+1      , r)
    }
}
```

- Initial call is **Quicksort**(*A*, 1, *n*), where *n* in the length of *A*

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the **index** of the "pivot" element separating the two subarrays

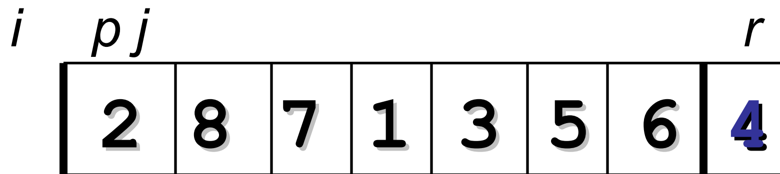
Partition Code

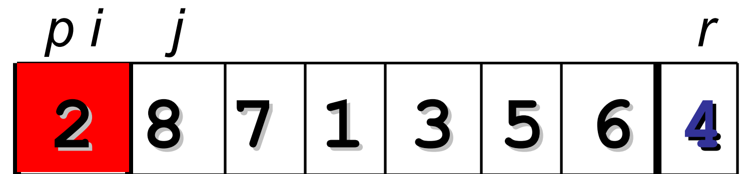
```
Partition(A, p, r)
{
     x = A[r]                  // x is pivot
     i = p - 1
     for j = p to r - 1
     {
          do if A[j] <= x
               then
               {
                  i = i + 1
                  exchange A[i] ↔ A[j]
               }
     }
     exchange A[i+1] ↔ A[r]
     return i+1
}
```

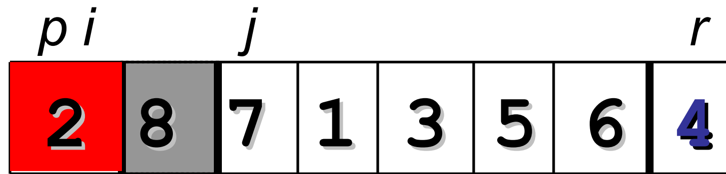
partition() runs in O(n) time

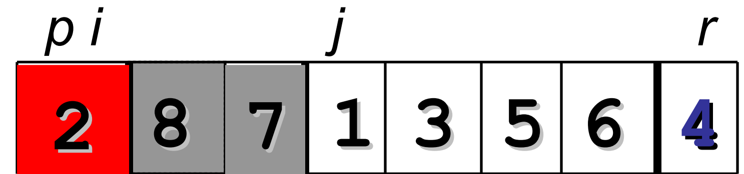
Partition Example

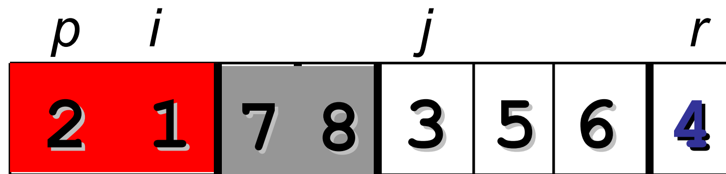
- $A = \{2, 8, 7, 1, 3, 5, 6, 4$

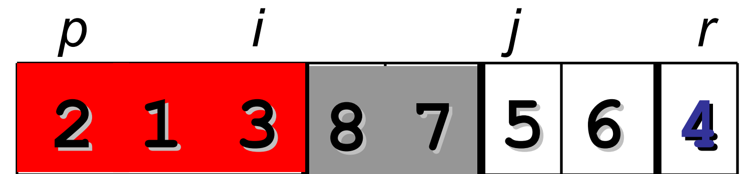


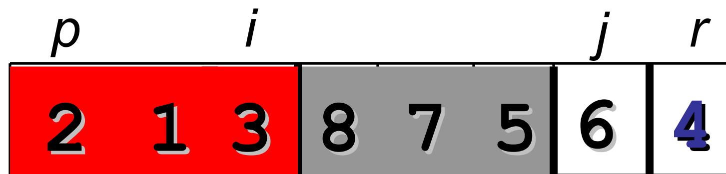


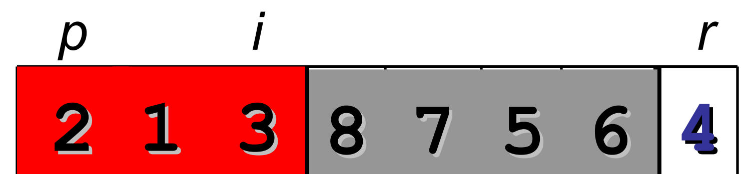


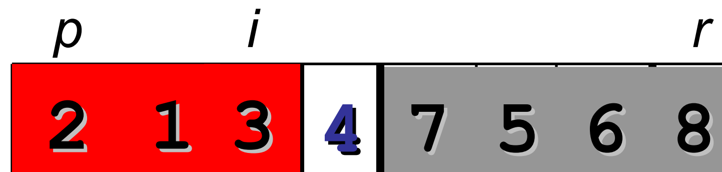












Partition Example Explanation

- Red shaded elements are in the first partition with values $\leq x$ (pivot)
- Gray shaded elements are in the second partition with values $\geq x$ (pivot)
- The unshaded elements have no yet been put in one of the first two partitions
- The final white element is the pivot

Review: Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is balanced

Summary: Quicksort

- In worst-case, efficiency is $\Theta(n^2)$
  - But easy to avoid the worst-case
- On average, efficiency is $\Theta(n \lg n)$
- Better space-complexity than mergesort.
- In practice, runs fast and widely used