QUICK SORT

COP 3502

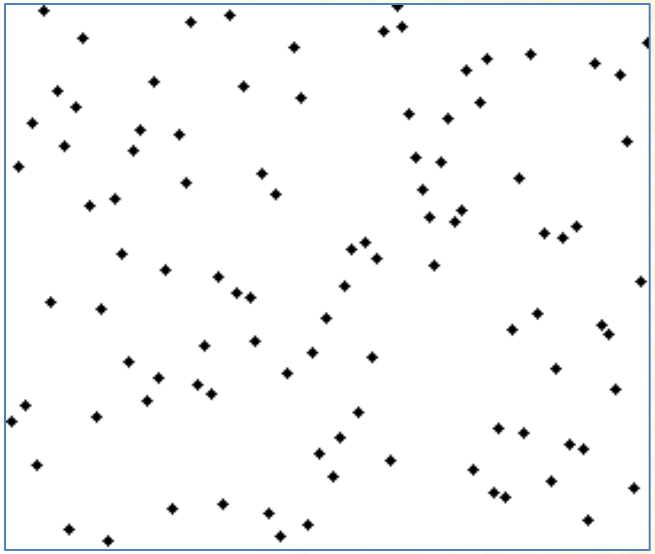
Motivation of Sorting

- Sorting algorithms contain interesting and important ideas for code optimization as well as algorithm design.

Merge Sort

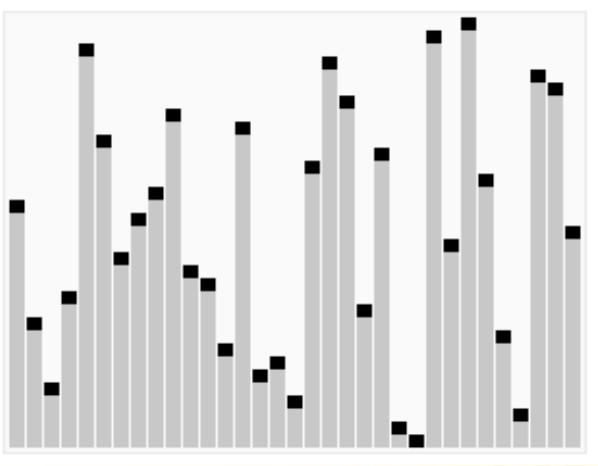
- Last time we talked about Merge Sort
  - Recursively calls:
    MergeSort(1st half of list)
    MergeSort(2nd half of list)

    Then Merges results



Quick Sort

- This probably the most common sort used in practice, since it is usually the quickest in practice.



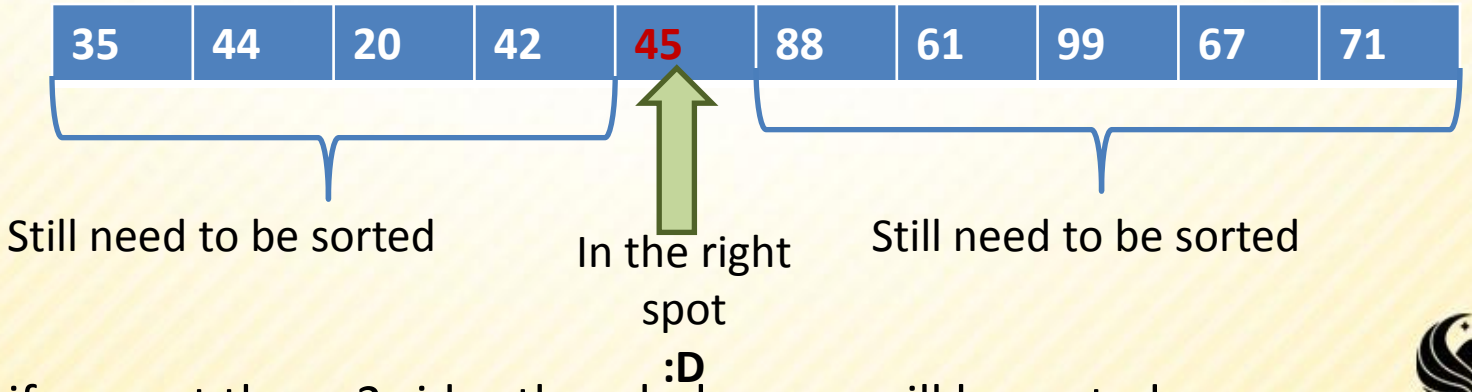
- It uses the idea of a partition, without using an additional array, and recursion to achieve this efficiency.

QuickSort

- Basically the partition works like this:
  - Given an array of n values you must randomly pick an element in the array to partition by.

88	35	44	99	71	20	45	42	67	61

  - Once you have picked this value, compare all of the rest of the elements to this value.
    - If they are ***greater***, put them to the ***"right"*** of the partition element.
    - If they are ***less***, put them to the ***"left"*** of the partition element.



    - So if we sort those 2 sides the whole array will be sorted.

QuickSort

- Thus, similar to MergeSort, we can use a partition to break the sorting problem into 2 smaller sorting problems.

  - QuickSort at a general level:

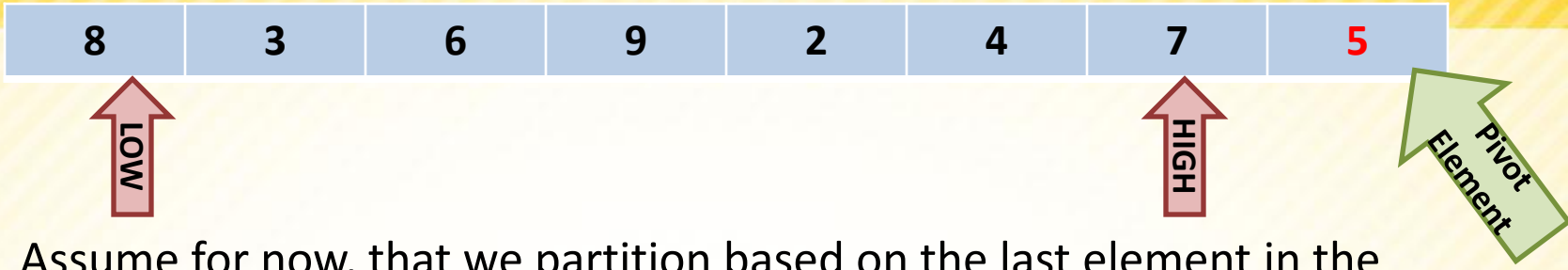
    1) Partition the array with respect to a random element.
    2) Sort the left part of the array, using Quick Sort.
    3) Sort the right part of the array, using Quick Sort.

QuickSort

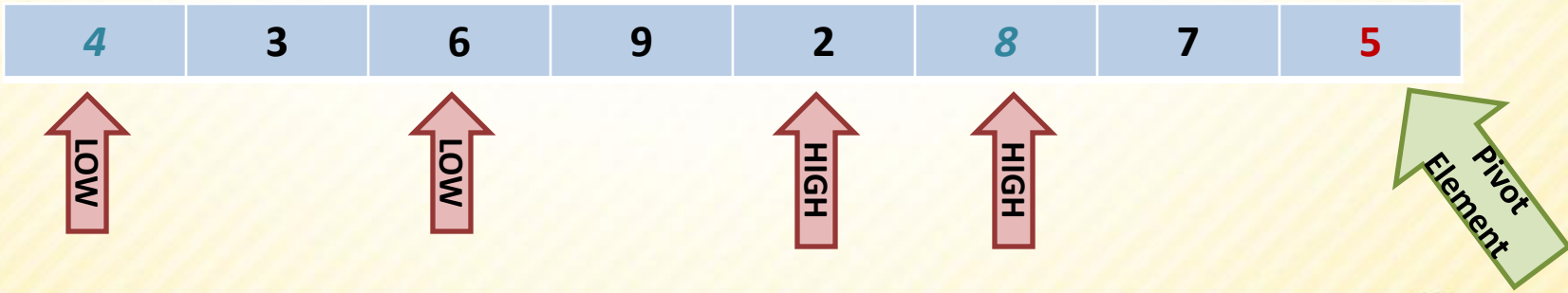
- It should be clear that this algorithm will work
  - But it may not be clear why it is faster than MergeSort.
    - Like MergeSort it recursively solves 2 sub problems and requires linear additional work.
    - BUT unlike MergeSort the sub problems are NOT guaranteed to be of equal size.

- The reason that QuickSort is faster is that the partitioning step can actually be performed in place and very efficiently.
    - This efficiency can more than make up for the lack of equal sized recursive calls.

How to Partition in Place

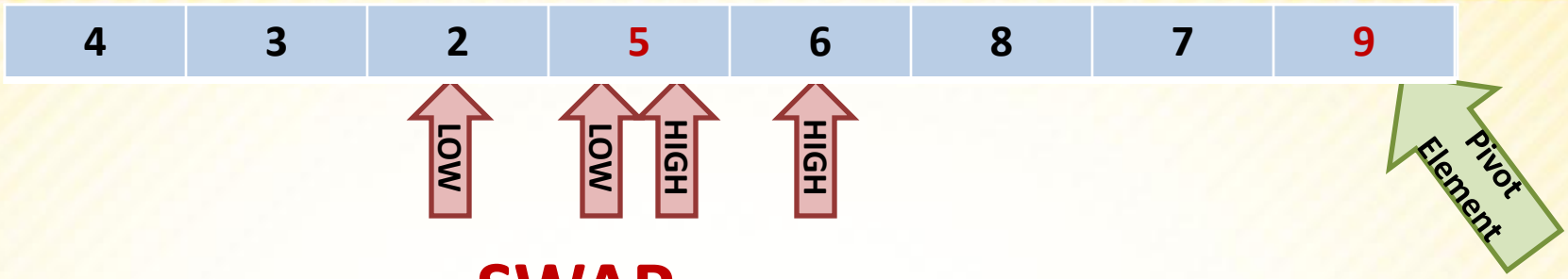


- Assume for now, that we partition based on the last element in the array, 5.
  - Start 2 counters: **Low** at array index 0 **High** at 2nd to last index in the array
    - Advance the **Low** counter forward until a value greater than the pivot is encountered.
    - Advance the **High** counter backward until a value less than the pivot is encountered.



- Now swap these 2 elements, since we know that they are both on the **"wrong"** side.
- Continue to advance the counters as before.

How to Partition in Place



SWAP

When both counters line up, SWAP the last element with the counter position to finish the partition.

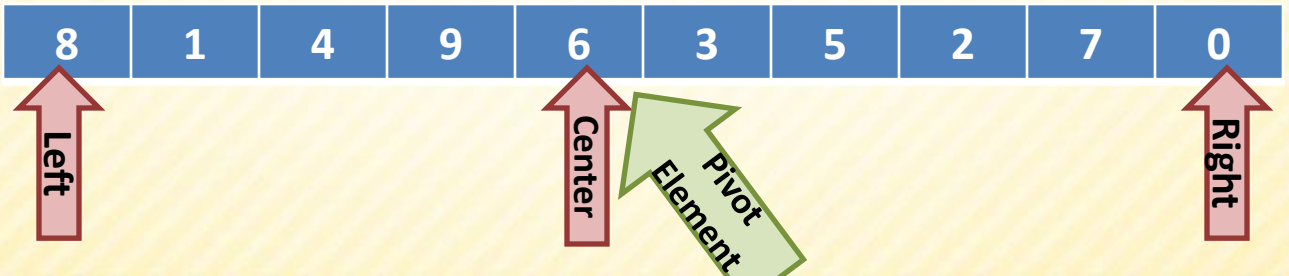
Now as you can see our array is partitioned into a “left” and a “right”.

Picking the Pivot

- Although the Partition algorithm works no matter which element is chosen as the pivot, some choices are better than others.
  - Wrong Choice:
    - Just use the first element in the list
      - If the input is random, this is acceptable.
      - BUT, what if the list is already sorted or in reverse order?
        - » Then all the elements go into S1 or S2, consistently throughout recursive calls.
      - So it would take $O(n^2)$ to do nothing at all! (If presorted)
        - » ***EMBARRASSING!***

Picking the Pivot

- A Safer Way
  - Choose the pivot randomly
    - Generally safe, since it's unlikely the random pivot would consistently be a poor partition.
    - Random number generation is generally expensive.
- Median-of-Three Partitioning
  - The **best** choice would be the median of the array.
    - But that would be hard to calculate and slow.
    - A good estimate is to pick 3 elements and use the median of those as the pivot.
    - The rule of thumb: Pick the left, center, and right elements and take the median as the pivot.



Analysis of Quicksort

- Shown on the board

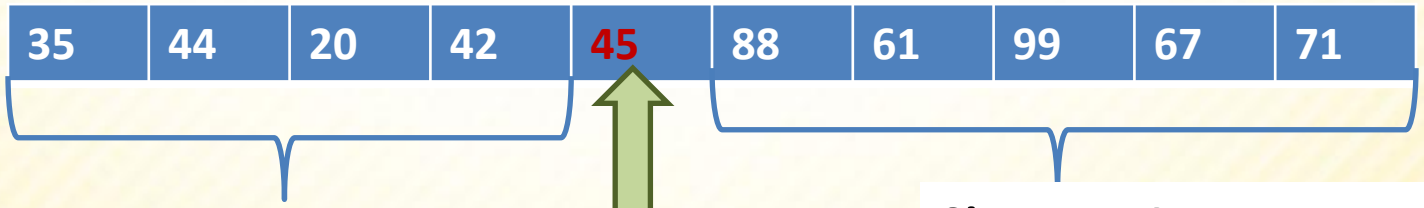
Quickselect

- Given an array of n elements, determine the kth smallest element.
  - Clearly k must lie in between 1 and n inclusive
  - The selection problem is different, but related to the sorting problem.

Quickselect

- **Given an array of n elements, determine the k^{th} smallest element.**

- The idea is:
  - Partition the array.
  - There are 2 subarrays:
    - One of size ***m***, with the ***m*** smallest elements.
    - The other of size ***n-m-1*** .
      - If ***k ≤ m***, we know the ***k^{th}*** smallest element is in the **1st partition**.
      - If ***k == m+1***, we know the ***k^{th}*** smallest element **IS the pivot**.
      - Otherwise, the ***k^{th}*** smallest element is in the **2nd partition**.



Size m:
if ($k \le m$) the k^{th} smallest element is in here.

if (k==m+1)
We know the k^{th} smalles is the pivot

Size n-m-1:
if ($k > m$) the k^{th} smallest element is in here.

Quickselect

- Algorithm:

Quickselect(A, low, high, k):

1) m=Partition(A, low, high) // m is how many values are less than the partition element.
2) if k≤m, return Quickselect(low, low+m-1, k)
3) if k==m+1 return the pivot, A[low+m]
4) else return Quickselect(low+m+1, high, k-m-1)

- So instead of doing 2 recursive calls, w only make one here.
  - It turns out that on average Quickselect takes O(n) time, which is far better than it's worst case performace of $O(n^2)$ time.

Quickselect Analysis

- Shown on the board