Divide and Conquer

Binary Search

Quick Sort

Divide and Conquer

- **Divide and Conquer Methodology**
  - Given a problem, identify a small number of significantly smaller sub-problems of the same type
  - Solve each sub-problem recursively (the smallest possible size of a sub-problem is a base-case)
  - Combine these solutions into a solution for the main problem
- **Examples**
  - **Binary search**
  - **Merge Sort**

Binary Search

- Finds the position of a specified input value within a sorted array
- For binary search, the array should be arranged in ascending or descending order.

How binary search works ?

In each step, the algorithm compares the search value with the value of the middle element of the array.

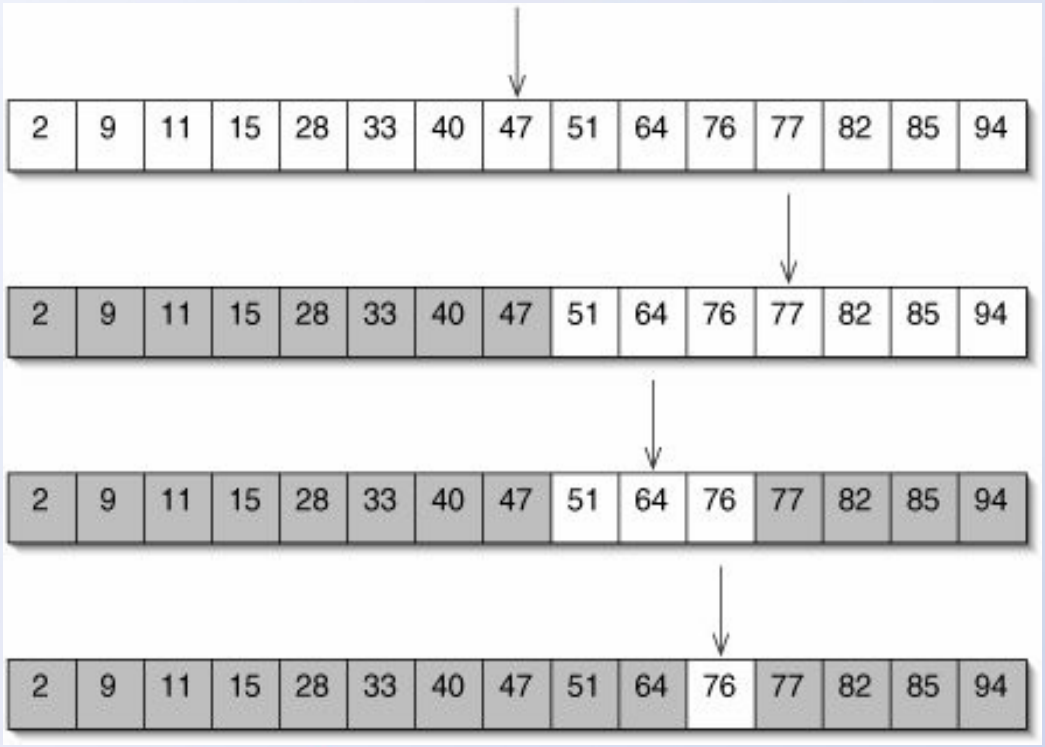
If the values match, then a match has been found and its index is returned.

If the search value is less than the middle element, then the algorithm repeats on the sub-array to the left of the middle element

If the search key is greater, then on the sub-array to the right

Binary Search

How it works ?



IT TALENTS
Training Camp

Task

Try to implement finding a number in sorted array of numbers.

Keep two variables – left and right to point where to search.

Sorting Algorithms

Bubble Sort

Worst Case	O(n^2)
Average Case	O(n^2)
Best Case	O(n)

Selection Sort

Worst Case	O(n^2)
Average Case	O(n^2)
Best Case	O(n^2)

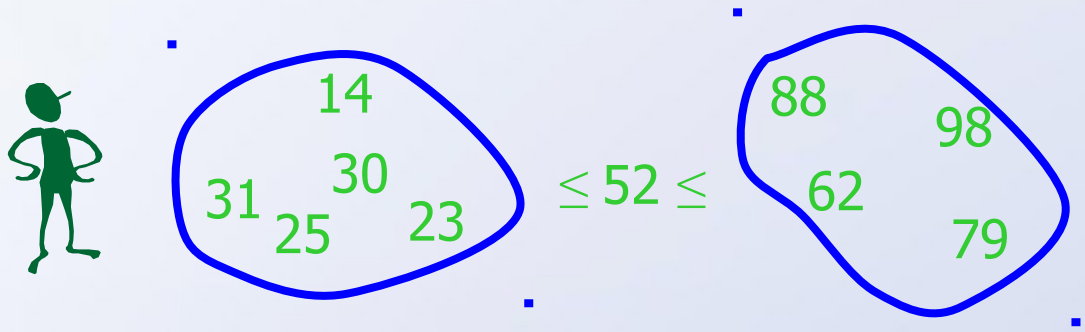
Counting Sort

Worst Case	O(n+k)
Average Case	O(n+k)
Best Case	O(n+k)

Quick Sort

Partition set into two using randomly chosen pivot

Quick Sort



sort the first half.

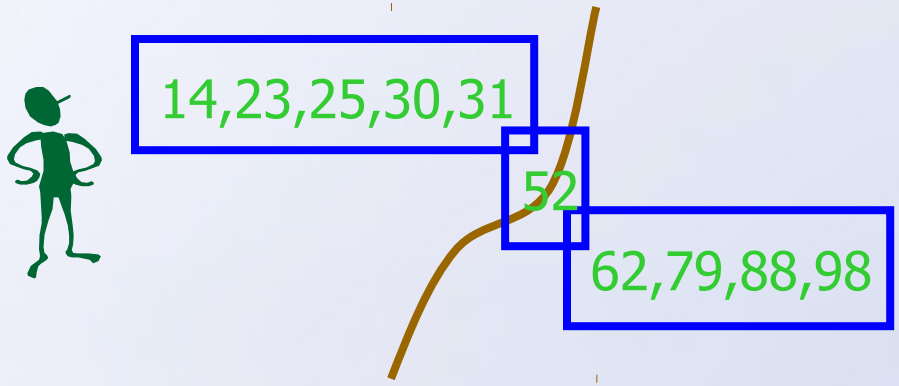
14,23,25,30,31

sort the second half.

62,79,98,88

IT TALENTS
Training Camp

Quick Sort



Glue pieces together.
(No real work)

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

Quicksort

- **Quicksort advantages:**
  - Sorts in place – no additional array needed
  - Sorts $O(n \lg n)$ in the average case
  - Very efficient in practice

- **Quicksort disadvantages :**
  - Sorts $O(n^2)$ in the worst case
  - not stable
    - Does not preserve the relative order of elements with equal keys
    - Sorting algorithm is stable if 2 records with same key stay in original order
  - But in practice, it's quick
  - And the worst case doesn't happen often … sorted

Quicksort

- ***Divide***: $A[p \ldots r]$ is partitioned (rearranged) into two nonempty subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ so that each element of $A[p \ldots q-1]$ is less than or equal to each element of $A[q+1 \ldots r]$. Index q is computed here, called pivot.

- ***Conquer***: two subarrays are sorted by recursive calls to quicksort.

- ***Combine***: 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 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:
    - $A_1 = \{x \in A-\{v\} \mid x \leq v\}$, and
    - $A_2 = \{x \in A-\{v\} \mid x \geq v\}$
  - return
    - { quicksort(A_1) followed by ***v*** followed by quicksort(A_2)}

Quicksort

- **Small instance has $n \leq 1$**
  - Every small instance is a sorted instance
- **To sort a large instance:**
  - select a pivot element from out of the ***n*** elements
- **Partition the *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**

Example

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot

2	5	4	1	3	6	7	9	10	11	8

Sort left and right groups recursively

Quicksort Code

```
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 sub-array in place
  - End result:
    - Two sub-arrays
  - Returns the **index** of the “pivot” element separating the two sub-arrays

Task

Try to reorder elements of given array so that first k elements are lower than or equal to the last element of the array and next elements are greater than it.

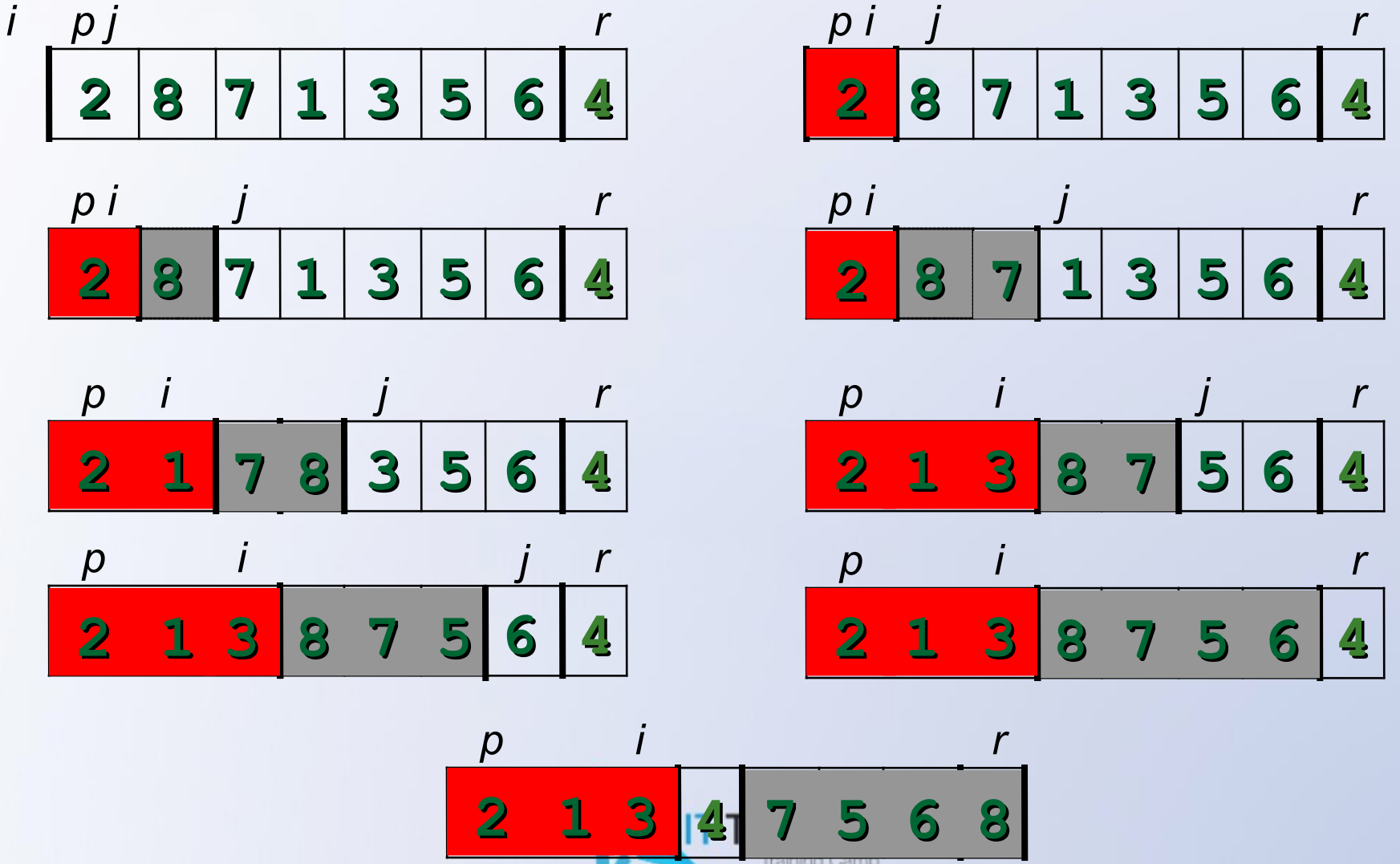
Your algorithm should work in O(N) time.

Partition Code

```
Partition(A, p, r)
{
   x = A[r];          // x is pivot
   i = p – 1;
     for(int j = p; j < r-1; j++)
     {
          if( A[j] <= x)
          {
            i = i + 1;
            exchange A[i] with A[j];
          }
     }
   exchange A[i+1] with A[r];
   return i+1;
}
```

`partition()` runs in O(n) time

Partition Example $A=\{2, 8$



Partition Example Explanation

- **Red shaded elements are in the first partition with values lower than the pivot.**
- **Gray shaded elements are in the second partition with values greater than the pivot.**
- **The unshaded elements have no yet been put in one of the first two partitions.**
- **The final white element is the pivot.**

Order Statistics

Task : given an array and number K, find K-th smallest element in the array.

1)Write the first solution that comes to your mind.

2)Next try to use partition function mentioned before to help you.