# Sorting

Joy Mukherjee

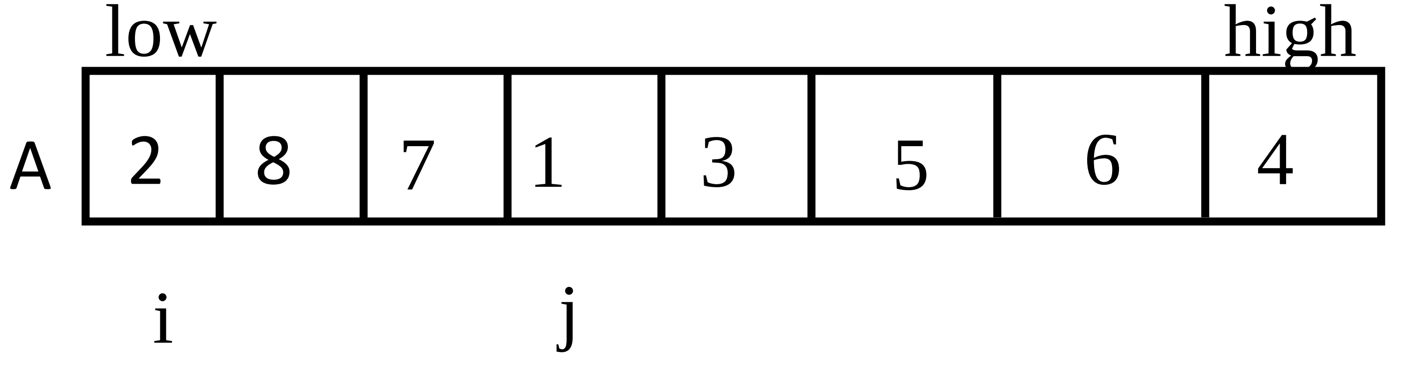
# Quick Sort

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 8 | 7 | 1 | 3 | 5 | 6 | 4 |

**GOAL**: Place A[n-1] in its sorted position i

1. All elements that are less than A[n-1] are placed on the left side of A[i] and
2. All elements that are greater than A[n-1] are placed on the right side of A[i].

- Since A[n-1] is placed in the sorted position, we can apply the same logic on the left and right hand side of the partition



# Partition Algorithm: Idea

Objective: Given an array A[low..high], place A[high] in a proper position (pivot)

Invariant (The property that remains valid throughout the execution of the algorithm): Maintain two indices i and j such that
A[low] to A[i] < A[high] and
A[i+1] to A[j-1] >= A[high]

```
i = low -1;
for(j = low; j <= high-1;j++) {
    if(a[j] < a[high]) {
        i++;
        swap(&a[i],&a[j]);
    }
}
```

| i=-1 | j=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 8 | 7 | 1 | 3 | 5 | 6 | 4 |

| | i=0 | j=1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | **8** | 7 | 1 | 3 | 5 | 6 | 4 |

| | i=0 | 1 | j=2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | **8** | **7** | 1 | 3 | 5 | 6 | 4 |

| | i=0 | 1 | 2 | j=3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | **8** | **7** | 1 | 3 | 5 | 6 | 4 |

| | 0 | i=1 | 2 | 3 | j=4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | **7** | **8** | 3 | 5 | 6 | 4 |

| | 0 | i=1 | 2 | 3 | j=4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | 7 | 8 | 3 | 5 | 6 | 4 |

| | 0 | 1 | i=2 | 3 | 4 | j=5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | 3 | 8 | 7 | 5 | 6 | 4 |

| | 0 | 1 | i=2 | 3 | 4 | j=5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | 3 | 8 | 7 | 5 | 6 | 4 |

| | 0 | 1 | i=2 | 3 | 4 | 5 | j=6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | 3 | 8 | 7 | 5 | 6 | 4 |

| | 0 | 1 | i=2 | 3 | 4 | 5 | 6 | j=7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | 3 | 8 | 7 | 5 | 6 | 4 |

| | 0 | 1 | 2 | i=3 | 4 | 5 | 6 | j=7 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 1 | 3 | 4 | 7 | 5 | 6 | 8 |

# Partition

```
int partition(int a[], int low, int high)
{
  int i, j;
  i = low-1;
  for(j = low; j < high; j++) {
        if(a[j] < a[high]) {
              i++;
              swap(&a[i],&a[j]);
        }
  }
  i++;
  swap(&a[i], &a[high]);
  return i;
}
```

# Quick Sort

```
void quicksort(int a[], int low, int high)
{
  if(low < high) {
        int pivot = partition(a, low, high); //It will place A[high] in a proper position of the sorted array, known as pivot.
        quicksort(a, low, pivot-1);
        quicksort(a, pivot+1, high);
  }
}
From main(): quicksort(a, 0, n-1);
```

# Time Complexity

- Philosophy: Place the rightmost element in the sorted position with the hope that this will partition the array in almost equal halves.
- Worst Case: When the partition does not divide the array evenly. $O(n^2)$

1. The array is sorted
2. The array is reverse sorted

$$(n-1) + (n-2) + \ldots + 1 = O(n^2)$$

- Best Case/ Average Case: O(nlog n) [will be discussed in Algorithm course]

# Randomized Quick Sort

```
void randomizedquicksort(int a[], int low, int high)
{
  if(low < high) {
        srand(time(NULL));
        int  x = rand()%(high-low+1) + low; // x ϵ [low, high]
        swap(&a[x], &a[high]);
         int pivot = partition(a, low, high);
         quicksort(a, low, pivot-1);
         quicksort(a, pivot+1, high);
  }
}
```

# Finding Maximum

- Input: Array of n distinct integers
- Output: maximum
- Question: What is number of comparisons? n-1

# Finding Maximum & Minimum (Even)

- Input: Array of n distinct integers
- Output: maximum & minimum
- Question: What is number of comparisons? 2(n-1) -> 2n
- 6 7   4 8   3 9   2 1
- Min = 6 Max = 7 (1)
- Compare 4 & 8. Compare their minimum with Min and compare their maximum with Max (3)   min = 4 max = 8
- Compare 3 & 9. Compare their minimum with Min and compare their maximum with Max (3) min =3 max = 9
- Compare 2 & 1. Compare their minimum with Min and compare their maximum with Max (3) min = 1 max = 9
- n is even: 1 + (n-2)/2*3 = (3n - 4)/2 = 1.5 n for large n

# Finding Maximum & Minimum (Odd)

- Input: Array of n distinct integers
- Output: maximum & minimum
- Question: What is number of comparisons? 2(n-1) -> 2n
- 7    4 8    3 9    2 1
- Min = 7 Max = 7 (0)
- Compare 4 & 8. Compare their minimum with Min and compare their maximum with Max (3)   min = 4 max = 8
- Compare 3 & 9. Compare their minimum with Min and compare their maximum with Max (3) min =3 max = 9
- Compare 2 & 1. Compare their minimum with Min and compare their maximum with Max (3) min = 1 max = 9
- n is odd: (n-2)/2*3 = (3n - 6)/2 -> 1.5 n for large n

# Finding Maximum & Second Maximum

- Input: Array of n distinct integers
- Output: maximum & second maximum
- Question: What is number of comparisons? (n-1) + (n-2) = 2n – 3

1 7 5 8 2 3 4 6

1 7 5 8 | 2 3 4 6

1 7 | 5 8 | 2 3 | 4 6

1 | 7 | 5 | 8 | 2 | 3 | 4 | 6

7 | 8 | 3 | 6 (4 comparisons)

8 | 6 (2 comparisons)

8 (1 comparisons)

In the divide and conquer process, 8 beats 5, 7, and 6 in succession to become the maximum element.

(n-1) comparisons needed to find the maximum and (ceil(log (n+1)) – 1) comparisons needed to find the second maximum. n + log n - 2 comparisons needed to find the maximum & the second maximum.

# Counting Sort

- If an unsorted array has n integers in the range [1, k] (k <= n), then the array can be sorted in non-decreasing order in O(n) time.
- N = 8
- A[] = {5  7  1   3   4   4  1  5}      A[i]є {1, 2, ..., 8}
- C[] = {2 0 1 2 2 0 1 0}
- C[i] tells how many times I appears in A[]
- C[] = {2 2 3 5 7 7 8 8}
- C[i] tells how many elements in A are <= i
- B[] ={1  1  3  4  4  5  5  7}

# Algorithm: Counting Sort

```
For(i=1;i<=n;i++)   C[i] = 0; O(n)
For(i=1;i<=n;i++)                O(n)
    C[A[i]] = C[A[i]] + 1;
For(i=2; i<=n;i++)         O(n)
    C[i] = C[i] + C[i-1];
For(i=n; i>=1;i--) {        O(n)
    B[C[A[i]]] = A[i];
    C[A[i]] = C[A[i]] – 1;
}
For(i=1;i<=n;i++) A[i] = B[i]; O(n)
```

Time complexity is O(n).