Sorting Algorithms
using
Divide and Conquer Technique

1. Merge Sort
2. Quick Sort

Sorting

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Sorting

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: Let's see!!

Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough ⇒ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

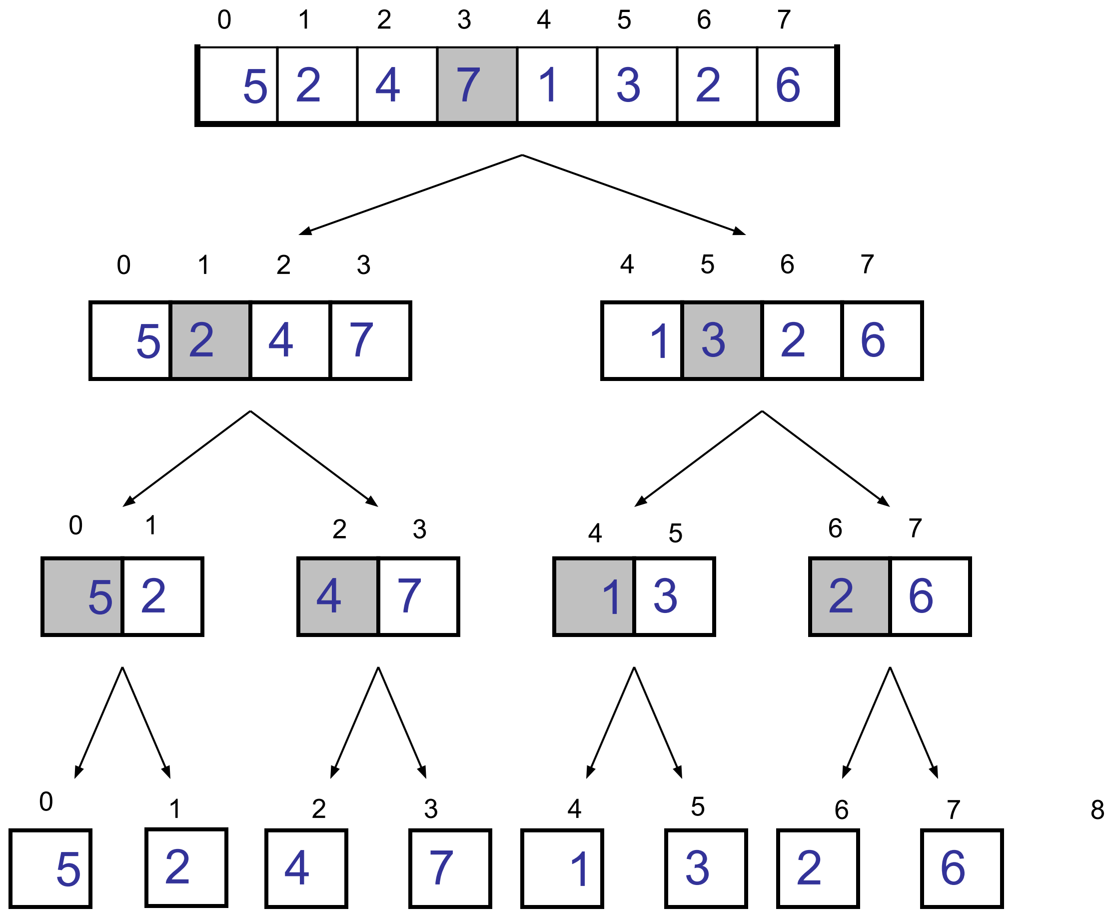
Merge Sort Approach

- To sort an array $A[l \, . \, . \, r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Example – *n* Power of 2

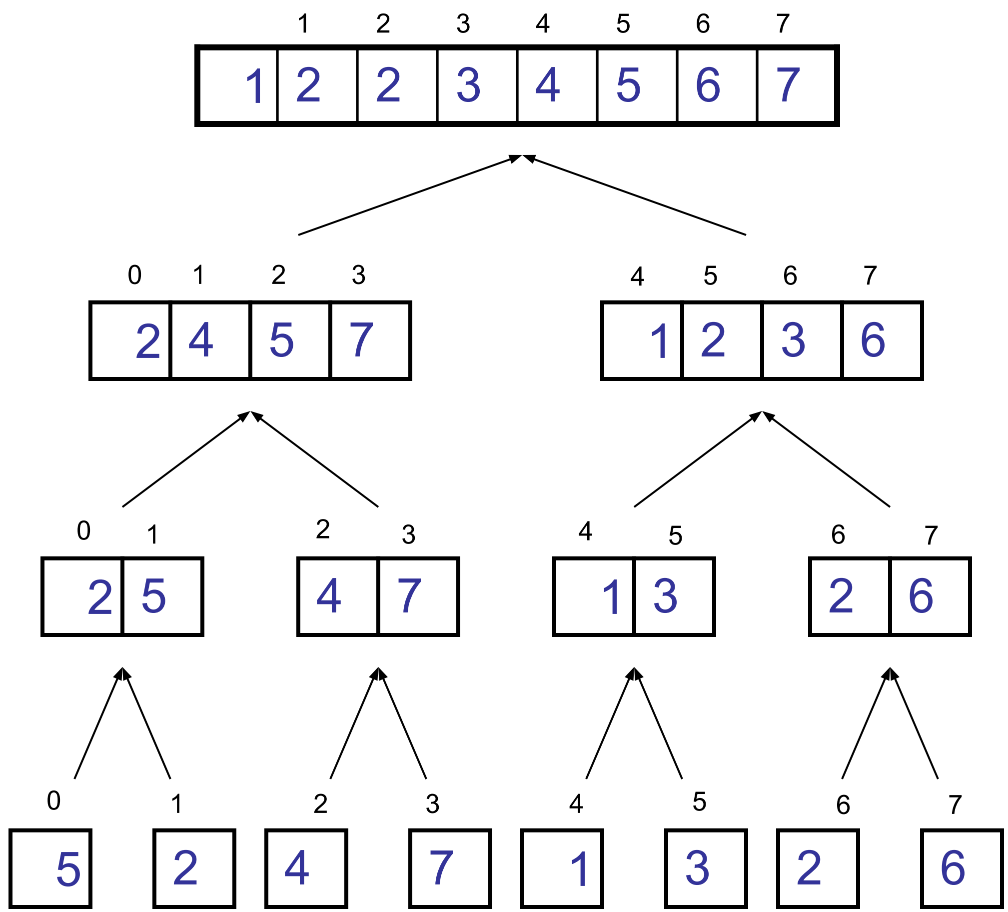
Divide

m = 3

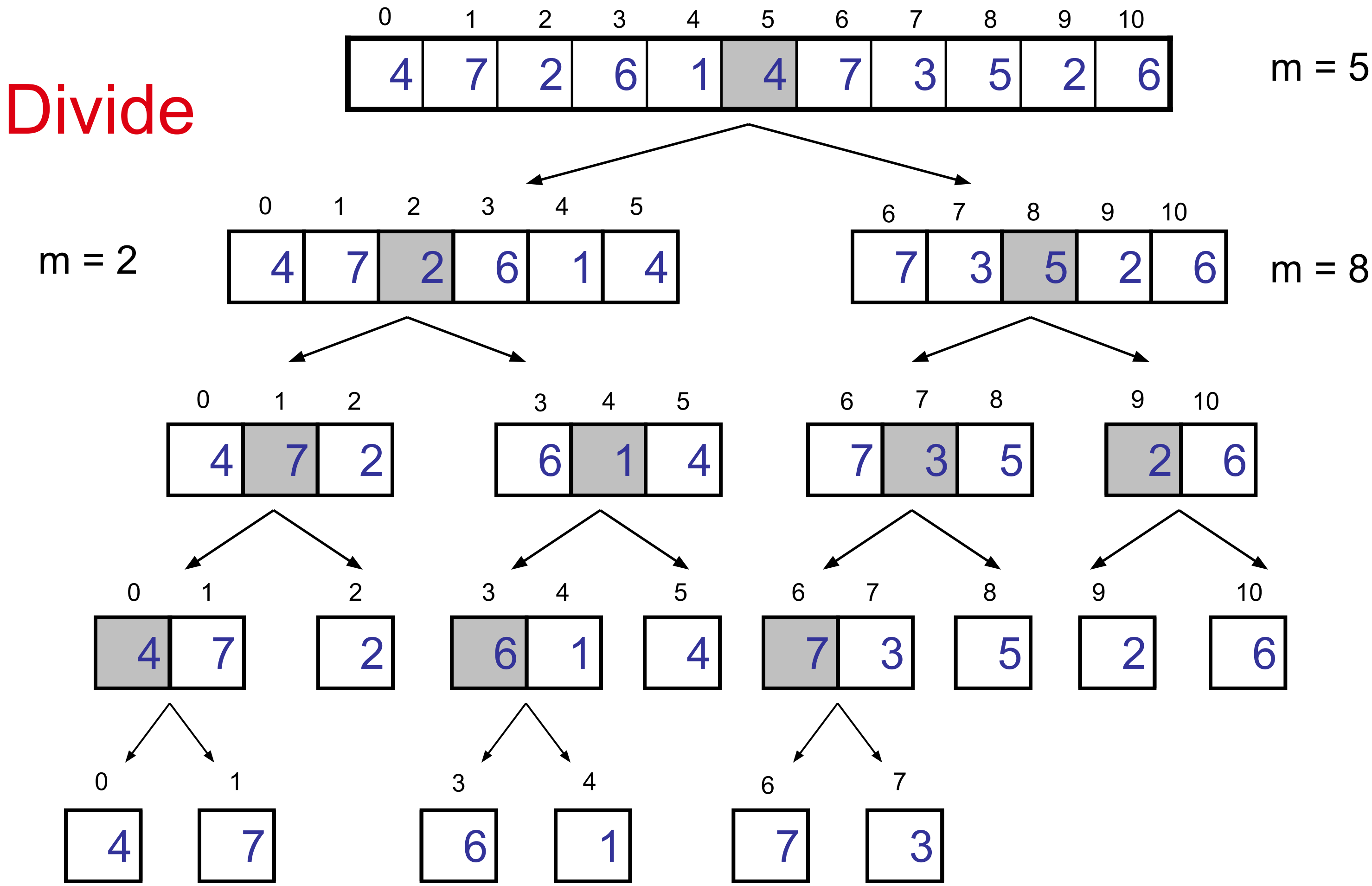


Example – n Power of 2

Conquer and Merge

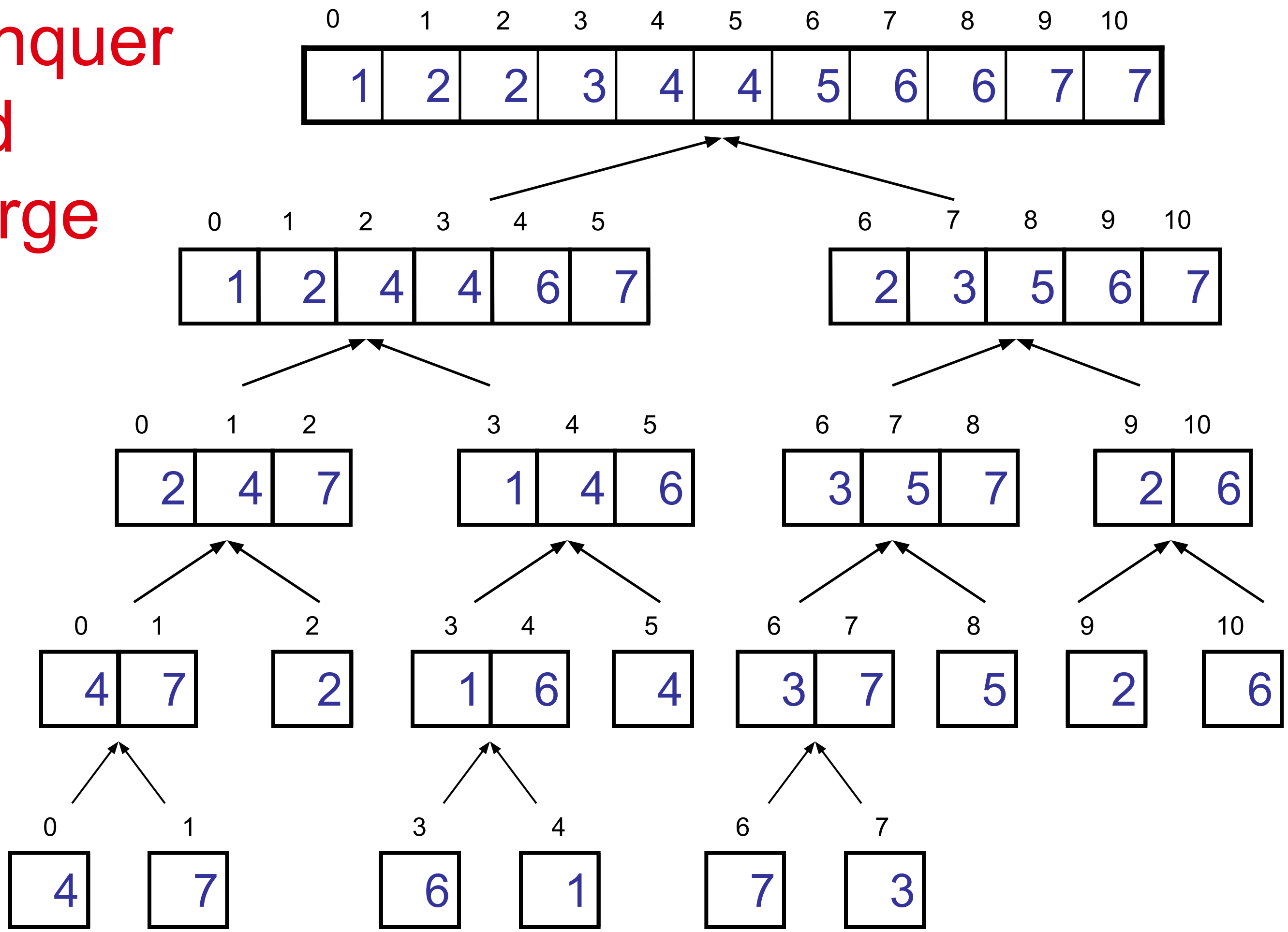


Example – n Not a Power of 2



Example – *n* Not a Power of 2

Conquer and Merge



Merge Sort

Alg.: MERGE-SORT(   )

l			m				r
0	1	2	3	4	5	6	7
5	2	4	7	1	3	2	6

- Initial call:

Merge Sort

l			m				r
0	1	2	3	4	5	6	7
5	2	4	7	1	3	2	6

Alg.: MERGE-SORT(A, l, r)

if $l < r$ # Check for base case

$m \leftarrow \lfloor (l + r)/2 \rfloor$ # Divide

MERGE-SORT(A, l, m) # Conquer

MERGE-SORT(A, $m + 1$, r) # Conquer

MERGE(A, l, m, r) # Combine

- Initial call: MERGE-SORT(A, 0, n-1)

Merging

l			m				r
0	1	2	3	4	5	6	7
5	2	4	7	1	3	2	6

- **Input:** Array A and indices l, m, r such that $l \le m < r$
  - Subarrays $A[l \,..\, m]$ and $A[m+1 \,..\, r]$ are sorted
- **Output:** One single sorted subarray $A[l \,..\, r]$

Merging

l			m				r
0	1	2	3	4	5	6	7
5	2	4	7	1	3	2	6

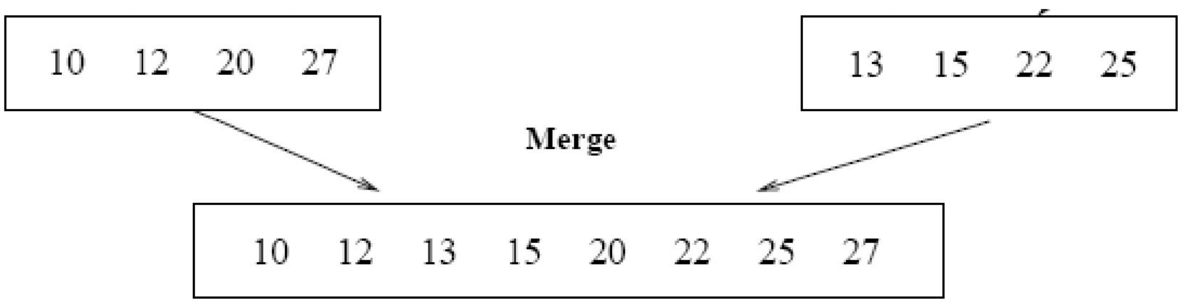
Merge - Pseudocode

```
merge(a,l,m,r):
  i=l,   j=m+1,    k=0
  while i<=m and j<=r:
    if a[i]<=a[j]:
      b[k] = a[i]
      k += 1
      i += 1
    else:
      b[k] = a[j]
      k += 1
      j += 1
```

```
  while i<=m:
      b[k] = a[i]
      k += 1
      i += 1
  while j<=r:
      b[k] = a[j]
      k += 1
      j += 1
  for (i=0 to k):
      a[l+i]=b[i]
```

Running Time of Merge (assume last **for** loop)

- Merging into temporary array:
  - $\Theta(n)$
- Copying the elements from temporary to the final array:
  - n iterations, $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



MERGE-SORT Running Time

- **Divide:**
  - compute m as the average of l and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + c(n) & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

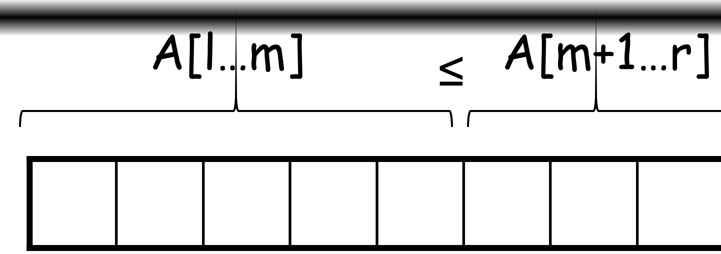
Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

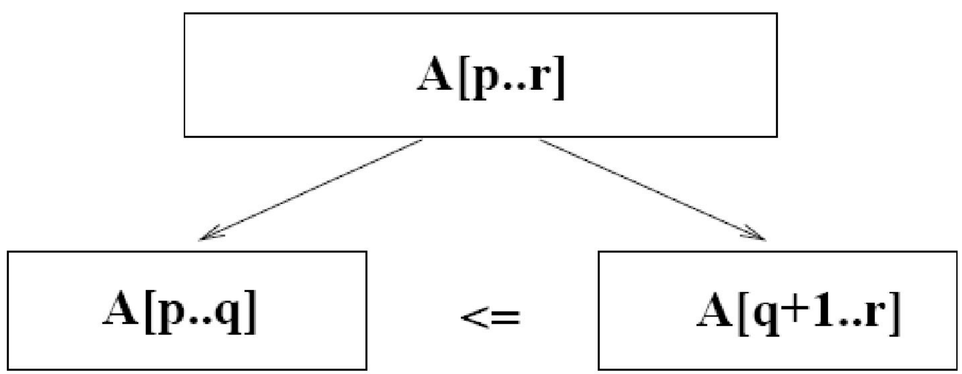
Merge Sort - Discussion

- Advantages:
  - Guaranteed to run in $\Theta(n \log n)$
- Disadvantage
  - Requires extra space $\approx N$

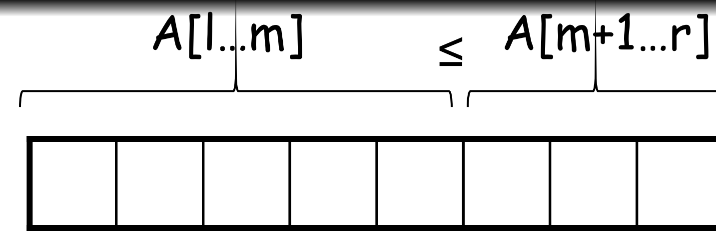
Quicksort



- Sort an array *A[l...r]*
- **Divide**
  - Partition the array *A* into 2 subarrays *A[l..m]* and *A[m+1..r]*, such that each element of *A[l..m]* is smaller than or equal to each element in *A[m+1..r]*
  - Need to find index m to partition the array



Quicksort



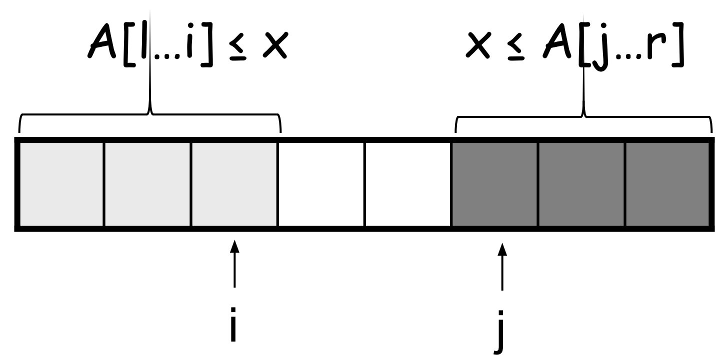
- **Conquer**
  - Recursively sort $A[l..m]$ and $A[m+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

Partitioning the Array

- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition (see prob. 7-1, page 159)
  - Select a pivot element **x** around which to partition
  - Grows two regions

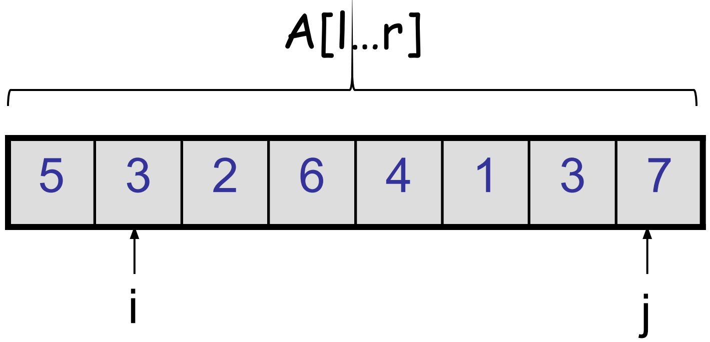
$A[l \dots i] \leq \mathbf{x}$

$\mathbf{x} \leq A[j \dots r]$

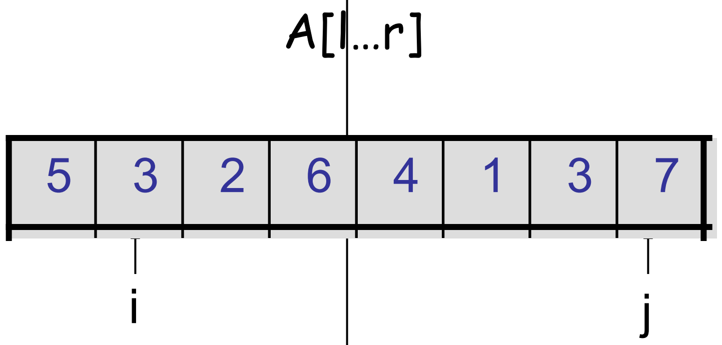


j

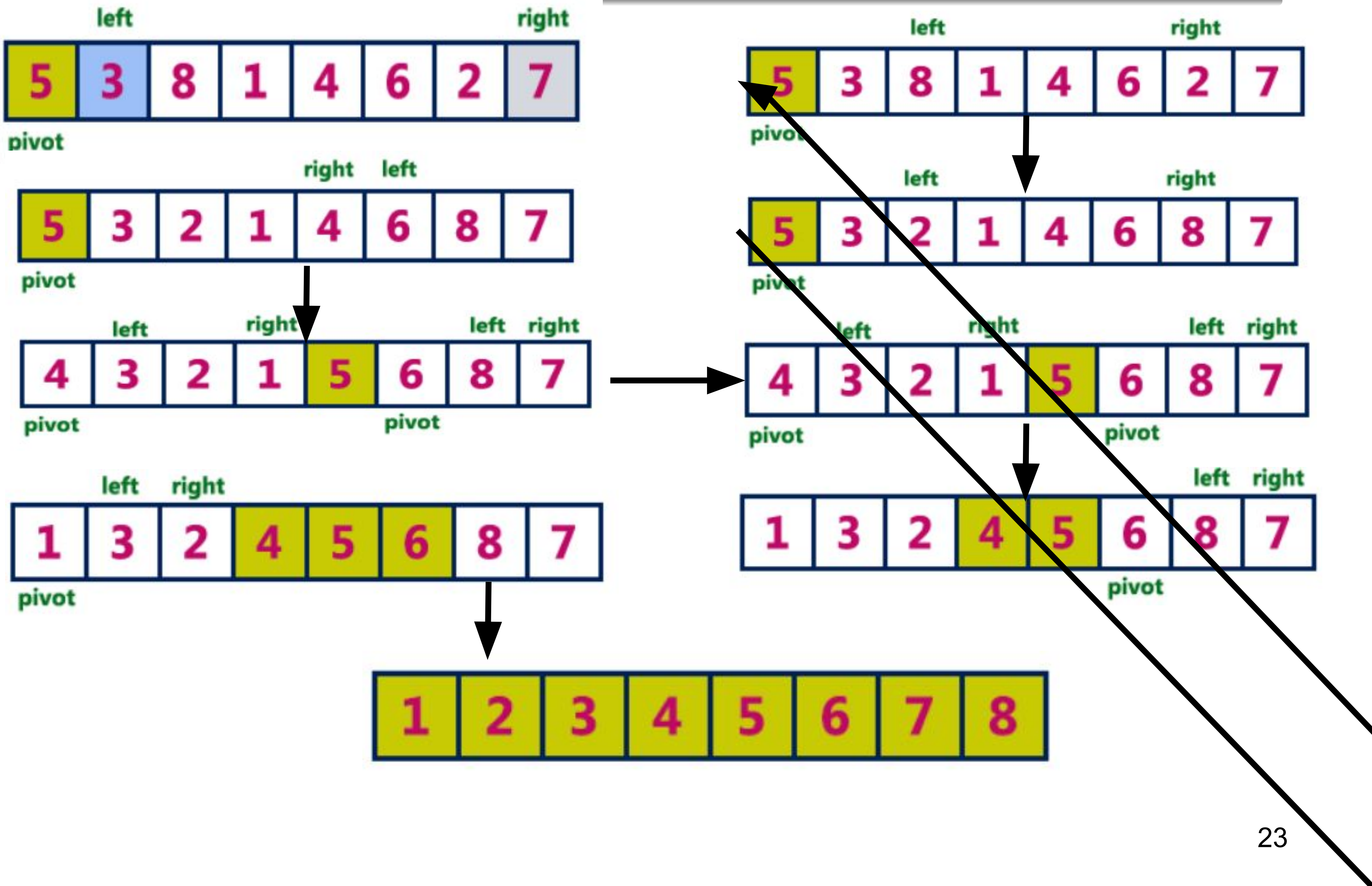
Example



pivot x=5



Example



Example

Partitioning the Array

```
def partition(A,l,h):
  pivot = A[l]
  i=l+1
  j=h
  while i<j:
    while A[i]<pivot and i<h:
      i += 1
    while A[j]>pivot and j>=l:
      j -= 1
    if i<j:
      t=A[i]
      A[i]=A[j]
      A[j]=t
```

```
  if A[l]>A[j]:
    t=A[l]
    A[l]=A[j]
    A[j]=t
  return j
```

Recurrence

Alg.: QUICKSORT(A, l, r)

Initially: l=1, r=n

if $l < r$ **then**

$m \leftarrow$ PARTITION(A, l, r)

QUICKSORT (A, l, $m-1$)

QUICKSORT (A, $m+1$, r)

Recurrence:

$$T(n) = T(m) + T(n - m) + n$$

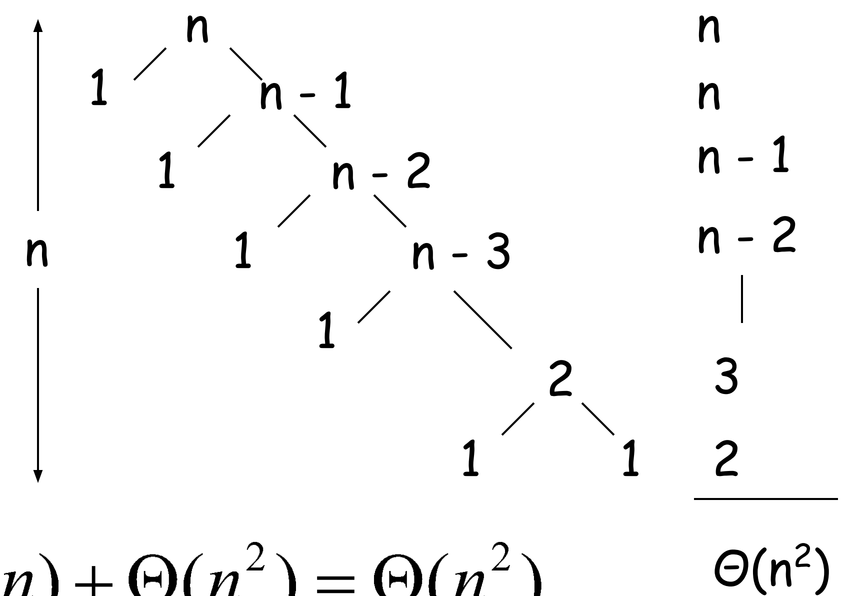
Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: m=1

T(n) = T(1) + T(n – 1) + n,

T(1) = Θ(1)

T(n) = T(n – 1) + n



$$= \quad n + \left(\sum_{k=1}^{n} k \right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$
- Recurrence: m=n/2

  $T(n) = 2T(n/2) + \Theta(n)$

  $T(n) = \Theta(n \log n)$ (Master theorem)