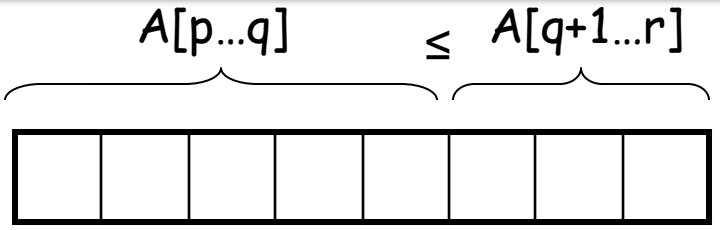
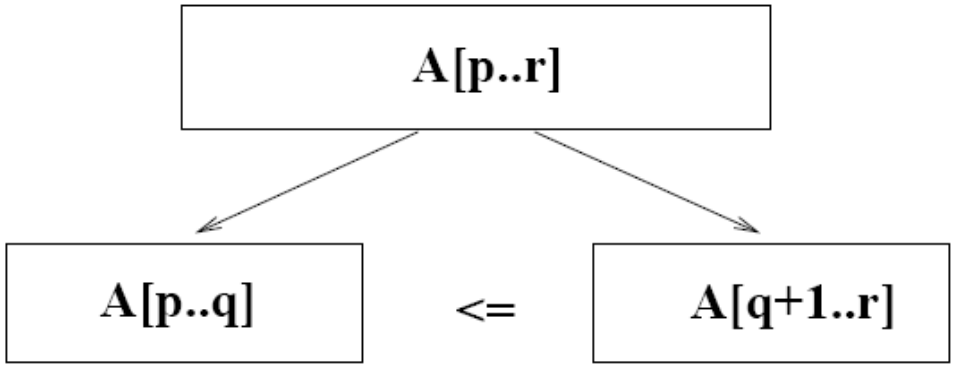
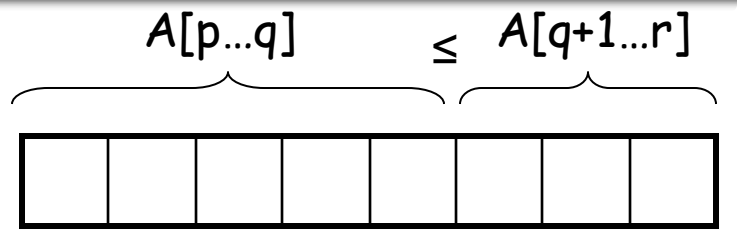
Quicksort

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





Quicksort



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

QUICKSORT

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

Partitioning the Array

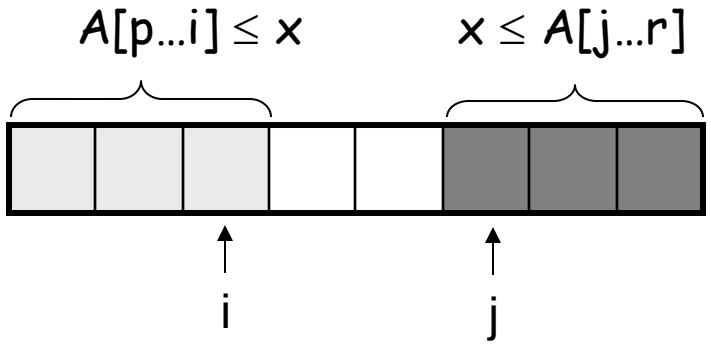
```
1    Algorithm Partition(a, m, p)
2    // Within a[m], a[m + 1], ..., a[p − 1] the elements are
3    // rearranged in such a manner that if initially t = a[m]
4    // then after completion a[q] = t for some q between m
5    // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6    // for q < k < p. q is returned. Set a[p] = ∞.
7    {
8         v := a[m]; i := m; j := p;
9         repeat
10        {
11             repeat
12                  i := i + 1;
13             until (a[i] ≥ v);

14             repeat
15                  j := j − 1;
16             until (a[j] ≤ v);

17             if (i < j) then Interchange(a, i, j);

18        } until (i ≥ j);

19        a[m] := a[j]; a[j] := v; return j;
20   }

1    Algorithm Interchange(a, i, j)
2    // Exchange a[i] with a[j].
3    {
4         p := a[i];
5         a[i] := a[j]; a[j] := p;
6    }
```

Partitioning the Array

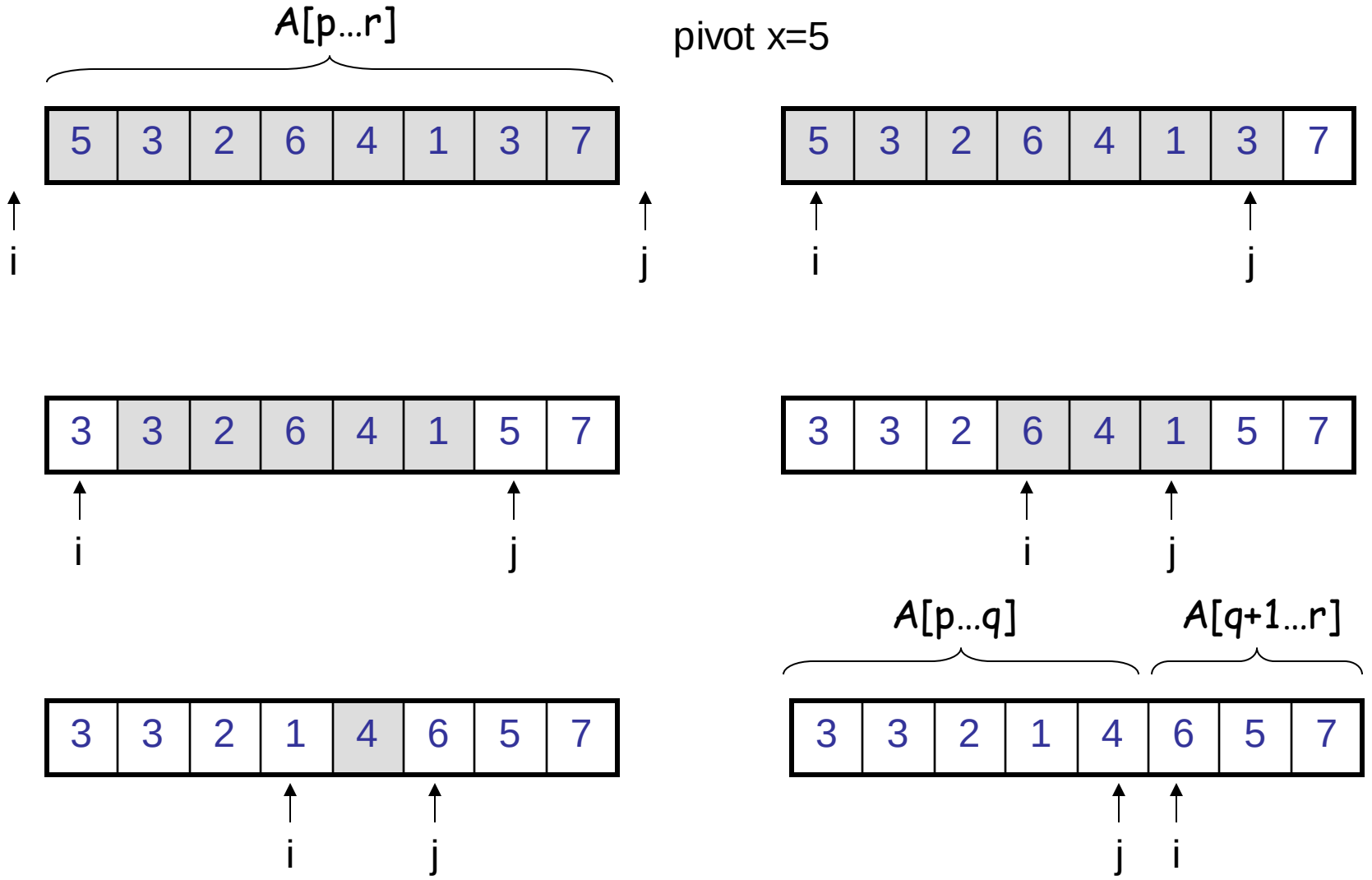
- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition (see prob. 7-1, page 159)
  - Select a pivot element x around which to partition
  - Grows two regions

    $A[p...i] \leq x$

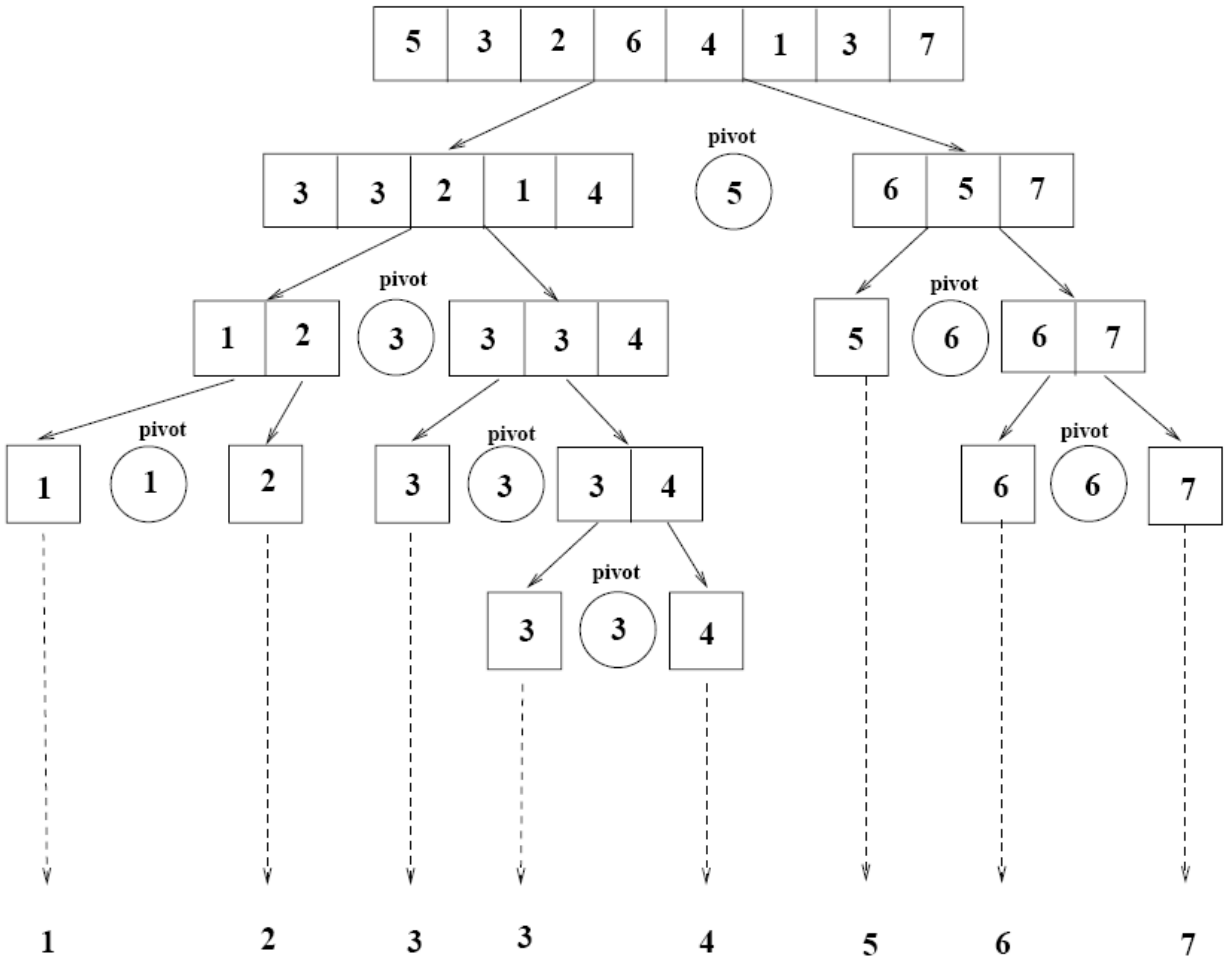
    $x \leq A[j...r]$



Example


A[p...r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
A[p...q]
A[q+1...r]
3 3 2 1 4 6 5 7
i
j
3 3 2 1 4 6 5 7
j
i

Example


5 3 2 6 4 1 3 7
pivot
5
3 3 2 1 4
6 5 7
pivot
3
1 2
3 3 4
pivot
6
5
6 7
pivot
1
1
2
3
3
3 4
pivot
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

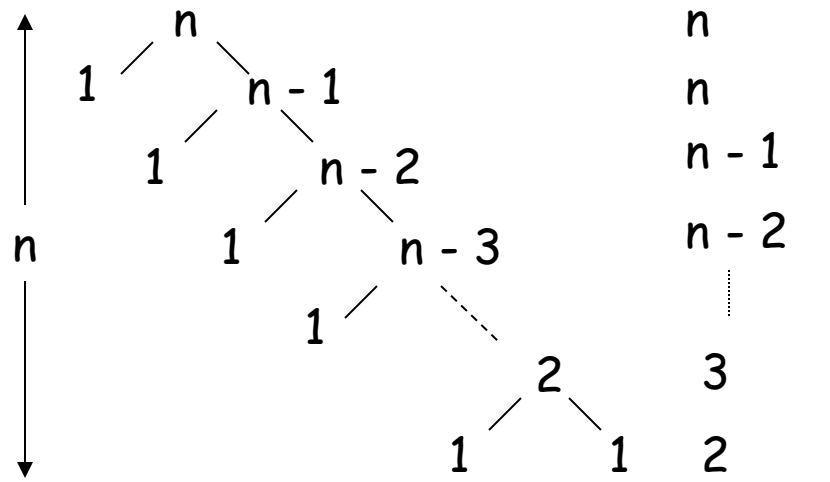
Worst Case Partitioning

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

$T(n) = T(1) + T(n-1) + n,$

$T(1) = \Theta(1)$

$T(n) = T(n-1) + n$



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

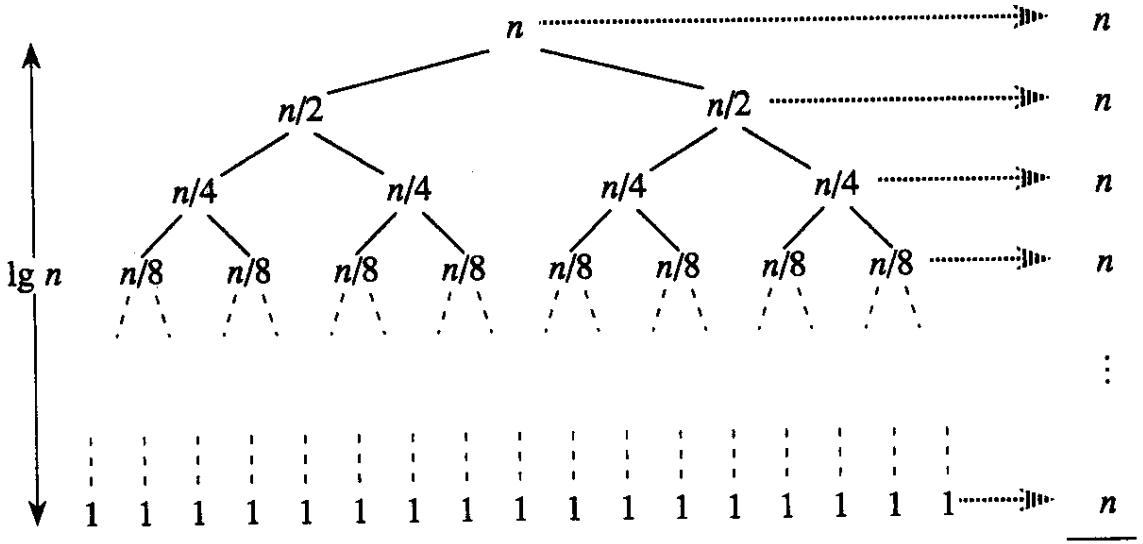
When does the worst case happen?

Best Case Partitioning

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

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

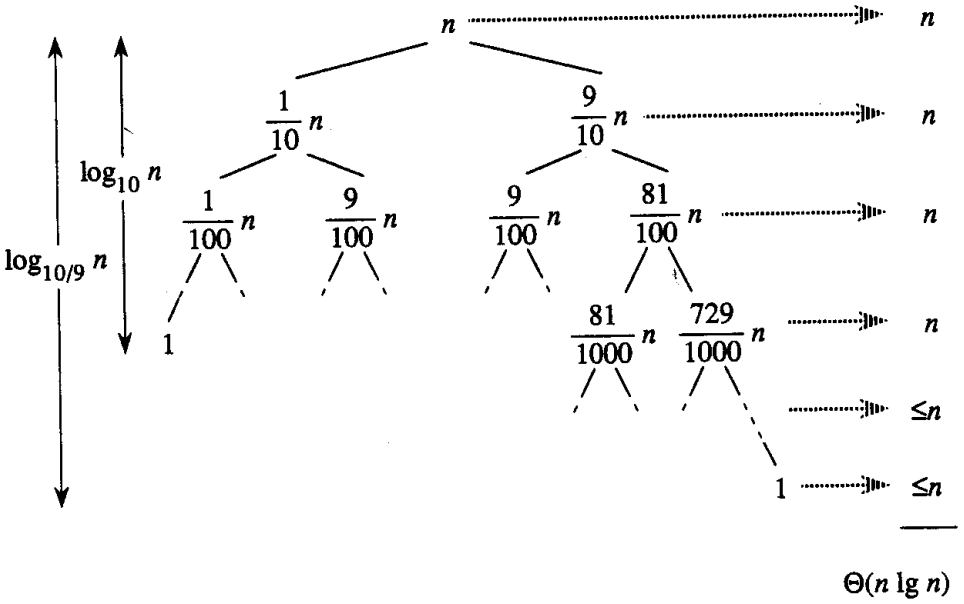
$T(n) = \Theta(n \lg n)$ (Master theorem)



Case Between Worst and Best

- 9-to-1 proportional split

$$Q(n) = Q(9n/10) + Q(n/10) + n$$



How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(nlgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

$$\text{ratio}=1/(n-1) \text{ not a constant !!!}$$

- Consider the $(n/2 : n/2)$ splitting:

$$\text{ratio}=(n/2)/(n/2) = 1 \text{ it is a constant !!}$$

- Consider the $(9n/10 : n/10)$ splitting:

$$\text{ratio}=(9n/10)/(n/10) = 9 \text{ it is a constant !!}$$