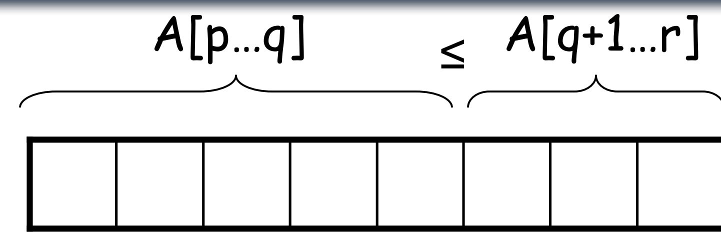
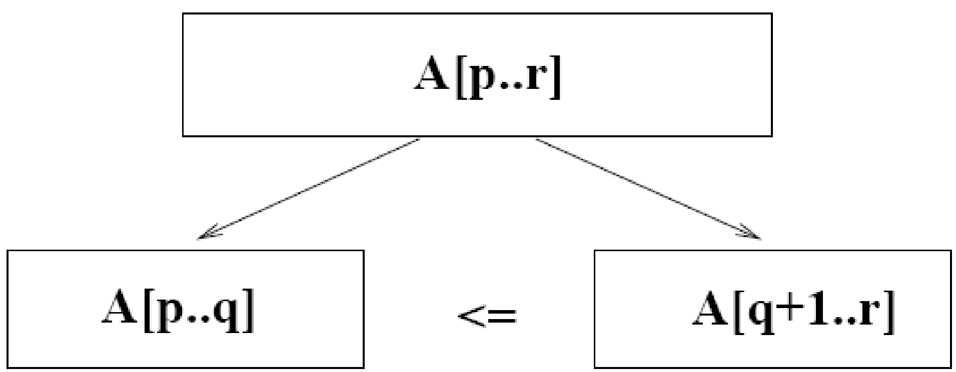
Quick Sort

SMITA SANKHE
Assistant Professor
Department of Computer Engineering

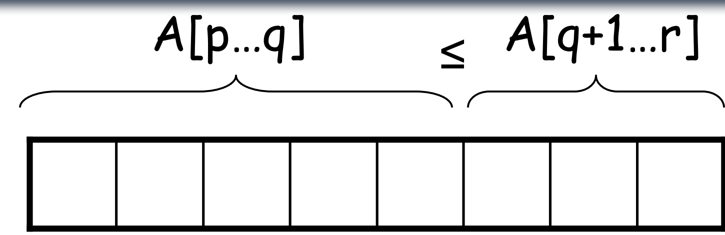
Quicksort



- 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

Quick Sort

- Quick Sort uses Divide and Conquer Strategy.
- There are three steps:

1. **Divide:**
   - Splits the array into sub arrays.
   - Splitting of array is based on **pivot element**.
   - Each element in left sub array is less than and equal to middle (pivot) element.
   - Each element in right sub array is greater than the middle (pivot) element.

2. **Conquer:** Recursively sort the two sub arrays
3. **Combine:** Combine all sorted elements in a group to form a list of sorted elements.

QUICKSORT

Algorithm QuickSort(p, q)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if $(p < q)$ **then** // If there are more than one element
    {
        // divide P into two subproblems.
            $j :=$ Partition$(a, p, q+1)$;
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort$(p, j-1)$;
            QuickSort$(j+1, q)$;
        // There is no need for combining solutions.
    }
}

Partitioning the Array

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

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

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

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

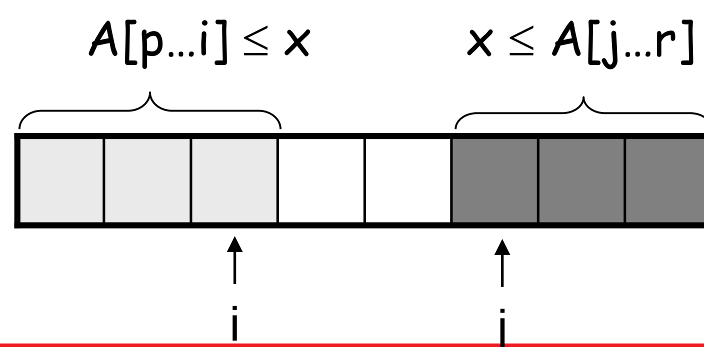
```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

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

$A[p \ldots i] \le \mathbf{x}$

$\mathbf{x} \le A[j \ldots r]$



Example

Step 1:

Low High

50	30	10	90	80	20	40	70

i / **Pivot** j

Step 2:

Increment i if A[i]<= Pivot and continue to increment it until element pointed by i is greater than A[Low]

High

50	30	10	90	80	20	40	70

Pivot i j

Step 3:

Decrement j if A[j]> Pivot and continue to decrement it until element pointed by j is less than A[Low]

High

50	30	10	90	80	20	40	70

Pivot i j

Example

Step 4:

As A[i] > A[Low], stop incrementing i

Low High

50	30	10	90	80	20	40	70

Pivot i j

Step 5:

Increment i if A[i]<= Pivot and continue to increment it until element pointed by i is greater than A[Low]

High

50	30	10	90	80	20	40	70

Pivot i j

Step 6:

Decrement j if A[j]> Pivot and continue to decrement it until element pointed by j is less than A[Low]

High

50	30	10	90	80	20	40	70

Pivot i j

Example

Step 7: As **A[j] > A[Low]**, stop decrementing j

Low High

50	30	10	**90**	80	20	**40**	70

i j

Pivot

Step 8: Since i and j cannot be further incremented and decremented, we will **swap A[i] and A[j]**

Low High

50	30	10	**40**	80	20	**90**	70

Pivot i j

Step 9: Continue incrementing i and decrementing j until false conditions are obtained

Low High

50	30	10	40	80	20	90	70

i j

Pivot

Example

Step 7:

Low High

50	30	10	40	**80**	**20**	90	70

i j

Pivot

Step 8:

swap A[i] and A[j]

Low High

50	30	10	40	**20**	**80**	90	70

Pivot i j

Step 9:

Again Increment i and decrement j. **As soon as i > j, swap A[Low] and A[j]**

Low High

50	30	10	90	**20**	80	90	70

j i

Pivot

Example

Step 10:

swap A[Low] and A[j]

Low | | | | Pivot | | | High

20	30	10	40	**50**	80	90	70

j | i

Step 11:

Low | High

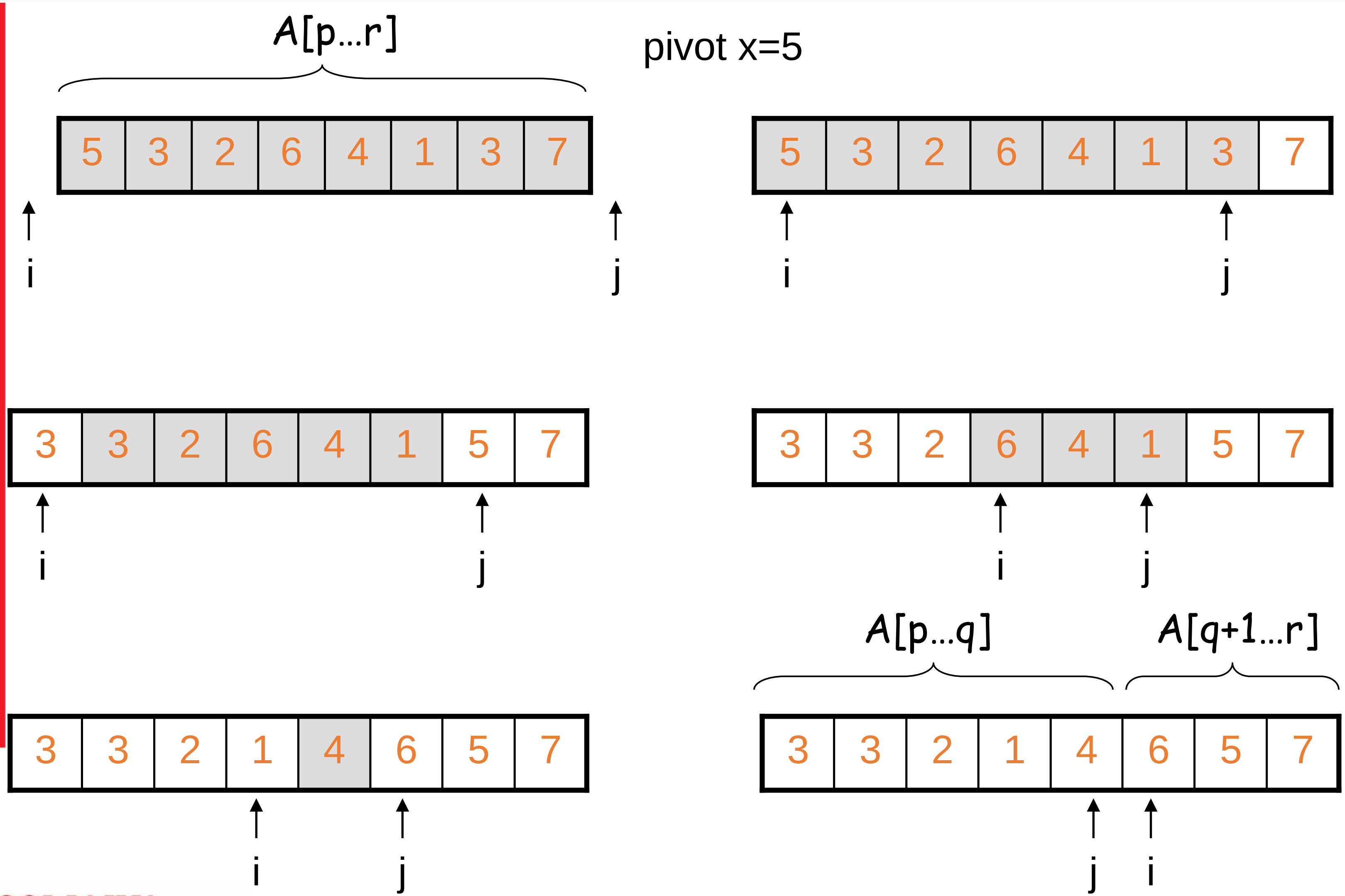
20	30	10	40	**50**	80	90	70

i / Pivot | j | **mid** | i/Pivot | j

Left Sub Array

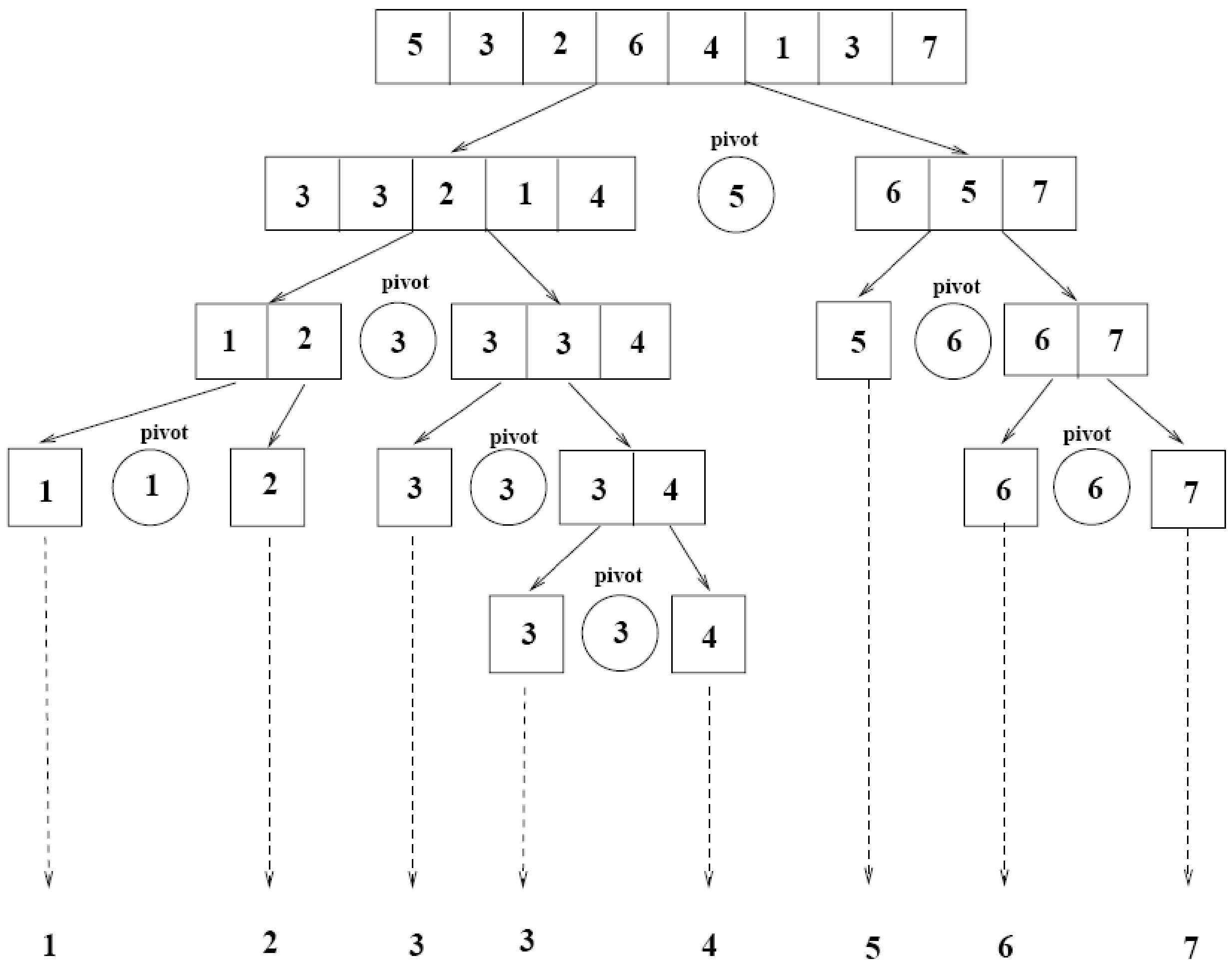
Right Sub Array

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

Worst Case Partitioning

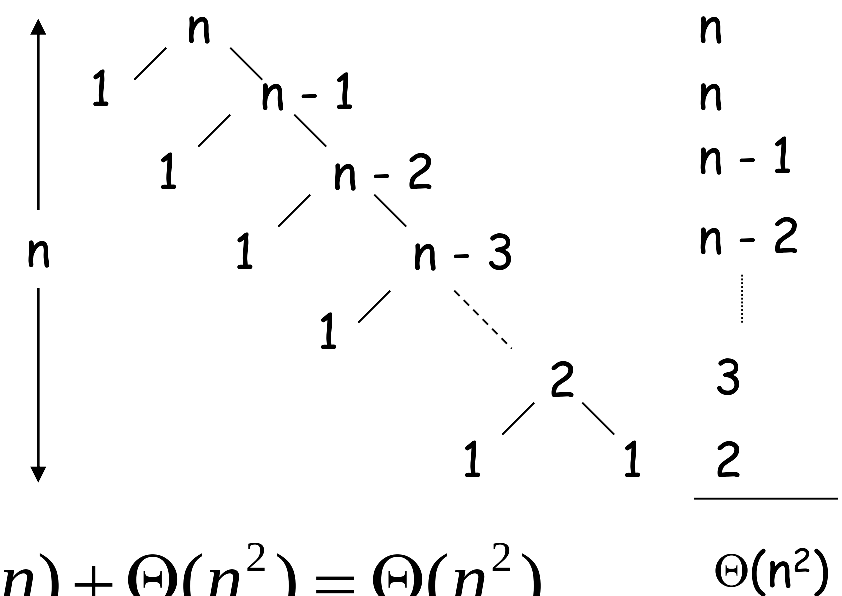
- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

T(n) = T(1) + T(n – 1) + n,

T(1) = Θ(1)

T(n) = T(n – 1) + n

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



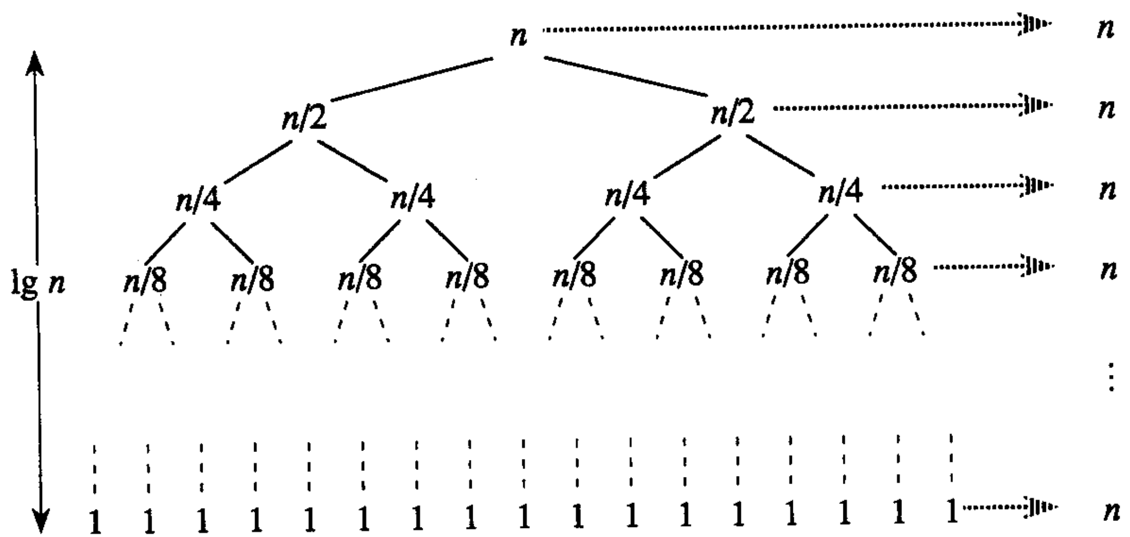
When does the worst case happen?

Best Case Partitioning

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

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

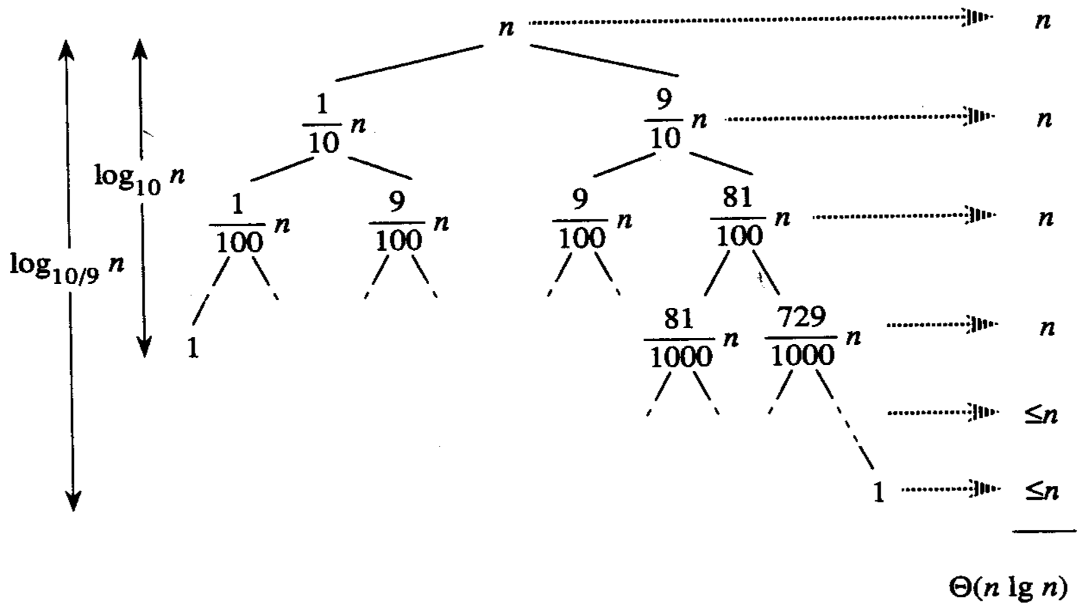
$T(n) = \Theta(n\lg n)$ (Master theorem)



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:

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