

Design and Analysis of Algorithms (UE17CS251)

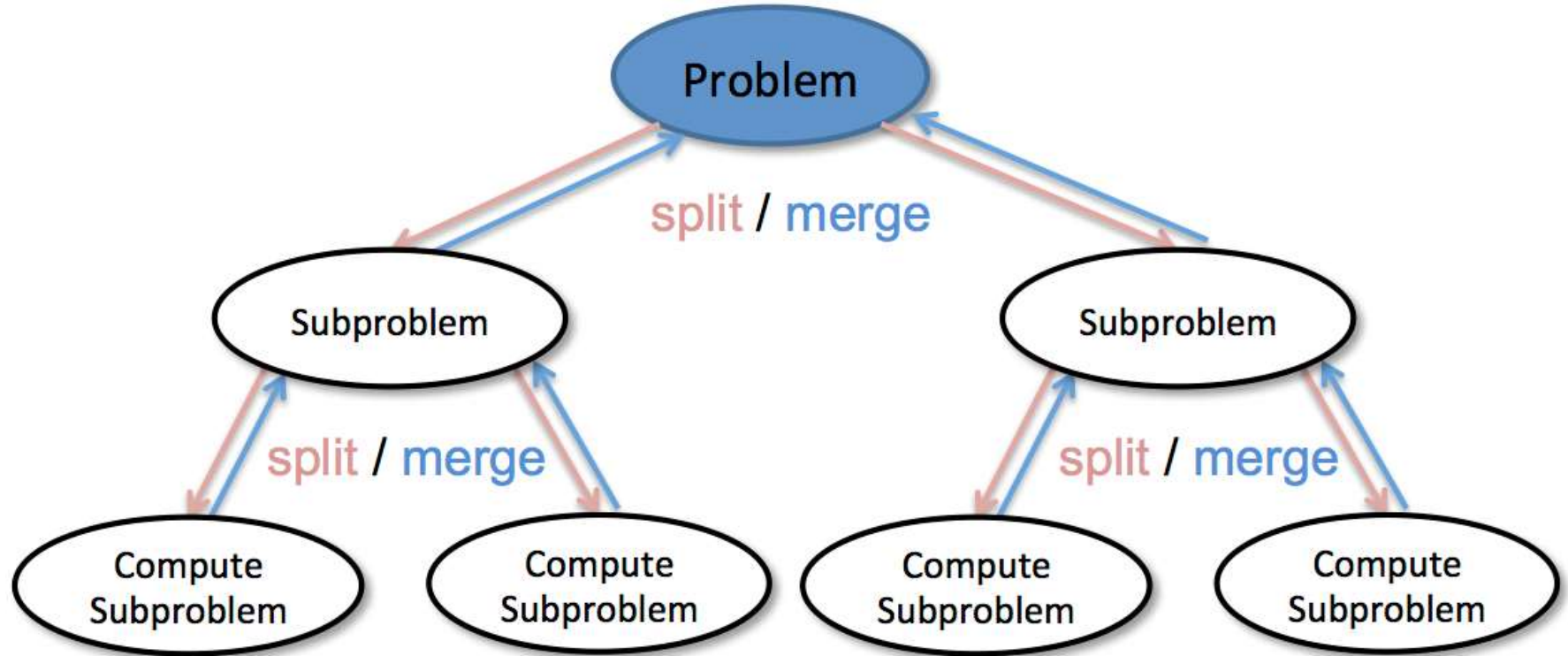
Unit II - Divide and Conquer

Mr. Channa Bankapur
channabankapur@pes.edu

Divide-and-Conquer!

 The picture can't be displayed.

Divide-and-Conquer!



It is a well-known algorithm design technique:

1. Divide instance of a problem into two or more smaller instances.
2. Solve the smaller instances of the same problem.
3. Obtain a solution to the original instance by combining the solutions of the smaller instances.

Q: Write an algorithm to find the sum of an array of n numbers using **Brute Force** approach.

Algorithm Sum(A[0..n-1])

//Sum of the numbers in an array

//Input: Array A having n numbers

//Output: Sum of n numbers in the array A

...

Q: Write an algorithm to find the sum of an array of n numbers using **Brute Force** approach.

Algorithm Sum(A[0..n-1])

//Sum of the numbers in an array

//Input: Array A having n numbers

//Output: Sum of n numbers in the array A

sum \leftarrow 0

for i \leftarrow 0 to n-1

sum \leftarrow sum + A[i]

return sum

$T(n) = n \in \theta(n)$

Q: Write an algorithm to find the sum of an array of n numbers using **Divide-and-Conquer** approach.

Algorithm Sum(A[0..n-1])

//Sum of the numbers in an array

//Input: Array A having n numbers

//Output: Sum of n numbers in the array A

...

Q: Write an algorithm to find the sum of an array of n numbers using **Divide-and-Conquer** approach.

Algorithm Sum(A[0..n-1])

//Sum of the numbers in an array

//Input: Array A having n numbers

//Output: Sum of n numbers in the array A

if (n = 0)

return 0

if (n = 1)

return A[0]

return Sum(A[0..[(n-1)/2]]) +

Sum(A[(n-1)/2+1..n-1])

$$T(n) = 2T(n/2) + 1, T(1) = 1$$

$$= 2n - 1 \in \Theta(n)$$

Q: Write an algorithm to find the sum of an array of n numbers using **Decrease-and-Conquer** approach.

Algorithm Sum(A[0..n-1])

//Sum of the numbers in an array

//Input: Array A having n numbers

//Output: Sum of n numbers in the array A

if (n = 0)

return 0

return Sum(A[0..(n-2)]) + A[n-1]

$$\begin{aligned} T(n) &= T(n-1) + 1, T(1) = 1 \\ &= n \in \Theta(n) \end{aligned}$$

This approach is called as **Decrease-and-Conquer**. It resonates more with the Math Induction.

- **Brute Force:**

- $\text{Sum}(A[0..n-1]) = A[0] + A[1] + \dots + A[n-1]$
- $T(n) \in \Theta(n)$

- **Divide-and-Conquer:**

- $\text{Sum}(A[0..n-1]) = \text{Sum}(A[0..n/2-1]) + \text{Sum}(A[n/2..n-1])$
- $C(n) = 2C(n/2) + 1, C(1) = 1$
 $T(n) \in \Theta(n)$

- **Decrease-and-Conquer:**

- $\text{Sum}(A[0..n-1]) = \text{Sum}(A[0..n-2]) + A[n-1]$
- $C(n) = C(n-1) + 1, C(1) = 1$
 $T(n) \in \Theta(n)$

Finding a^n using **Brute Force** approach.

Algorithm Power(a, n)

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

...

Finding a^n using **Brute Force** approach.

Algorithm Power(a, n)

//Computes $a^n = a * a * \dots a$ (n times)

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

p \leftarrow 1

for i \leftarrow 1 **to** n

p \leftarrow p * a

return p

$C(n) = n$

$T(n) \in \Theta(n)$

Finding a^n using **Divide-and-Conquer** approach.

Algorithm Power(a, n)

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

...

Finding a^n using **Divide-and-Conquer** approach.

Algorithm Power(a, n)

//Computes $a^n = a^{\lfloor n/2 \rfloor} * a^{\lfloor n/2 \rfloor}$

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

if ($n = 0$) **return** 1

if ($n = 1$) **return** a

return Power(a, $\lfloor n/2 \rfloor$) * Power(a, $\lfloor n/2 \rfloor$)

$$C(n) = 2C(n/2) + 1$$

$$T(n) \in \Theta(n)$$

Finding $\mathbf{a^n}$ using **Decrease-and-Conquer** approach.

Algorithm Power(a, n)

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I^+}$

//Output: a^n

...

Finding a^n using **Decrease-and-Conquer** approach.

Algorithm Power(a, n)

//Computes $a^n = a^{n-1} * a$

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

if ($n = 0$) **return** 1

return Power(a, n-1) * a

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

$$T(n) \in \Theta(n)$$

Finding a^n using **Decrease-by-a-constant-factor-and-Conquer** approach.

Algorithm Power(a, n)

//Computes $a^n = (a^{\lfloor n/2 \rfloor})^2 * a^{n \bmod 2}$

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

if ($n = 0$) **return** 1

$p \leftarrow \text{Power}(a, \lfloor n/2 \rfloor)$

$p \leftarrow p * p$

if (n is odd) $p \leftarrow p * a$

return p

What is its time complexity?

$C(n) = \dots$

$T(n) \in \boldsymbol{\Theta}(\dots)$

Finding a^n using **Decrease-by-a-constant-factor-and-Conquer** approach.

Algorithm Power(a, n)

//Computes $a^n = (a^{\lfloor n/2 \rfloor})^2 * a^{n \bmod 2}$

//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$

//Output: a^n

if ($n = 0$) **return** 1

$p \leftarrow \text{Power}(a, \lfloor n/2 \rfloor)$

$p \leftarrow p * p$

if (n is odd) $p \leftarrow p * a$

return p

$$C(n) = C(n/2) + 2$$

$$T(n) \in \boldsymbol{\Theta}(\log n)$$

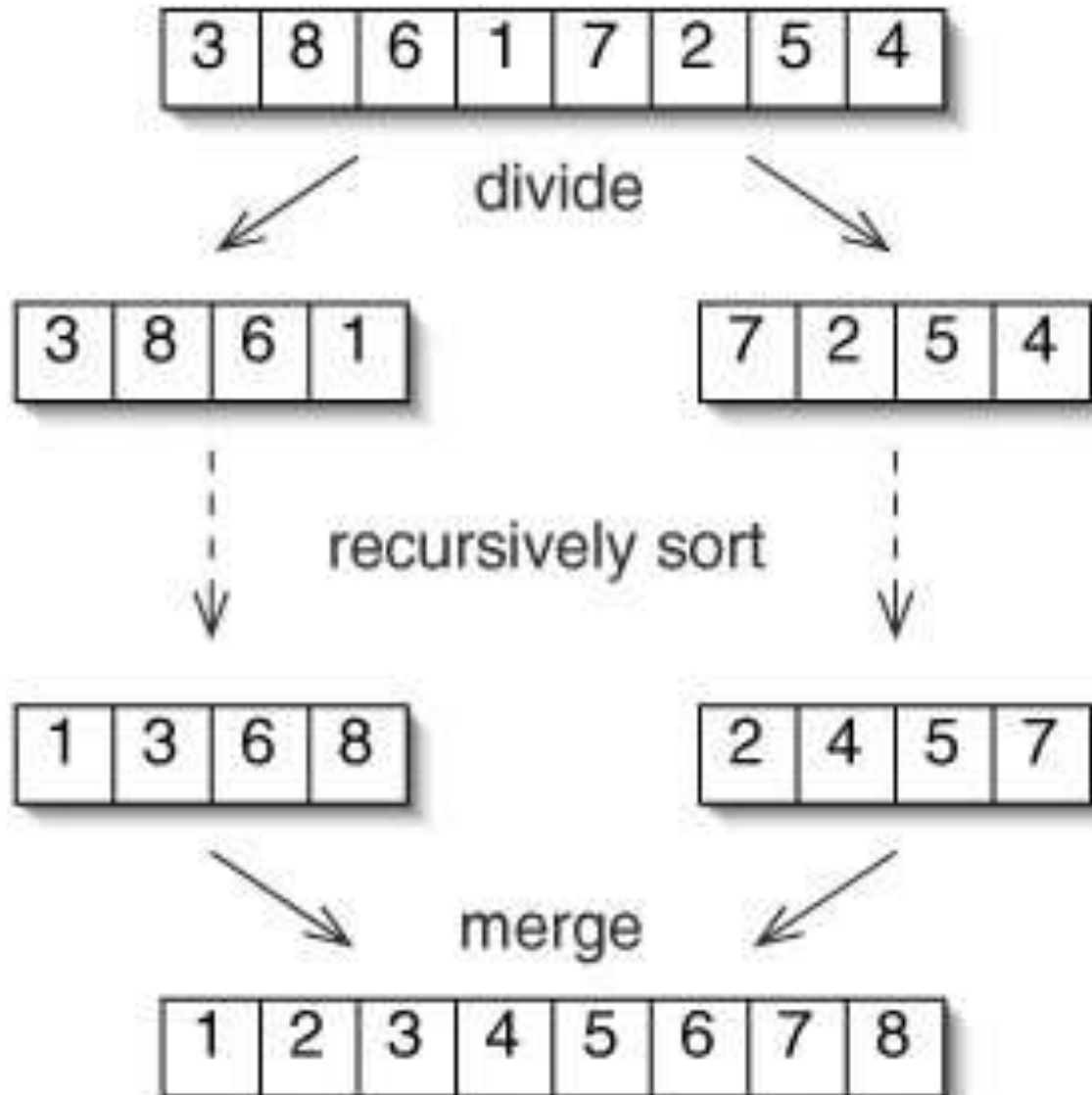
Finding a^n using different approaches.

- Brute-Force approach in $\Theta(n)$
 - $a^n = a * a * \dots a$ (n times)
- Divide-and-Conquer approach in $\Theta(n)$
 - $a^n = a^{\lfloor n/2 \rfloor} * a^{\lfloor n/2 \rfloor}$
- Decrease-by-a-constant-and-Conquer in $\Theta(n)$
 - $a^n = a^{n-1} * a$
- Decrease-by-a-constant-factor-and-Conquer in $\Theta(\log n)$
 - $a^n = (a^{\lfloor n/2 \rfloor})^2 * a^{n \bmod 2}, a^0 = 1$
 - $a^n = (a^{n/2})^2$ when n is even
 $a^n = a * (a^{(n-1)/2})^2$ when n is odd and
 $a^0 = 1$

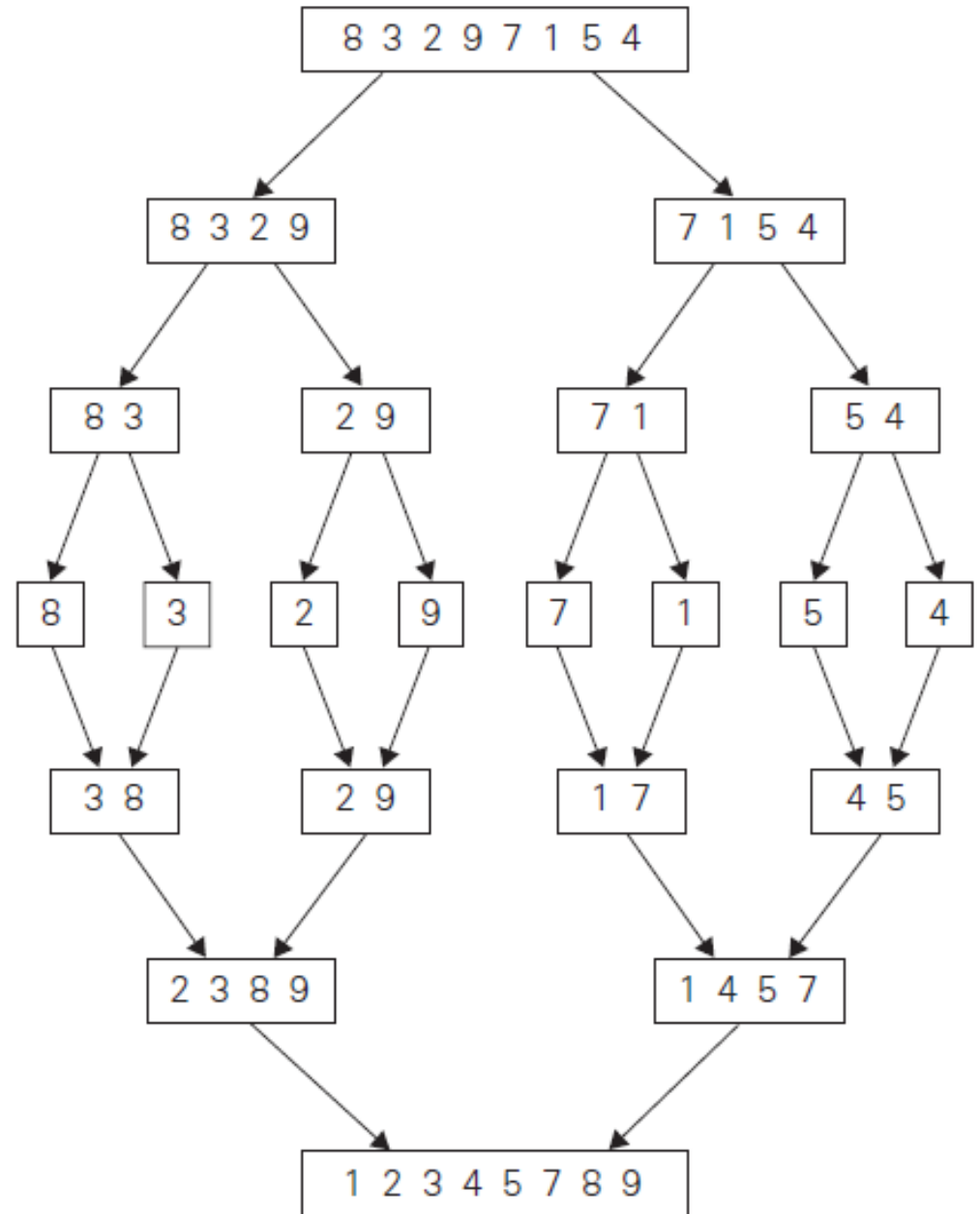
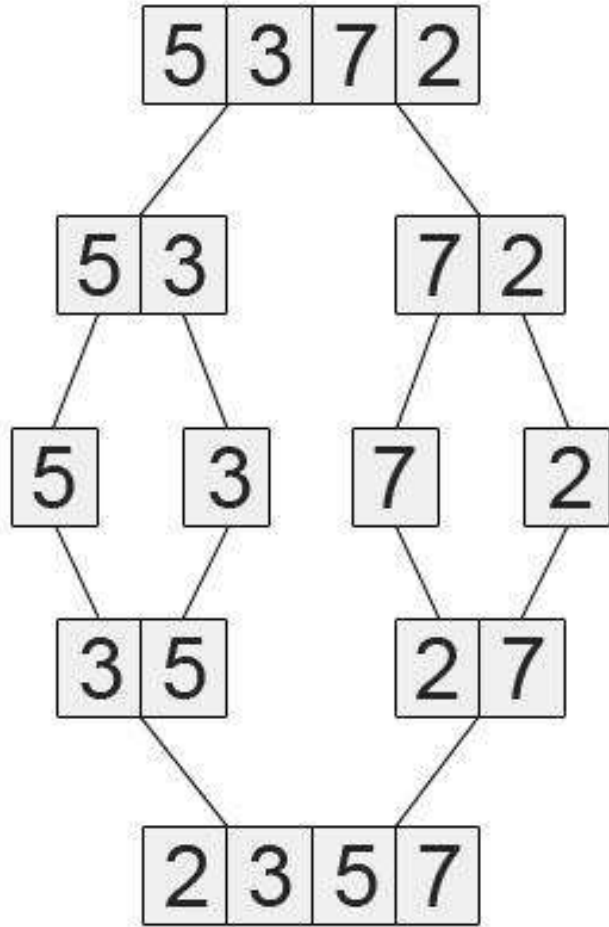
Divide-and-Conquer Examples:

- Sorting: Mergesort and Quicksort
- Search: Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Binary tree traversals

Idea of Merge Sort



Recursion tree of Merge Sort



Algorithm MergeSort($A[0..n-1]$)

```
//Sorts array  $A[0..n-1]$  by recursive Merge Sort  
//Procedure Merge( $A[0..n-1]$ ,  $m$ ) merges two  
// sorted subarrays  $A[0..m-1]$  and  $A[m..n-1]$   
// into a sorted array  $A[0..n-1]$ .
```

```
    if ( $n \leq 1$ ) return
```

```
     $m = \lfloor n/2 \rfloor$ 
```

```
    MergeSort( $A[0..m-1]$ )
```

```
    MergeSort( $A[m..n-1]$ )
```

```
    Merge( $A[0..n-1]$ ,  $m$ )
```

Merge two sorted arrays into a sorted array:

Array1	Array2	Merged
01	02	01
05	03	02
06	04	03
	07	04
	08	05
	09	06
		07
		08
		09

Two sorted arrays concatenated:

01, 05, 06, 02, 03, 04, 07, 08, 09

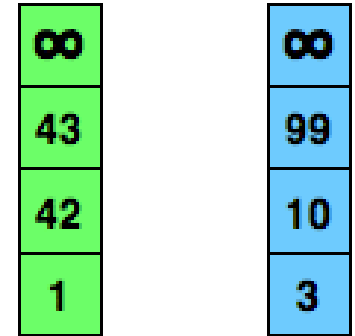
After merging: 01, 02, 03, 04, 05, 06, 07, 08, 09

Example for merging two sorted arrays:

List1: 2, 4, 5, 6, 8, 9

List2: 1, 3, 7

Merged: 1, 2, 3, 4, 5, 6, 7, 8, 9



Algorithm Merge($A[0..n-1]$, m)

//Merges two sorted arrays $A[0..m-1]$ and $A[m..n-1]$ into
//the sorted array $A[0..n-1]$

$i \leftarrow 0, j \leftarrow m, k \leftarrow 0$

while($i < m$ and $j < n$) **do**

if($A[i] \leq A[j]$) $B[k] \leftarrow A[i]; i \leftarrow i+1$

else $B[k] \leftarrow A[j]; j \leftarrow j+1$

$k \leftarrow k+1$

if($j = n$) **Copy** $A[i..m-1]$ **to** $B[k..n-1]$

else **Copy** $A[j..n-1]$ **to** $B[k..n-1]$

Copy $B[0..n-1]$ **to** $A[0..n-1]$

Analysis?

Algorithm Merge($A[0..n-1]$, m)

//Merges two sorted arrays $A[0..m-1]$ and $A[m..n-1]$ into
//the sorted array $A[0..n-1]$

$i \leftarrow 0, j \leftarrow m, k \leftarrow 0$

while($i < m$ and $j < n$) do

if($A[i] \leq A[j]$) $B[k] \leftarrow A[i]; i \leftarrow i+1$

else $B[k] \leftarrow A[j]; j \leftarrow j+1$

$k \leftarrow k+1$

if($j = n$) Copy $A[i..m-1]$ to $B[k..n-1]$

else Copy $A[j..n-1]$ to $B[k..n-1]$

Copy $B[0..n-1]$ to $A[0..n-1]$

Input Size: n

Basic Operation : Increment operation $k \leftarrow k+1$

$C(n) = n \in \Theta(n)$

Analysis of Mergesort

Algorithm MergeSort(A[0..n-1])

//Sorts array A[0..n-1] by recursive Merge Sort

//Procedure Merge(A[0..n-1], m) merges two

// sorted subarrays A[0..m-1] and A[m..n-1]

// into a sorted array A[0..n-1].

if($n \leq 1$) **return**

m = $\lfloor n/2 \rfloor$

MergeSort(A[0..m-1])

MergeSort(A[m..n-1])

Merge(A[0..n-1], m)

Algorithm: MergeSort (A[0..n-1])

Input Size: n

Basic Operation: Basic operation in **Merge ()**

$C(n) = cn + 2 * C(n / 2)$, $C(1) = 0$, when cn is the basic operation count of **Merge ()** with input size n .

$$\begin{aligned} C(n) &= 2 C(n / 2) + cn, C(1) = 0 \\ &= 2 * [2 C(n/4) + cn/2] + cn \\ &= 4 * C(n/4) + cn + cn \\ &= 4 * [2 C(n/8) + cn/4] + 2*cn \\ &= 2^3 C(n/2^3) + 3*cn \\ &= 2^i * C(n/2^i) + i*cn \end{aligned}$$

$C(n/2^i)$ is $C(1)$ when $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

$$\begin{aligned} C(n) &= n * C(1) + (\log_2 n) * cn \\ &= cn * \log_2 n \in \Theta(n \log n) \end{aligned}$$

Algorithm: MergeSort (A[0..n-1])

Input Size: **n**

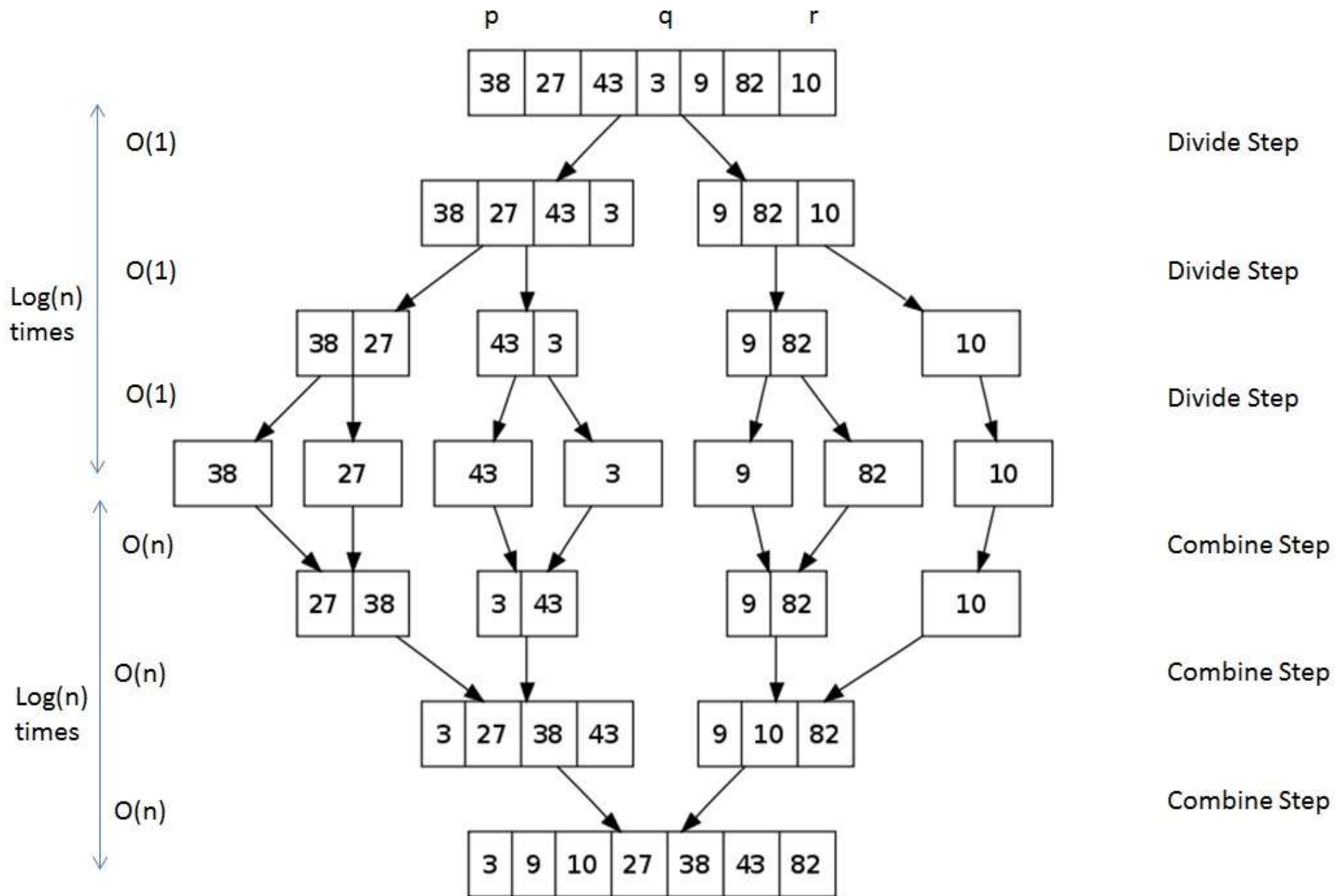
Basic Operation: ...

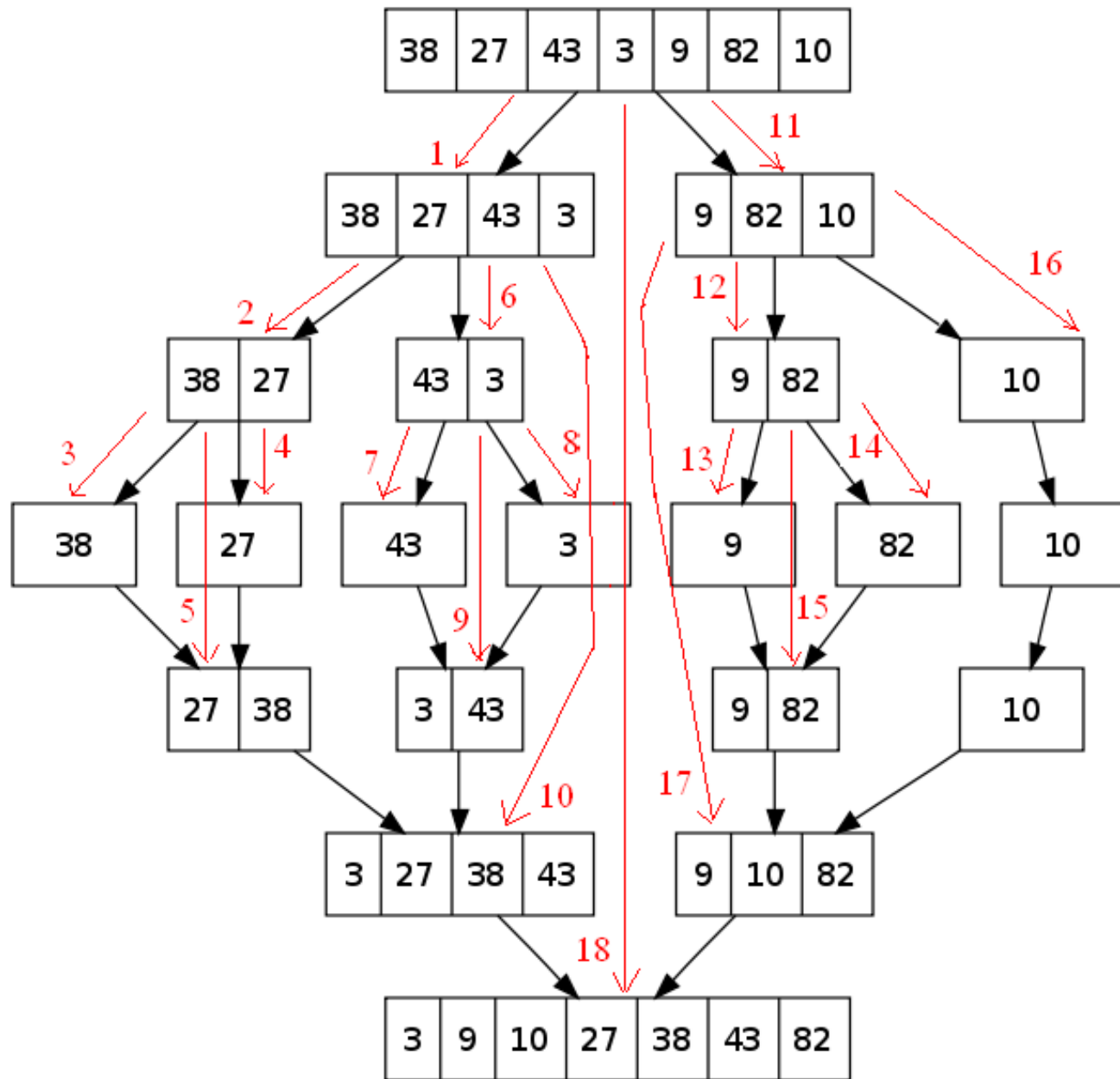
$C(n) = 2 * C(n / 2) + \mathbf{cn} + 1$, $C(1) = 1$, when **cn** is the basic operation count of **Merge ()** with input size **n**.

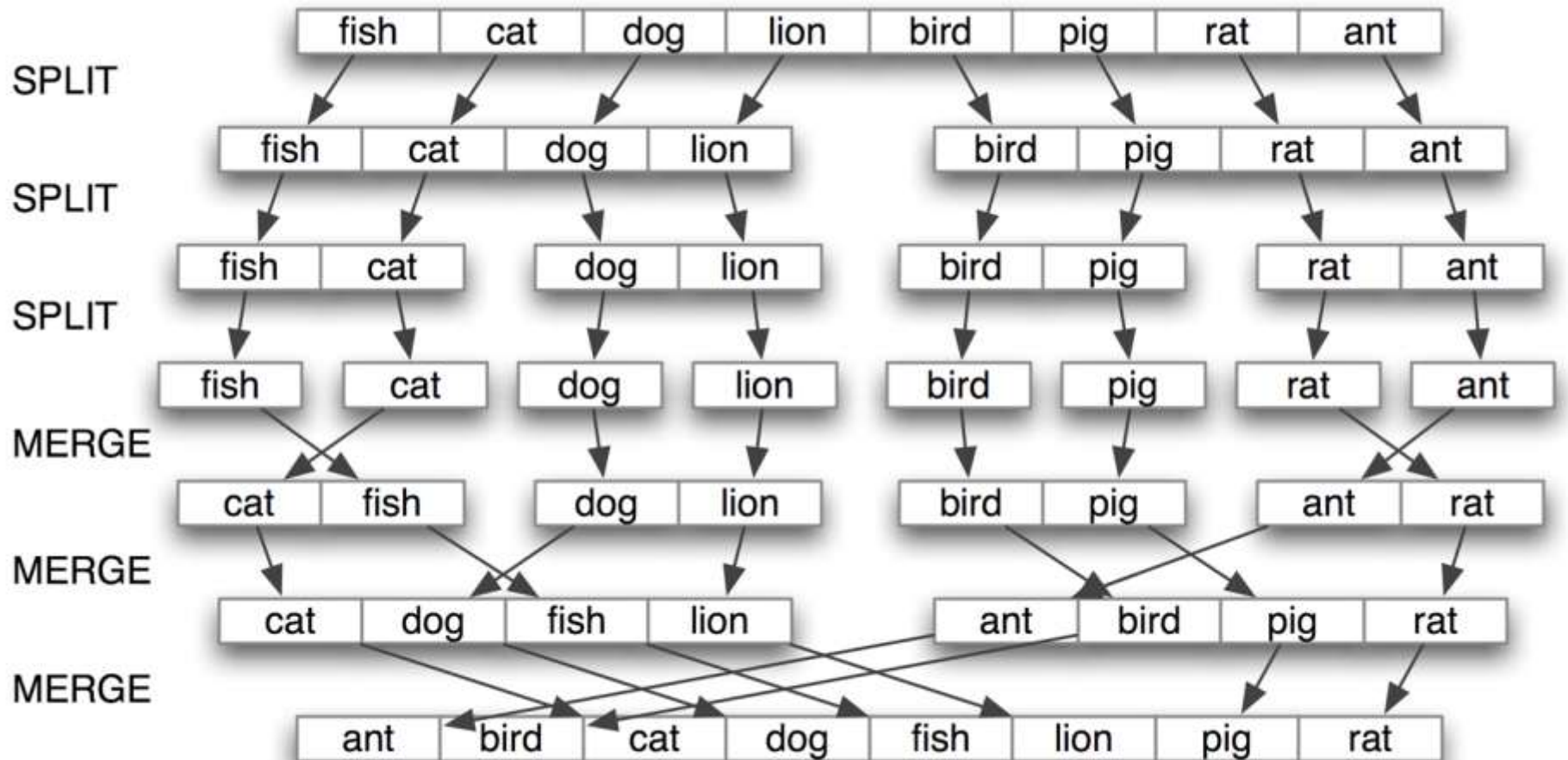
$$\begin{aligned} C(n) &= 2 C(n / 2) + cn + 1, C(1) = 0 \\ &= 2 * [2 C(n/4) + cn/2 + 1] + cn + 1 \\ &= 4 * C(n/4) + cn + cn + 2 + 1 \\ &= 4 * [2 C(n/8) + cn/4 + 1] + 2*cn + 2 + 1 \\ &= 2^3 C(n/2^3) + 3*cn + (2^3 - 1) \\ &= 2^i * C(n/2^i) + i*cn + (2^i - 1) \end{aligned}$$

$C(n/2^i)$ is $C(1)$ when $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

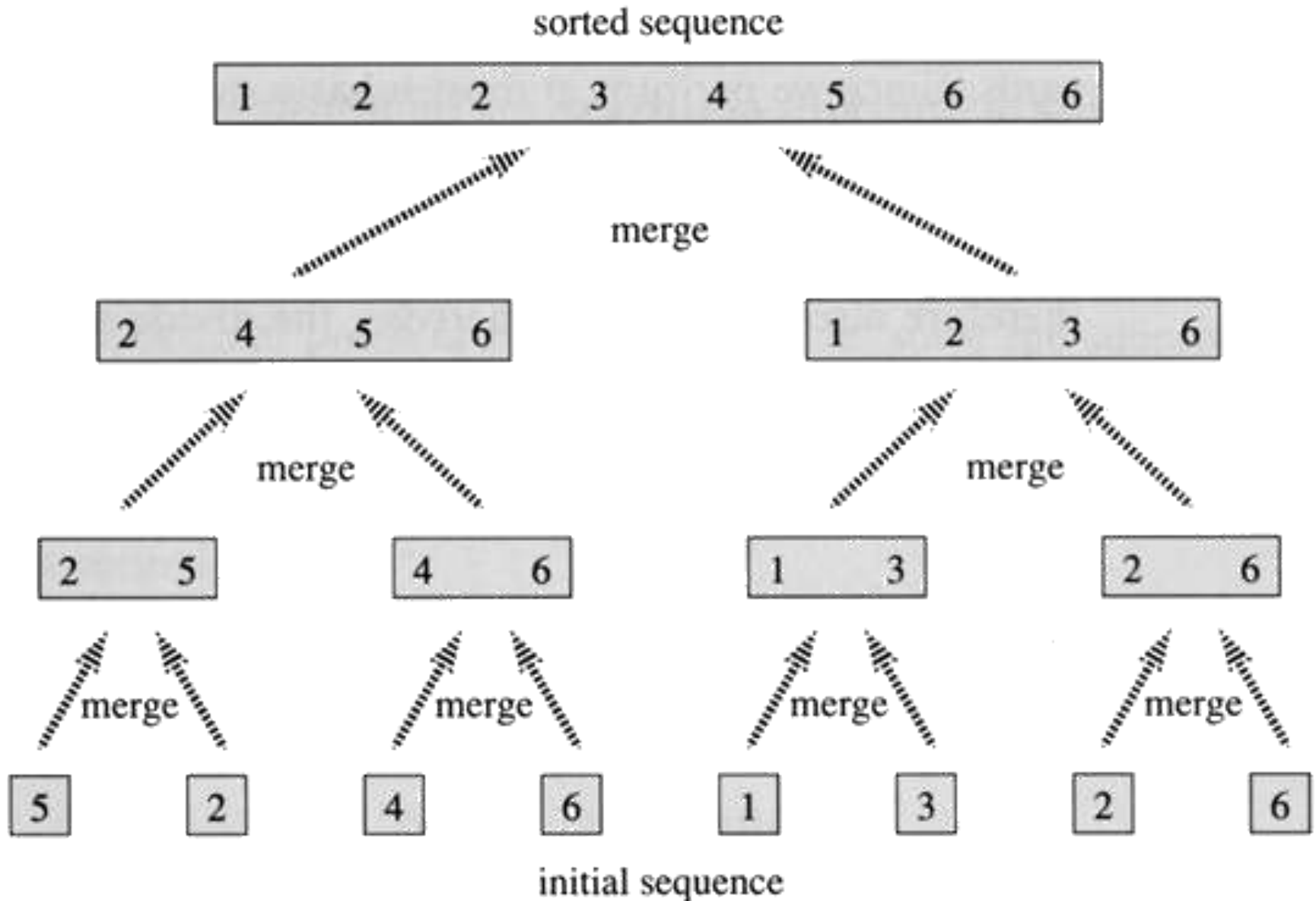
$$\begin{aligned} C(n) &= n * C(1) + (\log_2 n) * cn + (n - 1) \\ &= \mathbf{2n - 1 + cn * \log_2 n} \in \mathbf{\Theta(n \log n)} \end{aligned}$$







Non-recursive (Bottom-up) Mergesort:



Mergesort:

- Input size n being not a power of 2?
- Scope for parallelism in this algo?
- What's the basic operation in Merge Sort for Time Complexity analysis?
- Is Mergesort an in-place sorting algo?
- Is Mergesort a stable sorting algo?
- Implementation of Mergesort in iterative bottom-up approach skipping the divide stage.
- How far Mergesort is from the theoretical limit of any comparison-based sorting algos?

Problem:

Partition an array into two parts where the left part has elements \leq pivot element and the right part has the elements \geq pivot element.

Eg: 35 33 42 10 14 19 27 44 26 **31**

Let key = 31.

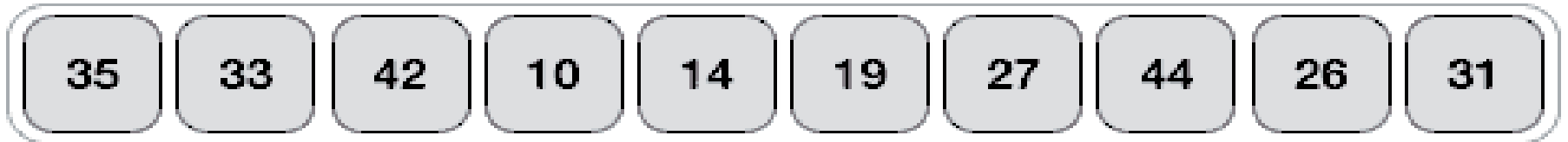
Array partitioned on the pivot element:

14 26 27 19 10 **31** 42 33 44 35

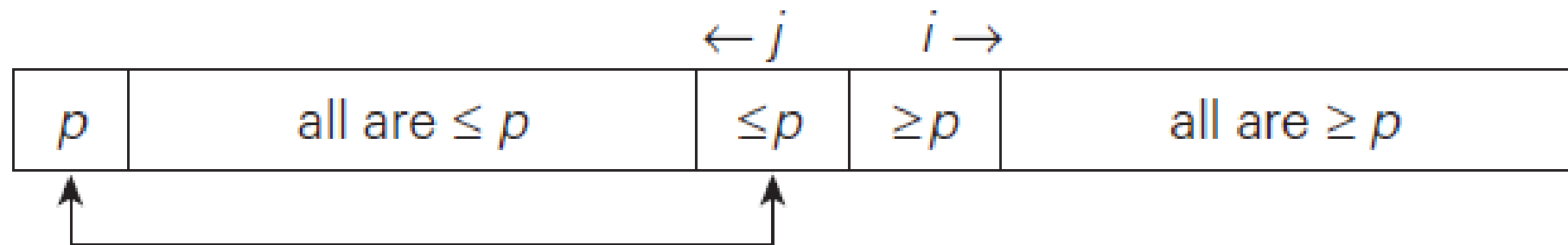
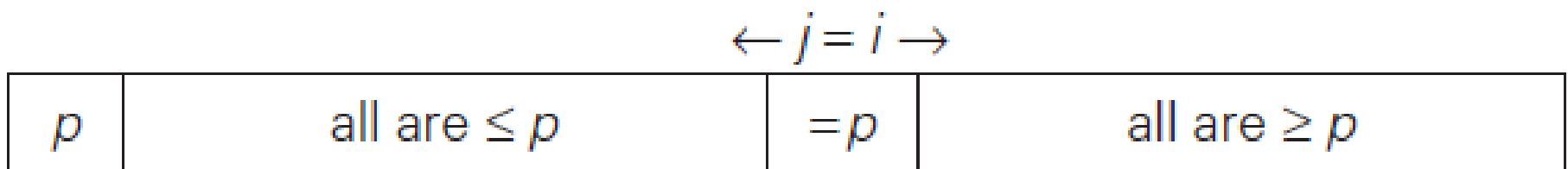
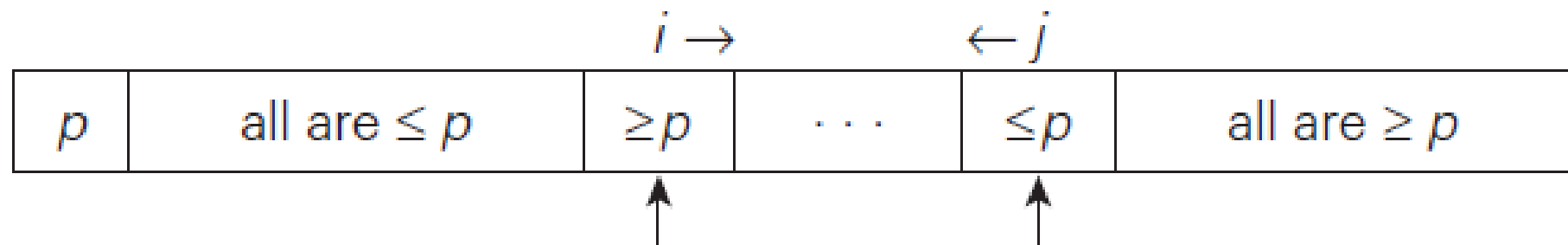
Partition an array into two parts where the left part has elements \leq key and the right part has the elements \geq key.

Eg: 35 33 42 10 14 19 27 44 26 31

Unsorted Array



$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$



ALGORITHM *HoarePartition*($A[l..r]$)

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot

//Input: Subarray of array $A[0..n - 1]$, defined by its left and right
// indices l and r ($l < r$)

//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value

$p \leftarrow A[l]$

$i \leftarrow l; j \leftarrow r + 1$

repeat

repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$

repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$

 swap($A[i], A[j]$)

until $i \geq j$

swap($A[i], A[j]$) //undo last swap when $i \geq j$

swap($A[l], A[j]$)

return j



Algorithm Partition($A[0..n-1]$)

$p \leftarrow A[0]$

$i \leftarrow 1, j \leftarrow n-1$

while($i \leq j$)

 while($i \leq j$ and $A[i] < p$) $i \leftarrow i + 1$

 while($i \leq j$ and $A[j] > p$) $j \leftarrow j - 1$

 if($i < j$)

 swap $A[i], A[j]$


$i \leftarrow i + 1$


$j \leftarrow j - 1$

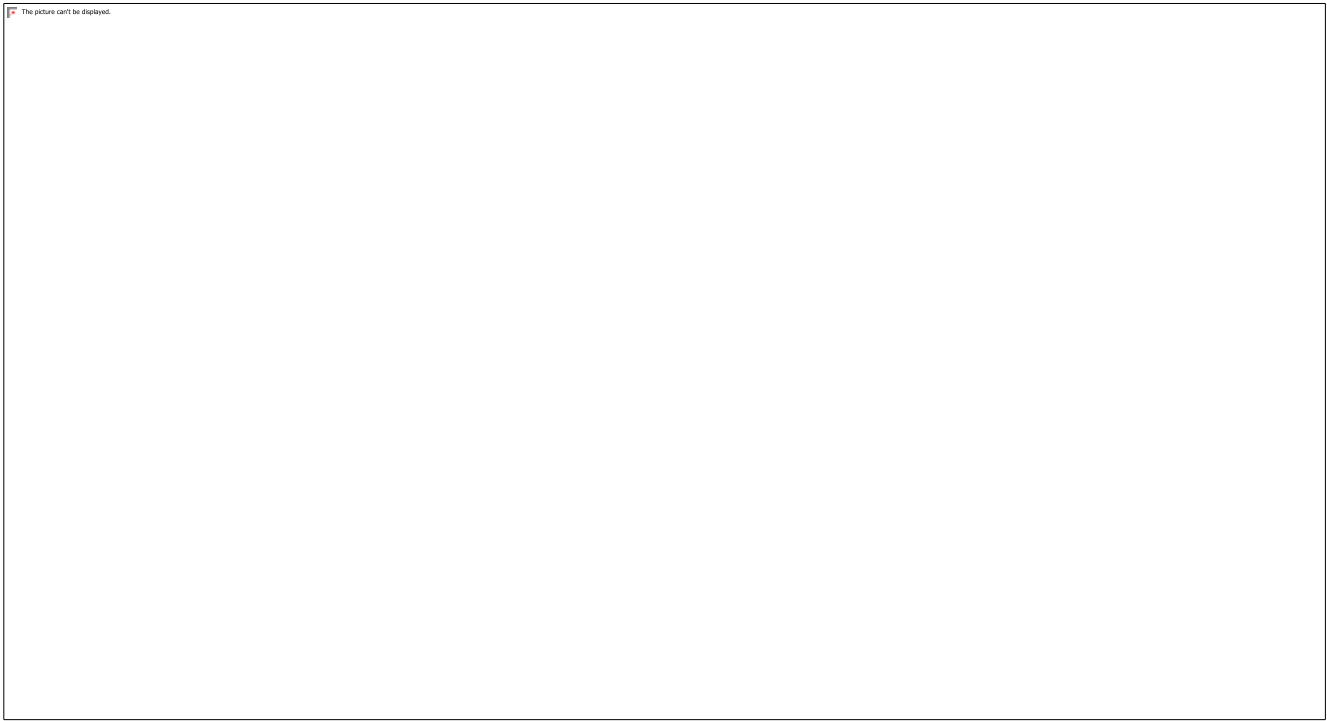
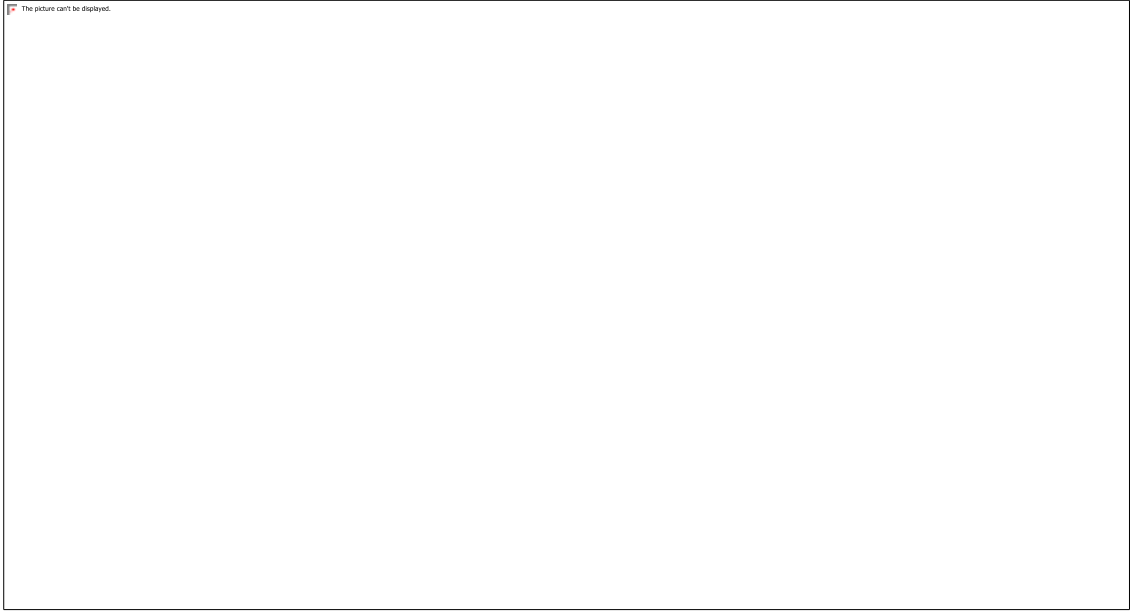
swap $A[j], A[0]$

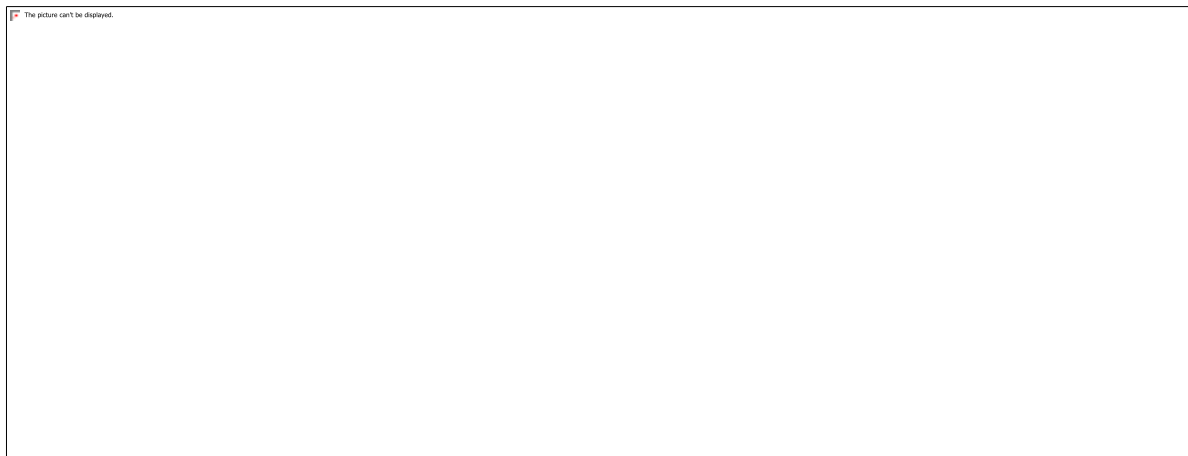
return j

Partition in the eyes of Ullas Aparanji :)


 The picture can't be displayed.

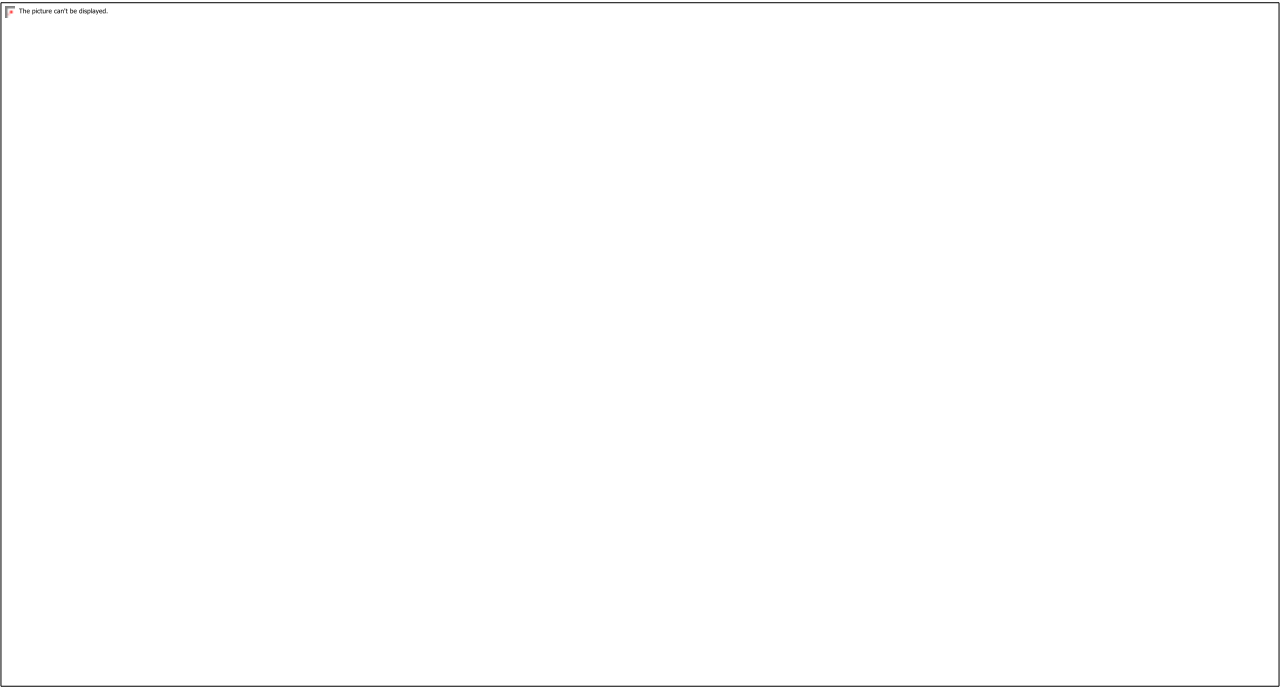
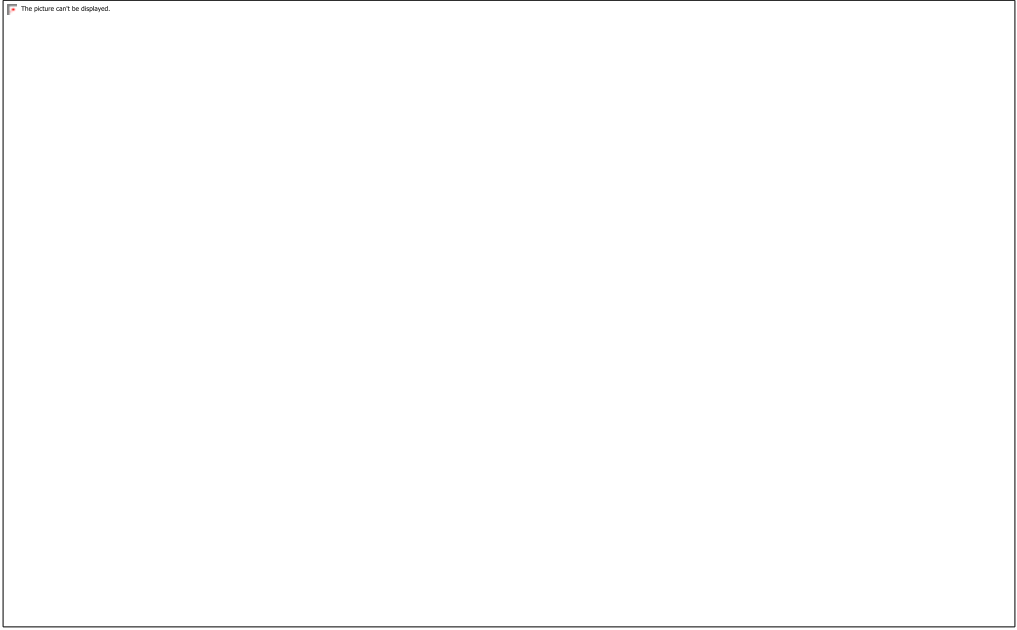
 The picture can't be displayed.

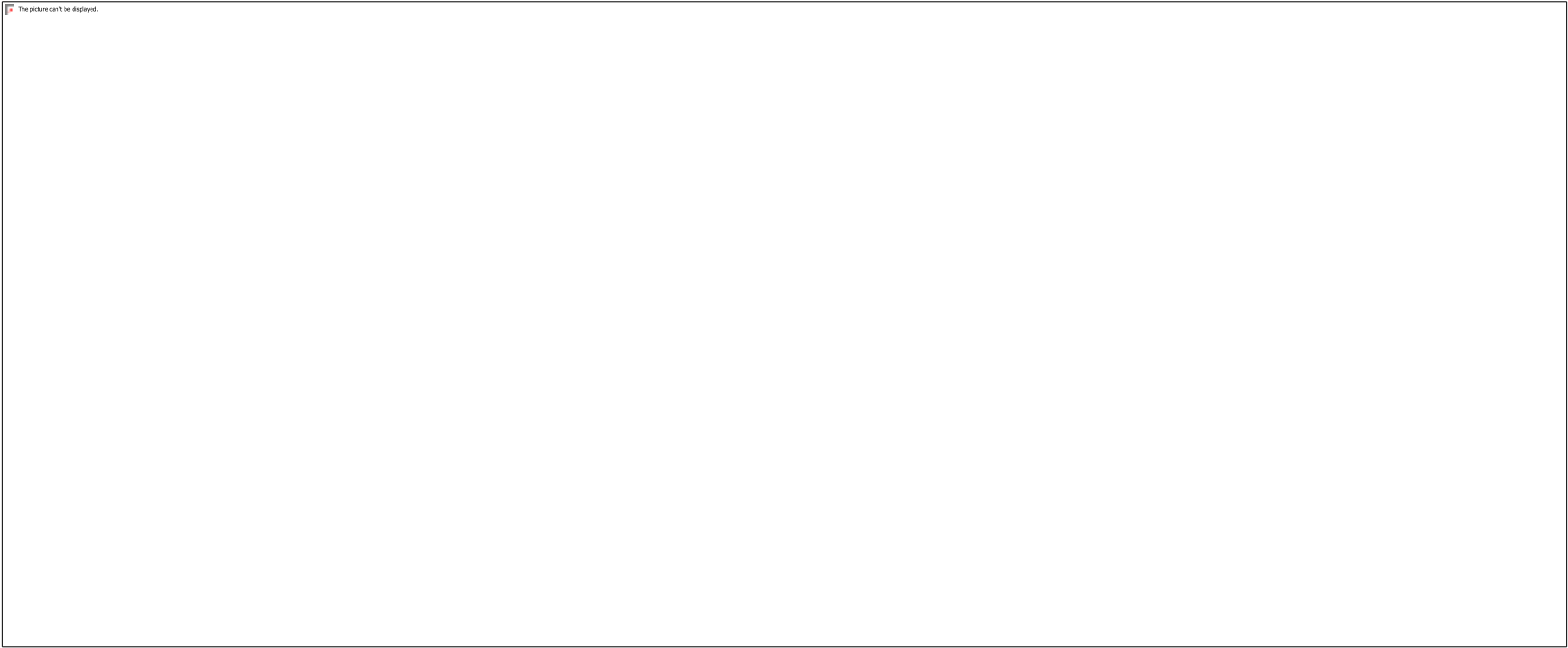
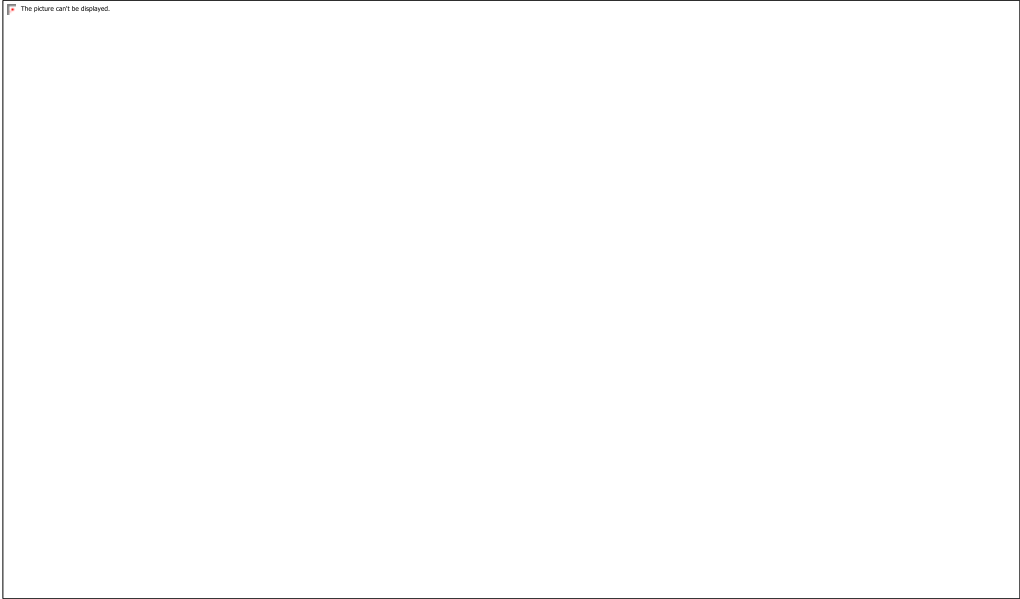


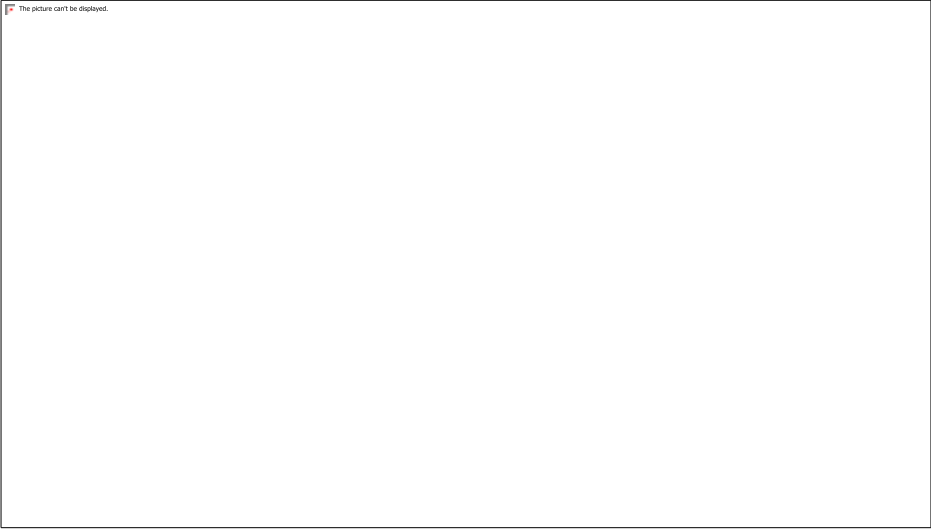


 The picture can't be displayed.

 The picture can't be displayed.





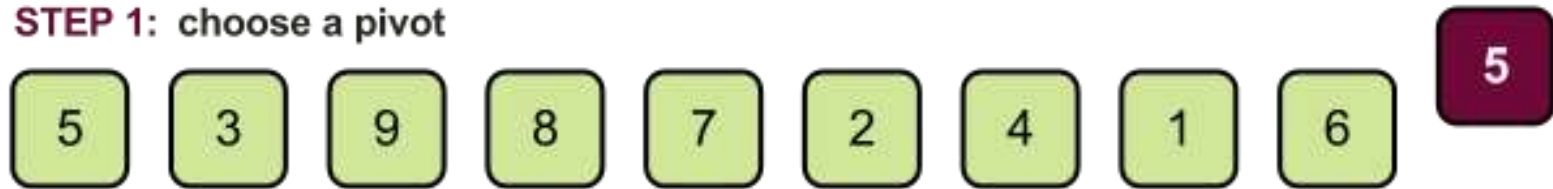


 The picture can't be displayed.

 The picture can't be displayed.

Idea of Quick Sort

STEP 1: choose a pivot



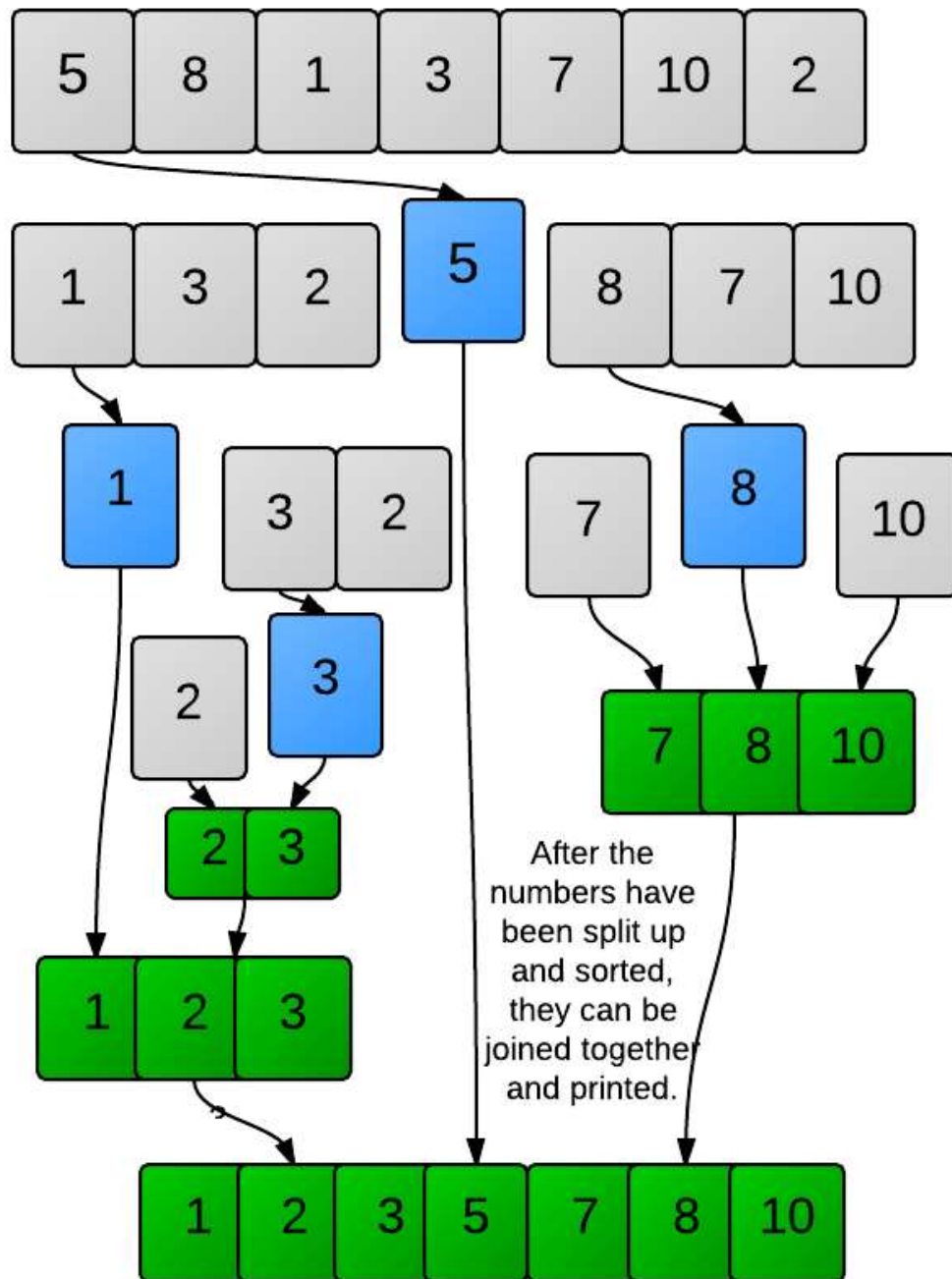
STEP 2: lesser values go to the left, greater values go to the right



STEP 3: repeat from step 1 with the two sub-lists



Quick Sort



```
public static void quicksort(char[] items, int left, int right)
{
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left + right) / 2];

    do
    {
        while ((items[i] < x) && (i < right)) i++;
        while ((x < items[j]) && (j > left)) j--;

        if (i <= j)
        {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while (i <= j);

    if (left < i) quicksort(items, left, i);
    if (i < right) quicksort(items, i, right);
}
```

Algorithm QuickSort($A[0..n-1]$)

if ($n \leq 1$) **return**

$s \leftarrow \text{Partition}(A[0..n-1])$

 QuickSort($A[0..s-1]$)

 QuickSort($A[s+1..n-1]$)

return

ALGORITHM *Quicksort*($A[l..r]$)

 //Sorts a subarray by quicksort

 //Input: Subarray of array $A[0..n-1]$, defined by its left and right

 // indices l and r

 //Output: Subarray $A[l..r]$ sorted in nondecreasing order

if $l < r$

$s \leftarrow \text{Partition}(A[l..r])$ // s is a split position

Quicksort($A[l..s-1]$)

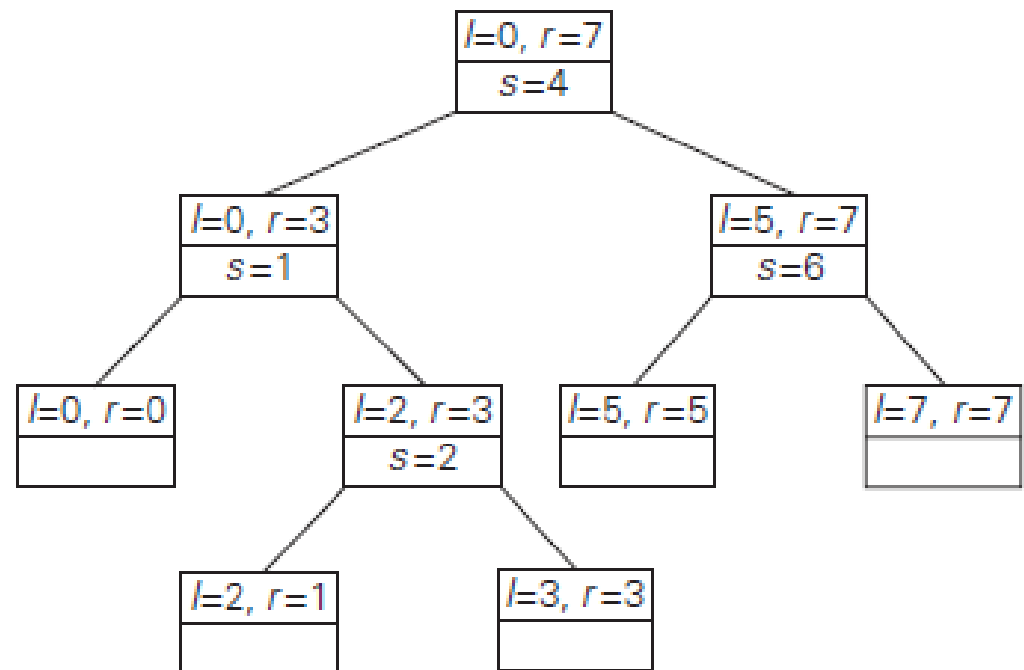
Quicksort($A[s+1..r]$)

Example

0	1	2	3	4	5	6	7
5	<i>i</i> 3	1	9	8	2	4	<i>j</i> 7
5	3	1	<i>i</i> 9	8	2	<i>j</i> 4	7
5	3	1	<i>i</i> 4	8	2	<i>j</i> 9	7
5	3	1	4	<i>i</i> 8	<i>j</i> 2	9	7
5	3	1	4	<i>i</i> 2	<i>j</i> 8	9	7
5	3	1	4	<i>j</i> 2	<i>i</i> 8	9	7
2	3	1	4	5	8	9	7
2	<i>i</i> 3	1	<i>j</i> 4		8	<i>i</i> 9	<i>j</i> 7
2	<i>i</i> 3	<i>j</i> 1	4		8	<i>i</i> 7	<i>j</i> 9
2	<i>i</i> 1	<i>j</i> 3	4		8	<i>j</i> 7	<i>i</i> 9
2	<i>j</i> 1	<i>i</i> 3	4		7	8	9
1	2	3	4				9
1		3	<i>ij</i> 4				
		<i>j</i> 3	<i>i</i> 4				
			4				

0	1	2	3	4	5	6	7
	<i>i</i>						<i>j</i>
5	3	1	9	8	2	4	7
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7
2	3	1	4				
2	3	1	4				
2	3	1	4				
2	1	3	4				
2	1	3	4				
1	2	3	4				
1							
		3	<i>ij</i>				
		3	4				
			4				
					8	9	7
					8	7	9
					8	7	9
					7	8	9
					7		
							9

Example



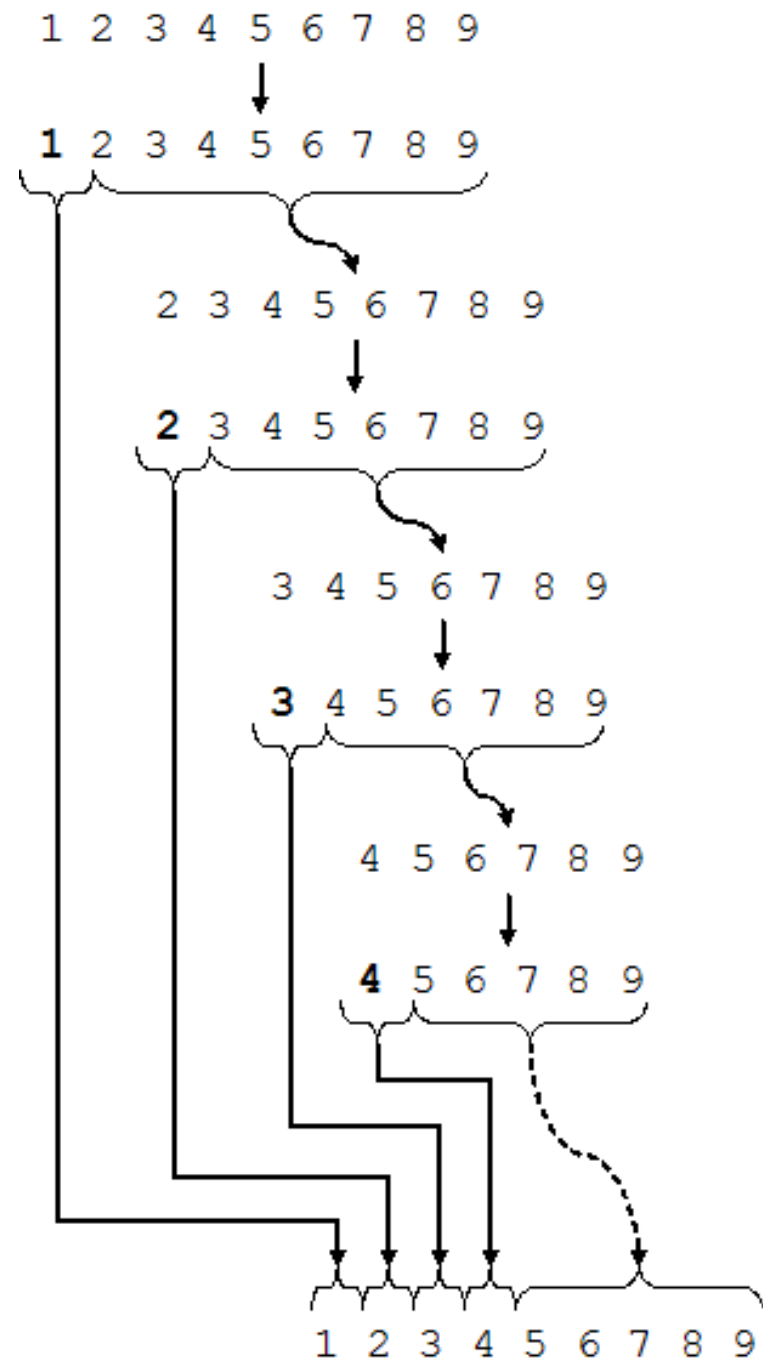
Examples of extreme cases:

Split at the end

1 2 3 4 5 6 7 8 9

Split in the middle

4 2 1 3 6 5 7



Best case:

$$C(n) = 1 + cn + 2 C(n/2), C(1) = 1$$

$$\begin{aligned} C(n) &= 2 C(n/2) + 1 + cn, C(1) = 1 \\ &= 2^i C(n/2^i) + i * cn + (2^i - 1) \end{aligned}$$

$$C(n) = \mathbf{2n - 1 + cn * \log_2 n \in \Theta(n \log n)}$$

Worst case:

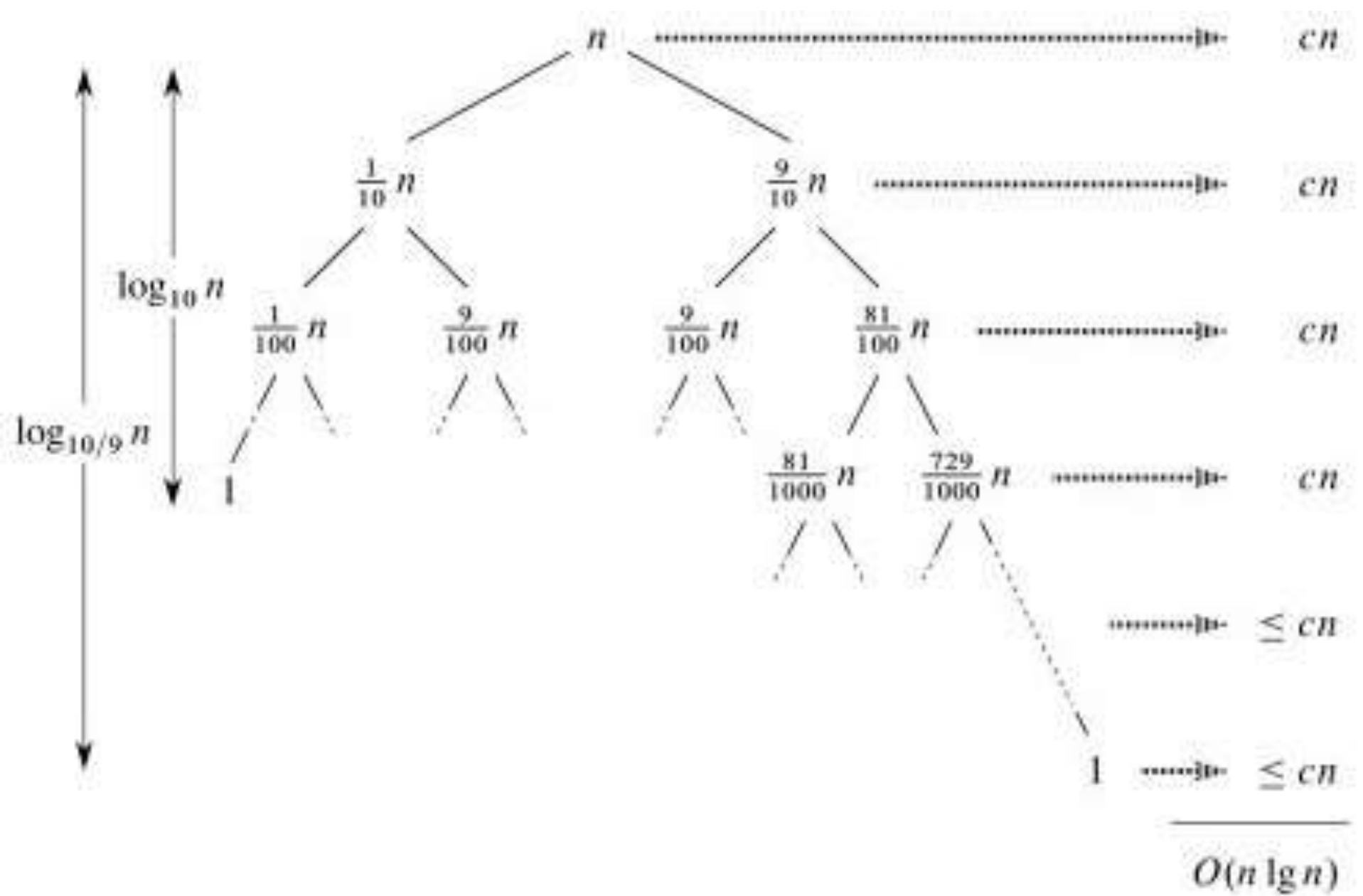
$$C(n) = 1 + cn + C(n-1), C(1) = 1$$

$$\begin{aligned} C(n) &= C(n-1) + 1 + cn, C(1) = 1 \\ &= C(n-i) + i + c(n + n-1 + \dots + n-i+1) \end{aligned}$$

$$C(n) = \mathbf{1 + (n-1) + cn(n+1)/2 \in \Theta(n^2)}$$

Avg case: $C(n) \in \mathbf{O(n^2)}$

$$C(n) \in \mathbf{\Theta(n \log n) ?}$$



Quicksort:

Avg case: $C(n) \in O(n^2)$

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)]$$

$$\text{for } n > 1, \quad C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n$$

$$\approx 1.39n \log_2 n \quad \in \quad \mathbf{\Theta(n \log n)}$$

Concluding remarks on Quicksort:

better pivot selection methods such as *randomized quicksort* that uses a random element or the *median-of-three* method that uses the median of the leftmost, rightmost, and the middle element of the array

switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array

modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot

Is Quicksort a Stable Sorting algorithm?

Recursion needs stack space. Skewed recursion needs more stack space. Deal with it?

...

Binary Search:

Efficient algorithm for searching in a **sorted array**.

Search for key element K in an Array A having n elements.

Let $m = \lfloor (n-1)/2 \rfloor$
 K

vs

$A[0] \quad . \quad . \quad . \quad A[m] \quad . \quad . \quad . \quad A[n-1]$

If $K = A[m]$, stop (successful search);

otherwise, continue searching by the same method

in $A[0..m-1]$ if $K < A[m]$

and in $A[m+1..n-1]$ if $K > A[m]$

Binary Search:

Algorithm BinarySearchRec ($A[0..n-1]$, K)

 if ($n \leq 0$)

 return -1

$m = \lfloor n/2 \rfloor$

 if ($k = A[m]$)

 return m

 if ($k < A[m]$)

 return BinarySearchRec ($A[0..m-1]$, K)

 else

 return BinarySearchRec ($A[m+1..n-1]$,

K)

ALGORITHM *BinarySearch*($A[0..n - 1]$, K)

//Implements nonrecursive binary search

//Input: An array $A[0..n - 1]$ sorted in ascending order and

// a search key K

//Output: An index of the array's element that is equal to K

// or -1 if there is no such element

$l \leftarrow 0$; $r \leftarrow n - 1$

while $l \leq r$ **do**

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

if $K = A[m]$ **return** m

else if $K < A[m]$ $r \leftarrow m - 1$

else $l \leftarrow m + 1$

return -1

Algorithm: BinarySearchRec (A[0..n-1] , k)

Input Size: n

Basic Operation : ($k = A[m]$)

Worst case:

$$\begin{aligned}C(n) &= C(n/2) + 1, C(1) = 1 \\&= C(n/4) + 1 + 1 = C(n/4) + 2 \\&= C(n/8) + 3 \\&= C(n/2^4) + 4 \\&= C(n/2^i) + i\end{aligned}$$

$C(n/2^i)$ is $C(1)$ when $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

$$\begin{aligned}C(n) &= C(1) + \log_2 n \\&= 1 + \log_2 n \in \Theta(\log n)\end{aligned}$$

Algorithm: BinarySearchRec (A[0..n-1] , K)

Input Size: n

Basic Operation : $(k = A[m])$

Worst case:

$$C(n) = C(n/2) + 1, C(1) = 1$$

$$C(n) = 1 + \log_2 n \in \Theta(\log n)$$

$$\text{Best case: } C(n) = 1 \in \Theta(1)$$

$$\text{Avg case: } C(n) \in O(\log n) \in \Theta(?)$$

We can prove $C(n) \in \Theta(\log n)$

Consider the problem of multiplying two (large) n -digit integers represented by arrays of their digits such as:

Brute-Force Strategy:

$$\begin{array}{r} 2135 \quad * \quad 4014 \\ 8540 \\ 2135+ \\ 0000++ \\ 8540+++ \\ \hline 8569890 \end{array}$$

Write a brute-force algorithm to multiply two arbitrarily large (of n digits) integers.

```

      12345678 * 32165487
      86419746
      98765424+
      49382712++
      61728390+++
      74074068++++
      12345678+++++
      24691356++++++
      37037034+++++++
      -----
    397104745215186

```

Basic Operation:
single-digit multiplication

$C(n) = n^2$ one-digit multiplications
 $C(n) \in \Theta(n^2)$

Multiplication of Large Integers by Divide-and-Conquer

Idea: To multiply $A = 23$ and $B = 54$.

$$A = (2 \cdot 10^1 + 3), \quad B = (5 \cdot 10^1 + 4)$$

$$A * B = (2 \cdot 10^1 + 3) * (5 \cdot 10^1 + 4)$$

$$= 2 * 5 \cdot 10^2 + (2 * 4 + 3 * 5) \cdot 10^1 + 3 * 4$$

For a base value 'x',

$$A = 2x + 3 \text{ and } B = 5x + 4$$

$$A = (2 \cdot x^1 + 3), \quad B = (5 \cdot x^1 + 4)$$

$$A * B = (2 \cdot x^1 + 3) * (5 \cdot x^1 + 4)$$

$$= 2 * 5 \cdot x^2 + (2 * 4 + 3 * 5) \cdot x^1 + 3 * 4$$

Multiplication of Large Integers by Divide-and-Conquer:

Idea:

To find $A * B$ where $A = 2140$ and $B = 3514$

$$A = (21 \cdot 10^2 + 40), \quad B = (35 \cdot 10^2 + 14)$$

$$\begin{aligned} \text{So, } A * B &= (21 \cdot 10^2 + 40) * (35 \cdot 10^2 + 14) \\ &= 21 * 35 \cdot 10^4 + (21 * 14 + 40 * 35) \cdot 10^2 + 40 * 14 \end{aligned}$$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n -digit, A_1 , A_2 , B_1 and B_2 are $n/2$ -digit numbers),

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

Multiplication of Large Integers by Divide-and-Conquer:

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n -digit, A_1 , A_2 , B_1 and B_2 are $n/2$ -digit numbers),

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

Trivial case: When $n = 1$, just multiply $A*B$ directly.

\therefore One multiplication of n -digit integers, requires four multiplications of $n/2$ -digit integers, when $n > 1$.

Basic operation: single-digit multiplication

$$C(n) = 4C(n/2), \quad C(1) = 1$$

$$\therefore C(n) \in \Theta(n^2)$$

Multiplication of Large Integers by **Karatsuba algorithm**:

$$A * B = \textcolor{red}{A}_1 * \textcolor{blue}{B}_1 \cdot 10^n + (\textcolor{red}{A}_1 * \textcolor{blue}{B}_2 + \textcolor{red}{A}_2 * \textcolor{blue}{B}_1) \cdot 10^{n/2} + \textcolor{red}{A}_2 * \textcolor{blue}{B}_2$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(\textcolor{red}{A}_1 * \textcolor{blue}{B}_2 + \textcolor{red}{A}_2 * \textcolor{blue}{B}_1) = (\textcolor{red}{A}_1 + \textcolor{red}{A}_2) * (\textcolor{blue}{B}_1 + \textcolor{blue}{B}_2) - \textcolor{red}{A}_1 * \textcolor{blue}{B}_1 - \textcolor{red}{A}_2 * \textcolor{blue}{B}_2,$$

which requires only 3 multiplications at the expense of 3 extra add/sub operations. Note that we are reusing $\textcolor{red}{A}_1 * \textcolor{blue}{B}_1$ and $\textcolor{red}{A}_2 * \textcolor{blue}{B}_2$ one more time.

$$A * B = \textcolor{red}{A}_1 * \textcolor{blue}{B}_1 \cdot 10^n +$$

$$[(\textcolor{red}{A}_1 + \textcolor{red}{A}_2) * (\textcolor{blue}{B}_1 + \textcolor{blue}{B}_2) - \textcolor{red}{A}_1 * \textcolor{blue}{B}_1 - \textcolor{red}{A}_2 * \textcolor{blue}{B}_2] \cdot 10^{n/2} + \textcolor{red}{A}_2 * \textcolor{blue}{B}_2$$

Multiplication of Large Integers by **Karatsuba algorithm**:

$$A = \mathbf{A_1} \mathbf{A_2}, B = \mathbf{B_1} \mathbf{B_2}$$

It needs three $n/2$ digits multiplications:

$$\mathbf{P_1} = \mathbf{A_1} * \mathbf{B_1},$$

$$\mathbf{P_2} = \mathbf{A_2} * \mathbf{B_2} \text{ and}$$

$$\mathbf{P_3} = (\mathbf{A_1} + \mathbf{A_2}) * (\mathbf{B_1} + \mathbf{B_2})$$

$$A * B = \mathbf{A_1} * \mathbf{B_1} \cdot 10^n +$$

$$((\mathbf{A_1} + \mathbf{A_2}) * (\mathbf{B_1} + \mathbf{B_2}) - \mathbf{A_1} * \mathbf{B_1} - \mathbf{A_2} * \mathbf{B_2}) \cdot 10^{n/2} + \mathbf{A_2} * \mathbf{B_2}$$

is equivalent to:

$$A * B = \mathbf{P_1} \cdot 10^n + (\mathbf{P_3} - \mathbf{P_1} - \mathbf{P_2}) \cdot 10^{n/2} + \mathbf{P_2}$$

Multiplication of Large Integers by **Karatsuba algorithm**:

```
Algorithm Karatsuba(a[0..n-1], b[0..n-1])
    if(n = 1) return a[0]*b[0]
    if(n is odd) n ← n+1 with a leading 0 padded.
    m = n/2
    a1, a2 = split_at(a, m)
    b1, b2 = split_at(b, m)
    p1 = Karatsuba(a1[0..m-1], b1[0..m-1])
    p2 = Karatsuba(a2[0..m-1], b2[0..m-1])
    p3 = Karatsuba((a1+a2)[0..m], (b1+b2)[0..m])
    return (p1.10n + (p3-p1-p2).10m + p2)[0..2n-1]
```

Time complexity of the **Karatsuba algorithm**:

$$C(n) = 3C(n/2), \quad C(1) = 1$$

$$C(n) = 3^{\log n} = n^{\log 3} \approx n^{1.585}$$

$$C(n) \in \Theta(n^{\log 3}) \approx \Theta(n^{1.585})$$

The number of single-digit multiplications needed to multiply two 1024-digit ($n = 1024 = 2^{10}$) numbers is:

Karatsuba algorithm requires $3^{\log n} = 3^{10} = 59,049$,

Classical D-n-C algorithm requires $4^{\log n} = 4^{10} = 1,048,576$.

Multiplication of Large Integers by **Karatsuba algorithm**:

$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0 \end{aligned}$$

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4$$

$$a = a_1a_0 \text{ and } b = b_1b_0$$

$$c = a * b = c_210^2 + c_110^1 + c_0$$

$$c_2 = a_1 * b_1 \text{ is the product of their first digits,}$$

$$c_0 = a_0 * b_0 \text{ is the product of their second digits,}$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0) \text{ is the product of the sum of the } a\text{'s digits and the sum of the } b\text{'s digits minus the sum of } c_2 \text{ and } c_0$$

Matrix Multiplication:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00}+a_{01}b_{10} & a_{00}b_{01}+a_{01}b_{11} \\ a_{10}b_{00}+a_{11}b_{10} & a_{10}b_{01}+a_{11}b_{11} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{bmatrix}$$

Multiplication of two 2X2 matrices requires
8 element-level multiplications and
4 element-level additions.

Matrix Multiplication by Divide-and-Conquer strategy:

Let A and B be two $n \times n$ matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B and their product C into four $n/2 \times n/2$ submatrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

Basic operation: atomic-element multiplication

$$C(n) = 8C(n/2), \quad C(1) = 1$$

$$\therefore C(n) \in \Theta(n^3)$$

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$

$$m_2 = (a_{10} + a_{11}) * b_{00},$$

$$m_3 = a_{00} * (b_{01} - b_{11}),$$

$$m_4 = a_{11} * (b_{10} - b_{00}),$$

$$m_5 = (a_{00} + a_{01}) * b_{11},$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

**Strassen's
Matrix
Multiplication**

Asymptotic Efficiency of Strassen's Matrix Multiplication:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \dots \\ &= 7^i M(2^{k-i}) \dots = 7^k M(2^{k-k}) = 7^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

which is smaller than n^3 required by the brute-force algorithm.

Asymptotic Efficiency of Strassen's Matrix Multiplication:

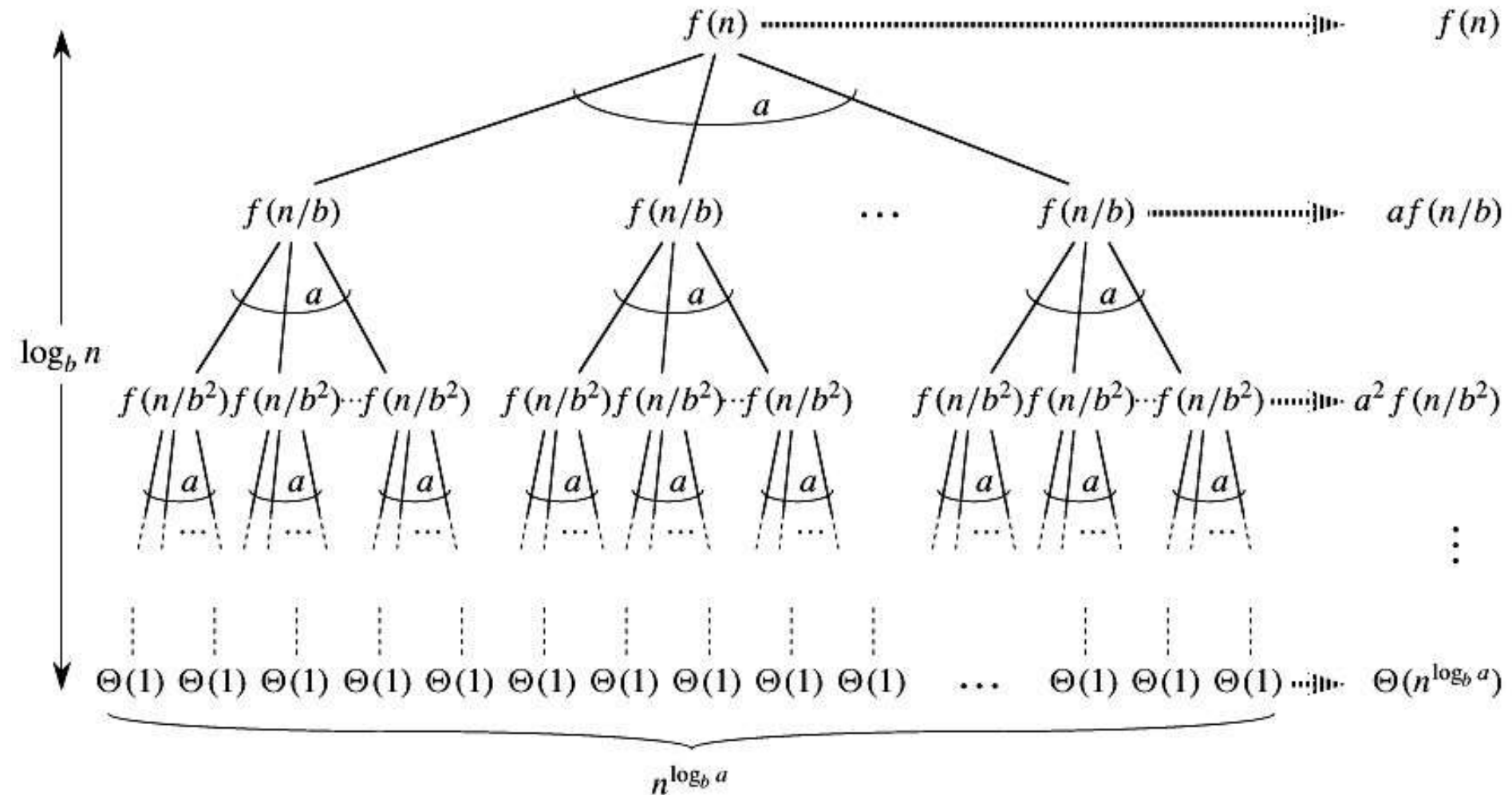
$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0$$

$$A(n) \in \Theta(n^{\log_2 7})$$

$$T(n) \in \Theta(n^{\log_2 7})$$

The fastest algorithm so far is that of Coopersmith and Winograd with its efficiency in $O(n^{2.376})$.

Divide-n-Conquer algorithms:



Master Theorem of Divide-n-Conquer:

$T(n) = a T(n/b) + c f(n)$ where $T(1) = c$ and $f(n) \in \Theta(n^d)$, $d \geq 0$

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Examples:

Array Sum: $T(n) = 2T(n/2) + 1 \in ?$

Mergesort: $T(n) = 2T(n/2) + n \in ?$

Bin Search: $T(n) = T(n/2) + 1 \in ?$

$T(n) = 4T(n/2) \in ?$, $T(n) = 3T(n/2) \in ?$

$T(n) = 3T(n/2) + n \in ?$, $T(n) = 3T(n/2) + n^2 \in ?$

Remarks on the Master Theorem:

$T(n) = a T(n/b) + c f(n)$ where $T(1) = c$ and $f(n) \in \Theta(n^d)$, $d \geq 0$

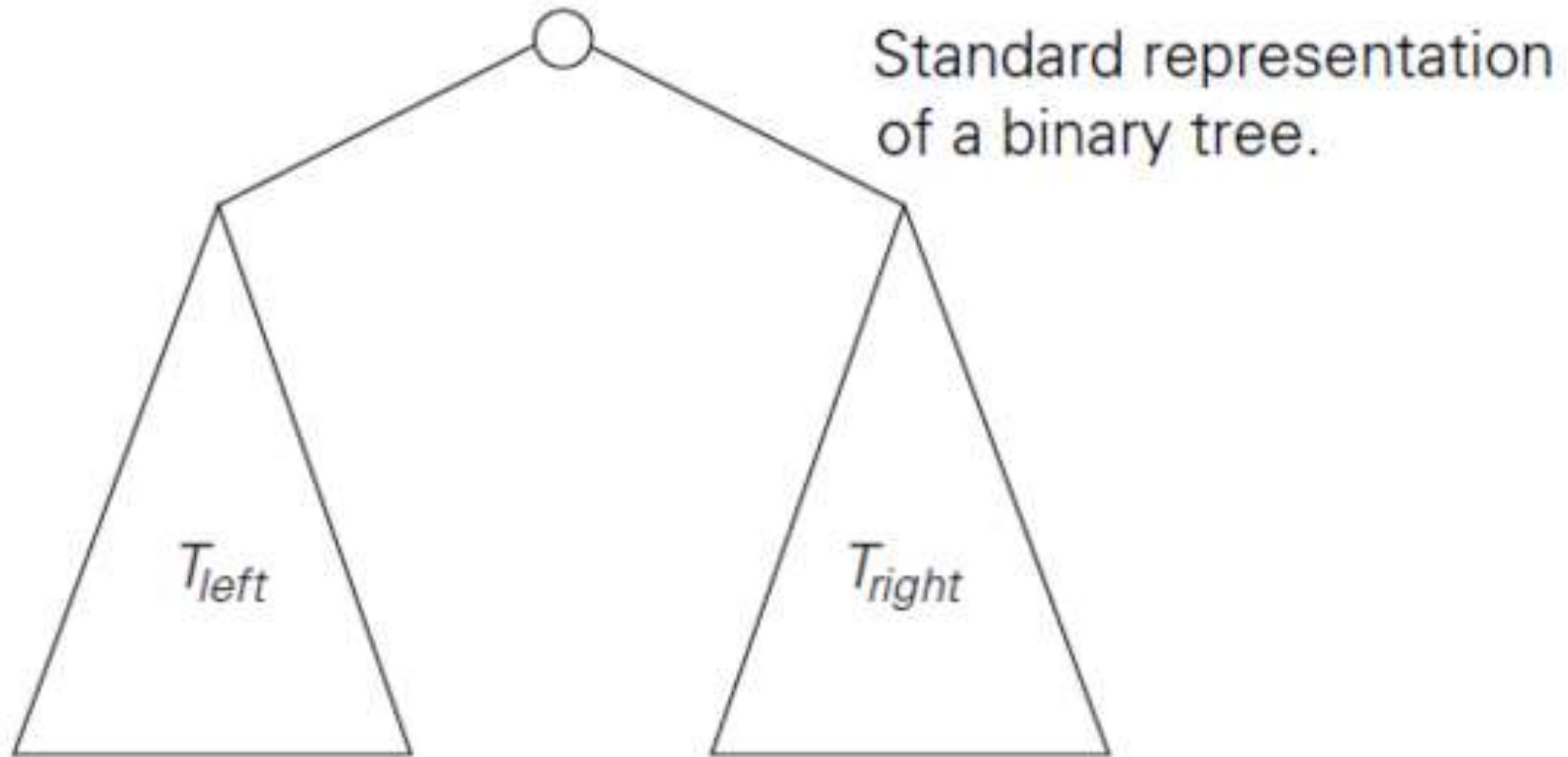
If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- When $f(n) \notin \Theta(n^d)$
- When division is not uniform
- Values of c and n_0

Binary Tree: is a divide-and-conquer ready data structure. A null node is a binary tree, and a non-null node having a left and a right binary trees is a binary tree.

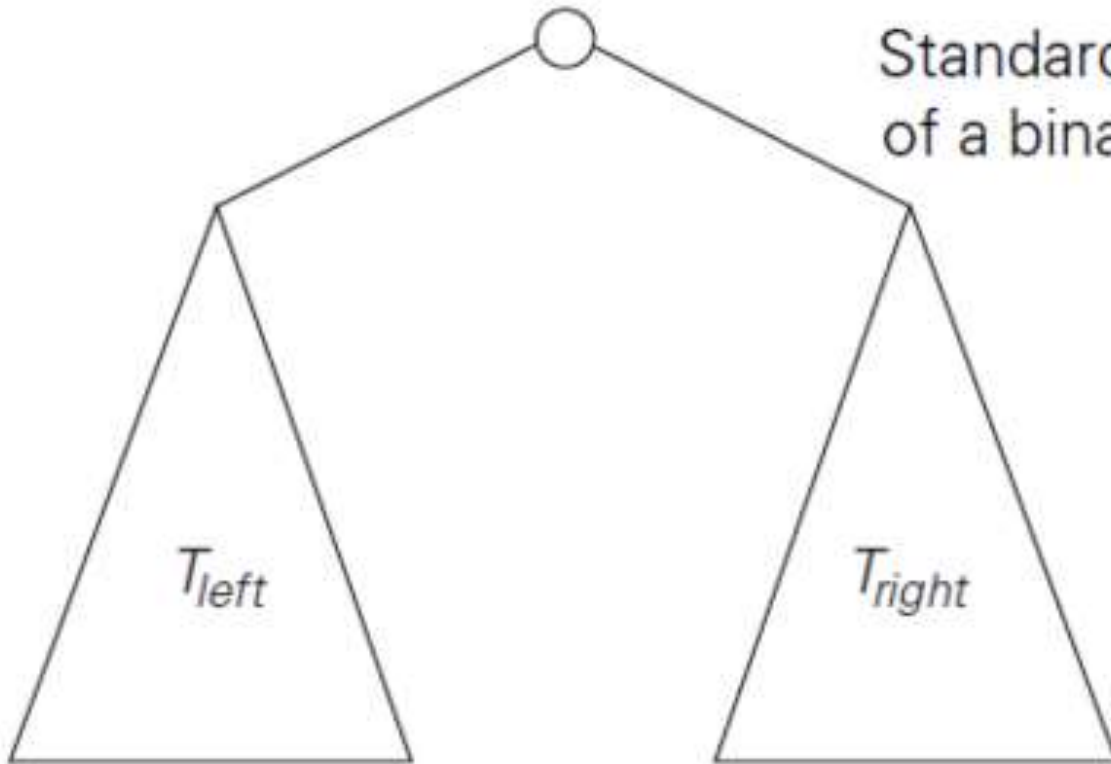


Q: Write an algorithm to find the height of a binary tree, where height of a binary tree is the length of the longest path from the root to a leaf.

E.g.: Height of a tree with only root node = 0

Height of a null tree = -1

Standard representation
of a binary tree.



ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if $T = \emptyset$ **return** -1

else return $\max\{Height(T_L), Height(T_R)\} + 1$

Input Size: $n(T)$, number of nodes in T

Basic Operation : **Addition**

$$\begin{aligned} C(n(T)) &= C(n(T_L)) + C(n(T_R)) + 1, C(0) = 0 \\ &= n(T) \in \Theta(n(T)) \end{aligned}$$

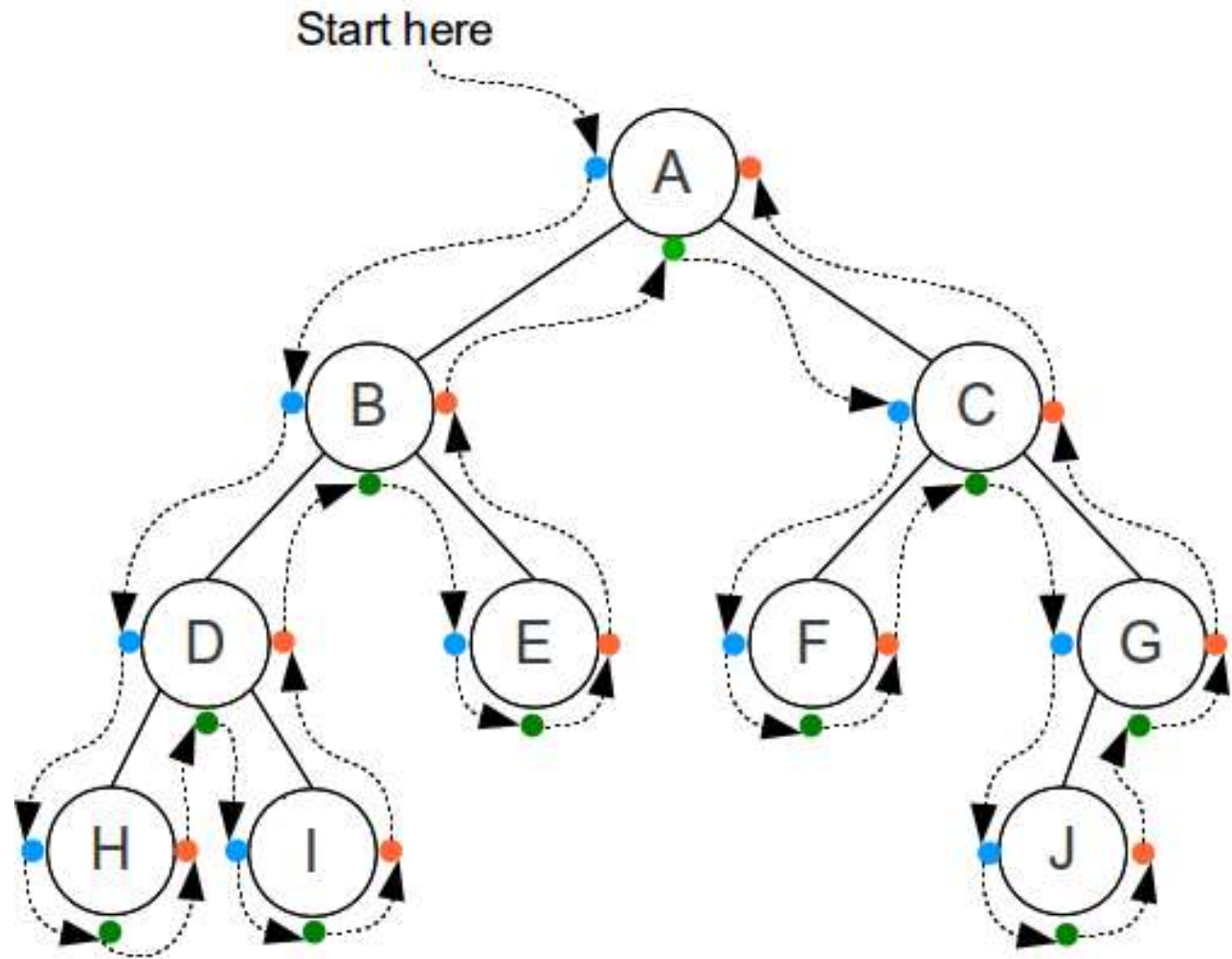
Basic Operation : **($T = \Phi$)**

$$\begin{aligned} C(n(T)) &= C(n(T_L)) + C(n(T_R)) + 1, C(0) = 1 \\ &= 2n(T) + 1 \in \Theta(n(T)) \end{aligned}$$

Binary Tree

Traversals:

- Pre-order
- In-order
- Post-order



Pre-Order

ABDHIECFGJ

In-Order

HDIBEAFCJG

Post-Order

HIDEBFJGCA

Q: Write an algorithm to count the number of nodes in a binary tree.

Algorithm CountNodes(T)

//Counts number of nodes in the binary tree

//Input: Binary tree T

//Output: Number of nodes in T

...

Q: Write an algorithm to count the number of leaf-nodes in a binary tree.

Algorithm CountLeafNodes(T)

//Counts number of leaf-nodes in the binary tree

//Input: Binary tree T

//Output: Number of leaf-nodes in T

...

</ End of Divide-and-Conquer >