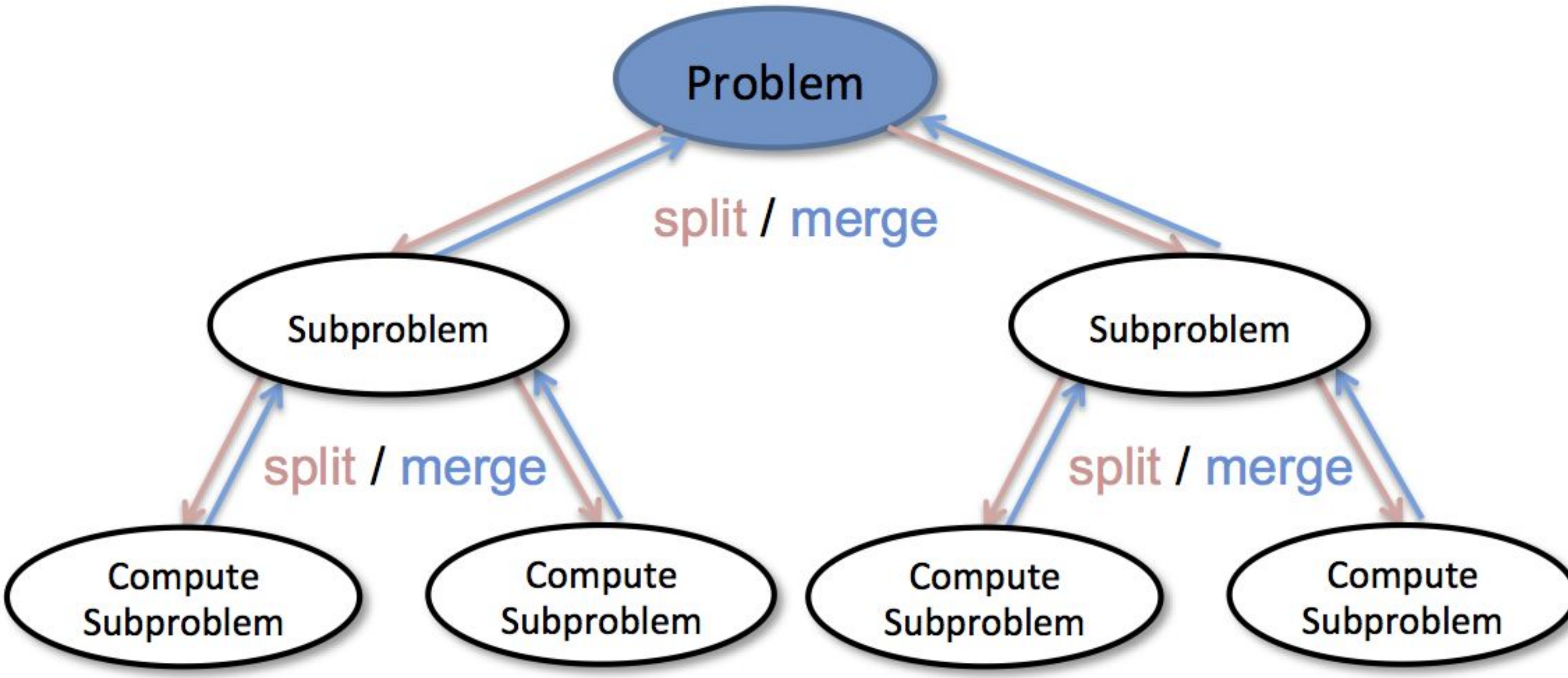
Design and Analysis of Algorithms (UE18CS251)

Unit III - Divide and Conquer

Mr. Channa Bankapur
channabankapur@pes.edu

Divide-and-Conquer!

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 ← 0
   for i ← 0 to n-1
      sum ← 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 **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
...
```

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]
```

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

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

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

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])
```

$$T(n) = 2T(n/2) + 1,\ T(1) = 1$$
$$= 2n - 1 \in \Theta(n)$$

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

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

$= 2\ [2T(n/4) + 1] + 1$

$= 2^2\ T(n/2^2)] + 2 + 1$

$= 2^i\ T(n/2^i)] + 2^{i-1} + 2^{i-2} + 2^1 + 2^0$

$= 2^i\ T(n/2^i)] + 2^i - 1$

$n/2^i = 1 \Rightarrow 2^i = n$

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

$= n + n - 1$

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

- **Brute Force:**
  - $Sum(A[0..n-1]) = A[0] + A[1] + \ldots + A[n-1]$
  - $T(n) \in \Theta(n)$
- **Decrease-and-Conquer:**
  - $Sum(A[0..n-1]) = Sum(A[0..n-2]) + A[n-1]$
  - $C(n) = C(n-1) + 1, C(1) = 1$

    $T(n) \in \Theta(n)$
- **Divide-and-Conquer:**
  - $Sum(A[0..n-1]) = Sum(A[0..n/2-1]) + Sum(A[n/2..n-1])$
  - $C(n) = 2C(n/2) + 1, C(1) = 1$

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

Finding $\mathbf{a^n}$ using **Brute Force** approach.

```
Algorithm Power(a, n)
//Input: a ∈ R and n ∈ I+
//Output: a^n
...
```

Finding $\mathbf{a}^{\mathbf{n}}$ using **Brute Force** approach.

```
Algorithm Power(a, n)
//Computes aⁿ = a*a*...a (n times)
//Input: a ∈ R and n ∈ I⁺
//Output: aⁿ
   p ← 1
   for i ← 1 to n
      p ← p * a
   return p
```

$C(n) = n$

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

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

```
Algorithm Power(a, n)
//Input: a ∈ R and n ∈ I+
//Output: a^n
…
```

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

```
Algorithm Power(a, n)
//Computes a^n = a^(n-1) * a
//Input: a ∈ R and n ∈ 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 $\mathbf{a^n}$ using

Decrease-by-a-constant-factor-and-Conquer approach.

```
Algorithm Power(a, n)
//Computes a^n = (a^⌊n/2⌋)^2 * a^(n mod 2)
//Input: a ∈ R and n ∈ I+
//Output: a^n
...
```

Finding $\mathbf{a^n}$ using

Decrease-by-a-constant-factor-and-Conquer approach.

```
Algorithm Power(a, n)
//Computes aⁿ = (a⌊n/2⌋)² * aⁿ ᵐᵒᵈ ²
//Input: a ∈ R and n ∈ I+
//Output: aⁿ
   if (n = 0) return 1
   p ← Power(a, ⌊n/2⌋)
   p ← p * p
   if (n is odd) p ← p * a
   return p
```

What is its time complexity?

C(n) = ...

T(n) ∈ **Θ(...)**

Finding $\mathbf{a^n}$ using

Decrease-by-a-constant-factor-and-Conquer approach.

```
Algorithm Power(a, n)
//Computes aⁿ = (a⌊n/2⌋)² * aⁿ ᵐᵒᵈ ²
//Input: a ∈ R and n ∈ I+
//Output: aⁿ
   if (n = 0) return 1
   p ← Power(a, ⌊n/2⌋)
   p ← p * p
   if (n is odd) p ← p * a
   return p
```

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

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

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

```
Algorithm Power(a, n)
//Input: a ∈ R and n ∈ I+
//Output: a^n
...
```

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

```
Algorithm Power(a, n)
//Computes a^n = a^⌊n/2⌋ * a^⌈n/2⌉
//Input: a ∈ R and n ∈ I+
//Output: a^n
   if (n = 0) return 1
   if (n = 1) return a
   return Power(a, ⌊n/2⌋) * Power(a, ⌈n/2⌉)
```

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

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

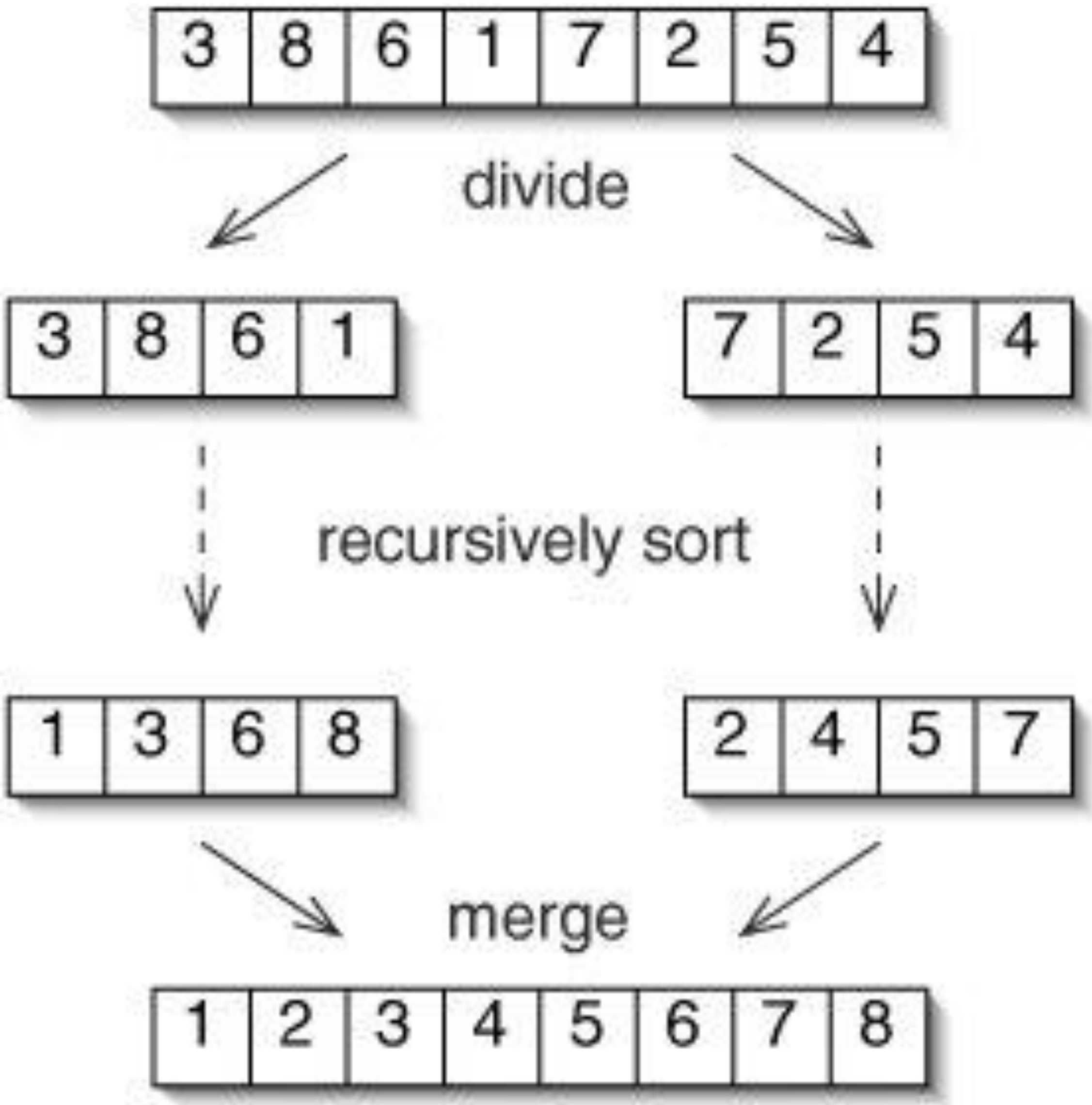
Finding $\mathbf{a^n}$ using different approaches.

- Brute-Force approach in **Θ(n)**
  - $a^n = a * a * \ldots a$ (n times), $a^0=1$
- Divide-and-Conquer approach in **Θ(n)**
  - $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$, $a^0=1$, $a^1=a$
- Decrease-by-a-constant-and-Conquer in **Θ(n)**
  - $a^n = a^{n-1} * a$, $a^0=1$
- Decrease-by-a-constant-factor-and-Conquer in **Θ(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
    $a^0 = 1$

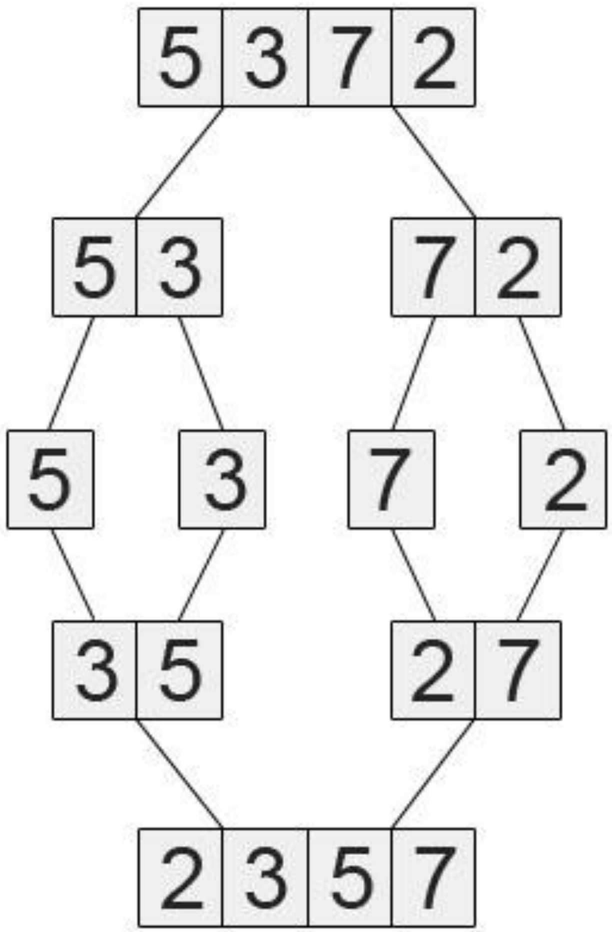
Divide-and-Conquer Examples:

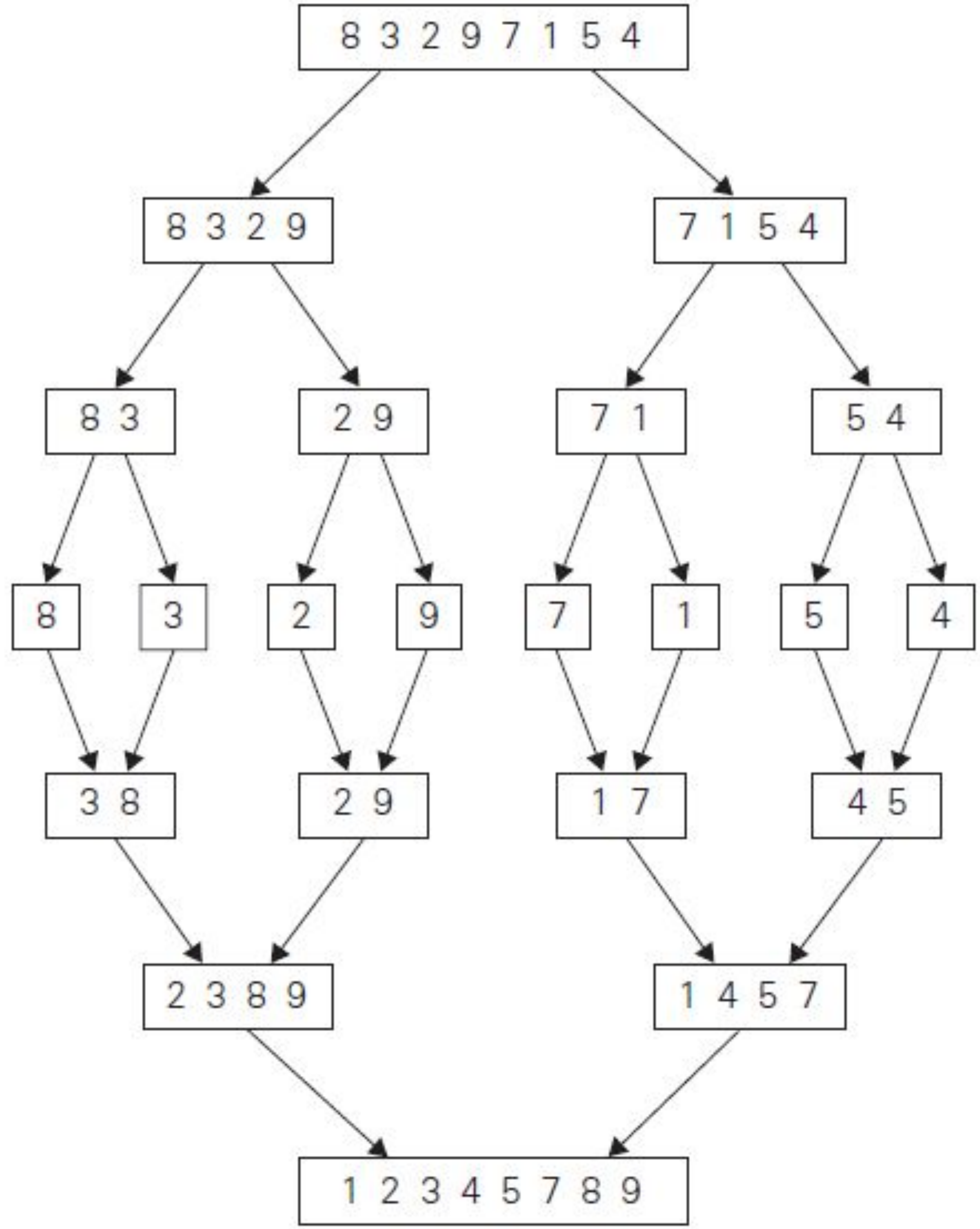
- Sorting: Mergesort and Quicksort
- Search: Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Binary tree traversals

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 ≤ 1)return
  m = ⌊n/2⌋
  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

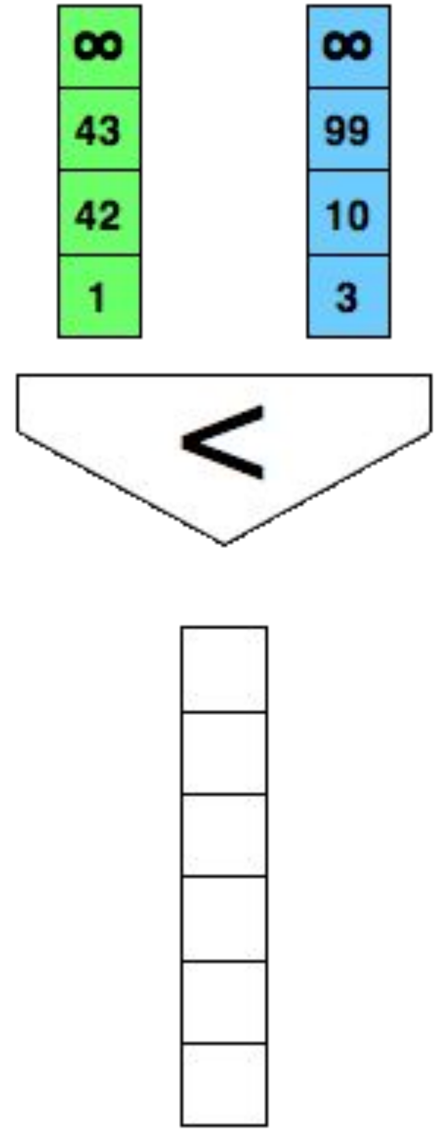
Two sorted arrays concatenated:

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 ← 0, j ← m, k ← 0
   while(i < m and j < n) do
      if(A[i] ≤ A[j]) B[k] ← A[i]; i ← i+1
      else B[k] ← A[j]; j ← j+1
      k ← 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 ← 0, j ← m, k ← 0
   while(i < m and j < n) do
      if(A[i] ≤ A[j]) B[k] ← A[i]; i ← i+1
      else B[k] ← A[j]; j ← j+1
      k ← 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←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 ≤ 1)return
    m = ⌊n/2⌋
    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 $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

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

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

Input Size: **`n`**

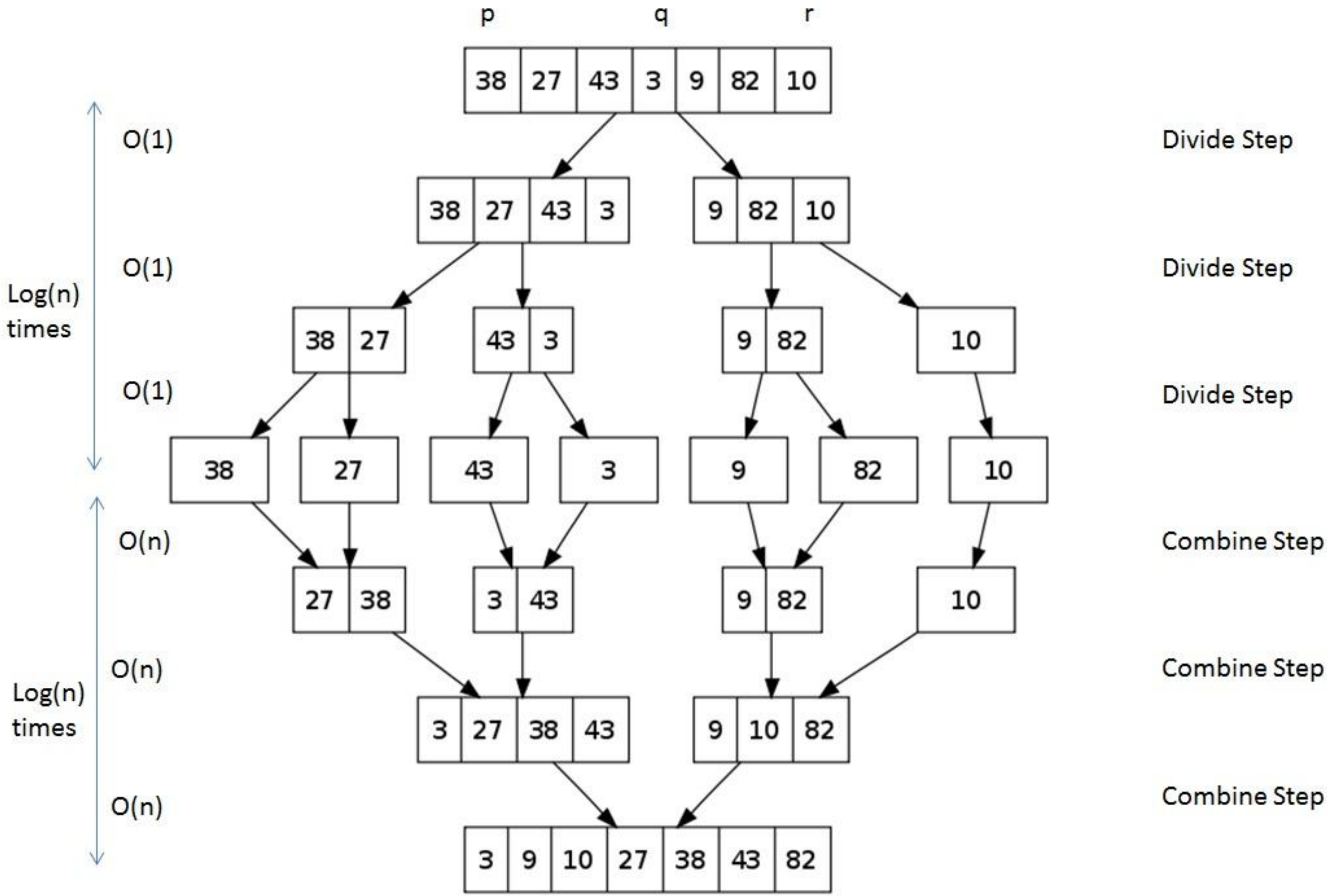
Basic Operation: **...**

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

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

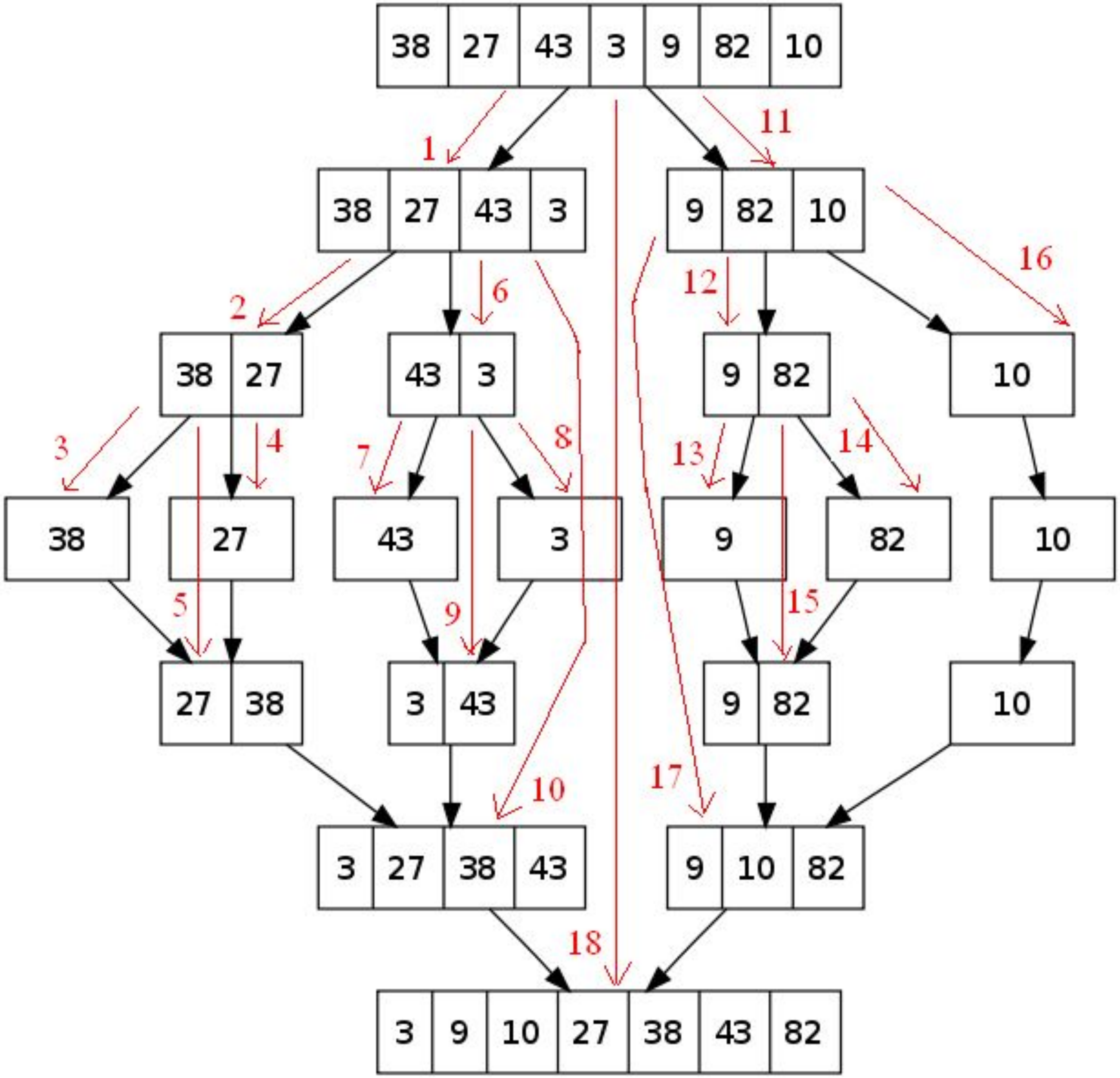
$C(n/2^i)$ is $C(1)$ when $n/2^i \mathbf{=} 1 \Rightarrow n \mathbf{=} 2^i \Rightarrow i \mathbf{=} \log_2 n$

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

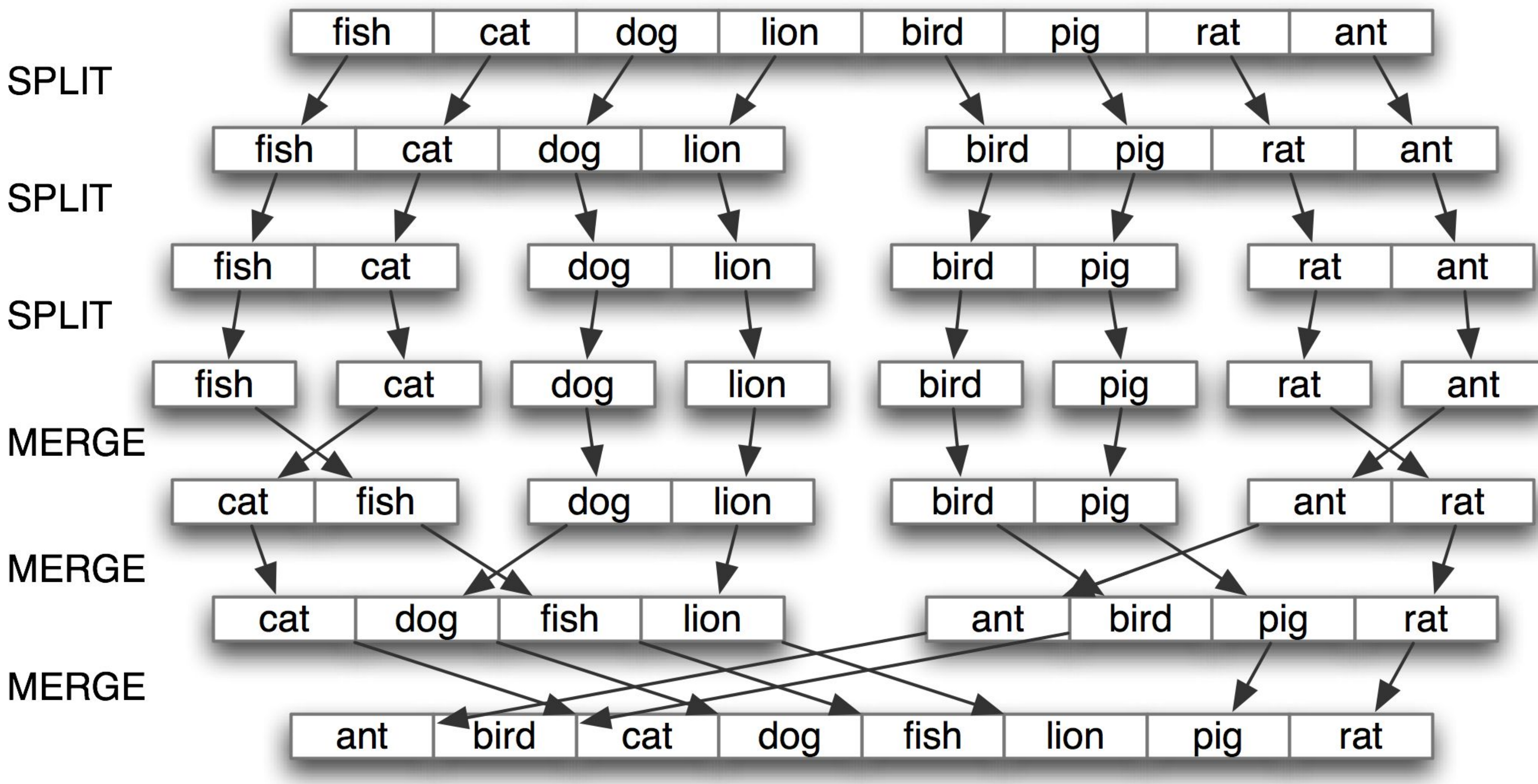


Total Runtime = Total time required in Divide + Total time required in Combine

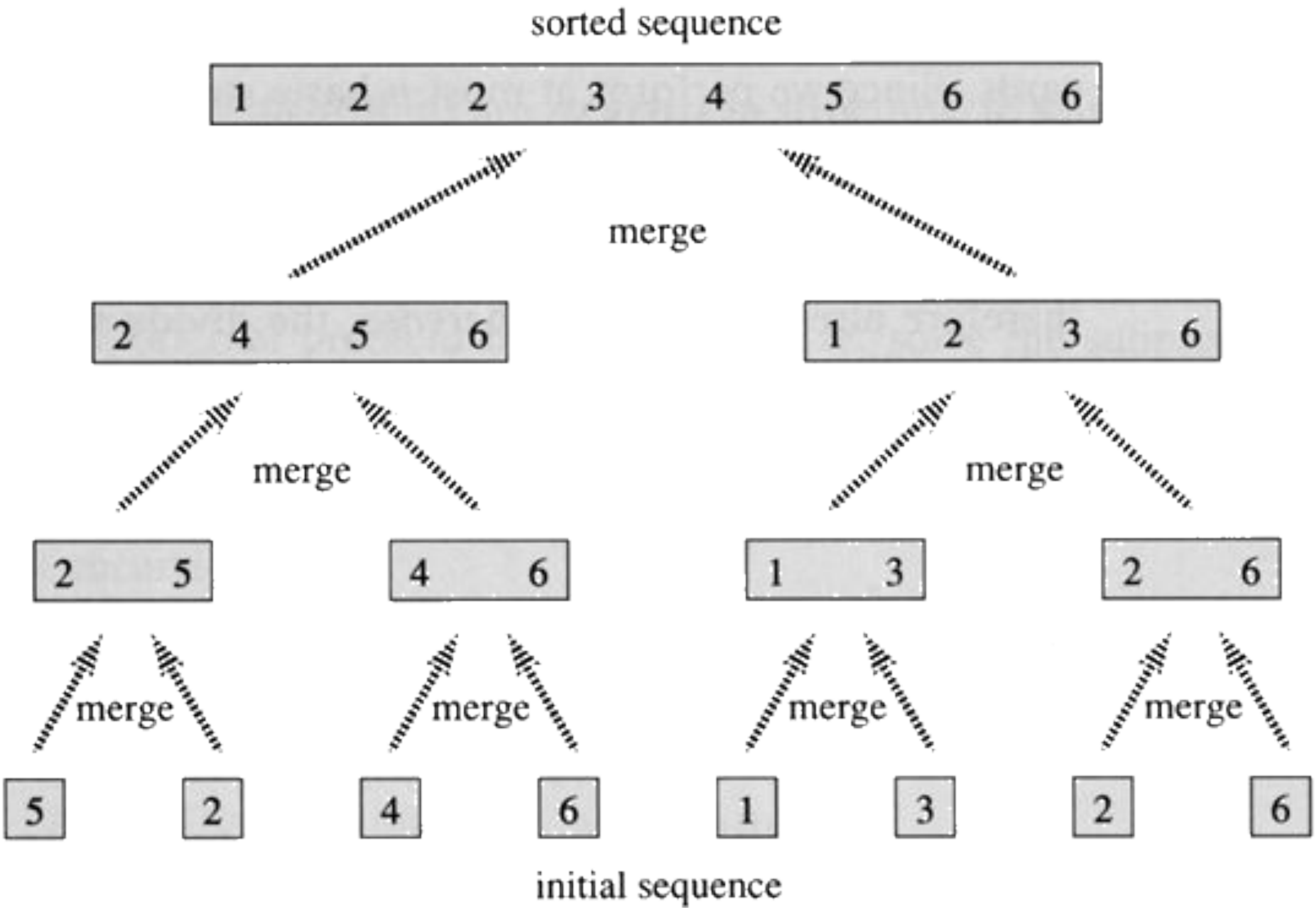
= 1 * Log(n) + n * Log(n) = n Log(n).



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18


fish cat dog lion bird pig rat ant
SPLIT
fish cat dog lion
bird pig rat ant
SPLIT
fish cat
dog lion
bird pig
rat ant
SPLIT
fish
cat
dog
lion
bird
pig
rat
ant
MERGE
cat fish
dog lion
bird pig
ant rat
MERGE
cat dog fish lion
ant bird pig rat
MERGE
ant bird cat dog fish lion pig rat

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 ≤ pivot element and
the right part has the elements ≥ pivot element.

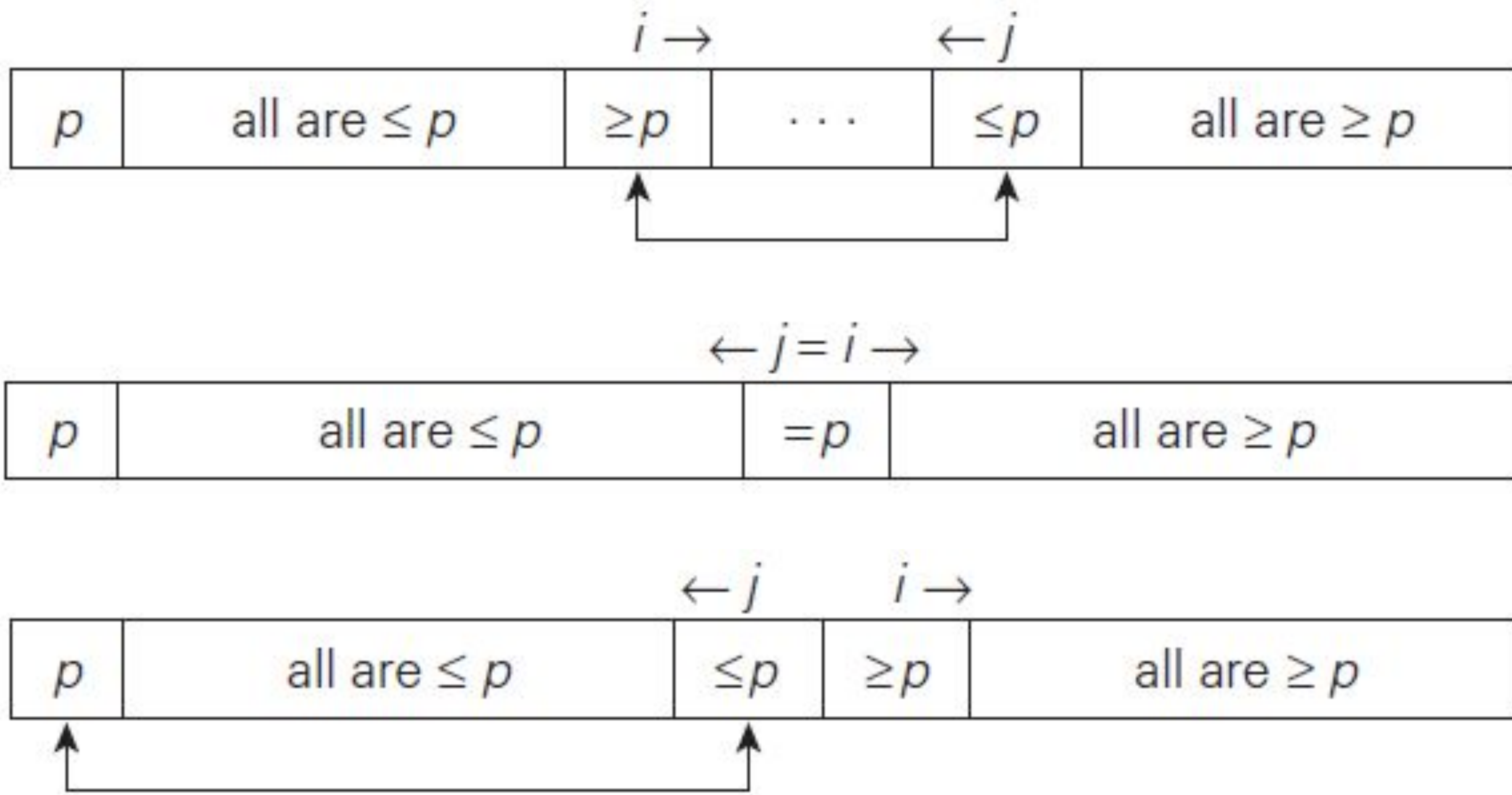
Eg: 35 33 42 10 14 19 27 44 26 **31**
Let key = 31.

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 ≤ key and the right part has the elements ≥ key.

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

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



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

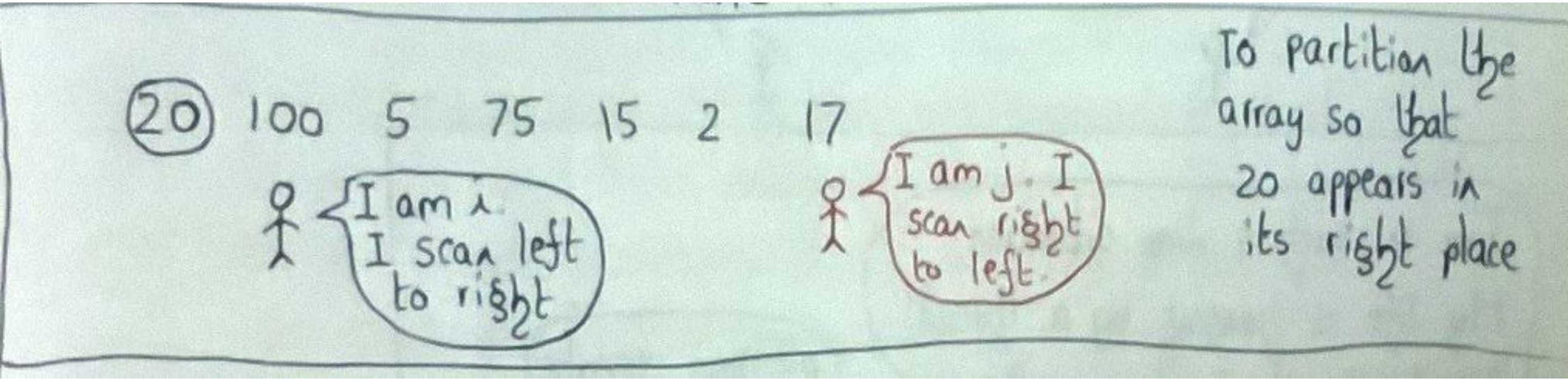
//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value

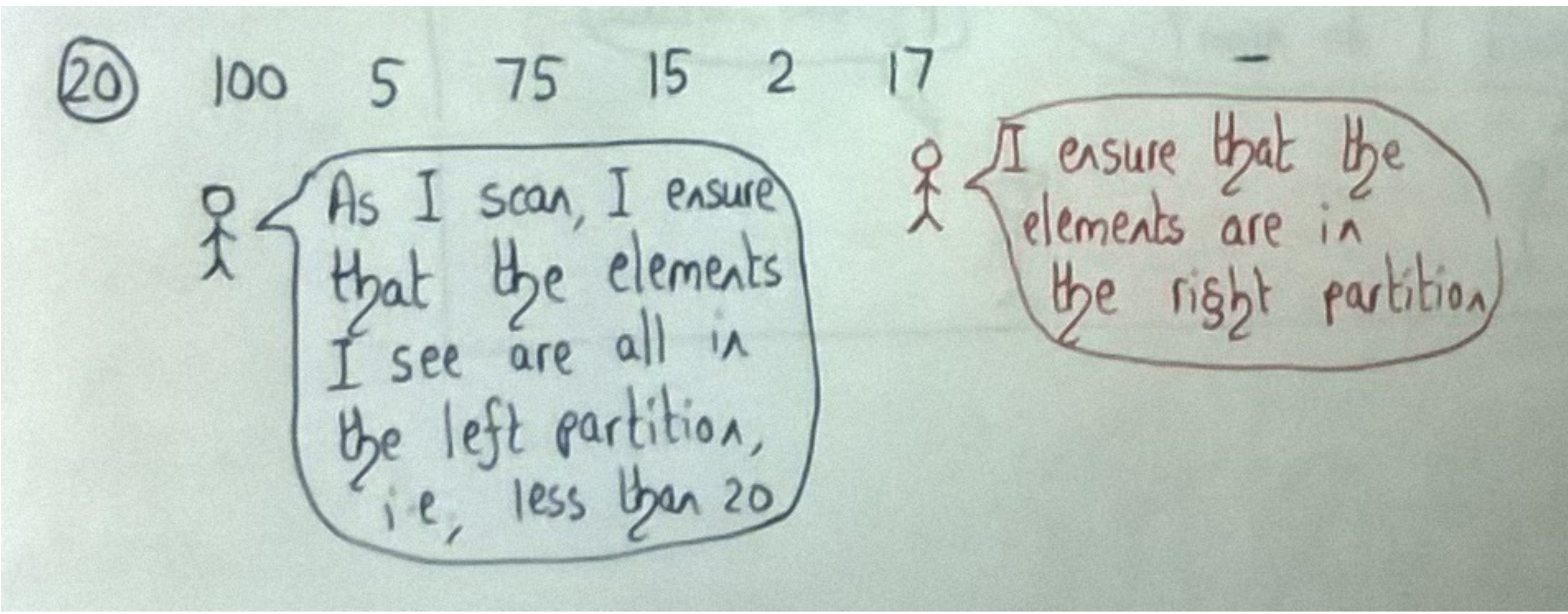
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

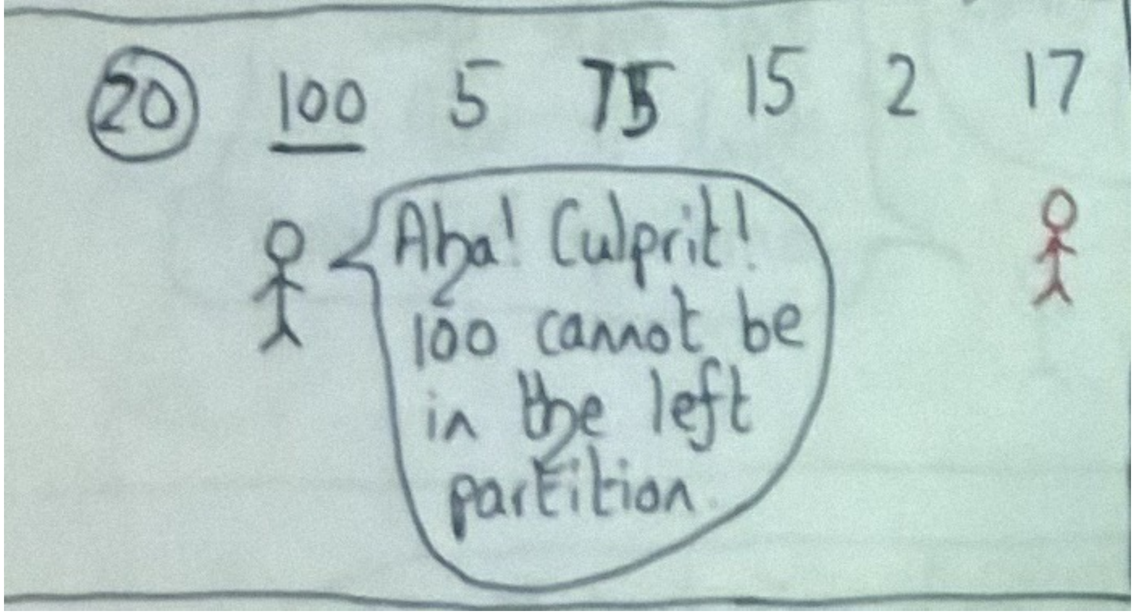
Algorithm Partition(A[0..n-1])

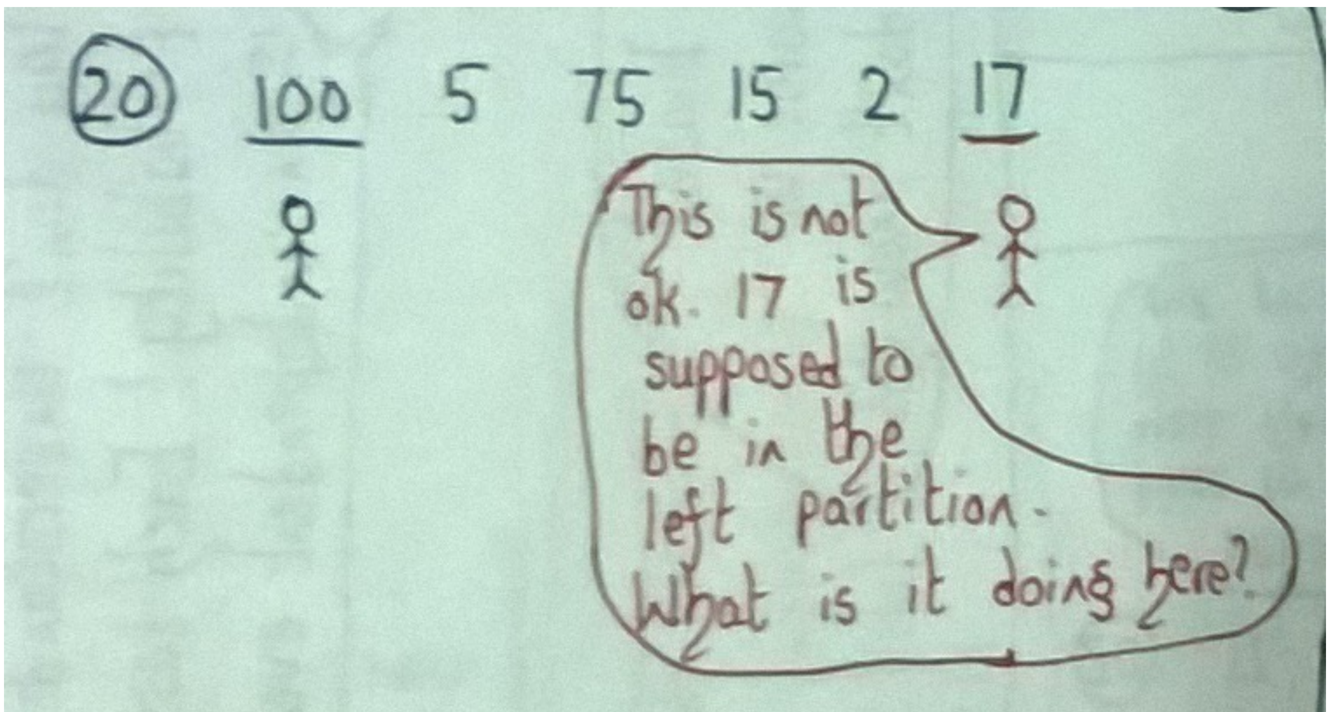
```
p ← A[0]
i ← 1, j ← n-1
while(i ≤ j)
   while(i ≤ j and A[i] < p) i ← i + 1
   while(i ≤ j and A[j] > p) j ← j - 1
   if(i < j)
      swap A[i], A[j]
      i ← i + 1
      j ← j - 1
swap A[j], A[0]
return j
```

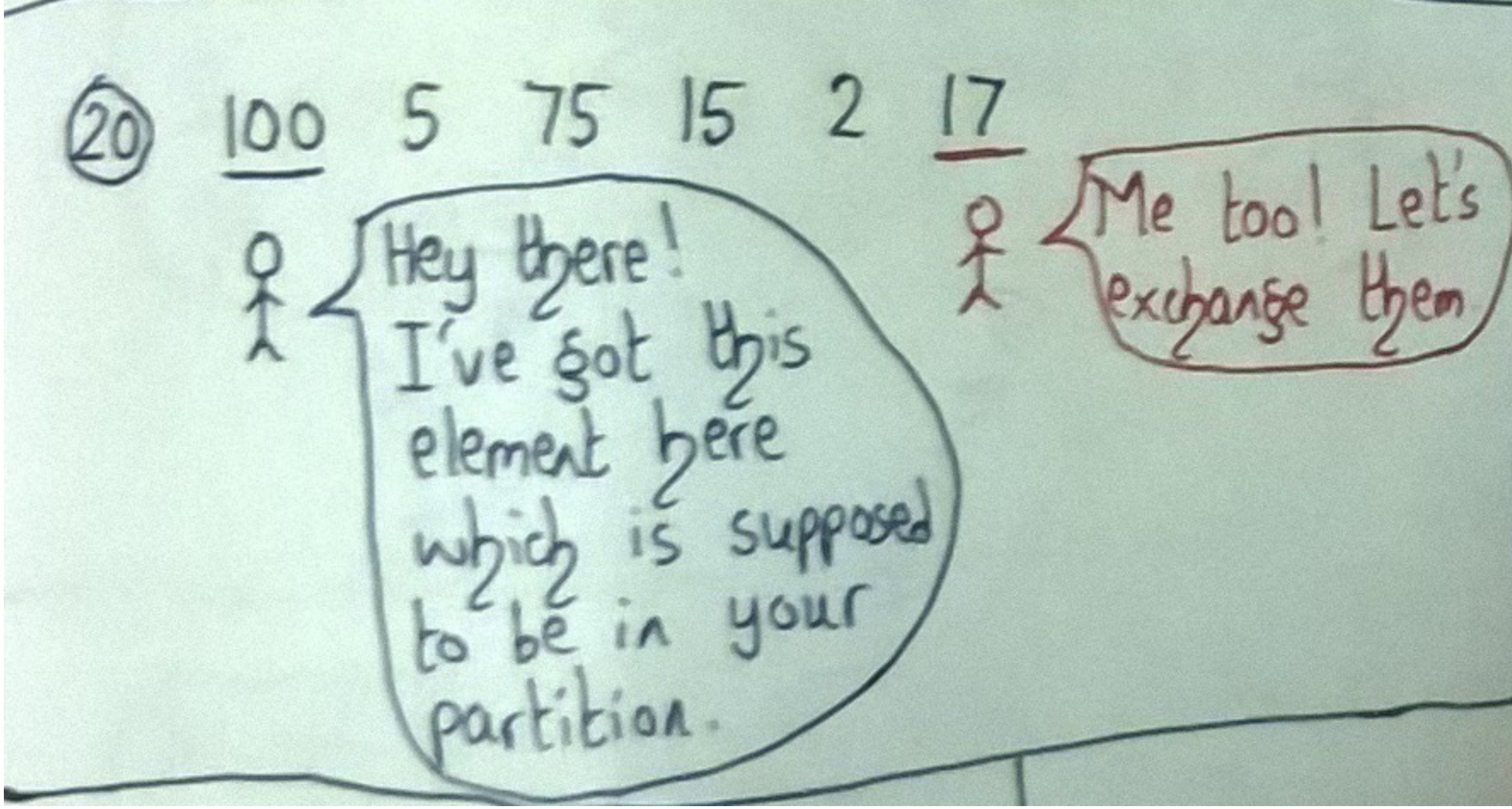
Partition in the eyes of Ullas Aparanji :)

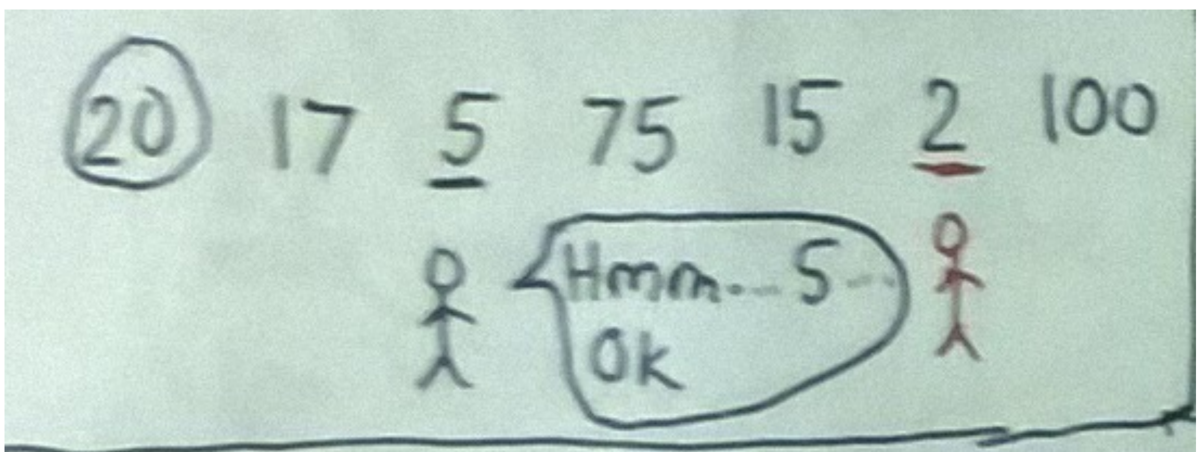


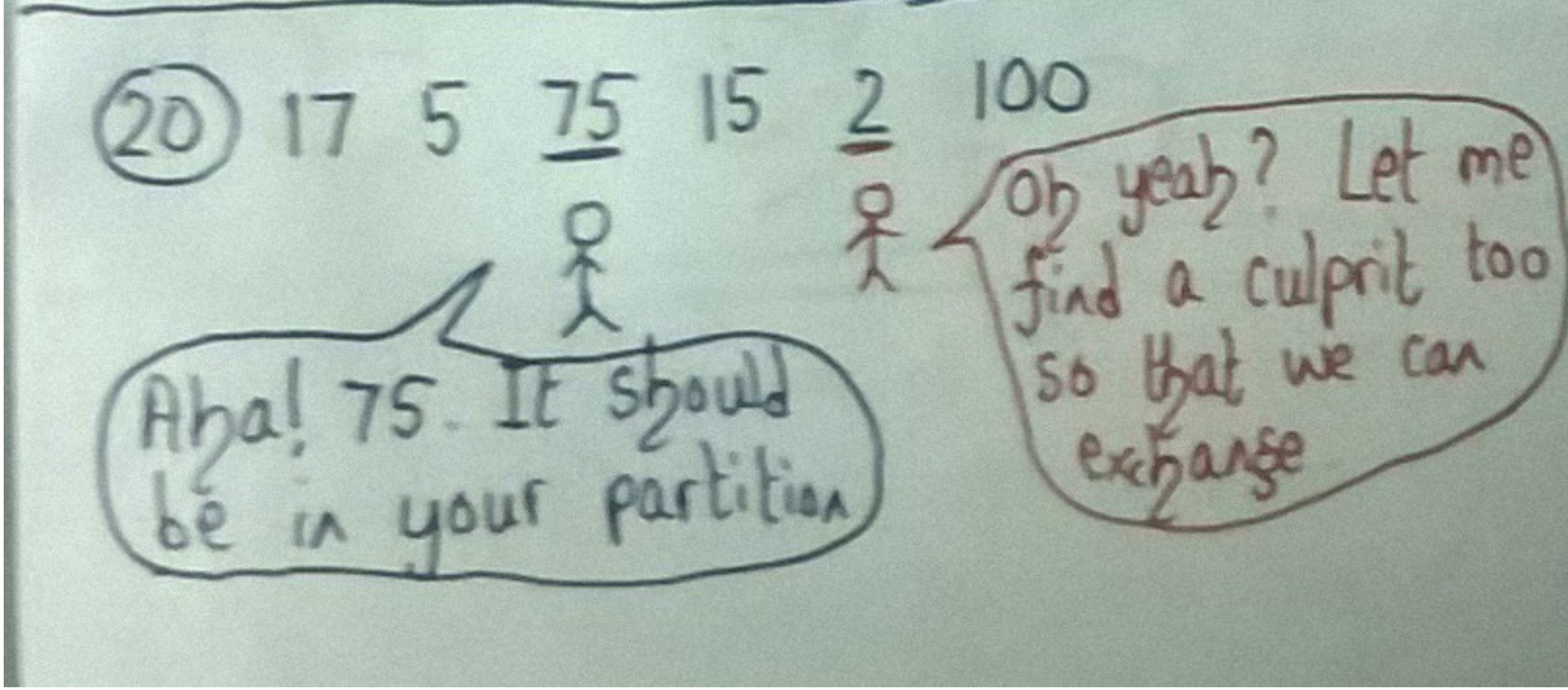


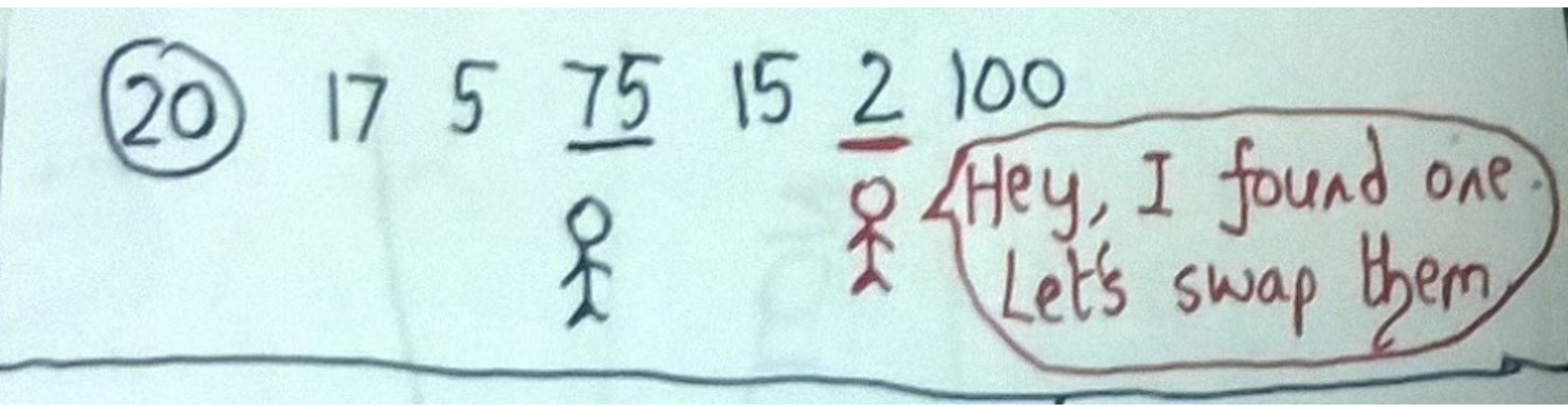

20 100 5 75 15 2 17
Aha! Culprit! 100 cannot be in the left partition.

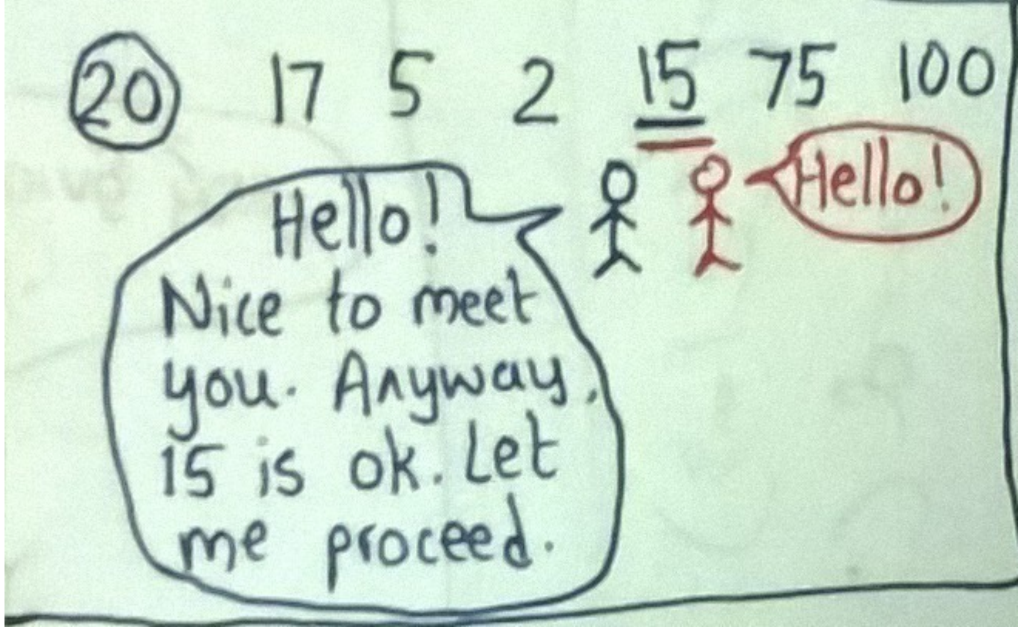

20 100 5 75 15 2 17
This is not ok. 17 is supposed to be in the left partition. What is it doing here?

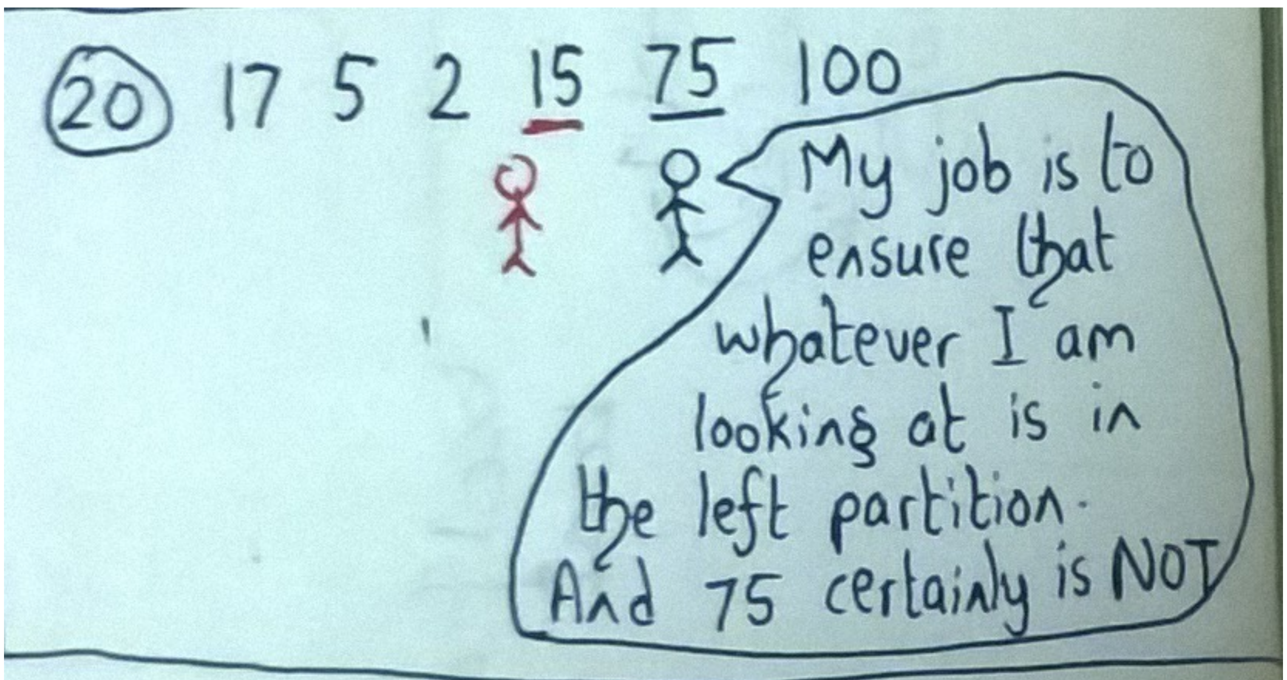

20 100 5 75 15 2 17
Hey there! I've got this element here which is supposed to be in your partition.
Me too! Let's exchange them

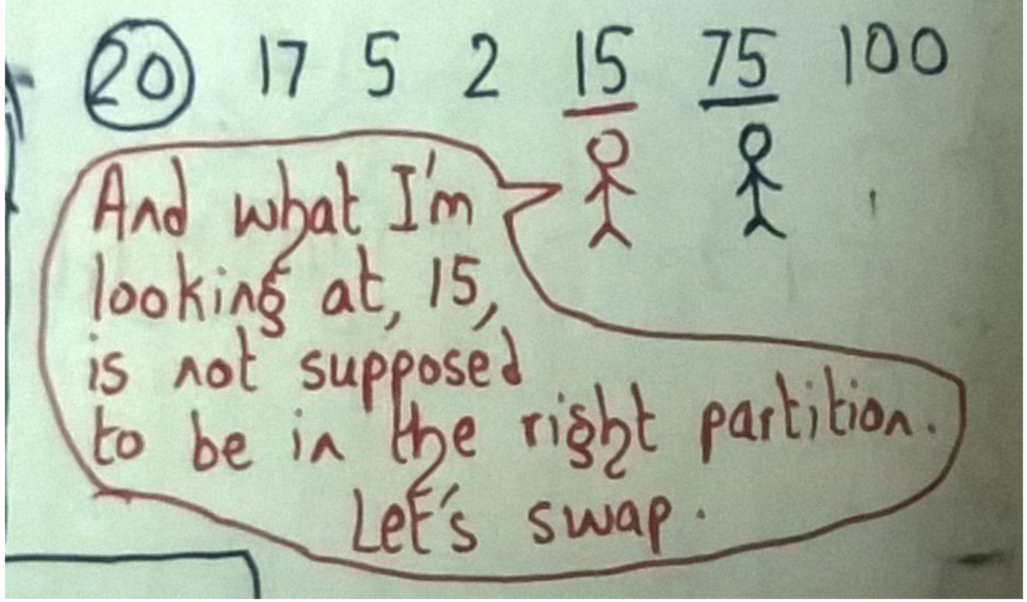

20 17 5 75 15 2 100
Hmm... 5... Ok

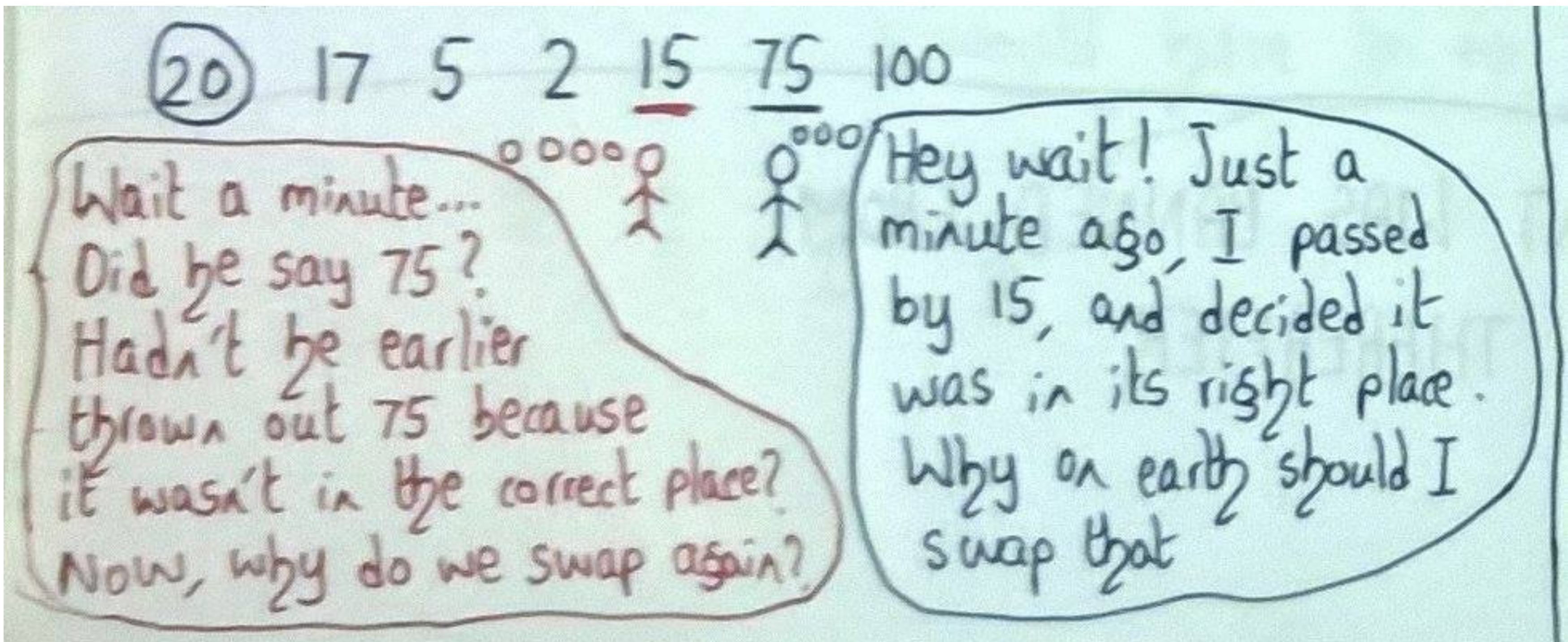

20 17 5 75 15 2 100
Aha! 75. It should be in your partition
Oh yeah? Let me find a culprit too so that we can exchange

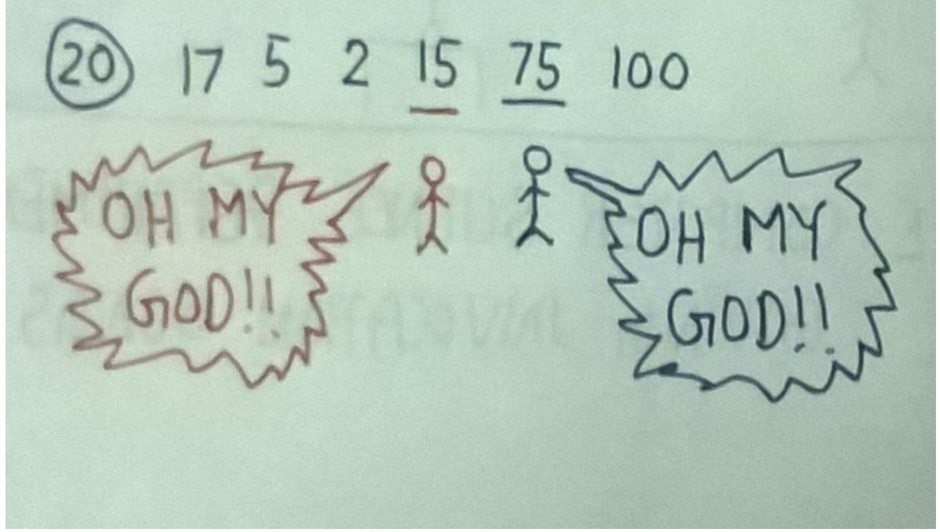

20 17 5 75 15 2 100
Hey, I found one. Let's swap them.

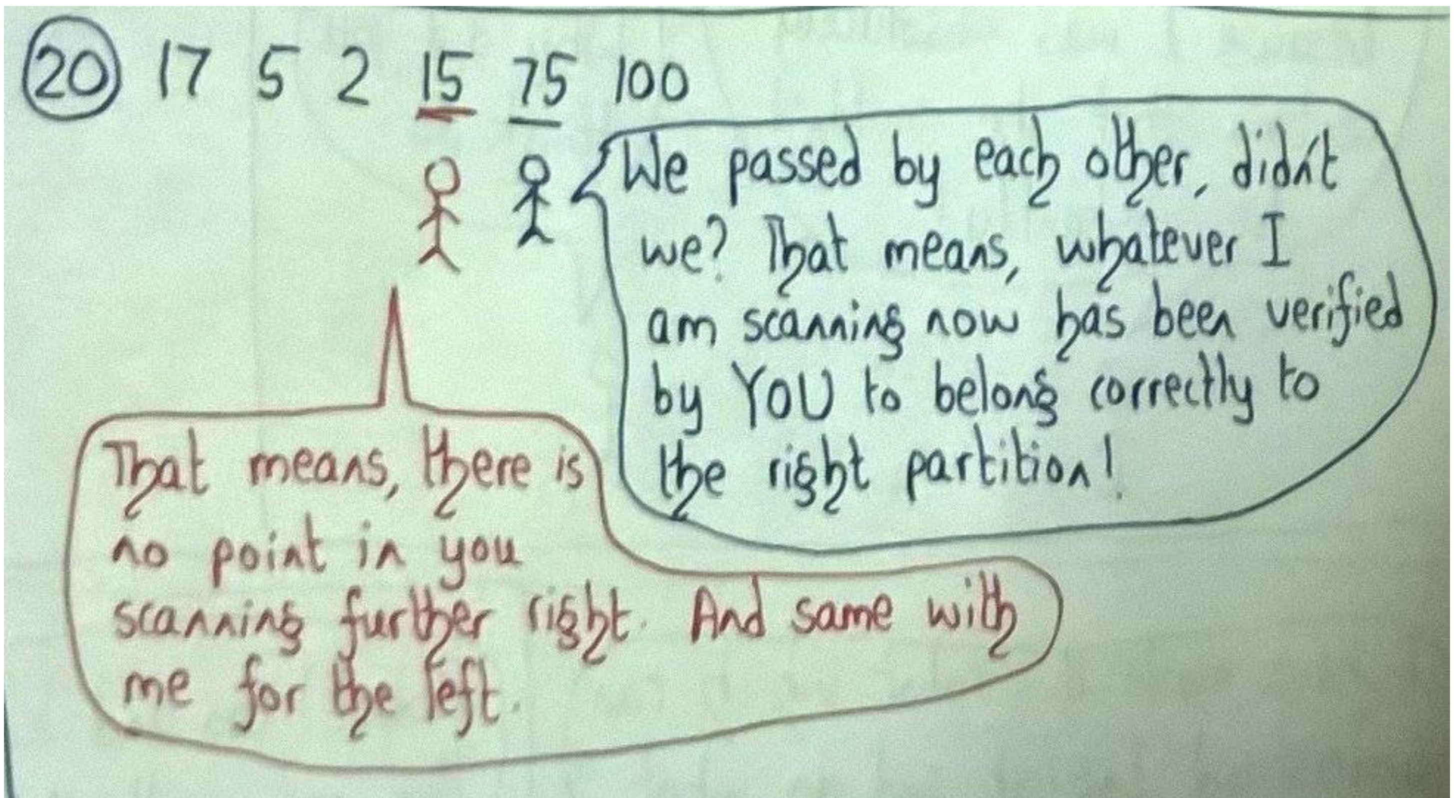

20 17 5 2 15 75 100
Hello! Nice to meet you. Anyway, 15 is ok. Let me proceed.
Hello!

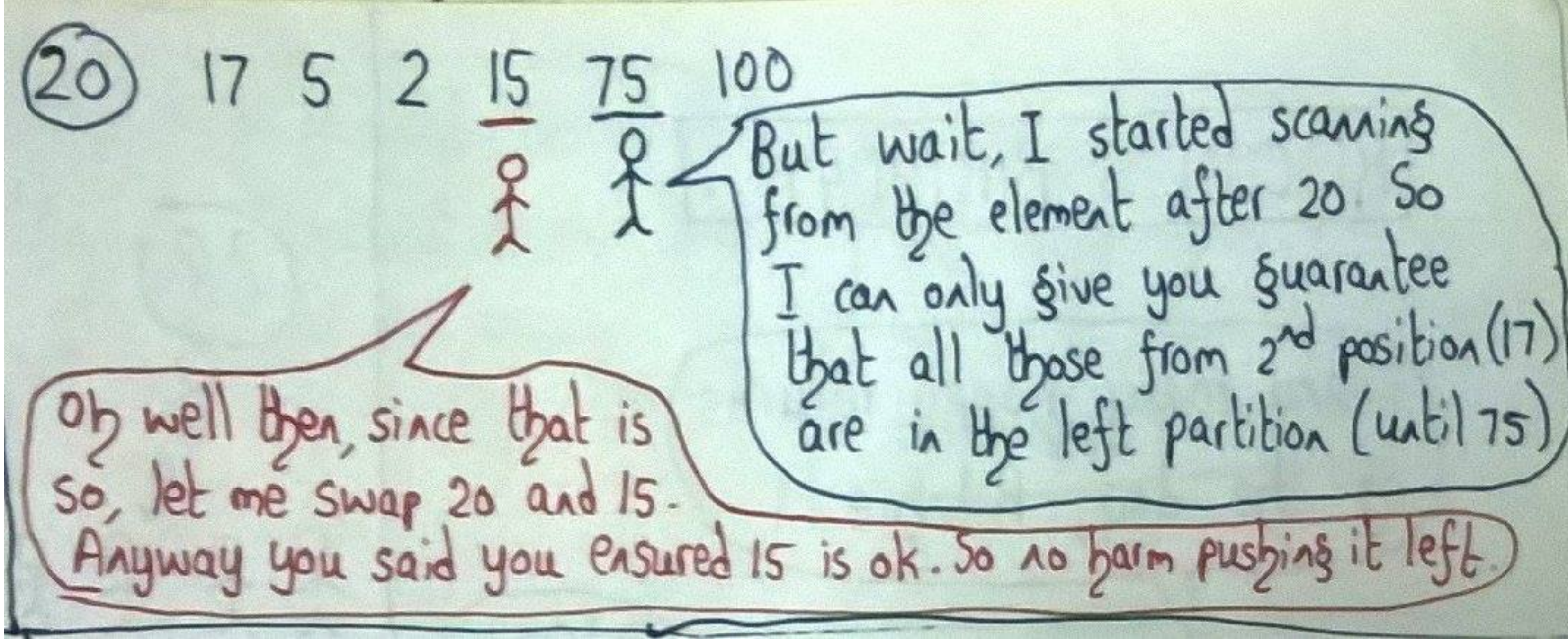

20 17 5 2 15 75 100
My job is to ensure that whatever I am looking at is in the left partition. And 75 certainly is NOT

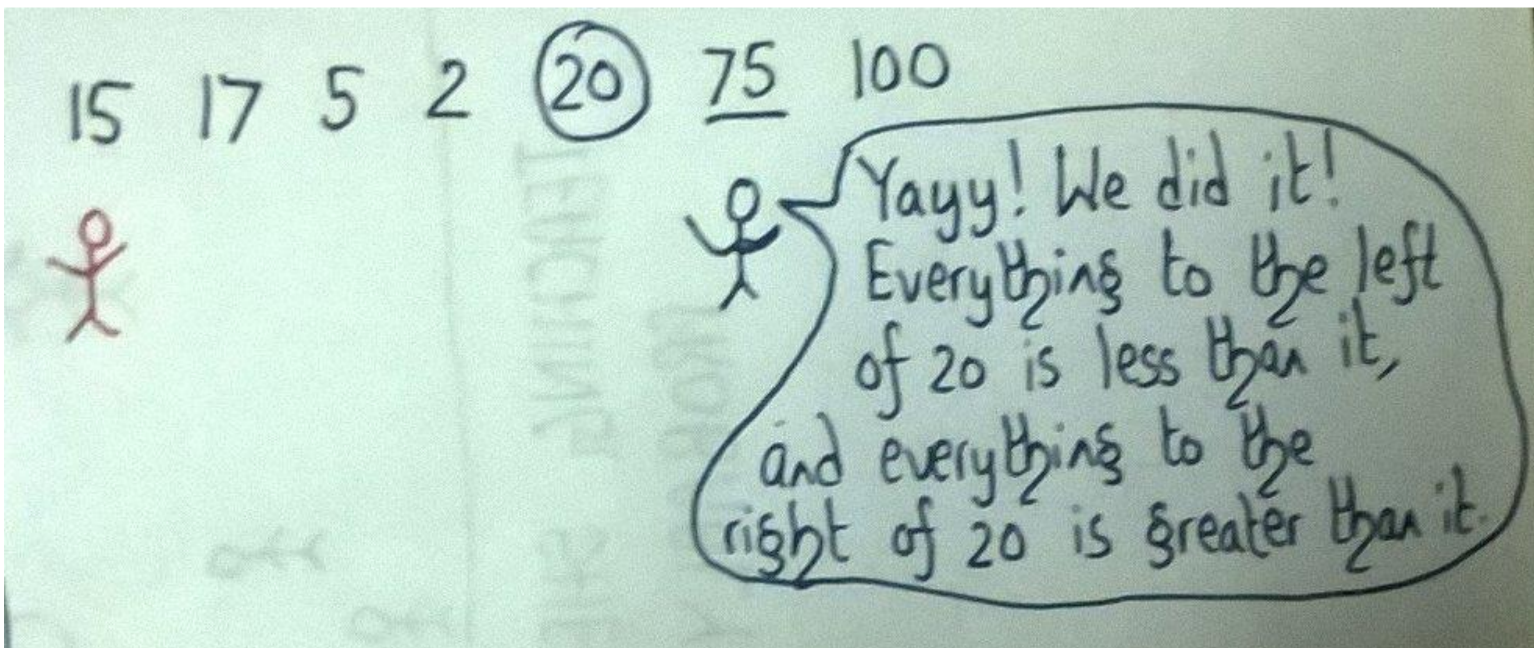

20 17 5 2 15 75 100
And what I'm looking at, 15, is not supposed to be in the right partition. Let's swap.


20 17 5 2 15 75 100
Wait a minute... Did he say 75? Hadn't he earlier thrown out 75 because it wasn't in the correct place? Now, why do we swap again?
Hey wait! Just a minute ago, I passed by 15, and decided it was in its right place. Why on earth should I swap that

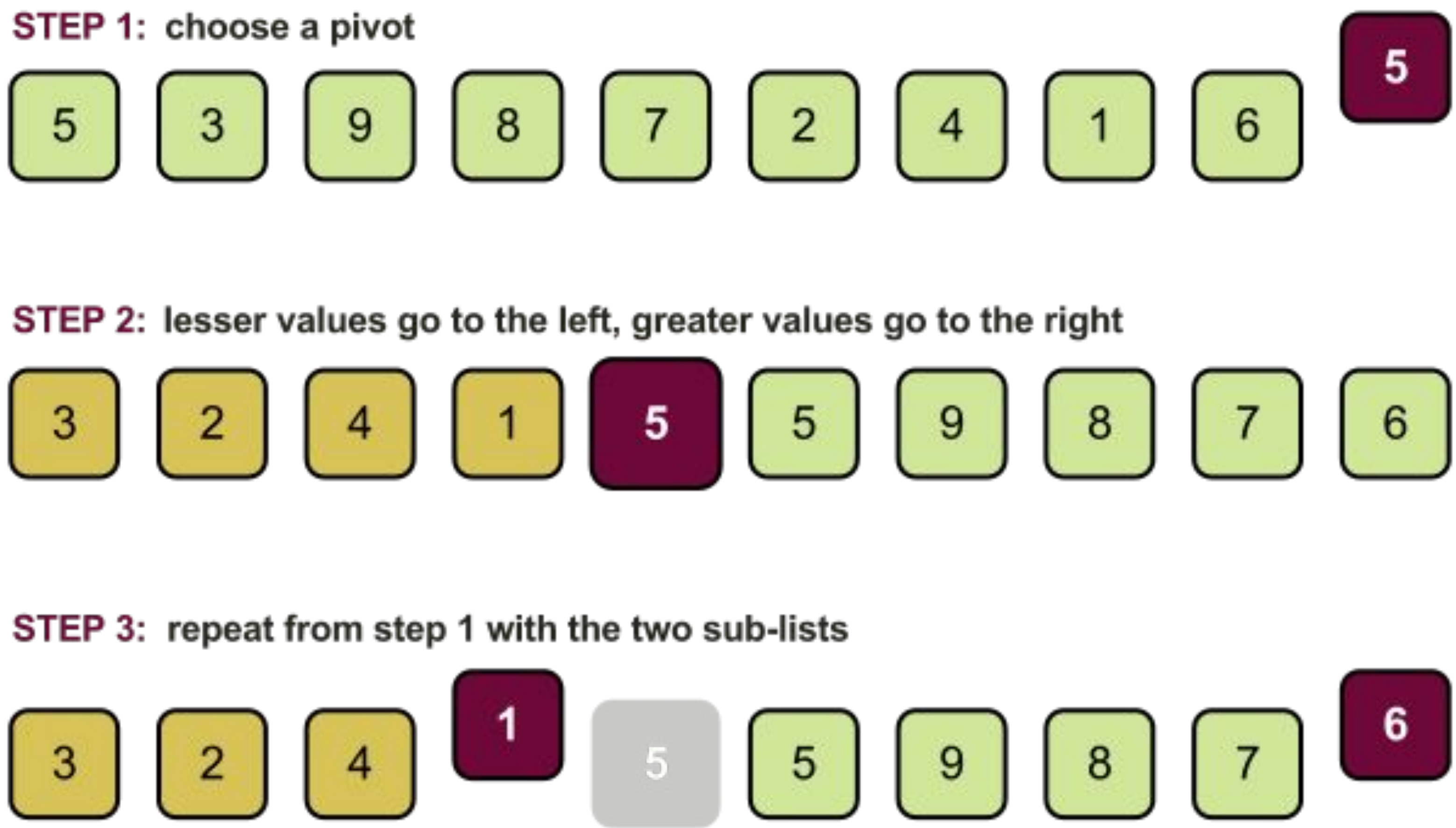

20 17 5 2 15 75 100
OH MY GOD!!
OH MY GOD!!


20 17 5 2 15 75 100
We passed by each other, didn't we? That means, whatever I am scanning now has been verified by YOU to belong correctly to the right partition!
That means, there is no point in you scanning further right. And same with me for the left.

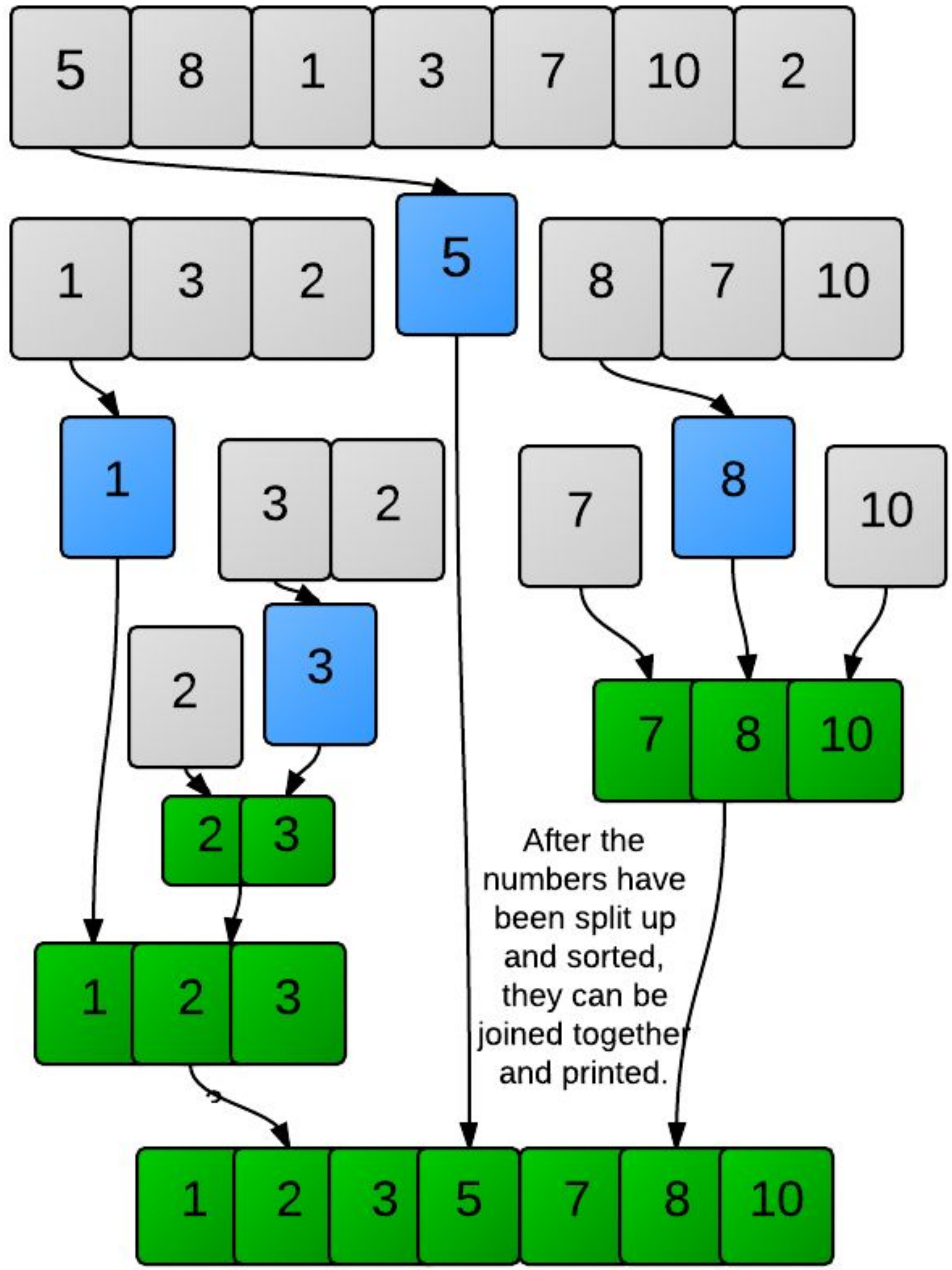

20 17 5 2 15 75 100
But wait, I started scanning from the element after 20. So I can only give you guarantee that all those from 2nd position (17) are in the left partition (until 75)
Oh well then, since that is so, let me swap 20 and 15. Anyway you said you ensured 15 is ok. So no harm pushing it left.

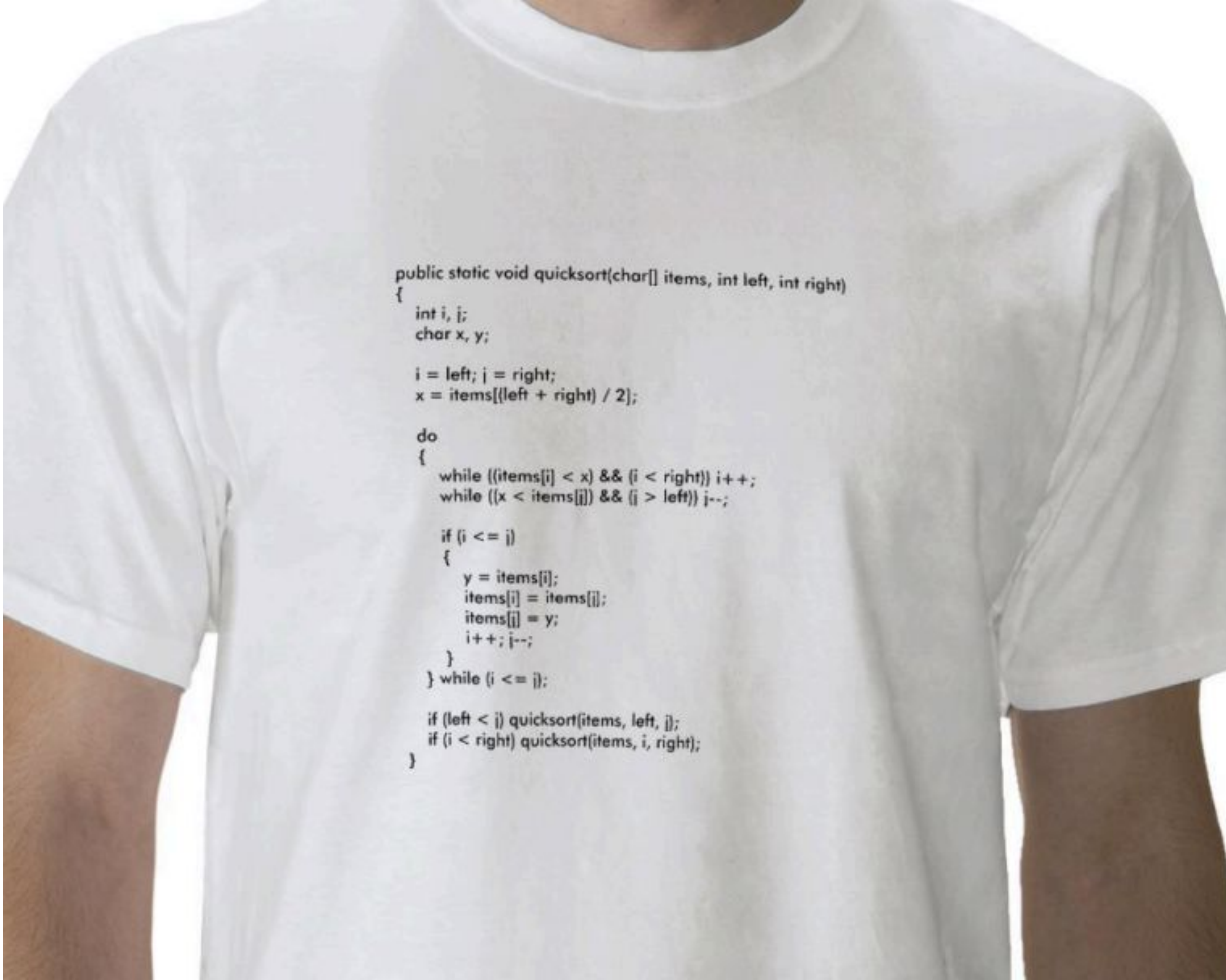

15 17 5 2 20 75 100
Yayy! We did it! Everything to the left of 20 is less than it, and everything to the right of 20 is greater than it.

Quick Sort - Idea



Quick Sort


5 8 1 3 7 10 2
5
1 3 2
8 7 10
1
3 2
7
8
10
2
3
7 8 10
2 3
After the numbers have been split up and sorted, they can be joined together and printed.
1 2 3
1 2 3 5 7 8 10


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

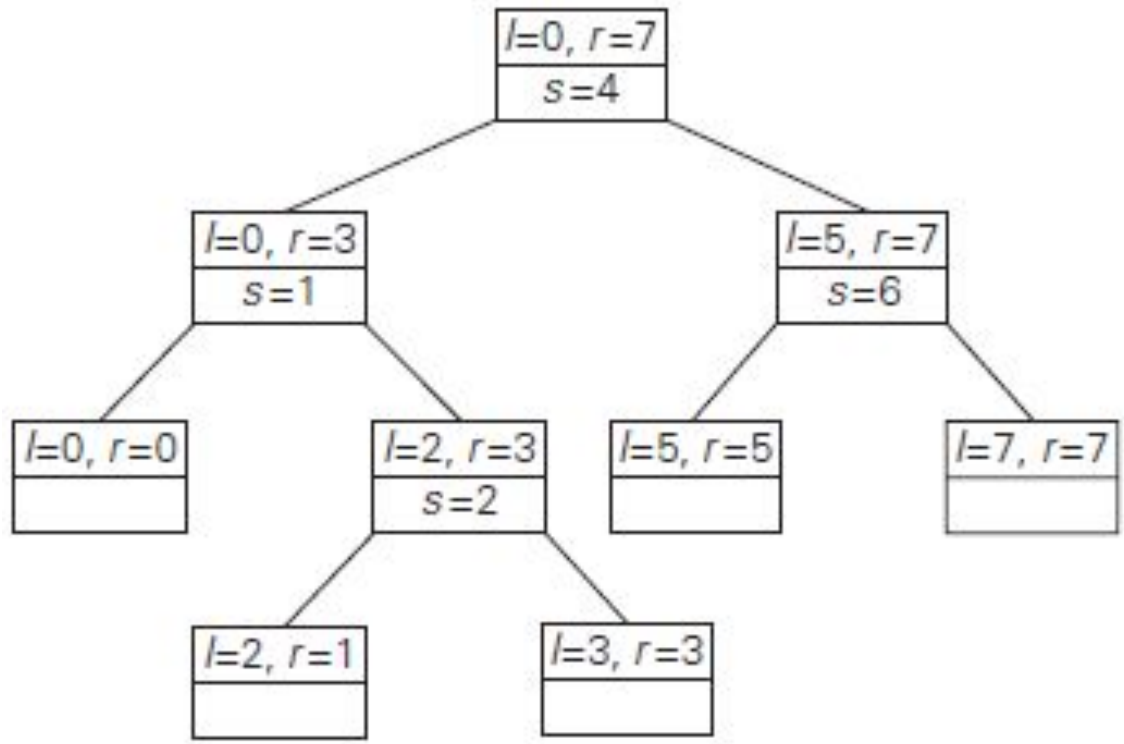
Quick Sort

```
Algorithm QuickSort(A[0..n-1])
   if(n ≤ 1) return
   s ← Partition(A[0..n-1])
   QuickSort(A[0..s-1])
   QuickSort(A[s+1..n-1])
   return
```

Quick Sort

Example

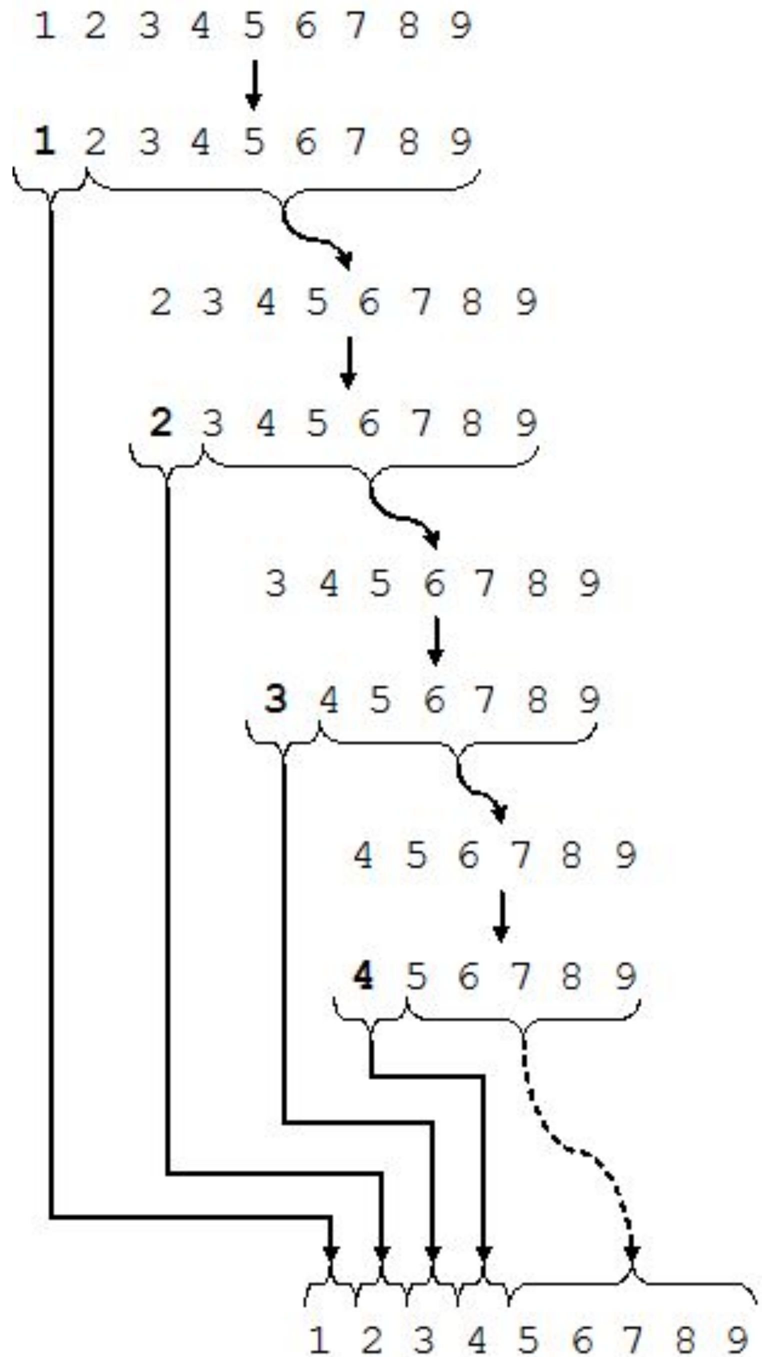
0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{i\,j}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9



Quick Sort
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:

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

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

$\quad\quad = 2^i\,C(n/2^i) + i{*}cn + (2^i - 1)$

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

Worst case:

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

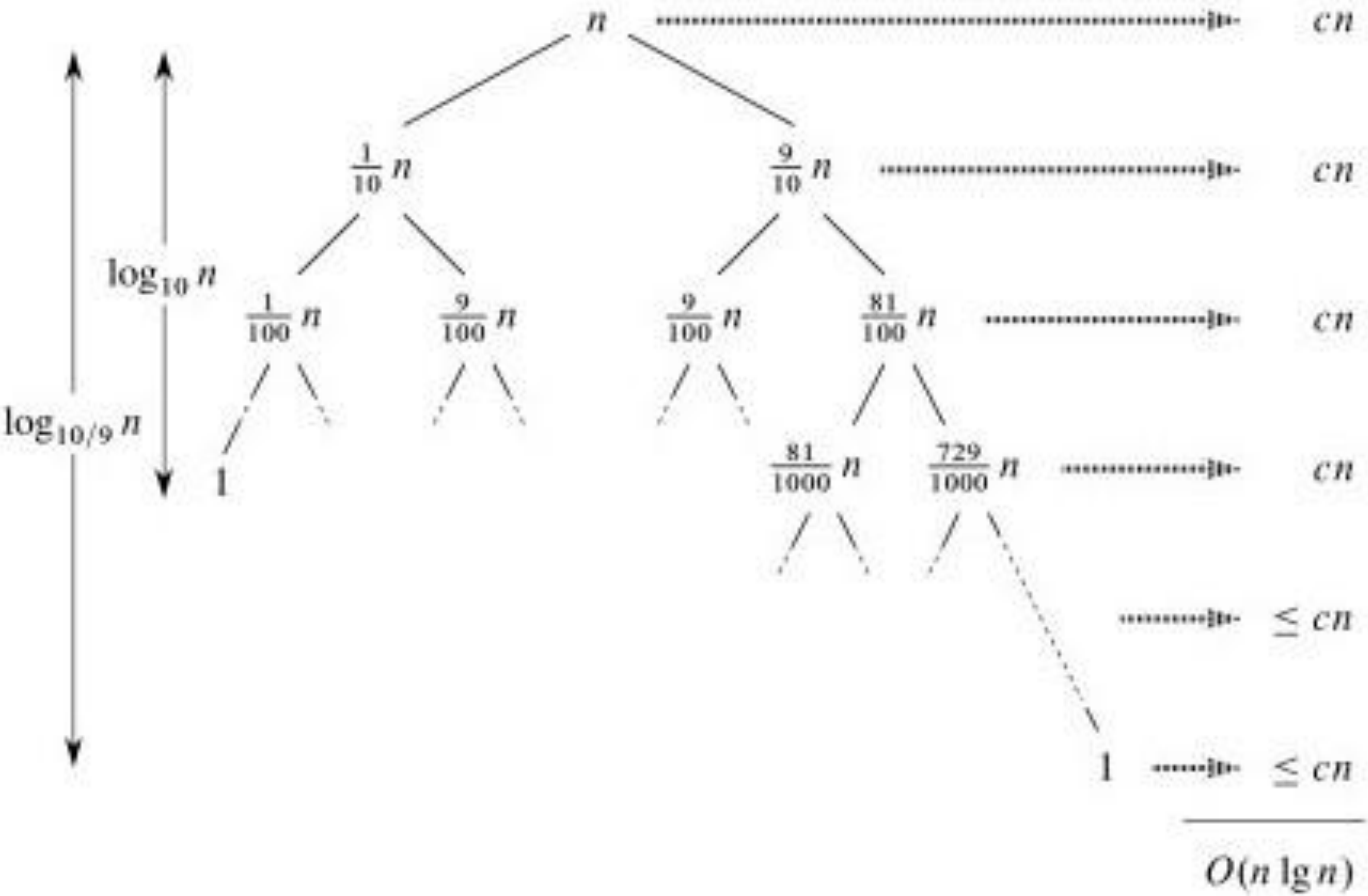
$C(n) = C(n-1) + 1+cn,\ C(1) = 1$

$\quad\quad = C(n-i) + i + c(n + n-1 + .. + n-i+1)$

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

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

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



n
cn
$\frac{1}{10}n$
$\frac{9}{10}n$
cn
$\log_{10} n$
$\log_{10/9} n$
$\frac{1}{100}n$
$\frac{9}{100}n$
$\frac{9}{100}n$
$\frac{81}{100}n$
cn
1
$\frac{81}{1000}n$
$\frac{729}{1000}n$
cn
$\leq cn$
1
$\leq cn$
$O(n \lg n)$

Quicksort:

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

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

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

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

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

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 $\mathbf{m} = \lfloor (n-1)/2 \rfloor$

$$K$$

vs

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

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:

// Finds the offset of an occurence of K

```
Algorithm BinarySearch(A[0..n-1], K)
   if(n = 0)
      return -1
   m = ⌊n/2⌋
   if(K = A[m])
      return m
   else if(K < A[m])
      return BinarySearch(A[0..m-1], K)
   else
      return BinarySearch(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 ← 0; r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

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

Input Size: **`n`**

Basic Operation : **`(K = A[m])`**

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

Worst case: $C(n) = C(n / 2) + 1, C(1) = 1$

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

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

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

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

Multiplication of Large Integers:

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

A = 12345678901357986429   B = 87654321284820912836

Brute-Force Strategy:

$$
\begin{array}{l}
\mathbf{a_1 \ \ a_2 \ \dots \ \ a_n * b_1 \ \ b_2 \ \dots \ \ b_n} \\
\qquad\qquad\qquad\qquad d_{10}d_{11}d_{12} \dots \ d_{1n} \\
\qquad\qquad d_{20}\ d_{21}d_{22} \dots \ d_{2n} \\
\quad \dots \ \dots \ \dots \ \dots \ \dots \ \dots \ \dots \\
d_{n0}d_{n1}d_{n2} \dots \ d_{nn}
\end{array}
$$

```
    2135 * 4014
    8540
   2135+
  0000++
 8540+++
________
 8569890
```

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) $\in$ $\boldsymbol{\Theta(n^2)}$

Multiplication of Large Integers by Divide-and-Conquer

Idea: To multiply A = **2**3 and B = **5**4.

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

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

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

For a base value 'x',

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

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

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

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

Multiplication of Large Integers by Divide-and-Conquer:

Idea:

To find A * B where A = **21**40 and B = **35**14

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

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

In general, if $A = \mathbf{A_1}A_2$ and $B = \mathbf{B_1}B_2$ (where A and B are *n*-digit, $\mathbf{A_1}$, A_2, $\mathbf{B_1}$ and B_2 are *n*/2-digit numbers),

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

Multiplication of Large Integers by Divide-and-Conquer:

In general, if A = $\mathbf{A_1}A_2$ and B = $\mathbf{B_1}B_2$ (where A and B are *n*-digit, $\mathbf{A_1}$, A_2, $\mathbf{B_1}$ and B_2 are *n*/2-digit numbers),

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

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

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

Basic operation: single-digit multiplication

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

∴ $C(n) \in \mathbf{\Theta(n^2)}$

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

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

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

$(\mathbf{A_1} * B_2 + A_2 * \mathbf{B_1}) = (\mathbf{A_1} + A_2) * (\mathbf{B_1} + B_2) - \mathbf{A_1} * \mathbf{B_1} - A_2 * B_2$, which requires only 3 multiplications at the expense of 3 extra add/sub operations. Note that we are reusing $\mathbf{A_1} * \mathbf{B_1}$ and $A_2 * B_2$ one more time.

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

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

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

It needs three n/2 digits multiplications:

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

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

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

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

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 (p1.10^n + (p3-p1-p2).10^m + p2)[0..2n-1]
```

Time complexity of the **Karatsuba algorithm:**

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

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

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

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

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

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

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 x 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 x n/2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

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

Basic operation: atomic-element multiplication

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

$\therefore$ $C(n) \in \boldsymbol{\Theta(n^3)}$

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

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

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

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

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

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

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

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

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

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 $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \cdots$$
$$= 7^iM(2^{k-i}) \cdots = 7^kM(2^{k-k}) = 7^k.$$

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:

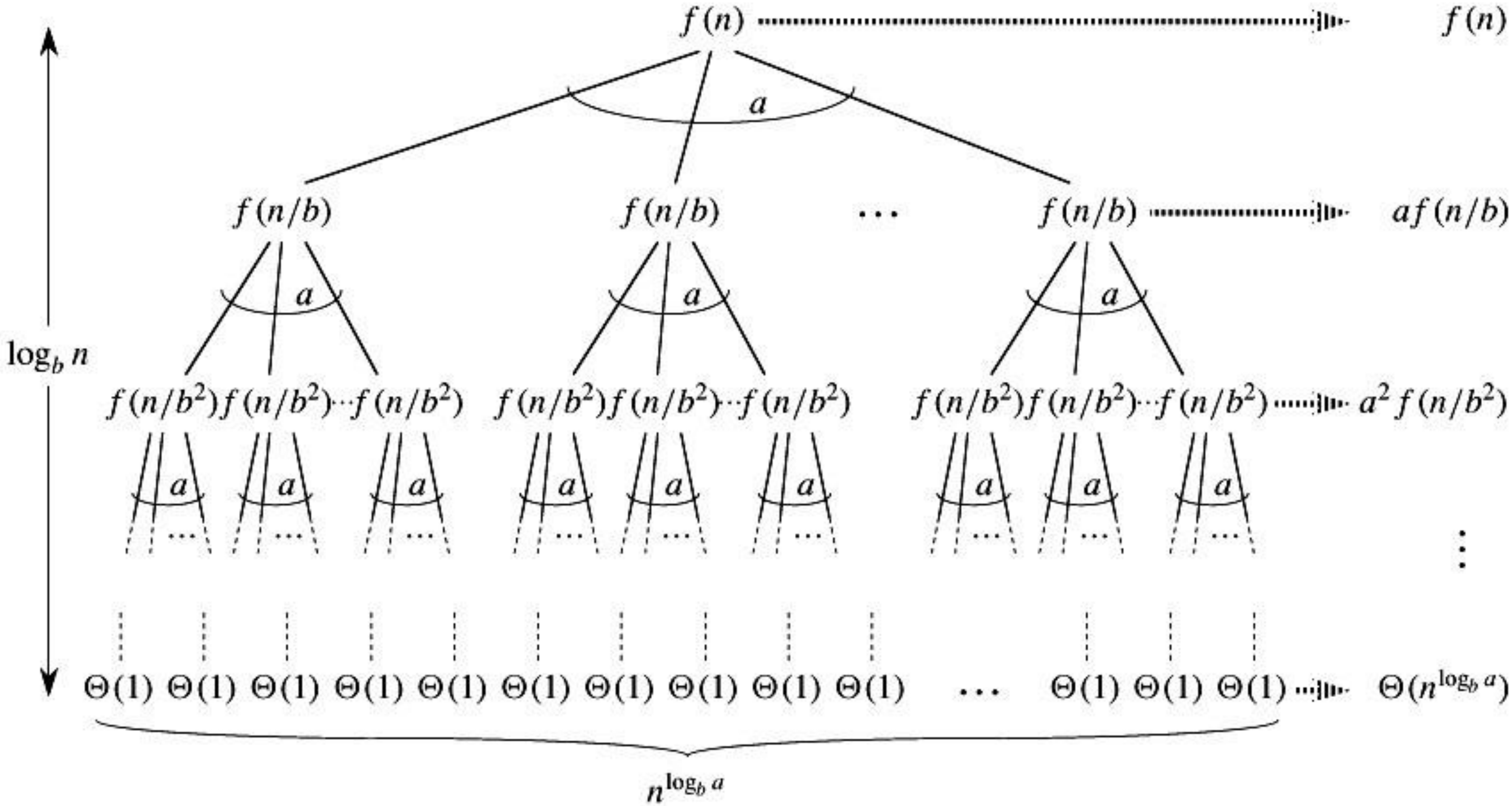
$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0$$

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

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

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 $f(n) \in \Theta(n^d)$, $d \geq 0$

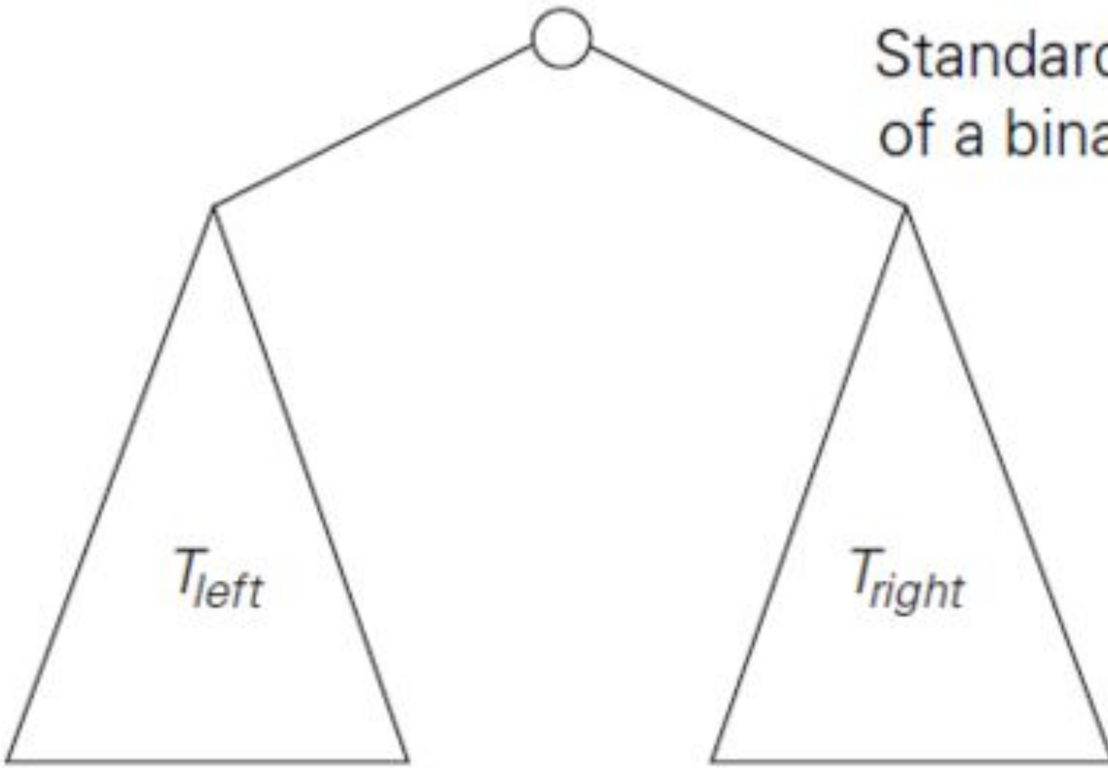
$$\text{If } a < b^d,\quad T(n) \in \Theta(n^d)$$

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

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

- 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.

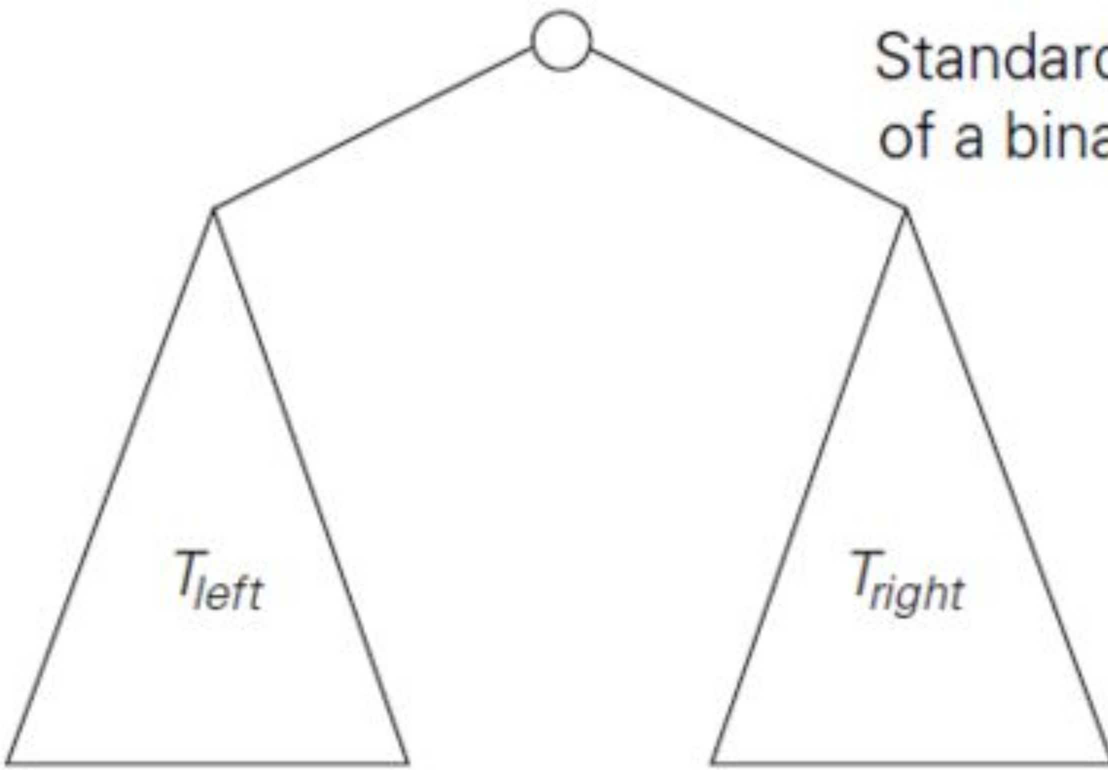


Standard representation of a binary tree.

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 $T = \varnothing$ **return** -1
else return $\max\{Height(T_L), Height(T_R)\} + 1$

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

Basic Operation : **`Addition`**

$C(n(T)) = C(n(T_L)) + C(n(T_R)) + 1,\ C(0) = 0$

$= \mathbf{n(T) \in \Theta(n(T))}$

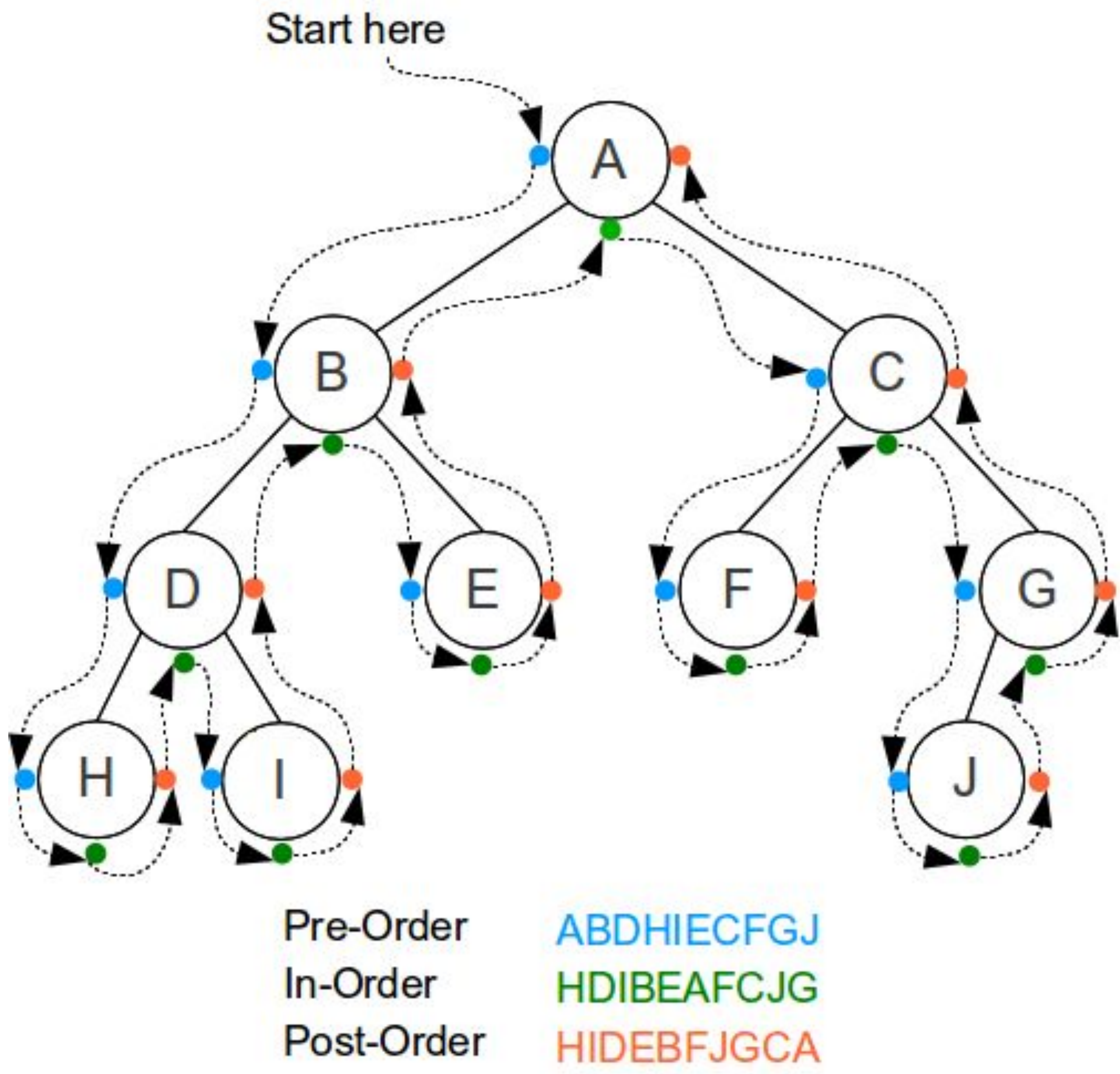
Basic Operation : **`(T = Φ)`**

$C(n(T)) = C(n(T_L)) + C(n(T_R)) + 1,\ C(0) = 1$

$= \mathbf{2n(T) + 1 \in \Theta(n(T))}$

Binary Tree Traversals:

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



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 >