late winter blossoms
crow perches on a bare branch
nature's calm order

Application Area	Description
Search Optimization	Enables efficient algorithms like binary search.
Data Organization	Critical for databases and file systems, facilitating quick data retrieval.
Algorithm Efficiency	Enhances strategies in divide and conquer algorithms and graph algorithms.
Data Compression	Groups similar data for more effective compression techniques.
Statistical Analysis	Facilitates the calculation of statistics such as medians and quartiles.
Machine Learning	Used in feature scaling and preprocessing for algorithms.
Rendering and Visualization	Essential for graphics rendering and coherent data visualizations.
Scheduling and Resource Allocation	Optimizes job scheduling and task prioritization.
Network Routing	Ensures correct packet sequencing in data transmission.

Problem: Find common elements between two sets of numbers

```python
def find_common_naive(set1, set2):
    """Find common elements between two sets (naive approach)."""
    common = []
    for elem in set1:
        if elem in set2:
            common.append(elem)
    return common
```

Example Input

set1	**set2**
{1, 3, 5, 7, 9}	{1, 2, 3, 4, 5}

Output:
{1, 3, 5}

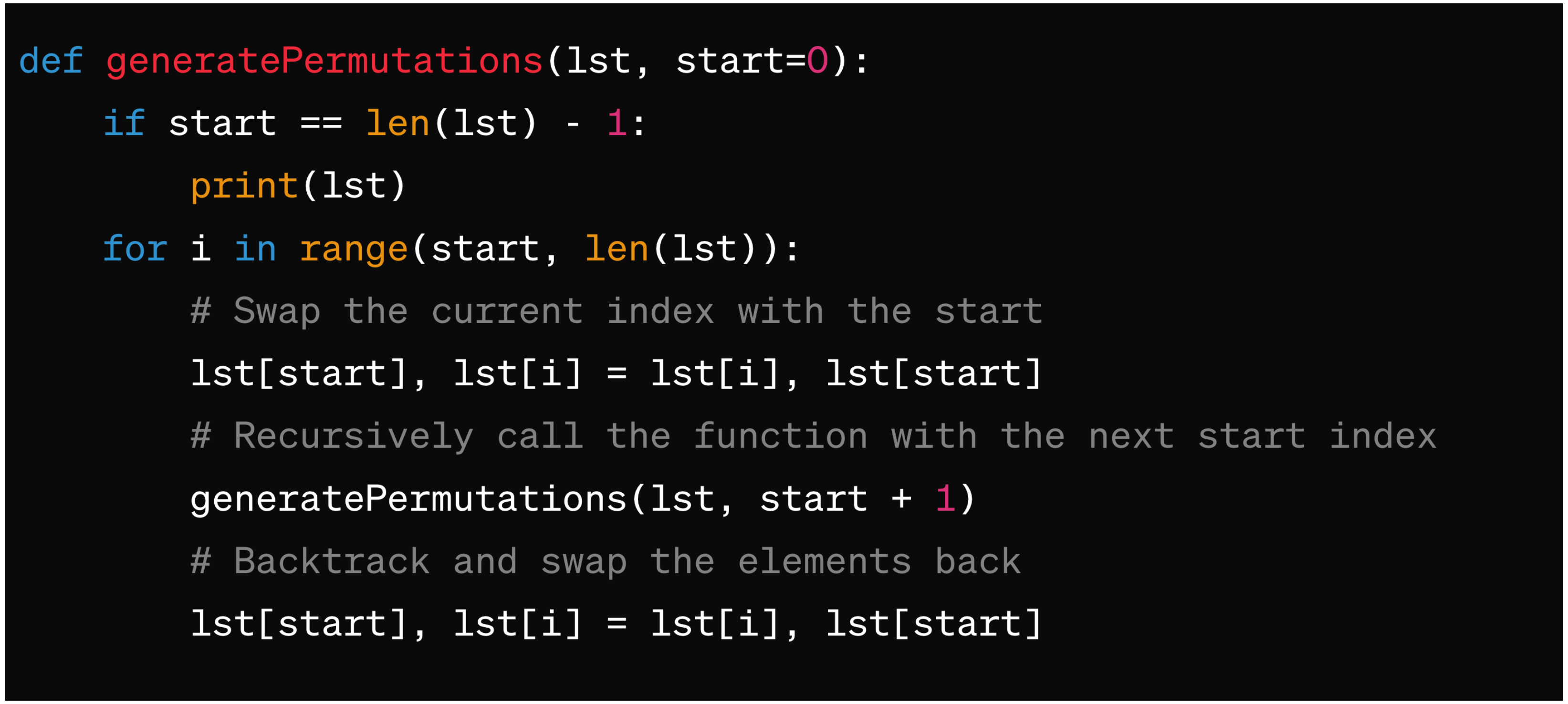
```python
def find_common_sorted(set1, set2):
    """Find common elements between two sets (efficient approach using sorting)."""
    common = []
    sorted_set1 = sorted(set1)
    sorted_set2 = sorted(set2)
    i, j = 0, 0
    while i < len(sorted_set1) and j < len(sorted_set2):
        if sorted_set1[i] == sorted_set2[j]:
            common.append(sorted_set1[i])
            i += 1
            j += 1
        elif sorted_set1[i] < sorted_set2[j]:
            i += 1
        else:
            j += 1
    return common
```

A **permutation** of a set is any rearrangement of its members into a sequence or linear order.

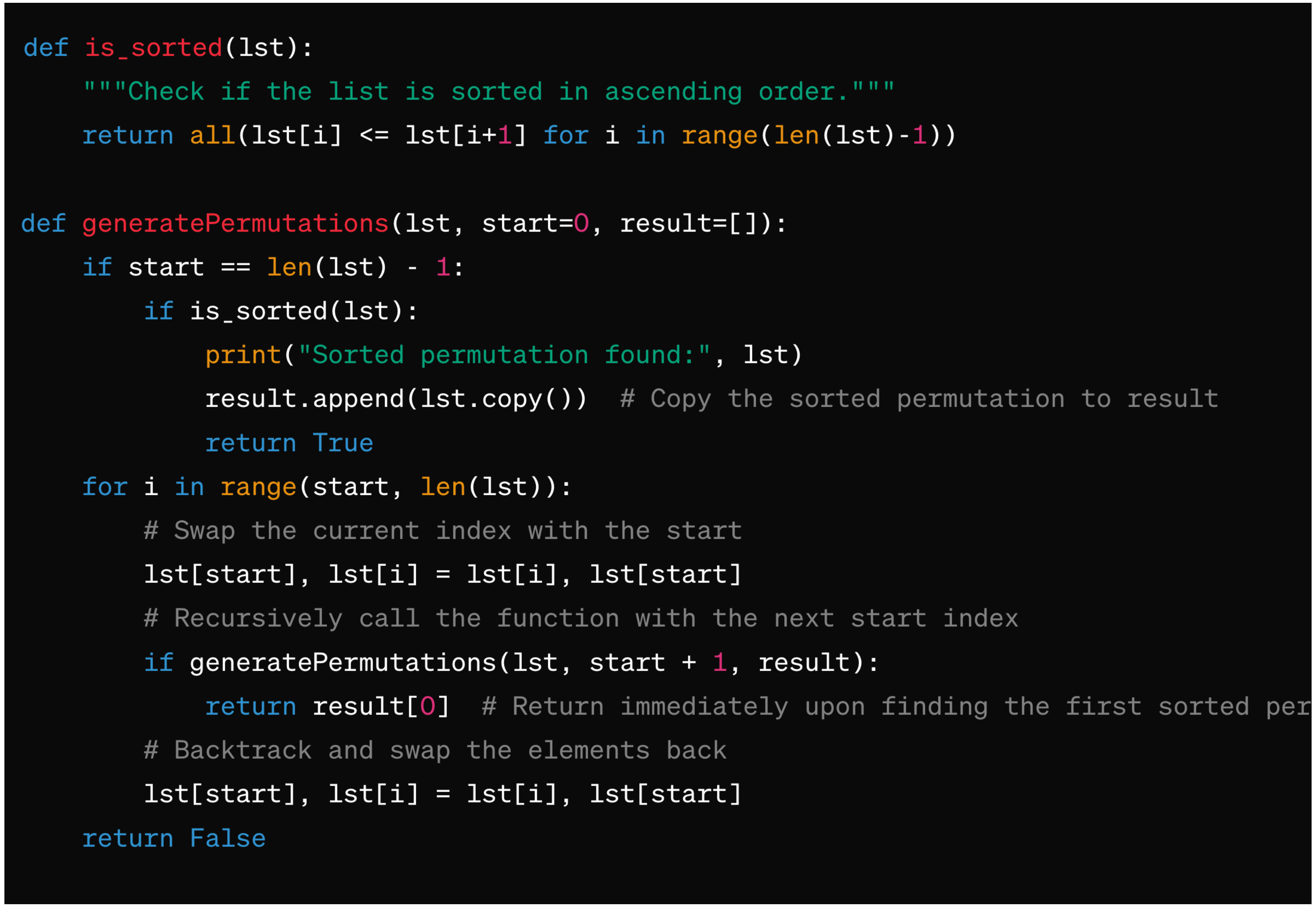
For a set of ***n*** elements, there are ***n!*** (factorial of ***n***) possible permutations.

For example, the set {1, 2, 3} has ***3! = 6*** permutations:

```
123, 132, 213, 231, 312, 321
```



```python
def generatePermutations(lst, start=0):
    if start == len(lst) - 1:
        print(lst)
    for i in range(start, len(lst)):
        # Swap the current index with the start
        lst[start], lst[i] = lst[i], lst[start]
        # Recursively call the function with the next start index
        generatePermutations(lst, start + 1)
        # Backtrack and swap the elements back
        lst[start], lst[i] = lst[i], lst[start]
```



```python
def is_sorted(lst):
    """Check if the list is sorted in ascending order."""
    return all(lst[i] <= lst[i+1] for i in range(len(lst)-1))

def generatePermutations(lst, start=0, result=[]):
    if start == len(lst) - 1:
        if is_sorted(lst):
            print("Sorted permutation found:", lst)
            result.append(lst.copy())  # Copy the sorted permutation to result
            return True
    for i in range(start, len(lst)):
        # Swap the current index with the start
        lst[start], lst[i] = lst[i], lst[start]
        # Recursively call the function with the next start index
        if generatePermutations(lst, start + 1, result):
            return result[0]  # Return immediately upon finding the first sorted per
        # Backtrack and swap the elements back
        lst[start], lst[i] = lst[i], lst[start]
    return False
```

[Nietzsche, Weil, Kant, Foucault, Wittgenstein, de Beauvoir, Plato, Sartre, Arendt, Lao Tzu]

[1 april, 2 janurary, 9 december, 14 june, 18 october]

[Star Wars, Star Trek, Dune, Blade Runner]

Sorting can be broadly defined as the process of arranging items in a systematic sequence (ascending or descending) based on some ***comparative*** criteria.

The efficiency of a sorting algorithm largely depends on how it ***compares elements*** and ***rearranges*** them into the correct order.

A comparison between two elements a and b in a set S can be mathematically represented by defining a binary relation $\leq$ on S, where for any two elements $a, b \in S$, one and only one of the following is true:

$$a < b, \quad a = b, \quad or \quad a > b$$

1. **Transitivity**: If $a \leq b$ and $b \leq c$, then $a \leq c$.
2. **Reflexivity**: For any element a, $a \leq a$.
3. **Antisymmetry**: If $a \leq b$ and $b \leq a$, then $a = b$.

Exploitable Properties

Transitivity of Order: If element A is less than element B, and B is less than C, then A is less than C. Efficient sorting algorithms rely on this property to reduce the number of comparisons needed to determine order across a set.

Stability: The concept that equal elements retain their relative order before and after sorting. Some algorithms maintain stability by carefully managing how elements are moved. This property is particularly important when the sort order depends on multiple fields.

Existence of Natural Runs: In real-world data, sequences of elements are often already in order or nearly in order. Adaptive sorting algorithms take advantage of these "runs" to minimize work, adjusting their approach based on the degree of existing order.

Existence of Identifiable Substructures: Many sorting problems contain inherent substructures that can be isolated and addressed independently before integrating them into a larger solution. For example, divide and conquer algorithms break the problem into smaller, more manageable chunks (subarrays) that are easier to sort.

Distribution of Input: Algorithms like Counting Sort and Radix Sort exploit specific characteristics of the input distribution, such as the range of numerical values or the length of strings, to organize data more efficiently than comparison-based methods.

Relative Ordering (Binary Relations): Sorting fundamentally relies on the ability to determine a binary relation (e.g., less than, greater than) between any two elements. Efficient algorithms optimize how many and which comparisons are made to determine the overall order with fewer operations.

Element Uniqueness and Repetition: Some sorting algorithms are particularly well-suited to handling data with many repeated values (e.g., counting sort) or ensure unique outcomes by managing how duplicate values are treated.

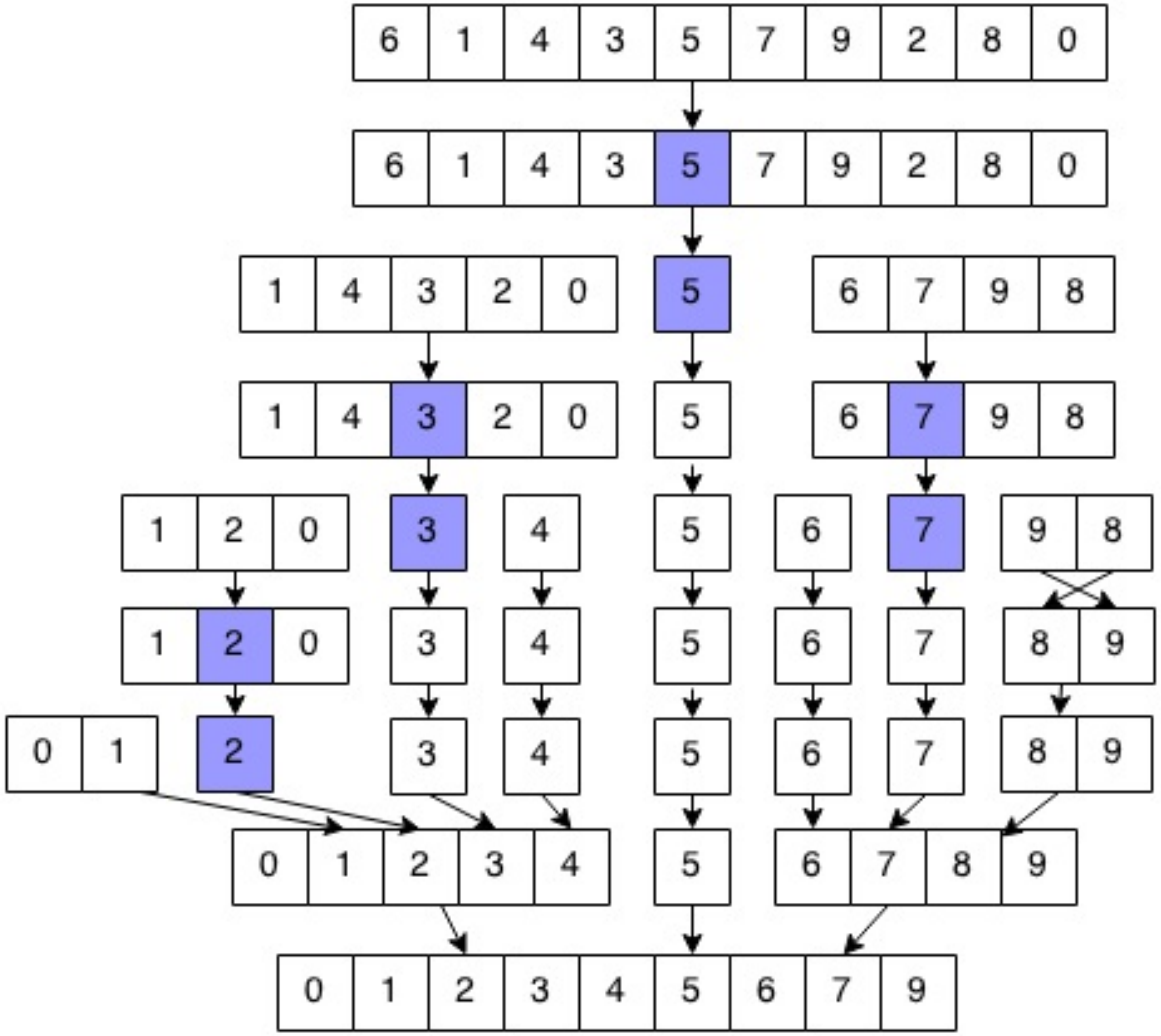
- **Divide and Conquer**: Exploits substructure by breaking the problem down and solving smaller problems independently.
- **Hybrid Approaches**: Use the stability and natural runs properties to combine sorting methods for optimal performance on varied data.
- **Adaptive Sorting**: Adjusts algorithmic behavior in real-time based on the degree of existing order, leveraging natural runs.
- **Counting and Positioning**: Utilize the distribution of input and relative ordering properties to sort without direct element-to-element comparisons.

Property	Algorithms	Description
Transitivity of Order	All comparison-based sorts (e.g., Bubble, Quick, Merge)	These algorithms rely on the transitive nature of the comparison to sort the entire dataset by comparing pairs of elements.
Identifiable Substructures	MergeSort, QuickSort	Divide and conquer algorithms break the dataset into smaller pieces, sort those pieces, and then combine them, exploiting the problem's substructures.
Stability	MergeSort, Bubble Sort, Insertion Sort	These algorithms maintain the relative order of equal elements, making them suitable for multi-field sorting.
Natural Runs	Timsort, Insertion Sort	These adaptive algorithms exploit existing order within the dataset to reduce the number of operations required.
Distribution of Input	Counting Sort, Radix Sort, Bucket Sort	These non-comparison sorts organize data based on the distribution of input values, often leading to linear time sorting under certain conditions.
Relative Ordering	All comparison-based sorts	Fundamental to comparison-based sorting, this property is directly used to determine the sorted order through element comparisons.
Element Uniqueness and Repetition	Counting Sort, Radix Sort	Efficient in handling datasets with many repeated values by organizing elements based on their counts or digit/character positions.

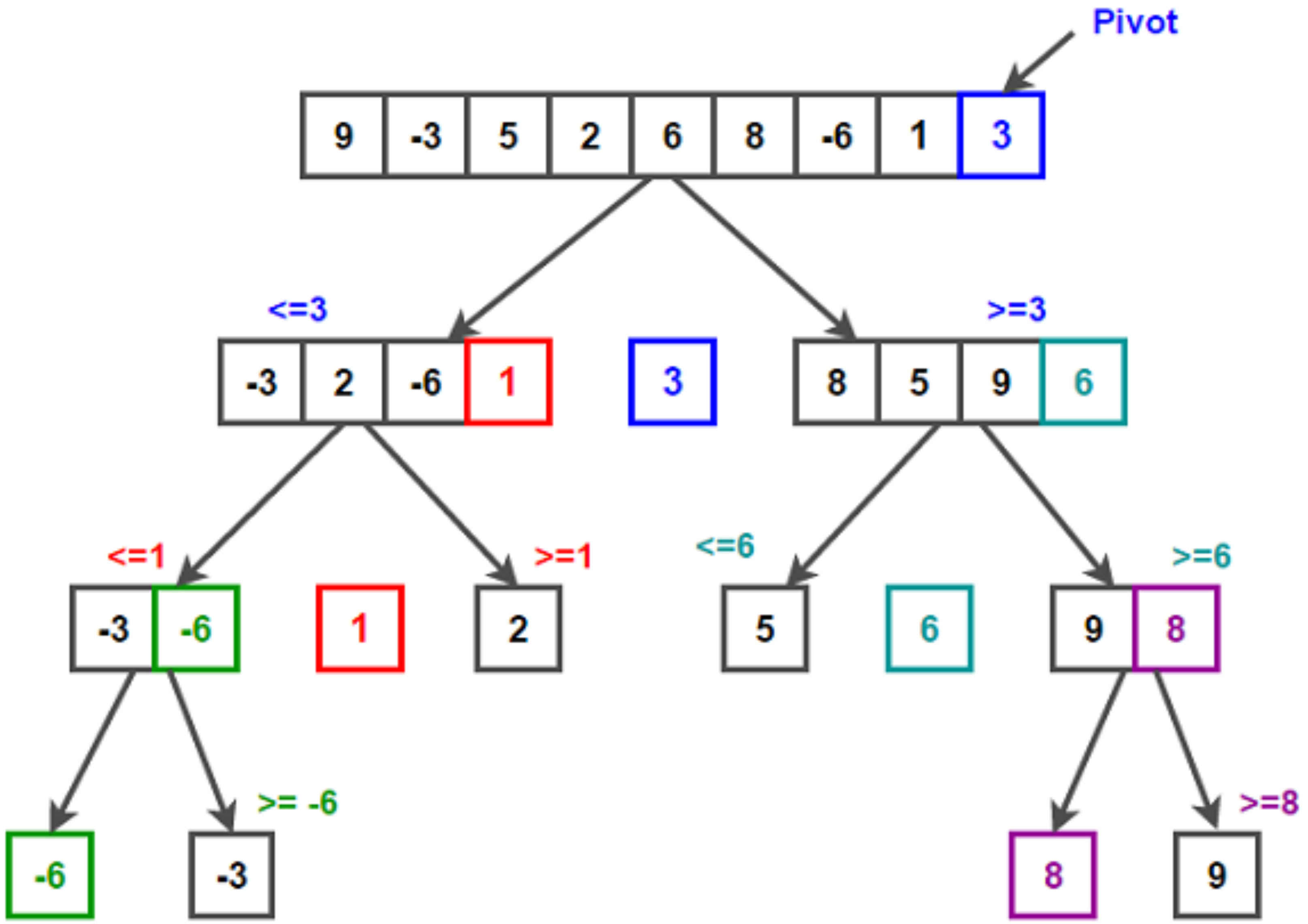
2
3
4
5
6
7
8
9
10
J
Q
K
A

https://visualgo.net/en/sorting

Quick Sort

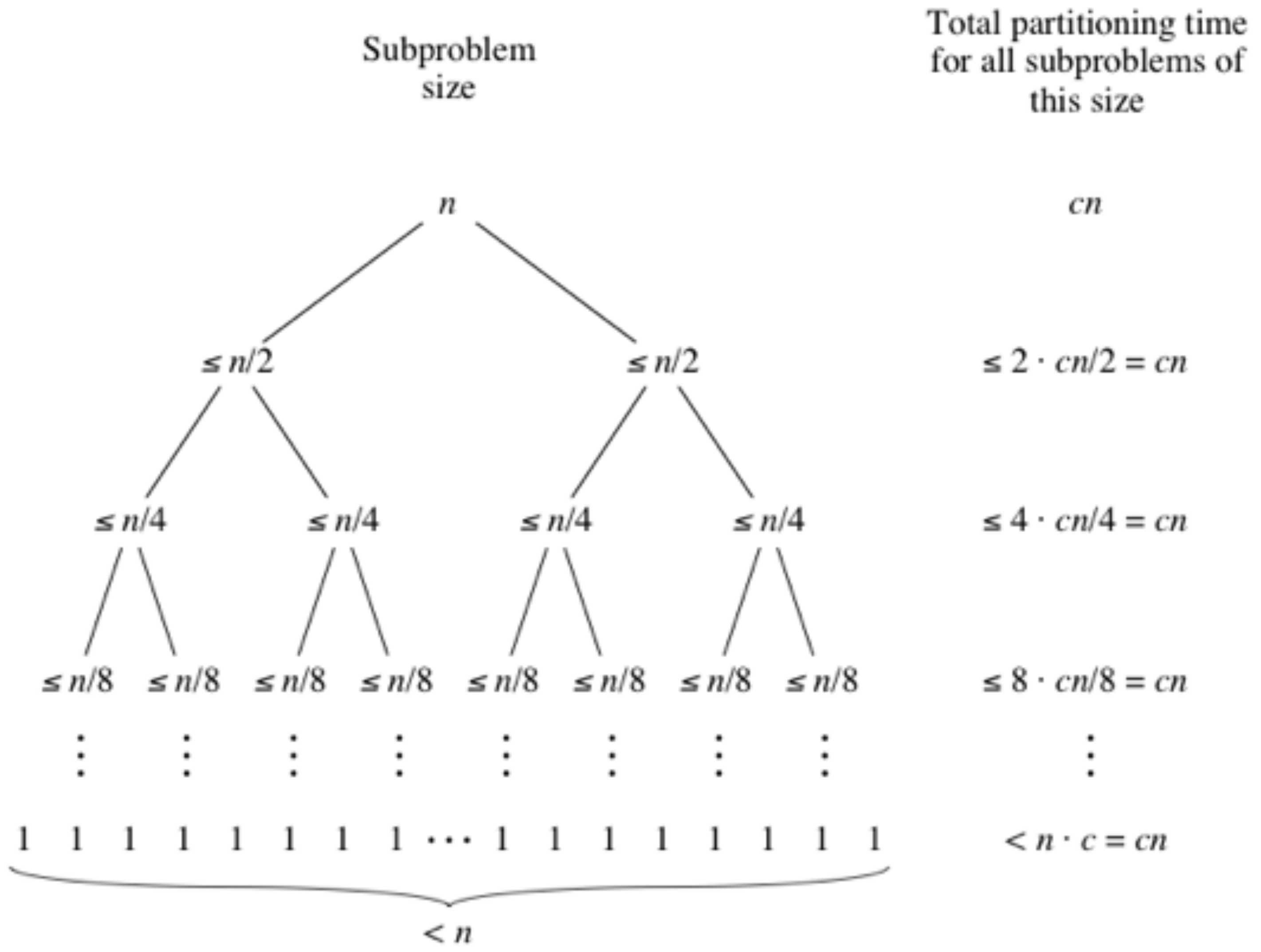


6 1 4 3 5 7 9 2 8 0
6 1 4 3 5 7 9 2 8 0
1 4 3 2 0 5 6 7 9 8
1 4 3 2 0 5 6 7 9 8
1 2 0 3 4 5 6 7 9 8
1 2 0 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 9

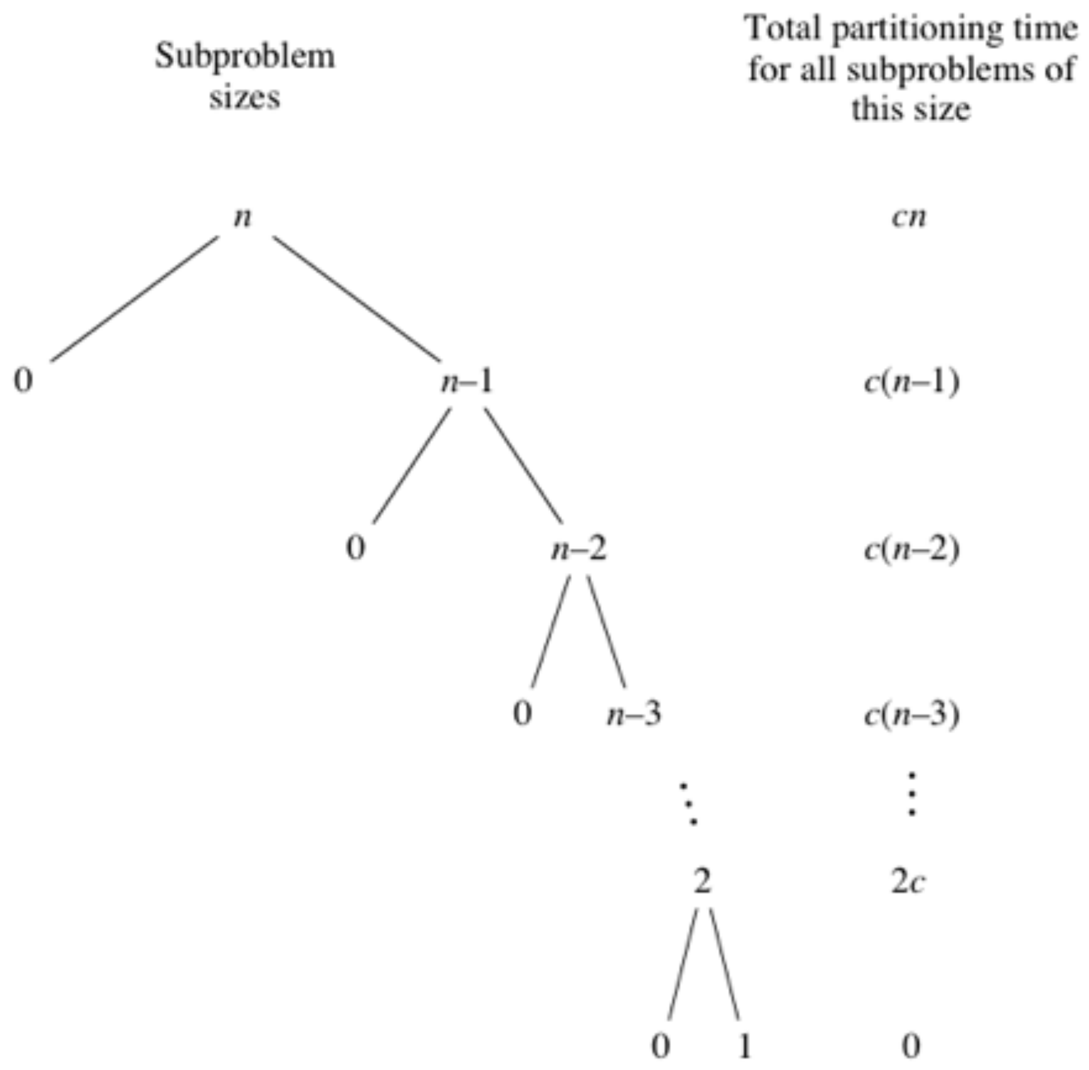

Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
>=3
8 5 9 6
<=1
-3 -6
1
>=1
2
<=6
5
6
>=6
9 8
-6
>= -6
-3
8
>=8
9

Quick Sort - Pivot

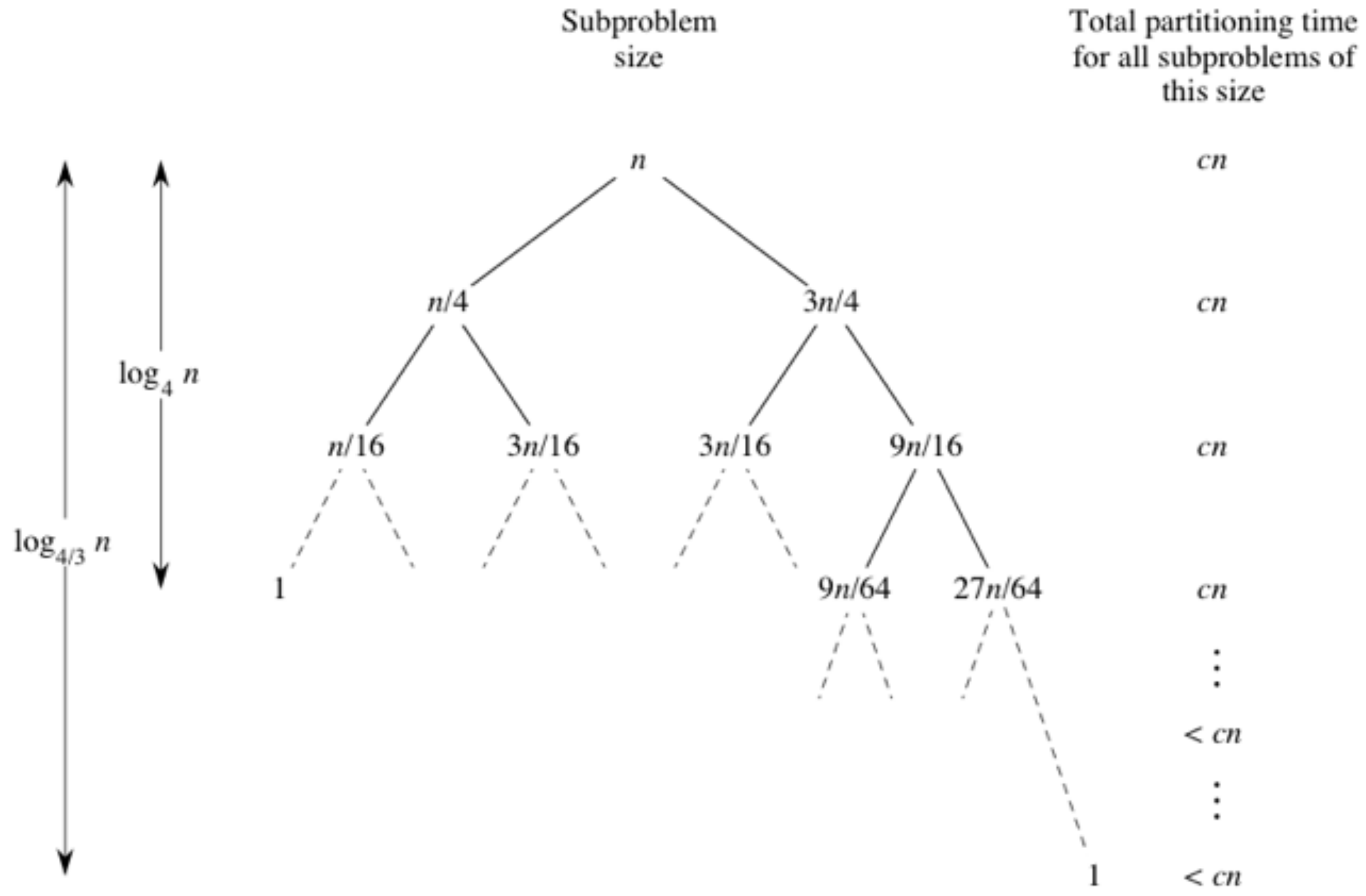
Quick Sort Complexity (best case)



QuickSort Complexity (worst case)



QuickSort Complexity (‘example’ case)



$(4/3)^x = n$ for which x?

answer:

$\log_{4/3} n$

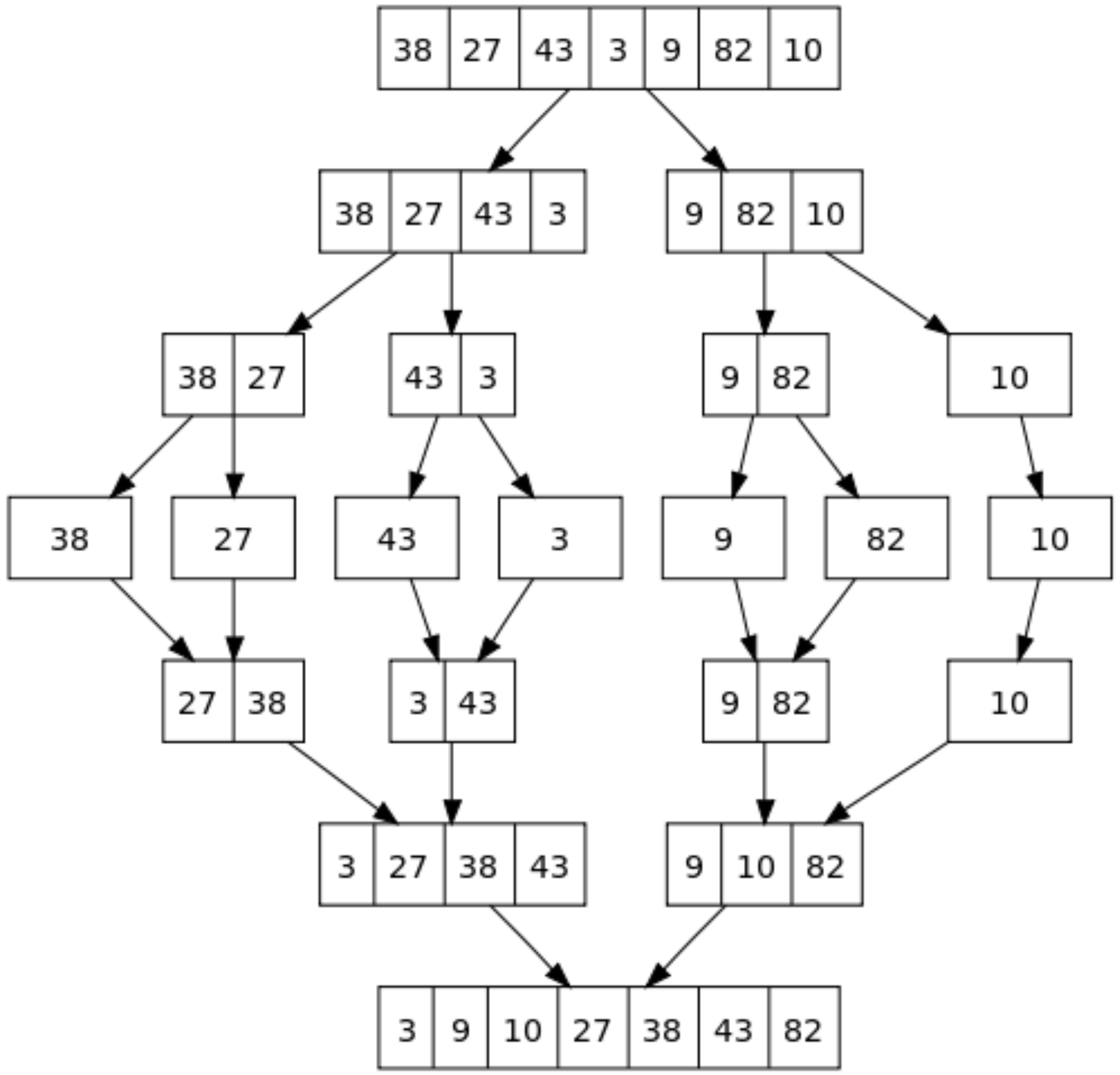
and:

$$\log_b(x) = \log_c(x) / \log_c(b)$$

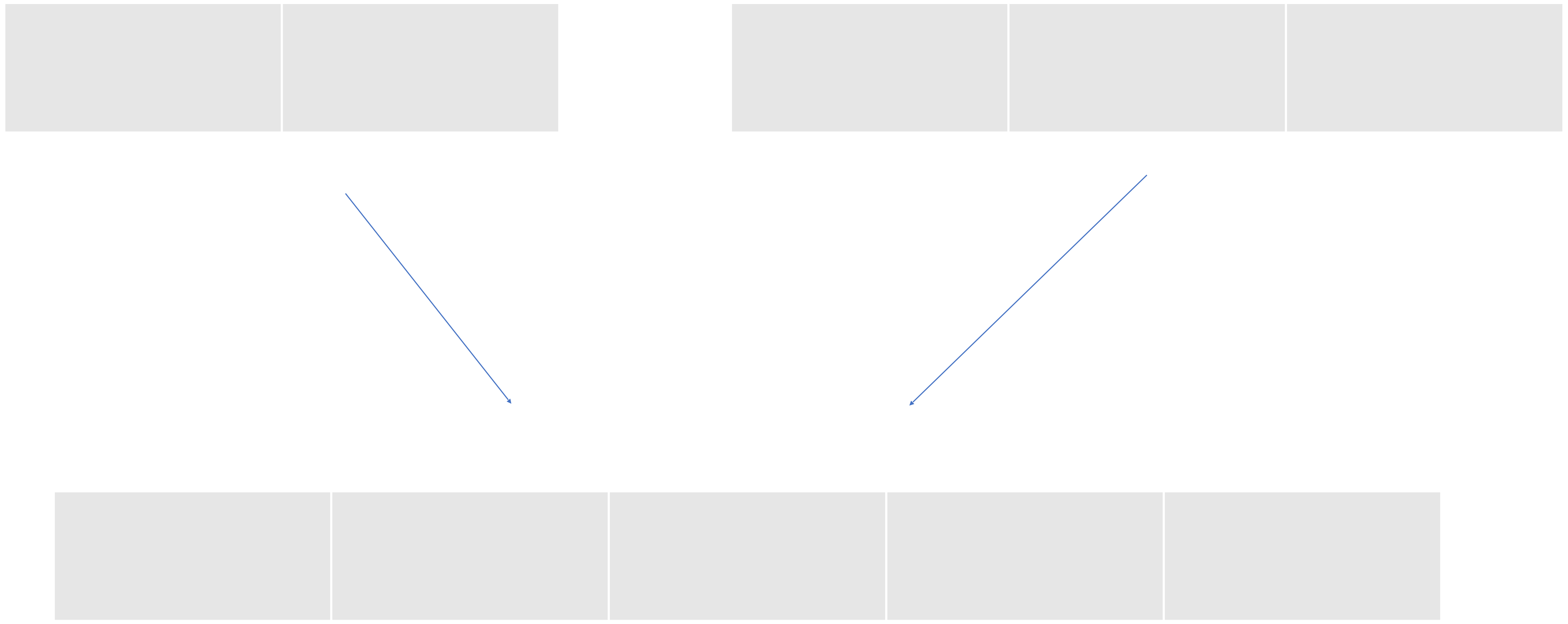
thus:

$$\log_{4/3} n = \frac{\log_2 n}{\log_2(4/3)}$$

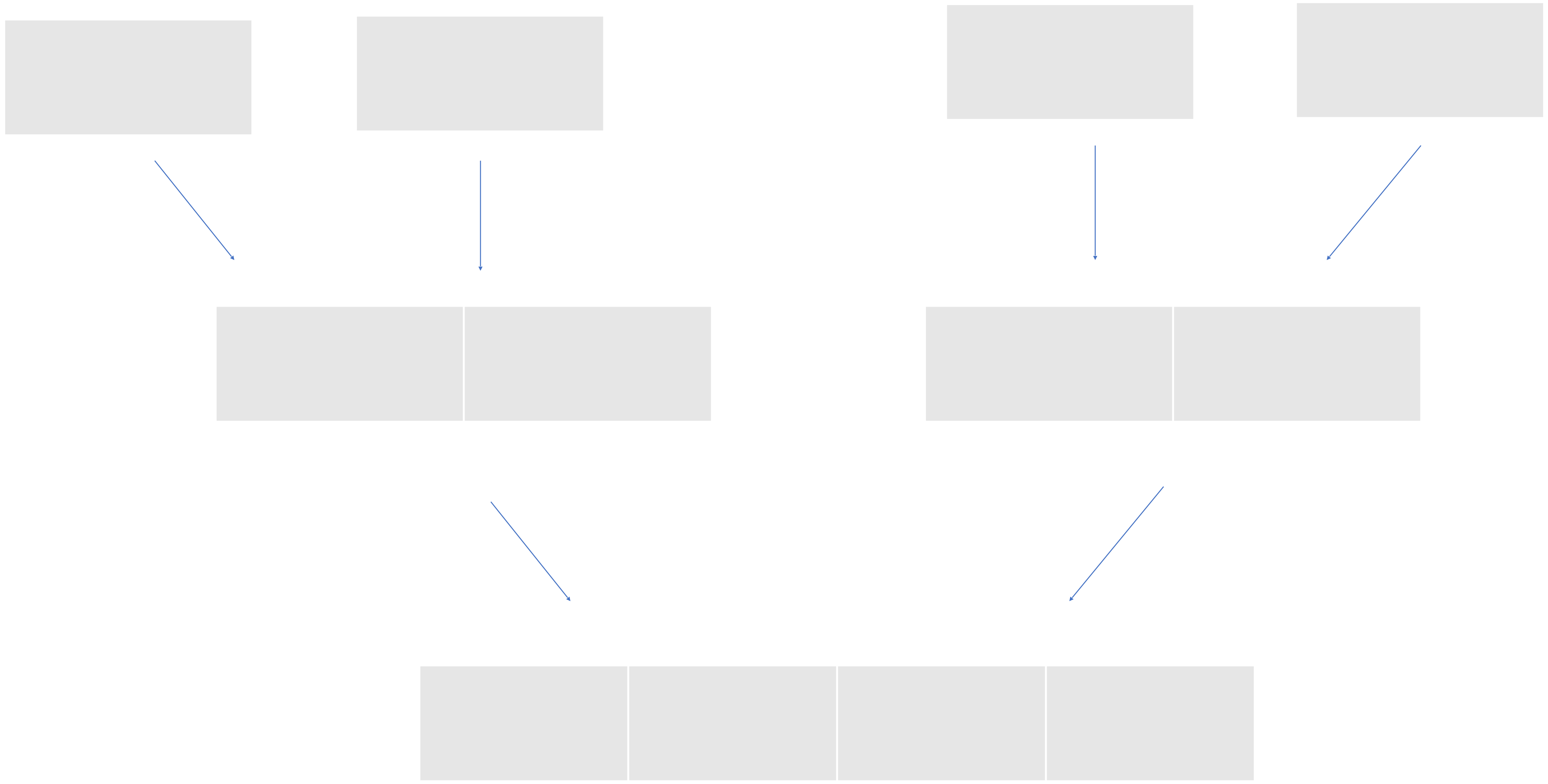
Merge Sort

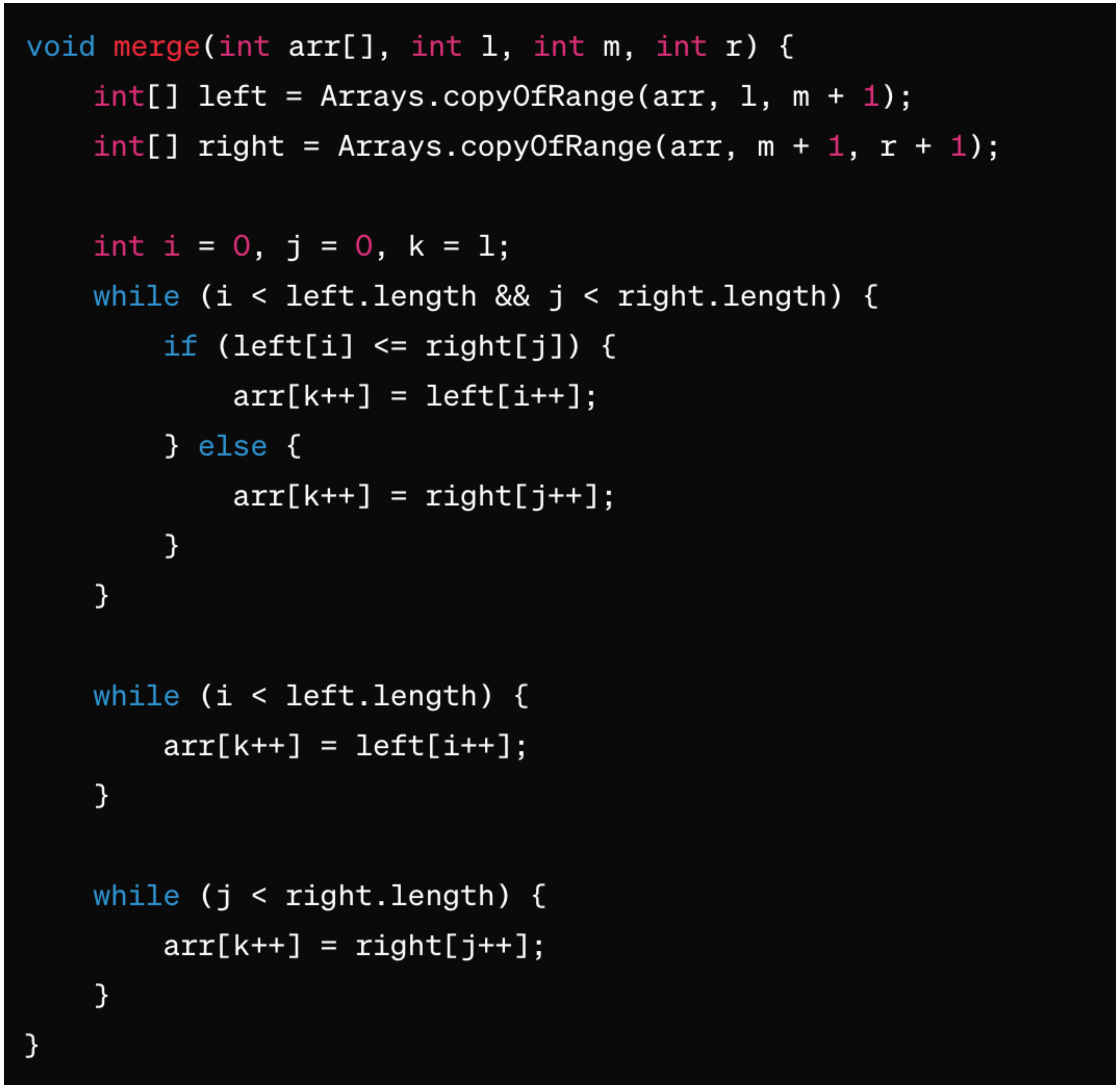

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort - Merging



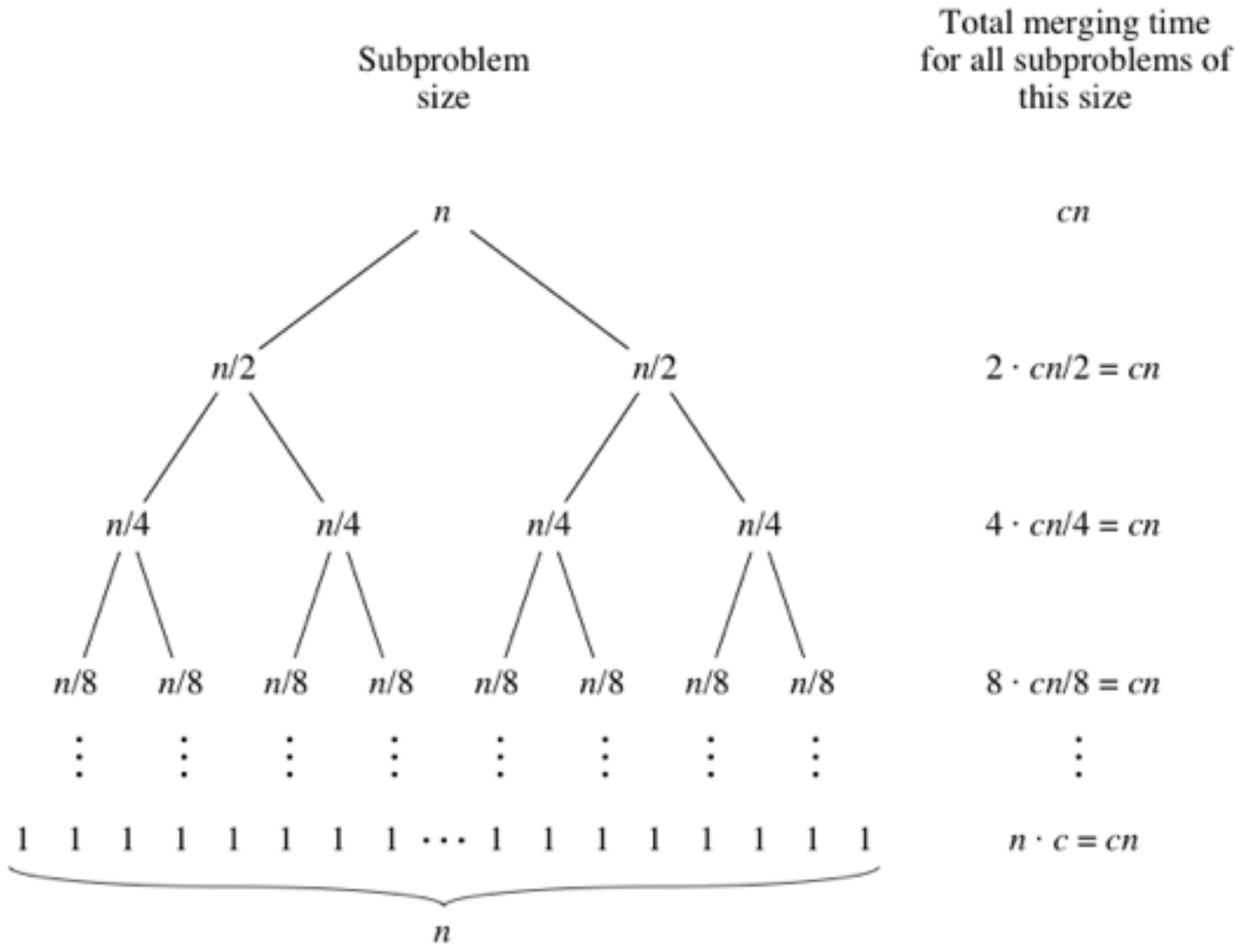
Merge Sort - Merging





```java
void merge(int arr[], int l, int m, int r) {
    int[] left = Arrays.copyOfRange(arr, l, m + 1);
    int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);

    int i = 0, j = 0, k = l;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }

    while (i < left.length) {
        arr[k++] = left[i++];
    }

    while (j < right.length) {
        arr[k++] = right[j++];
    }
}
```

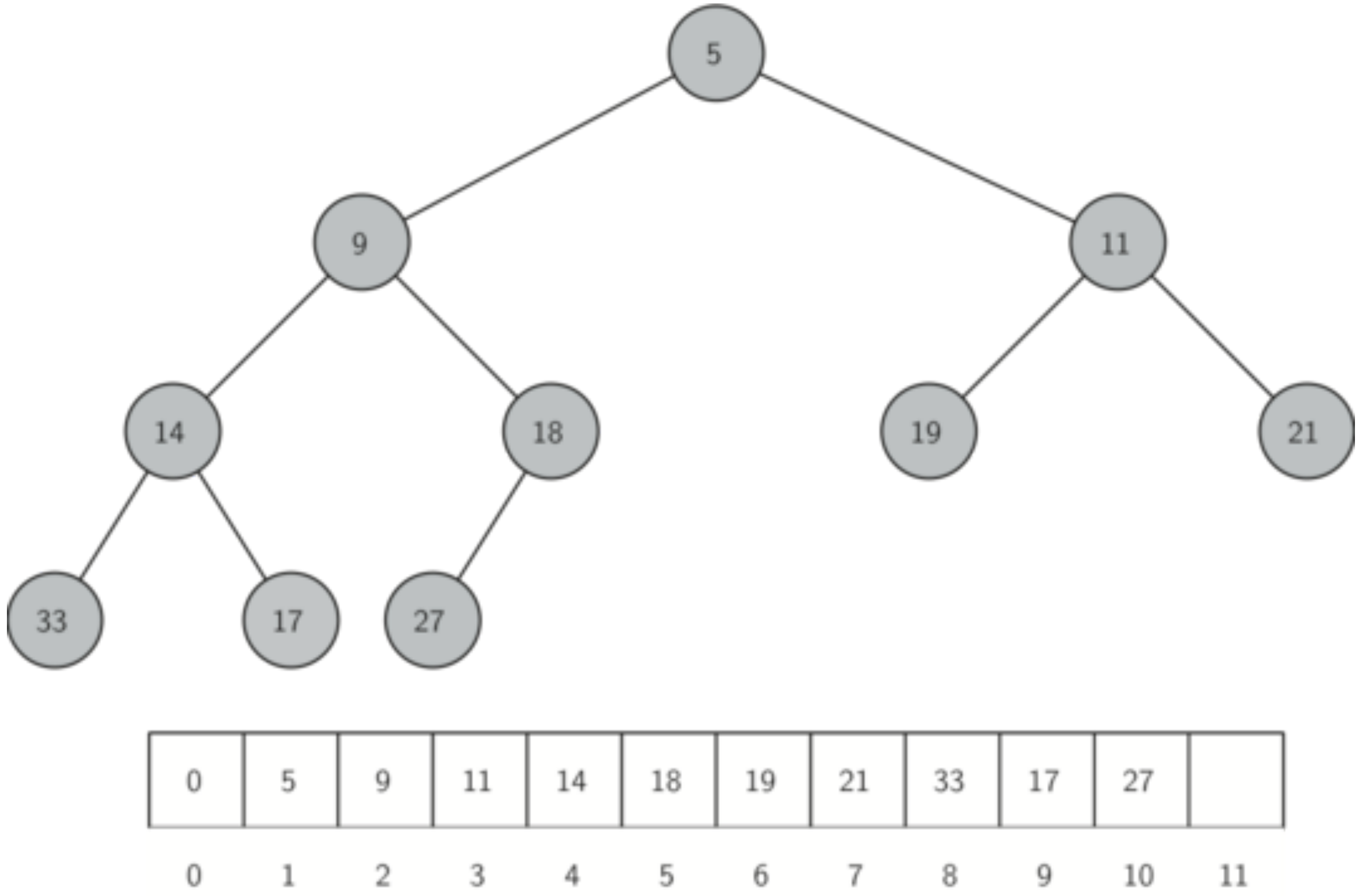
Merge Sort Complexity

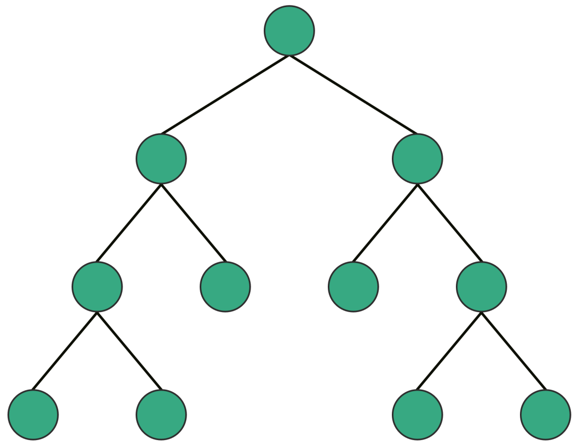


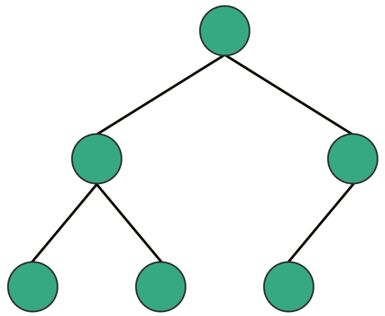
https://www.youtube.com/watch?v=dENca26N6V4

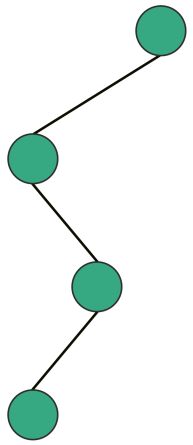
https://www.youtube.com/watch?v=3San3uKKHgg

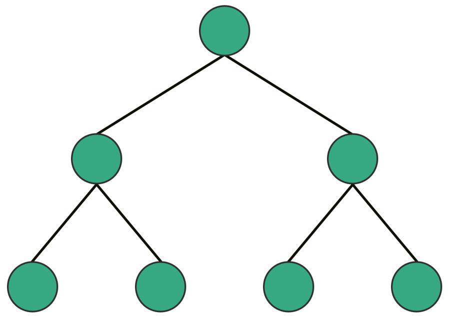
Heap

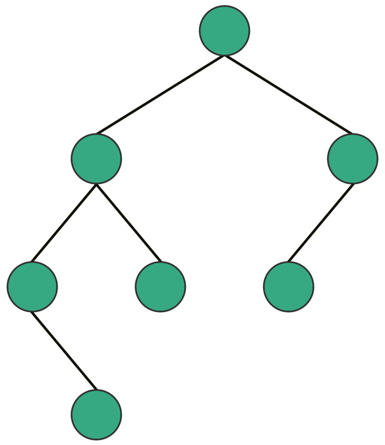

5
9
11
14
18
19
21
33
17
27
0 5 9 11 14 18 19 21 33 17 27
0 1 2 3 4 5 6 7 8 9 10 11


Full


Complete


Degenerate


Perfect


Balanced

The number of nodes, N, in a perfectly complete binary tree (one that is completely filled, including the last level) of height h can be calculated using the formula:
$N = 2^0 + 2^1 + 2^2 + ... + 2^h = 2^{h+1} - 1$

Given a heap with N nodes, we want to find the relationship between N and the height h.

- For the minimum number of nodes at height h, we have $N \geq 2^h$.
- For the maximum number of nodes before increasing the height to $h + 1$, we have $N \leq 2^{h+1} - 1$.

$$\lfloor \log_2(N) \rfloor \leq h \leq \lceil \log_2(N + 1) - 1 \rceil$$

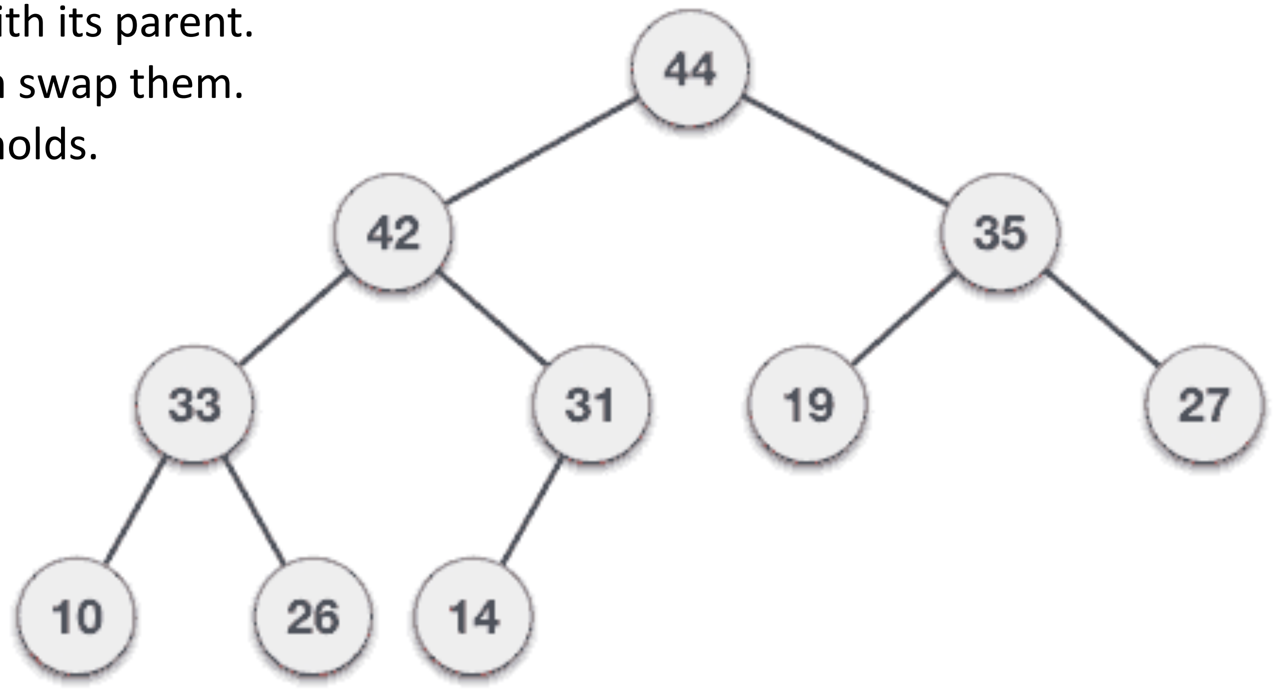
Heap Insertion

Input 35 33 42 10 14 19 27 44 26 31

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

Heap Deletion

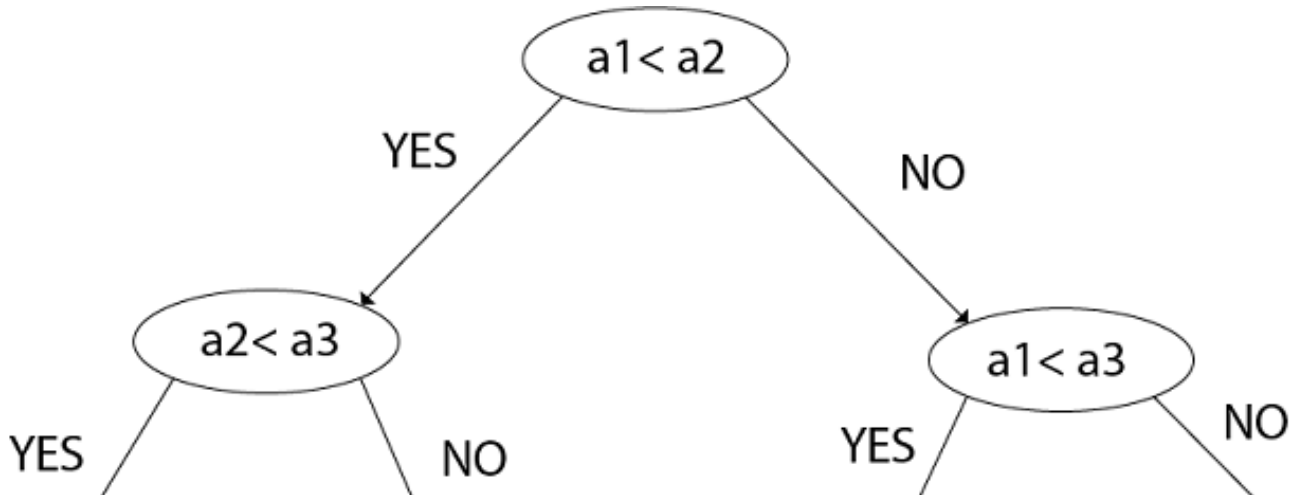
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.



Swap with its smaller child in a min-heap and its larger child in a max-heap

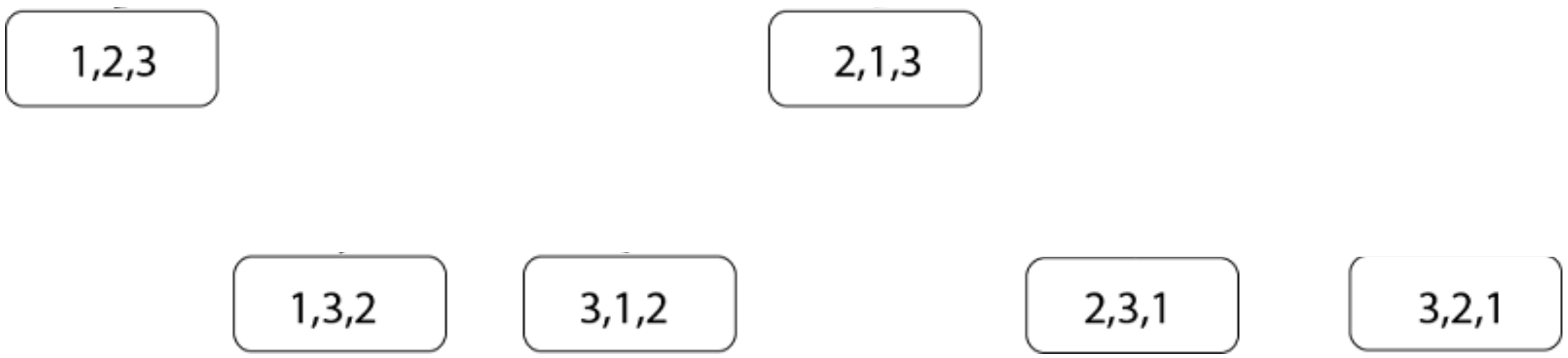
Lower Bound

```
function sortArray(arr):

    if arr is sorted:

        print "Sorted array:", arr
        return

    for i from 0 to length(arr)-1:

        for j from i+1 to length(arr):

            if arr[i] > arr[j]:

                swap(arr[i], arr[j])
                sortArray(arr)
                swap(arr[i], arr[j])
```


a1< a2
YES
NO
a2< a3
YES
NO
a1< a3
YES
NO

Lower Bound

```
function sortArray(arr):
    if arr is sorted:
        print "Sorted array:", arr
        return

    for i from 0 to length(arr)-1:
        for j from i+1 to length(arr):
            if arr[i] > arr[j]:
                swap(arr[i], arr[j])
                sortArray(arr)
                swap(arr[i], arr[j])
```

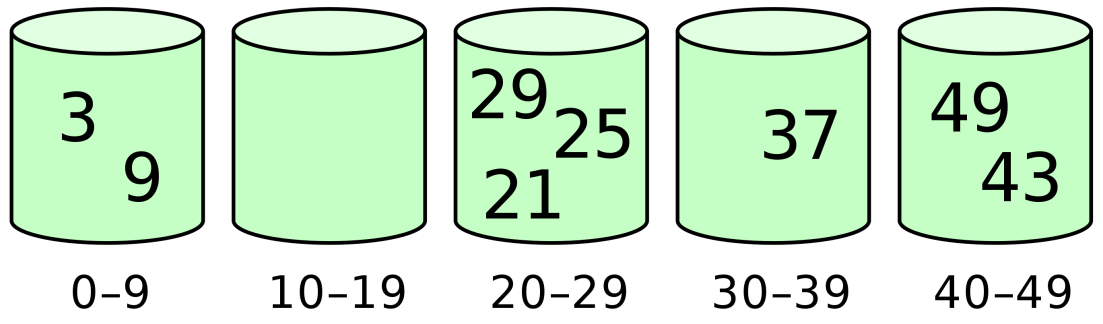


Stirling's Approximation:

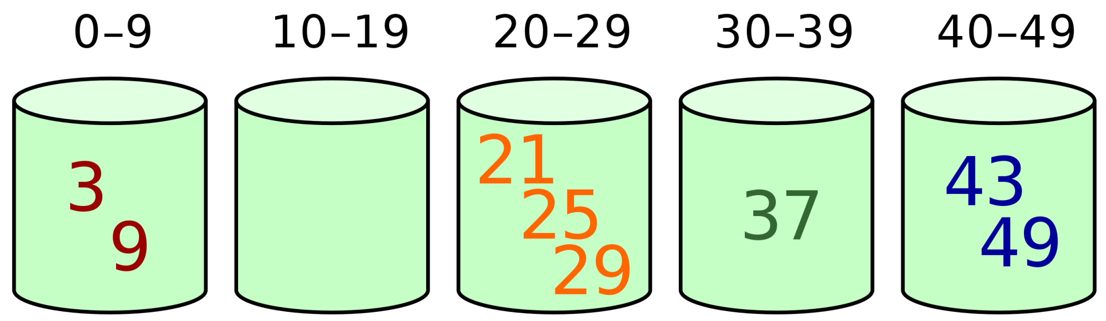
- Using Stirling's approximation for factorials: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$, we can approximate $\log_2(n!)$ to show the lower bound of comparisons. For simplicity, the key takeaway from applying Stirling's approximation is that $\log_2(n!)$ is $\Omega(n \log n)$.

Bucket Sort

29 25 3 49 9 37 21 43

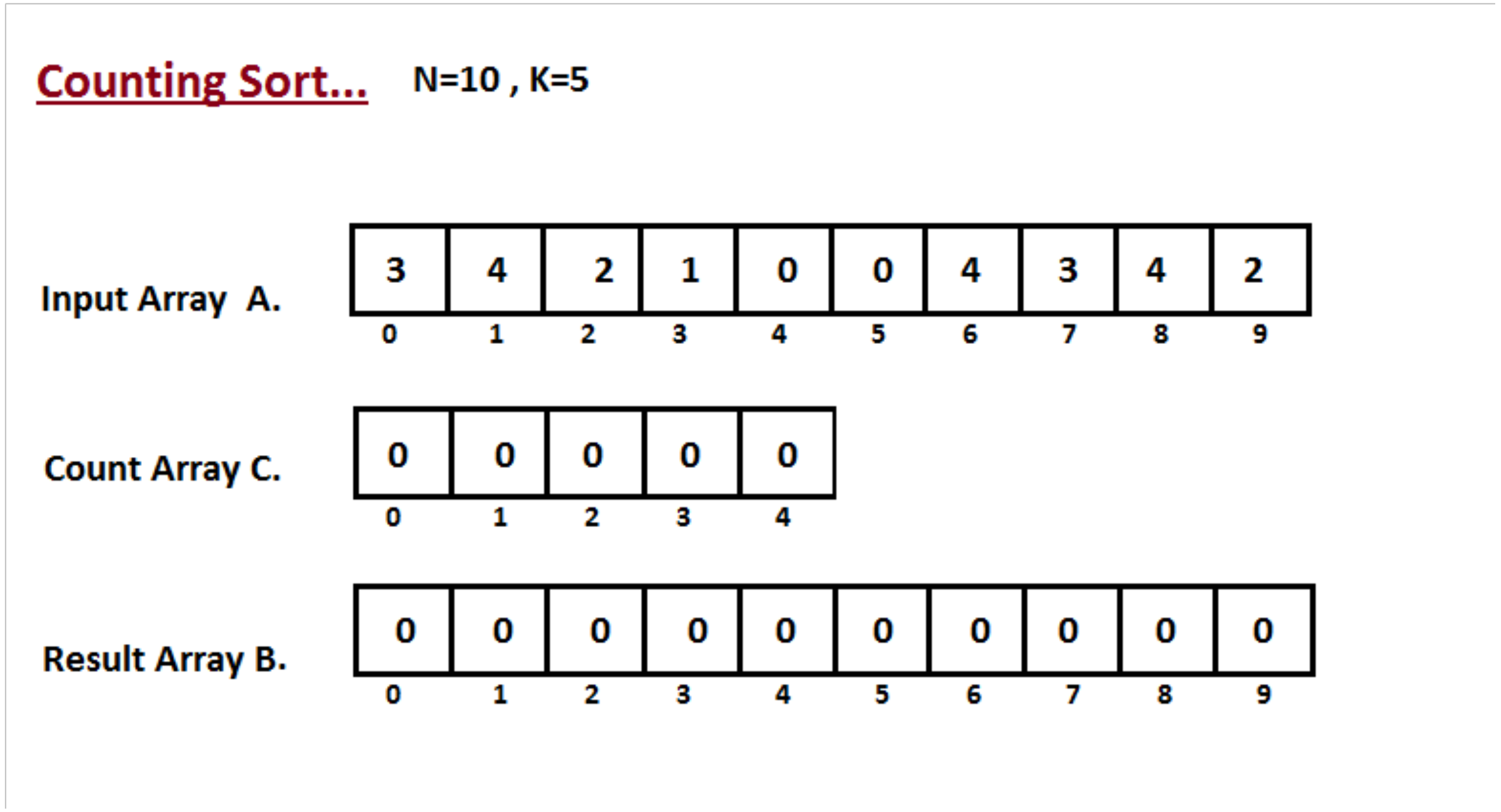


```
function bucketSort(array, bucketSize):
    // 1. Create buckets and distribute the elements of the array into
buckets
    buckets = create array of bucketSize lists
    for each element in array:
        index = determine the appropriate bucket for element
        add element to buckets[index]

    // 2. Sort individual buckets
    for each bucket in buckets:
        sort(bucket) // Use a different sorting algorithm

    // 3. Concatenate all buckets back into original array
    index = 0
    for each bucket in buckets:
        for each element in bucket:
            array[index++] = element
```



3 9 21 25 29 37 43 49

Counting Sort


Counting Sort... N=10 , K=5

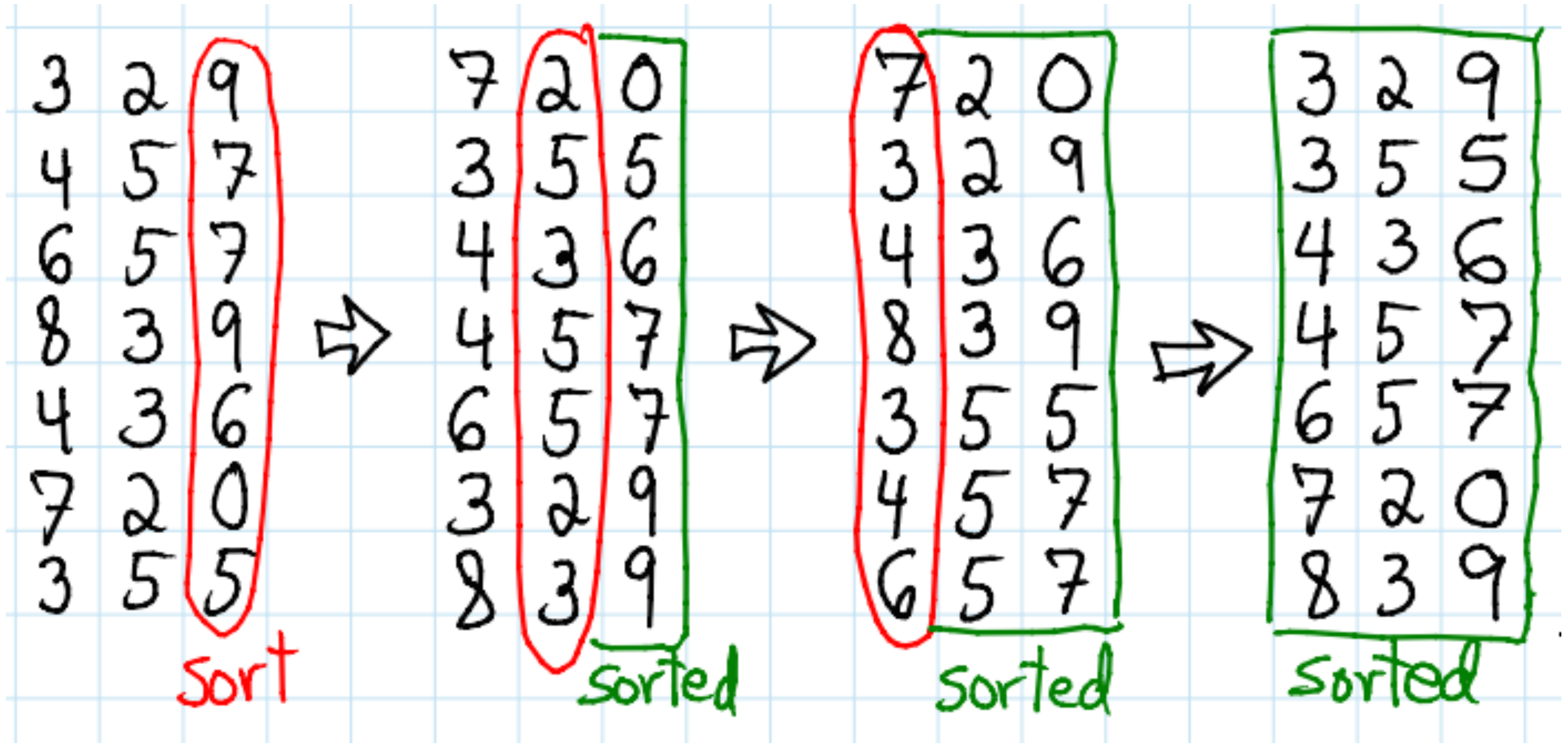
Input Array A.

Input Array A.	3	4	2	1	0	0	4	3	4	2
	0	1	2	3	4	5	6	7	8	9

Count Array C.	0	0	0	0	0
	0	1	2	3	4

Result Array B.	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Radix Sort


3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
Sort
7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9
sorted
7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7
sorted
3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9
sorted

- ***n***: The number of elements in the array.
- ***b***: The base of the numbering system. In the case of decimal numbers, b=10.
- ***d***: The number of digits in the maximum number in the array, which determines how many passes Radix Sort makes over the data.

Complexity: $O(d(n+b))$