UNIT-1
DIVIDE AND CONQUER

Dr. K. Raghava Rao
Professor inCSE,
Dept. of MCA, KL University
krraocse@gmail.com
http://mcadaa.blog.com

BASIC IDEA

- Pick one element in the array, which will be the *pivot.*
- Make one pass through the array, called a *partition* step, re-arranging the entries so that:
  - entries smaller than the pivot are to the left of the pivot.
  - entries larger than the pivot are to the right

BASIC IDEA

- **Recursively apply quicksort** to the part of the array that is to the left of the pivot, and to the part on its right.

- **No merge step**, at the end all the elements are in the proper order

CHOOSING THE PIVOT

Some fixed element: e.g. the first, the last, the one in the middle.

Bad choice - may turn to be the smallest or the largest element, then one of the partitions will be empty

Randomly chosen (by random generator) - still a **bad choice**

CHOOSING THE PIVOT

The **median of the array**

(if the array has N numbers, the median is the [N/2] largest number).

This is **difficult to compute** - increases the complexity.

CHOOSING THE PIVOT

The median-of-three choice:

take the first, the last and the middle element.

Choose the median of these three elements.

QUICK SORT

- Result:
  - All elements to the left of pivot are smaller or equal than pivot, and
  - All elements to the right of pivot are greater or equal than pivot
  - pivot in correct place in sorted array/list
- Need: Clever split procedure (Hoare)

QUICK SORT

Divide: Partition into subarrays (sub-lists)

Conquer: Recursively sort 2 subarrays

Combine: Trivial

QUICKSORT (HOARE 1962)

Problem: Sort n keys in nondecreasing order

Inputs: Positive integer n, array of keys S indexed from 1 to n

Output: The array S containing the keys in nondecreasing order.

quicksort (*low*, *high*)

1. if *high* > *low*
2. then partition(*low*, *high*, *pivotIndex*)
3. quicksort(*low*, *pivotIndex* -1)
4. quicksort(*pivotIndex* +1, *high*)

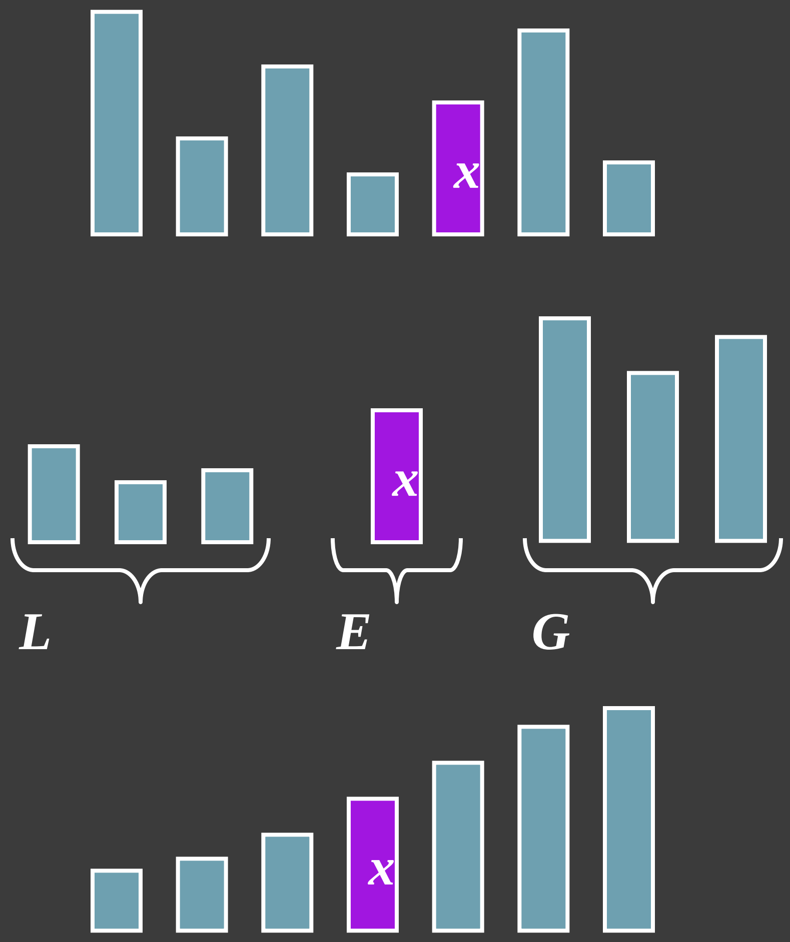
PARTITION ARRAY FOR QUICKSORT

partition (*low*, *high*, *pivot*)

1. *pivotitem* = *S* [*low*]
2. *k* = *low*
3. for *j* = *low* +1 to *high*
4. do if *S* [*j*] < *pivotitem*
5. then *k* = *k* + 1
6. exchange *S* [*j*] and *S* [*k*]
7. *pivot* = *k*
8. exchange *S*[*low*] and *S*[*pivot*]

QUICK-SORT

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element ***x*** (called pivot) and partition ***S*** into
    - ***L*** elements less than ***x***
    - ***E*** elements equal ***x***
    - ***G*** elements greater than ***x***
  - Recur: sort ***L*** and ***G***
  - Conquer: join ***L***, ***E*** and ***G***

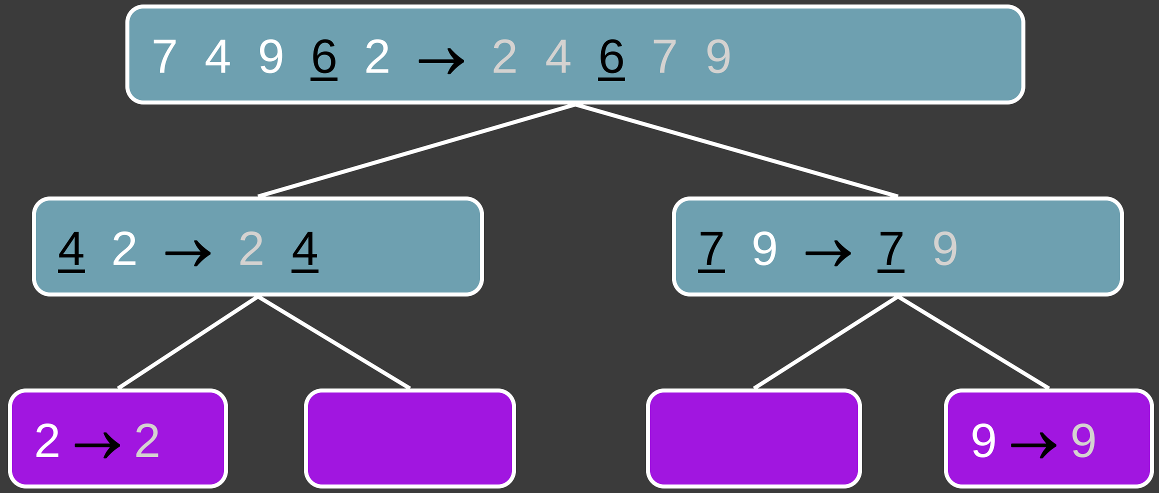


PARTITION

- We partition an input sequence as follows:
  - We remove, in turn, each element $\boldsymbol{y}$ from $\boldsymbol{S}$ and
  - We insert $\boldsymbol{y}$ into $\boldsymbol{L}$, $\boldsymbol{E}$ or $\boldsymbol{G}$, depending on the result of the comparison with the pivot $\boldsymbol{x}$
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $\boldsymbol{O}(1)$ time
- Thus, the partition step of quick-sort takes $\boldsymbol{O}(\boldsymbol{n})$ time

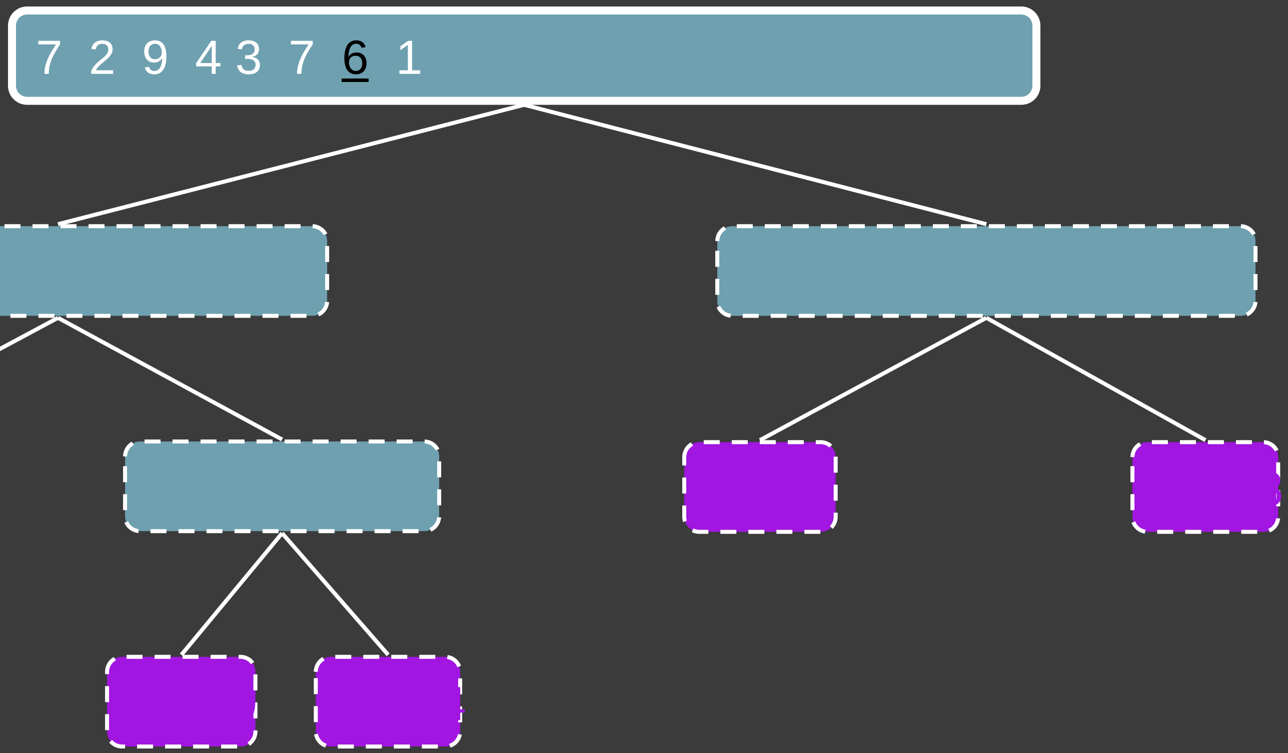
QUICK-SORT TREE

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



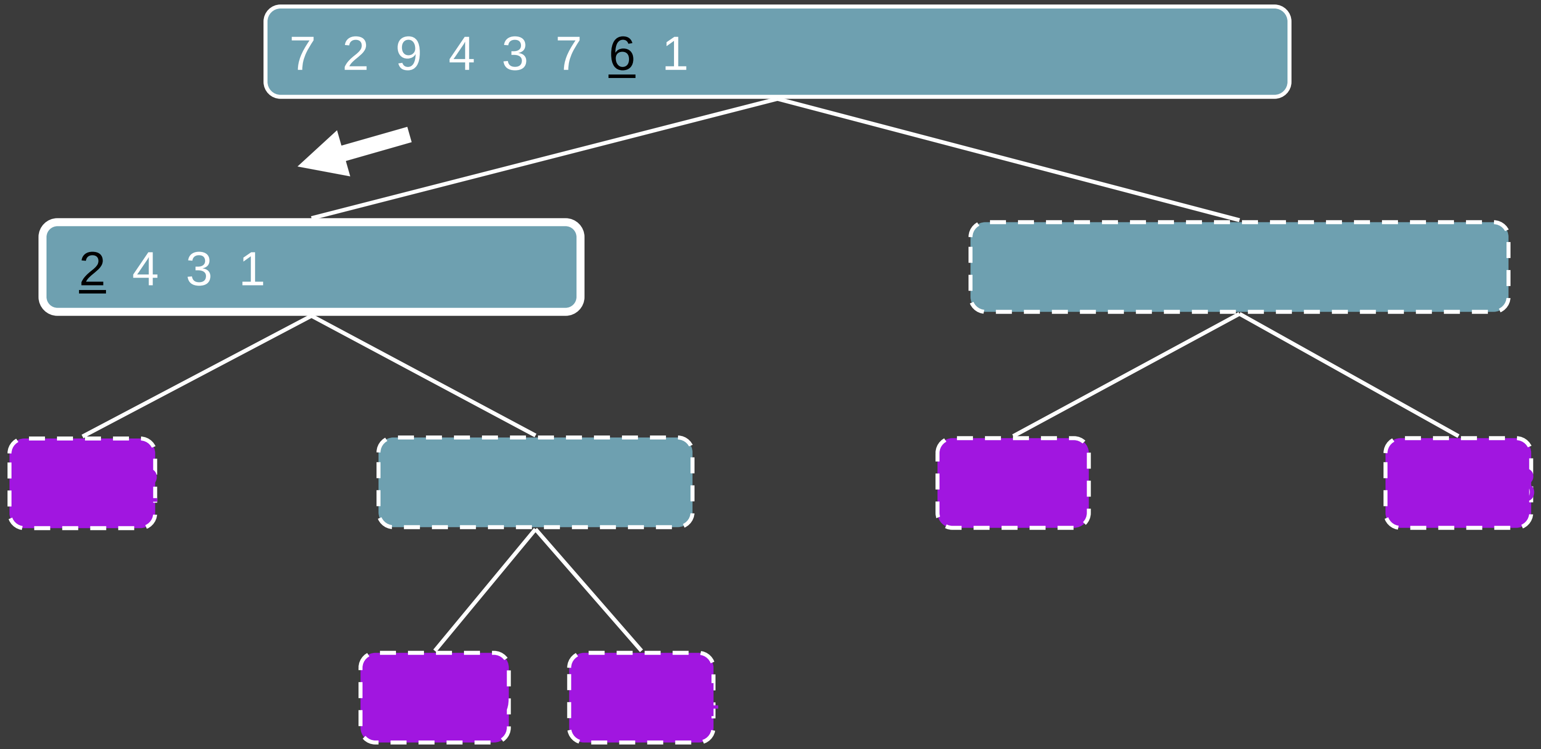
EXECUTION EXAMPLE

- Pivot selection



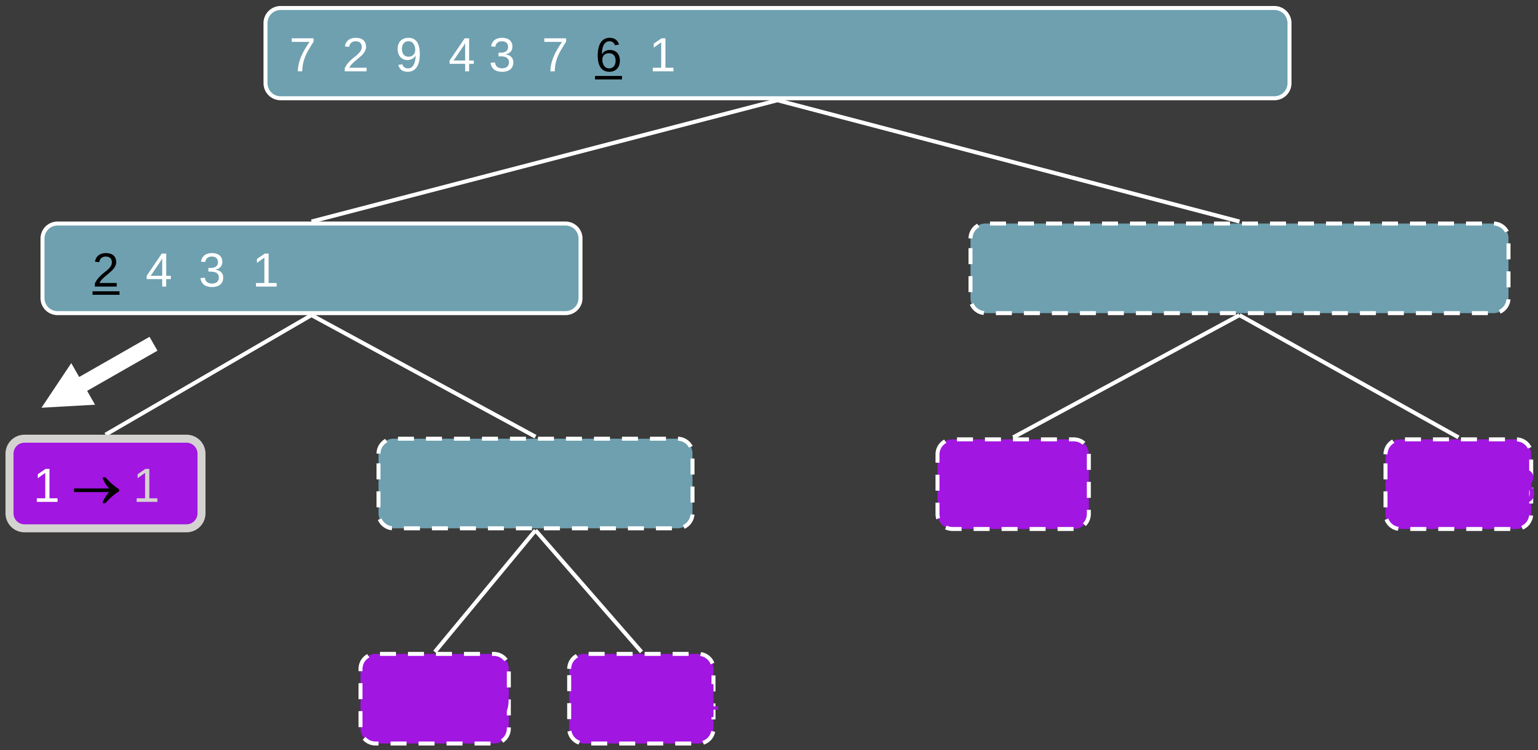
EXECUTION EXAMPLE (CONT.)

- Partition, recursive call, pivot selection



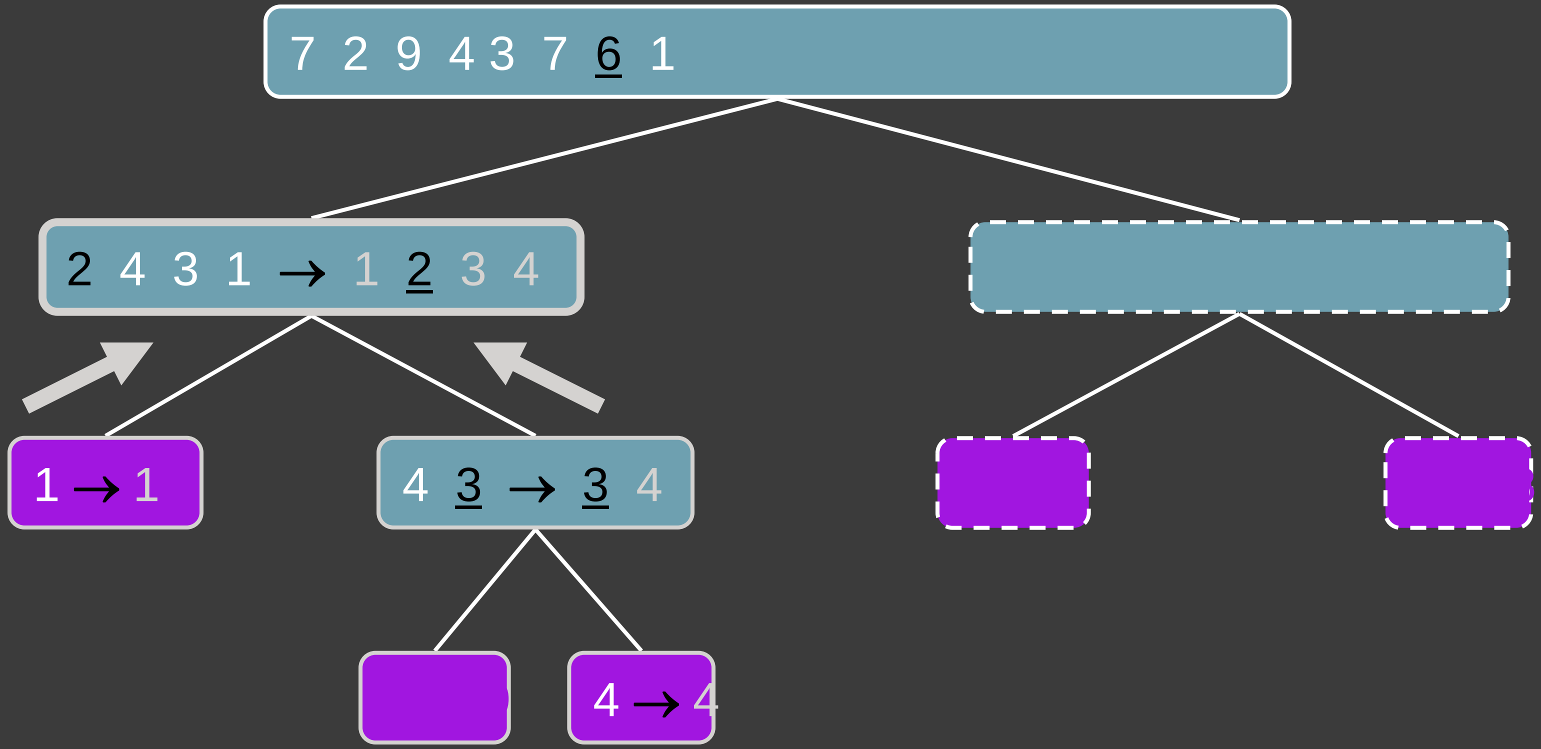
EXECUTION EXAMPLE (CONT.)

- Partition, recursive call, base case



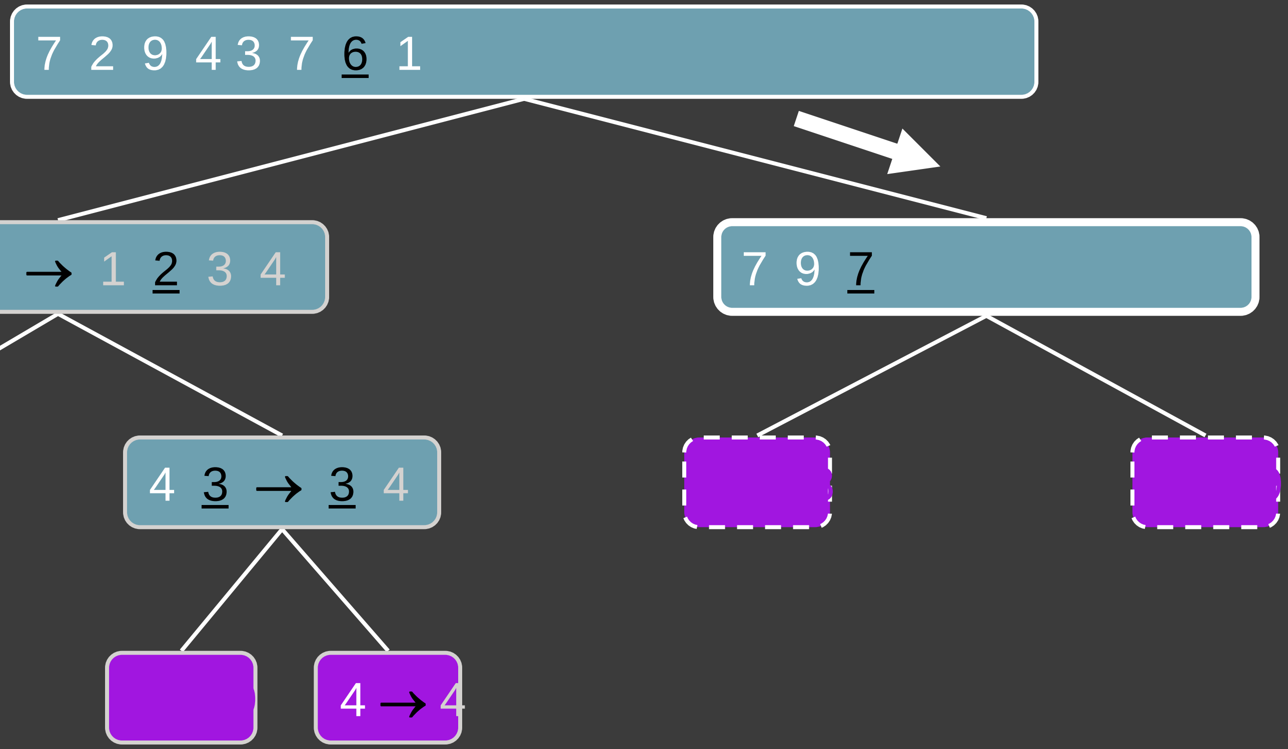
EXECUTION EXAMPLE (CONT.)

- Recursive call, …, base case, join



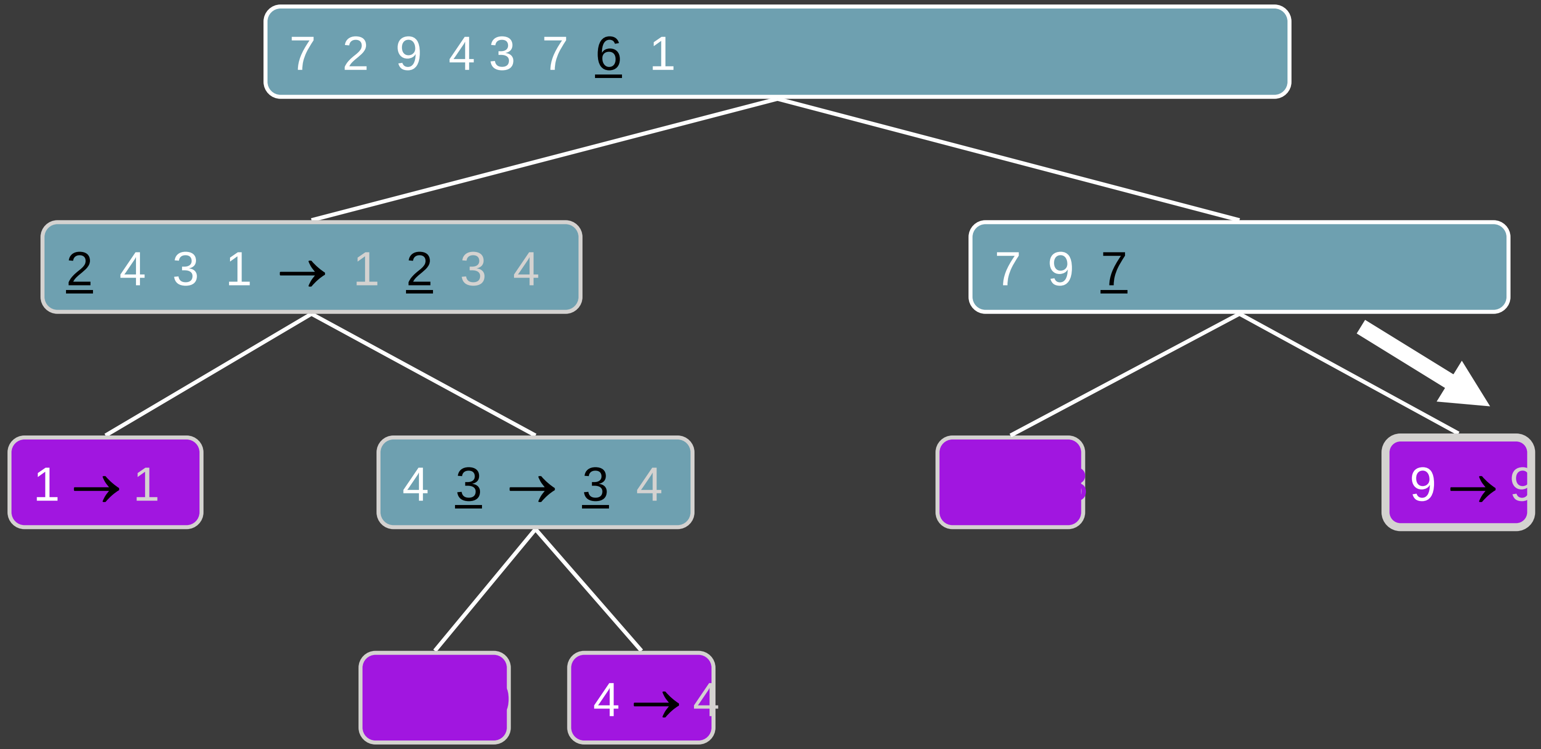
EXECUTION EXAMPLE (CONT.)

- Recursive call, pivot selection



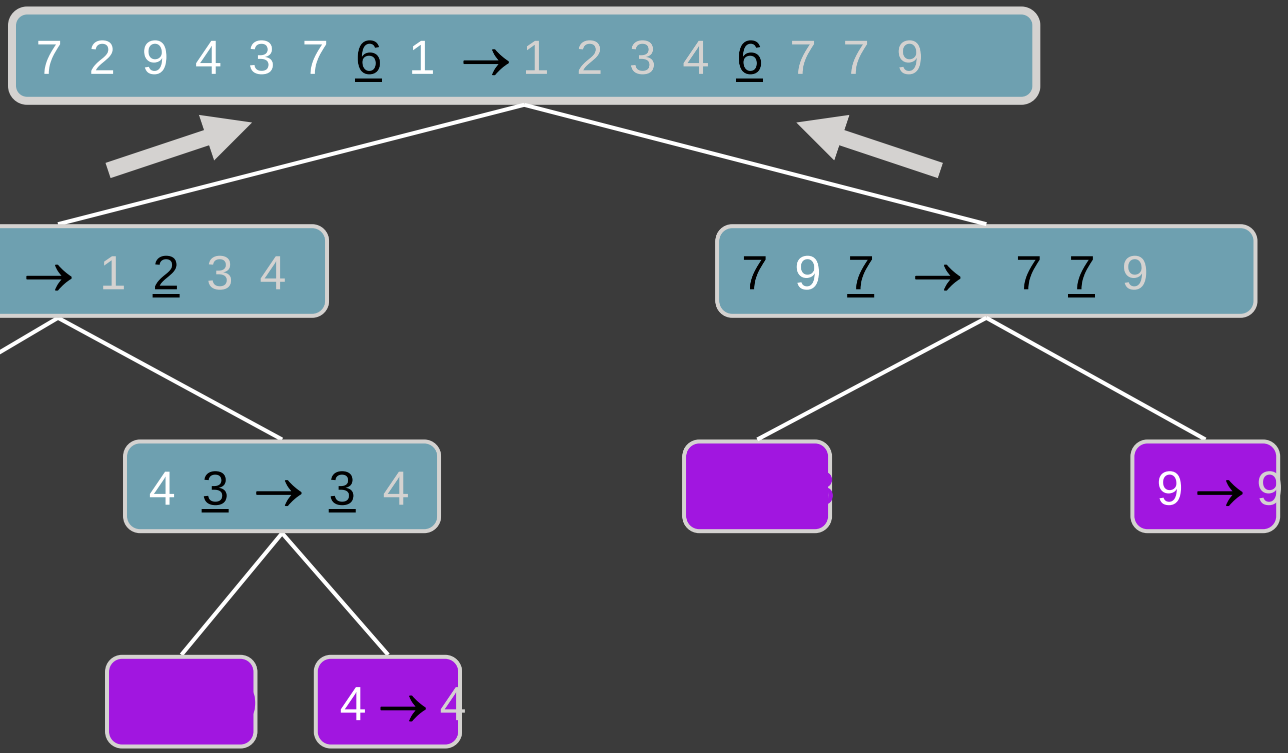
EXECUTION EXAMPLE (CONT.)

- Partition, …, recursive call, base case



EXECUTION EXAMPLE (CONT.)

- Join, join



EXAMPLE

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

PICK PIVOT ELEMENT

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

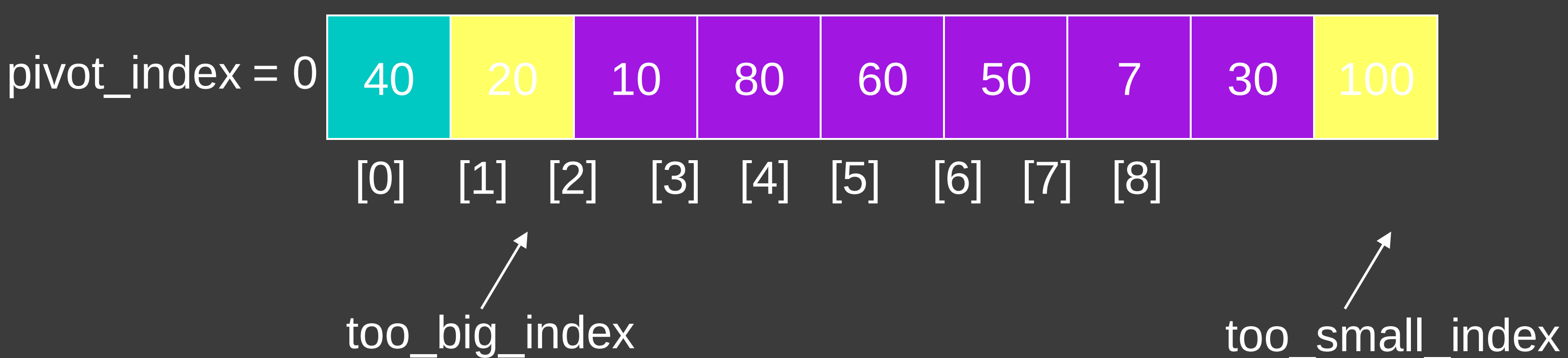
PARTITIONING ARRAY

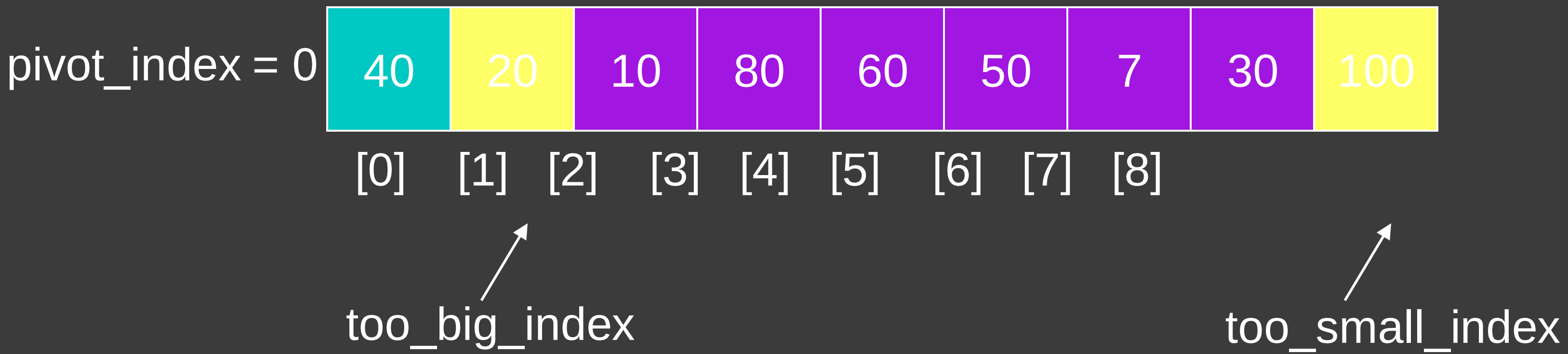
Given a pivot, partition the elements of the array such that the resulting array consists of:

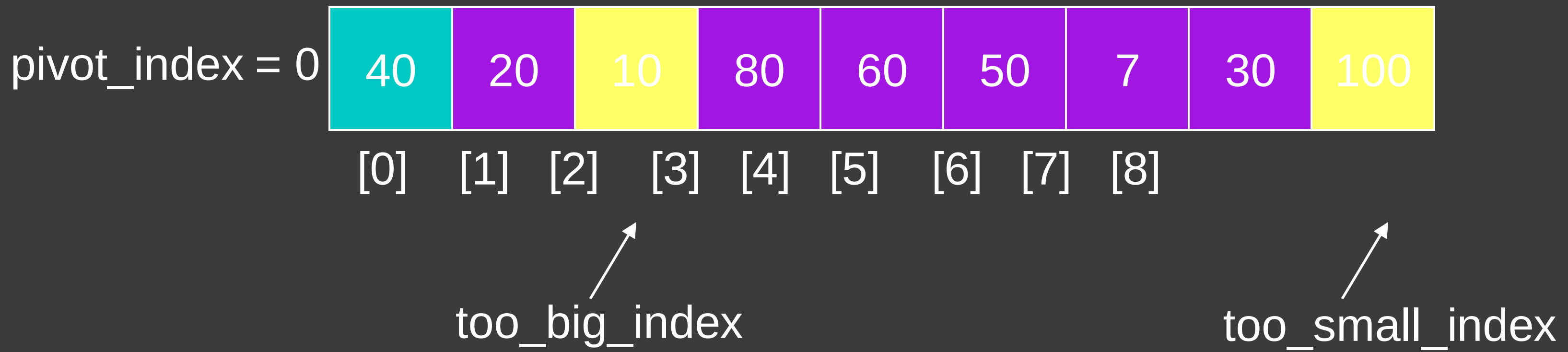
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

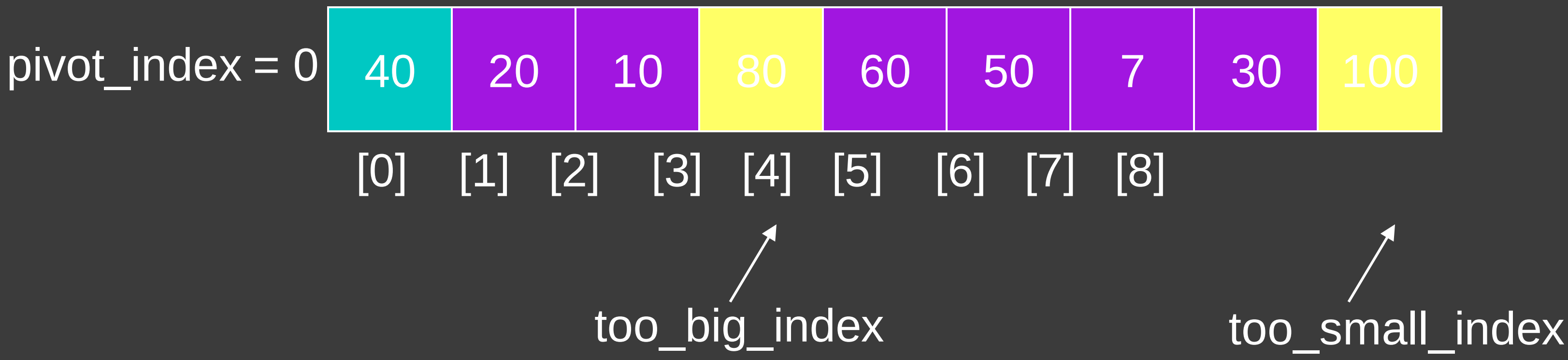
The sub-arrays are stored in the original data array.

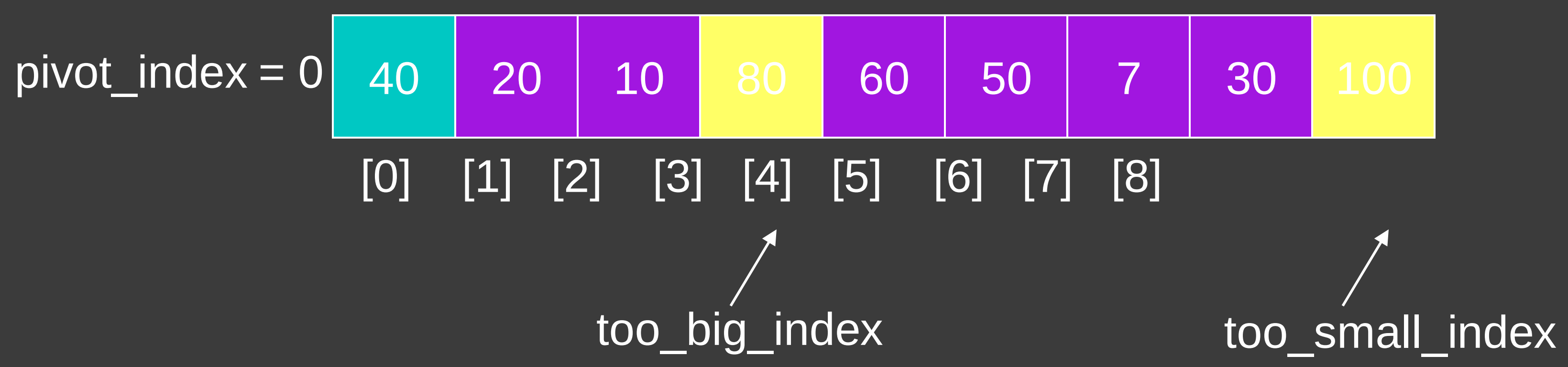
Partitioning loops through, swapping elements below/above pivot.

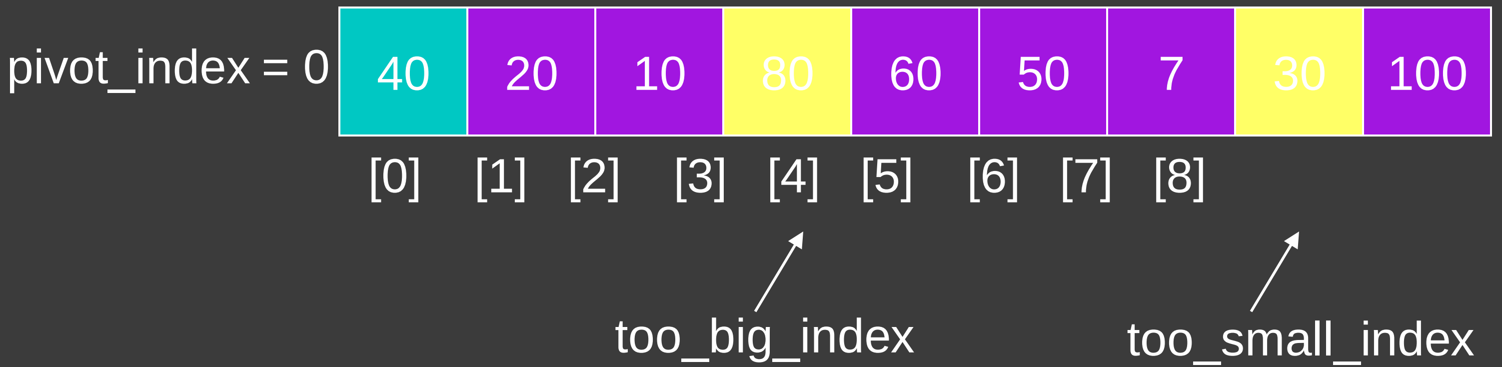

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

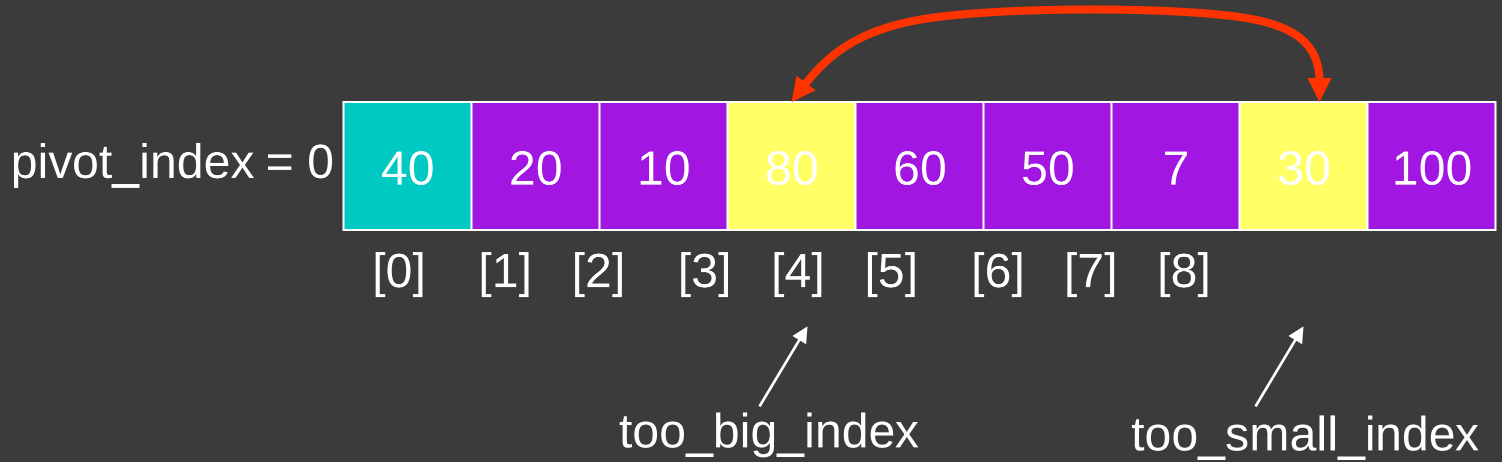

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

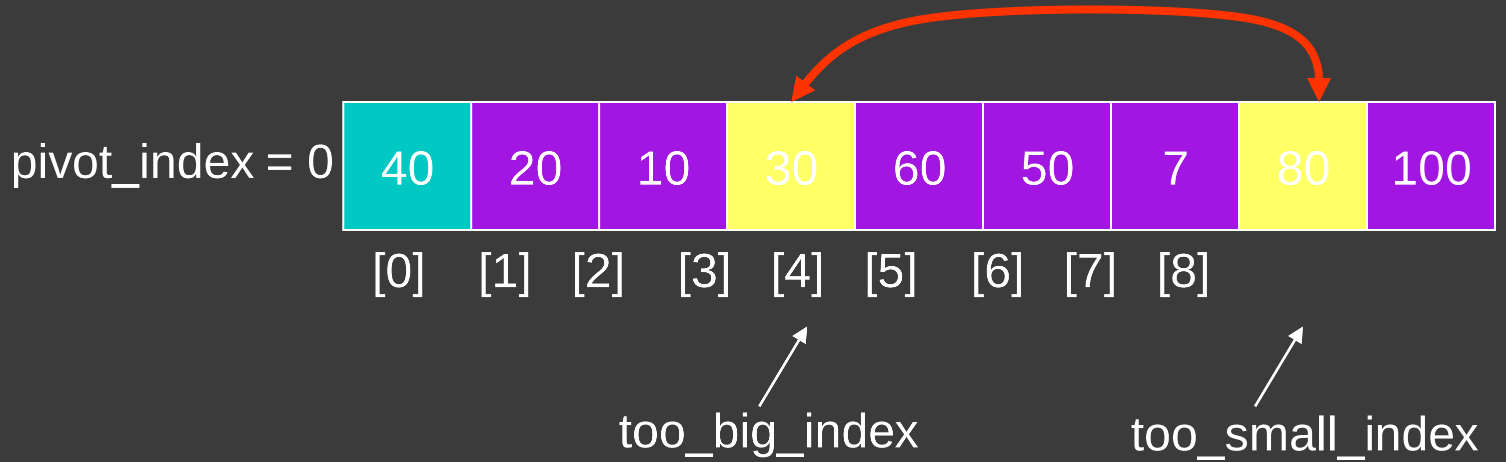

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

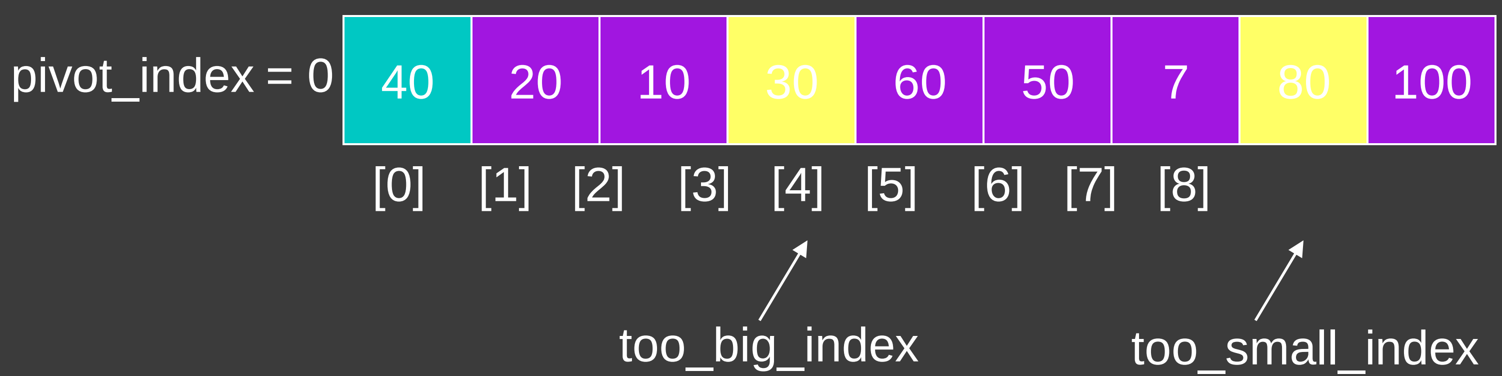

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

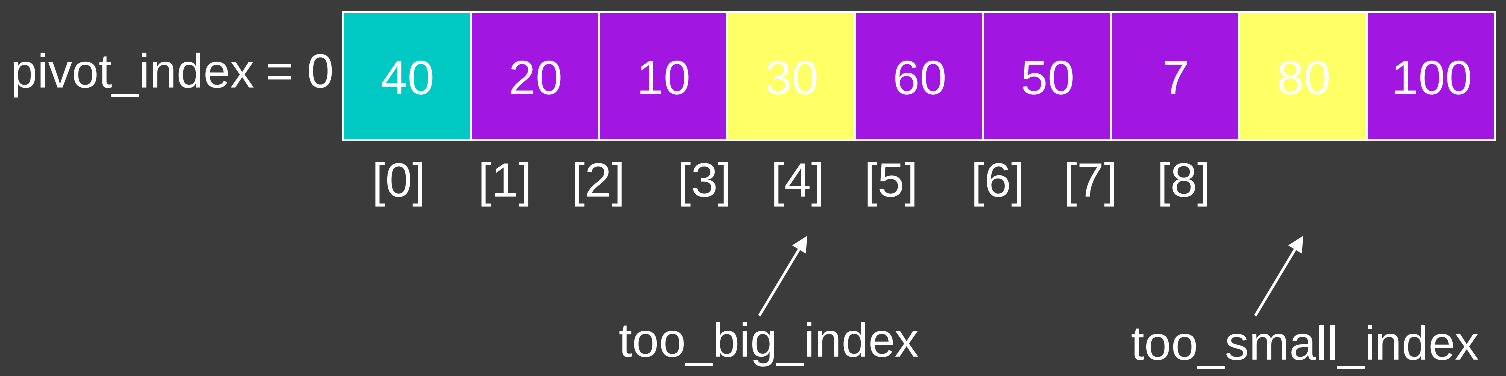

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

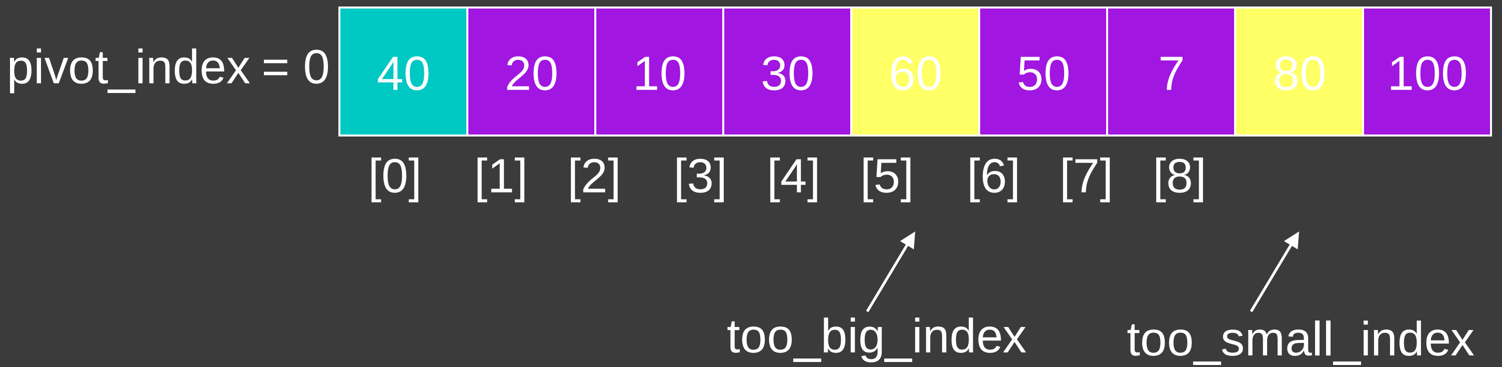

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

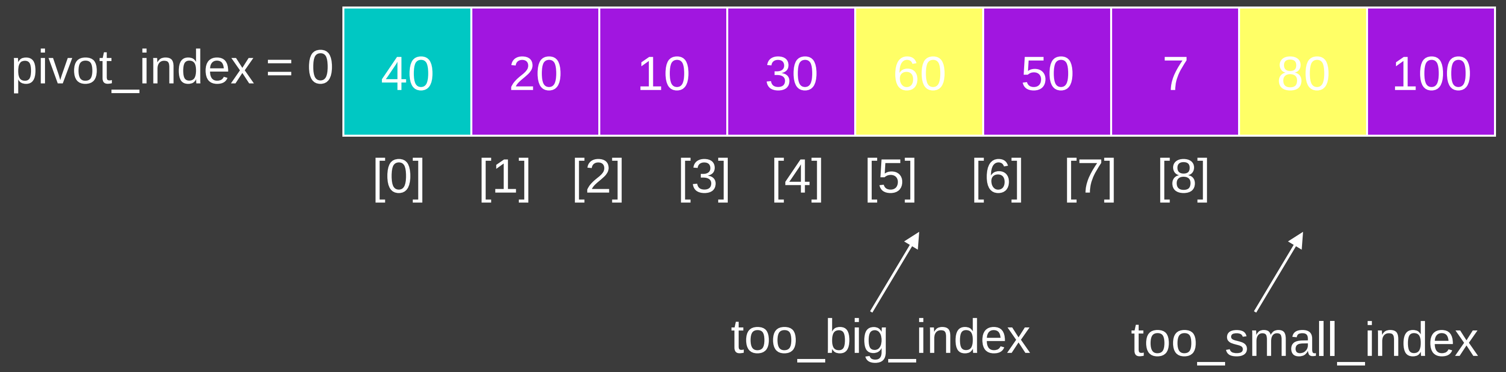

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

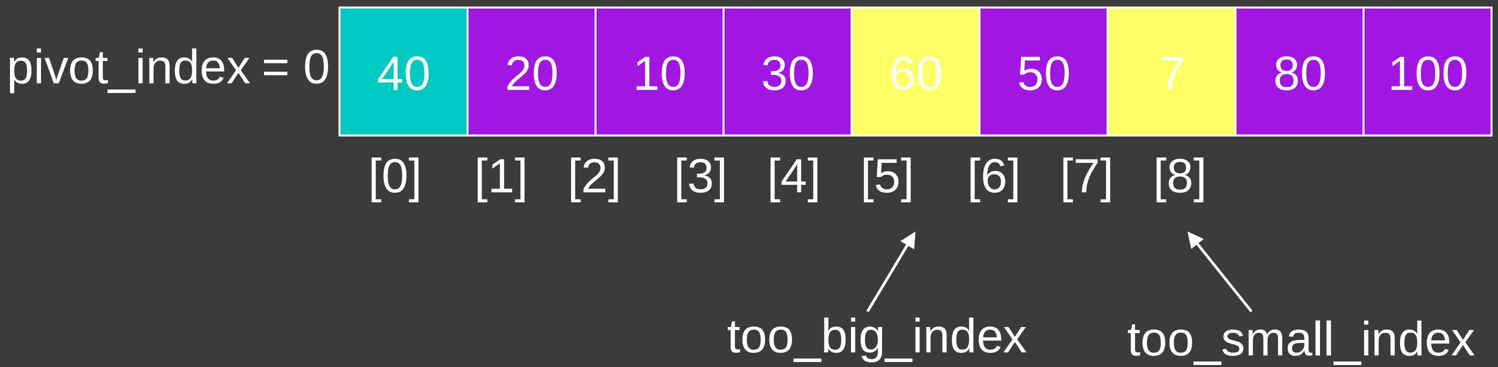

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

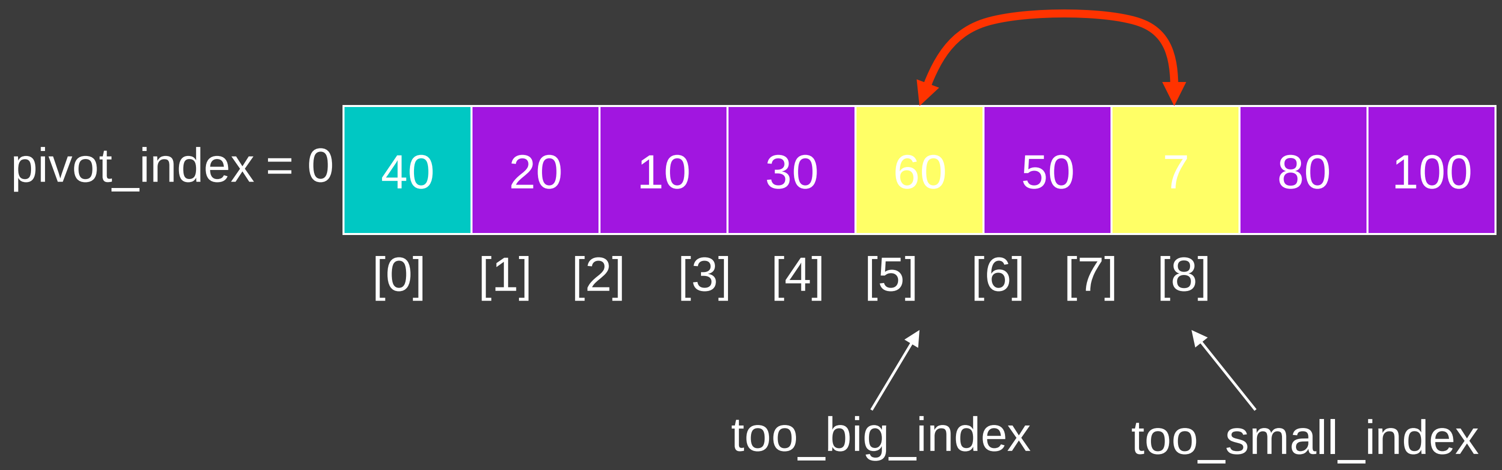

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

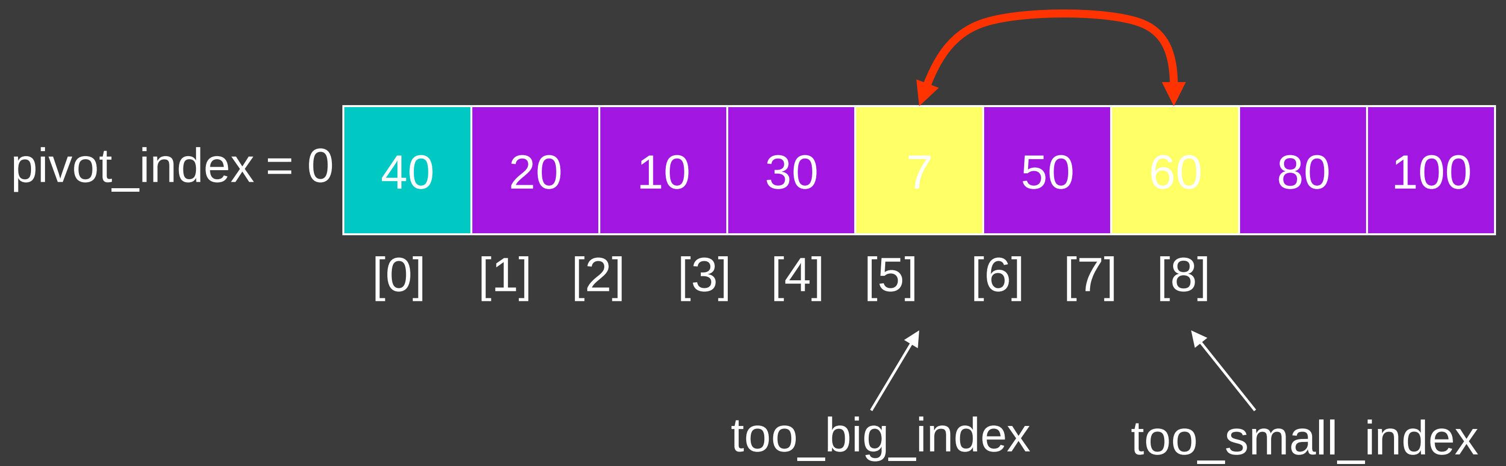

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

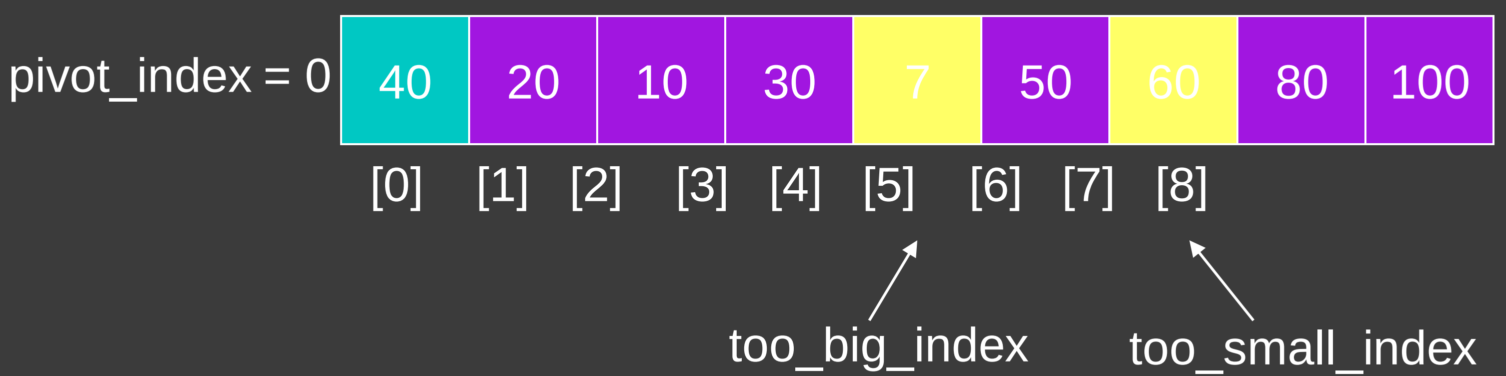

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

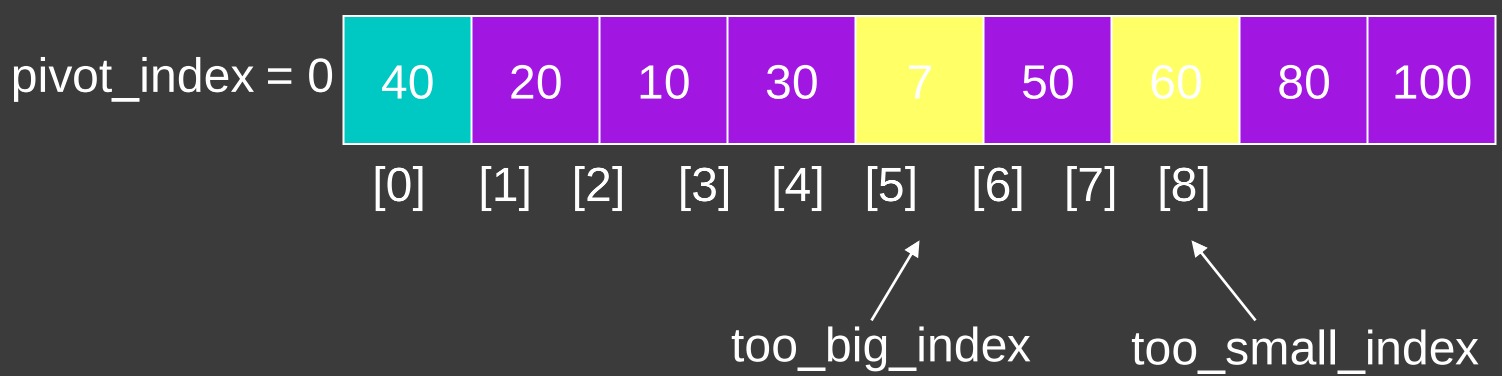

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

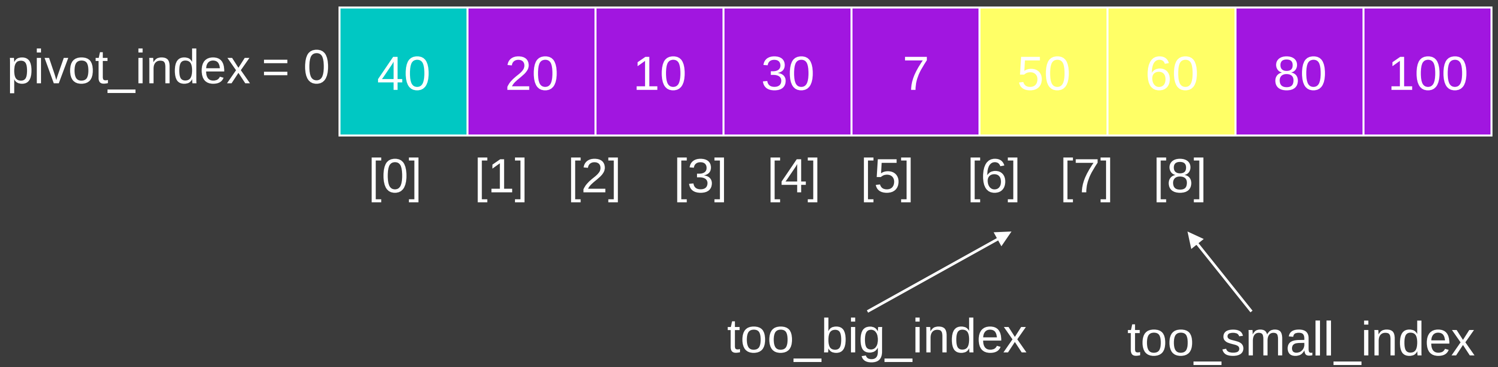

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

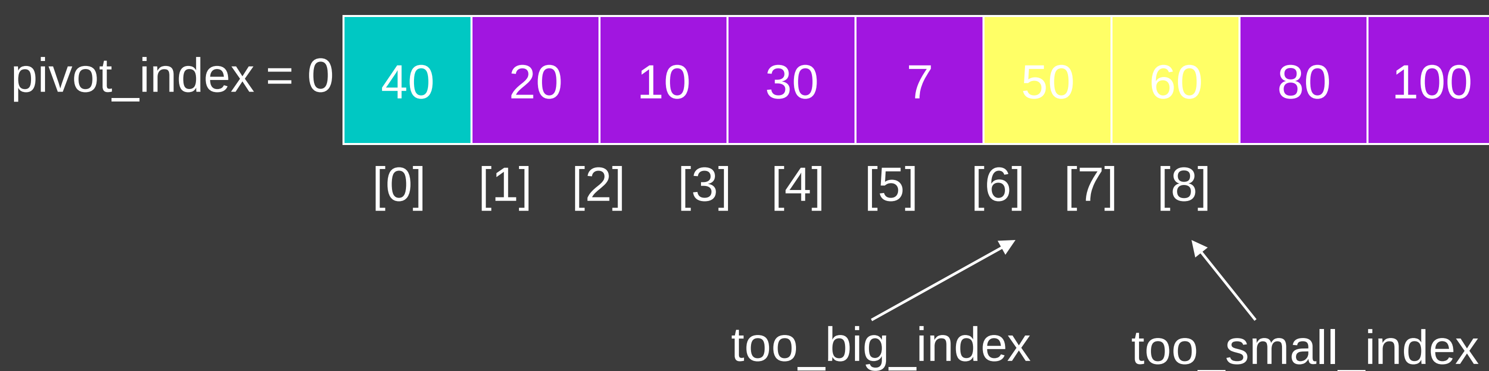

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

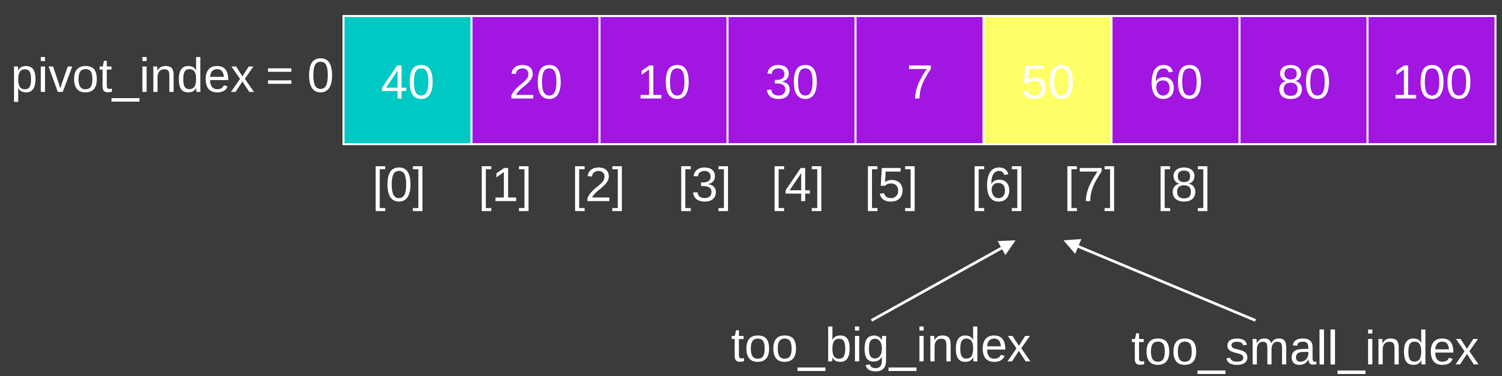

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

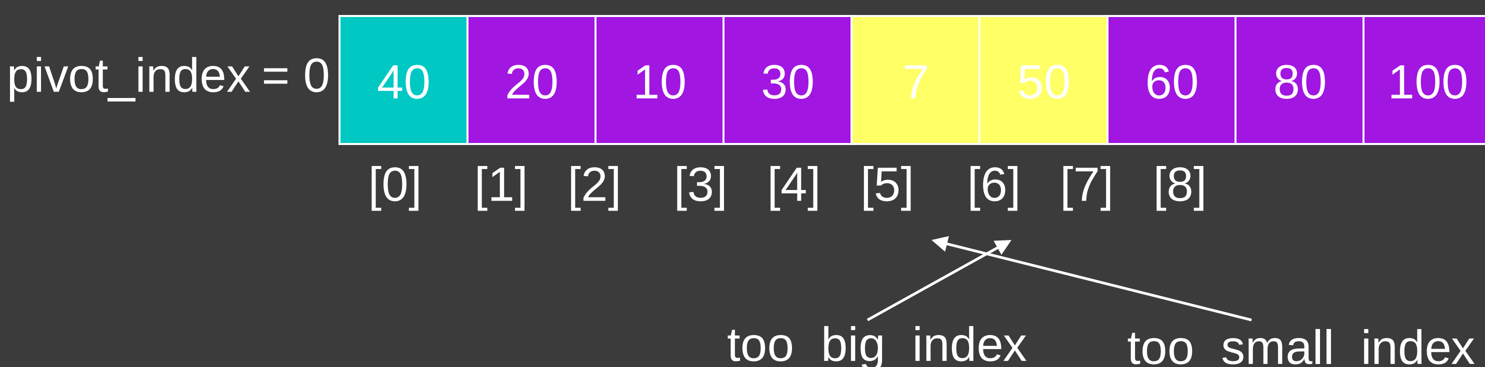

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

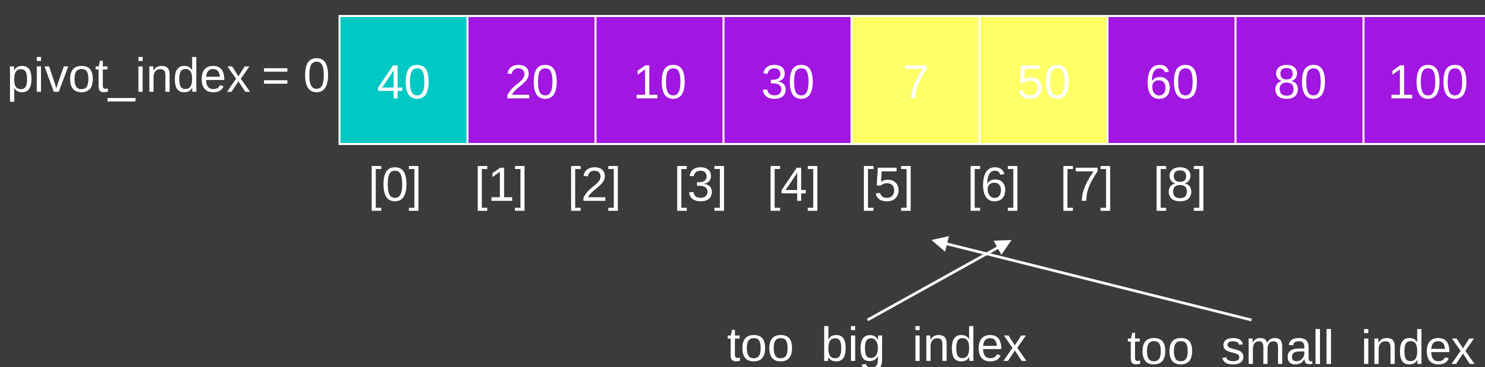

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

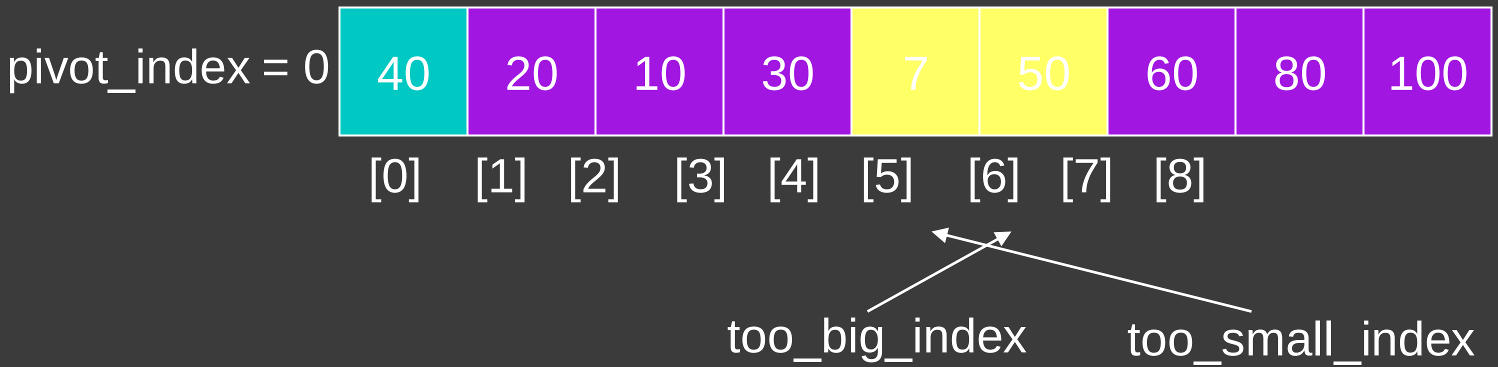

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

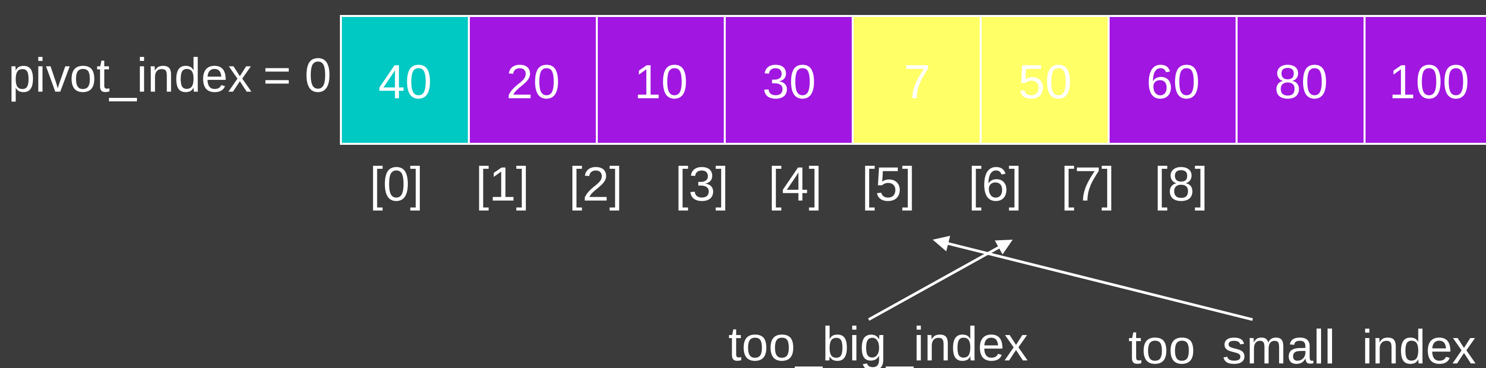

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

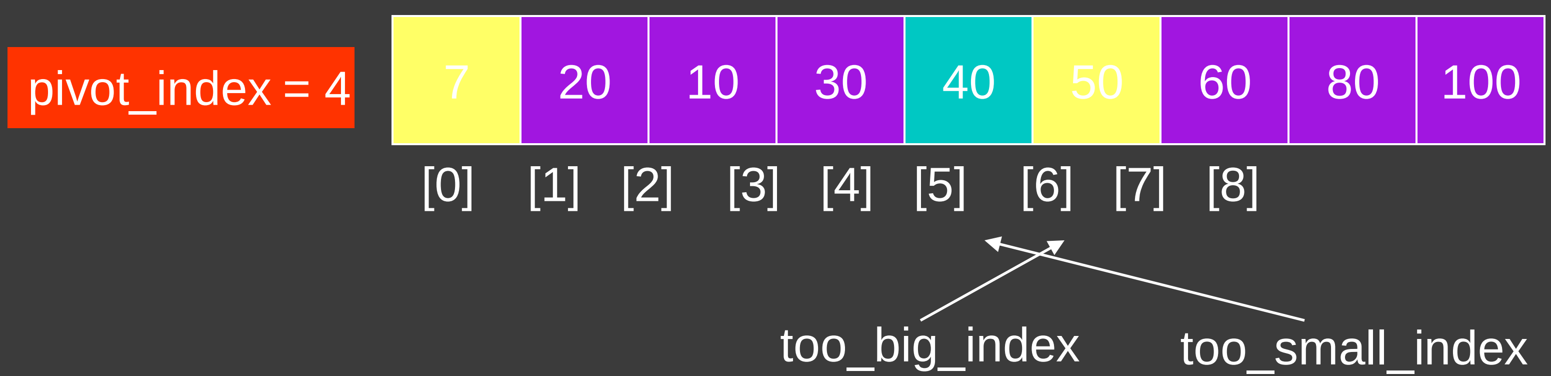

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index


pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

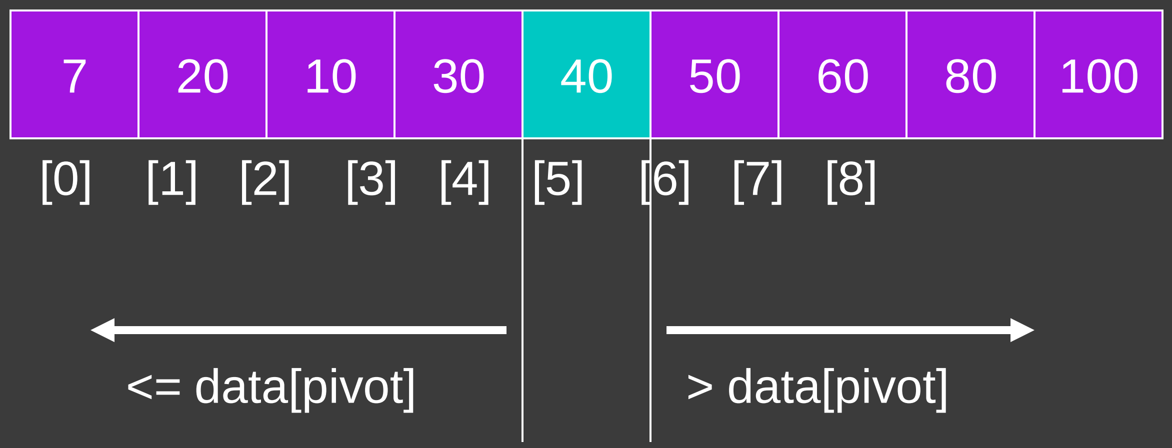

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index


pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

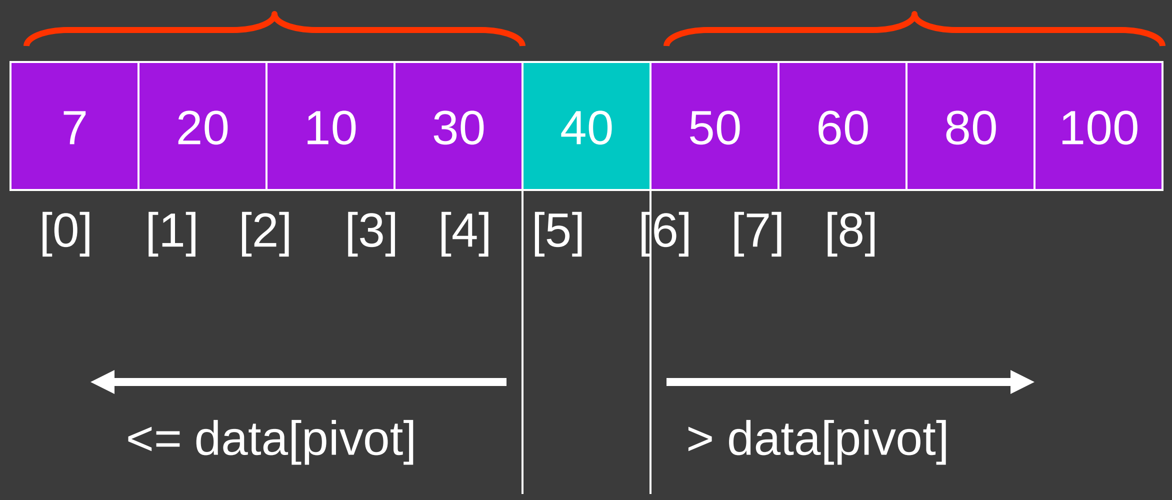

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index


pivot_index = 4
7
20
10
30
40
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

PARTITION RESULT

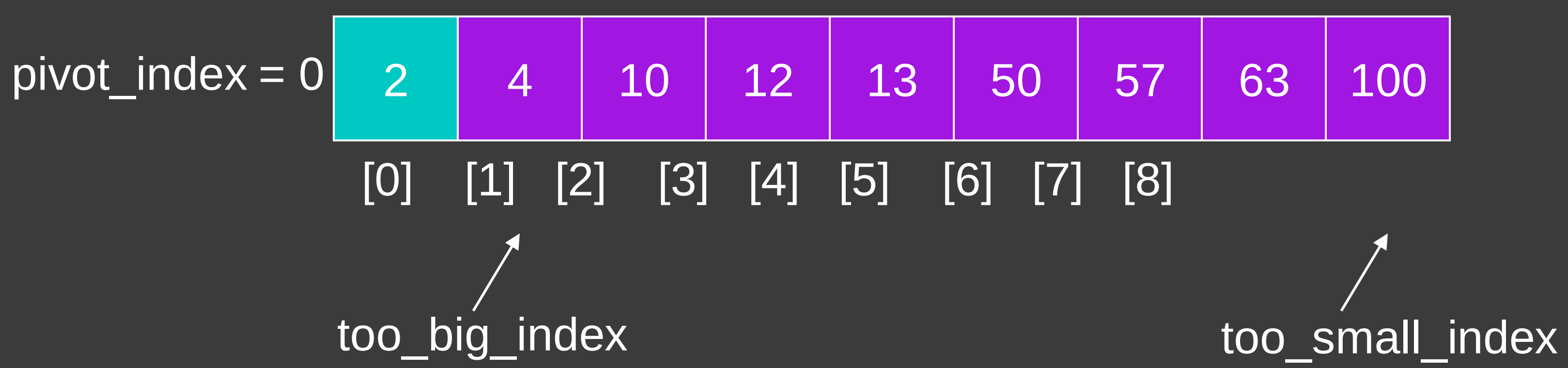


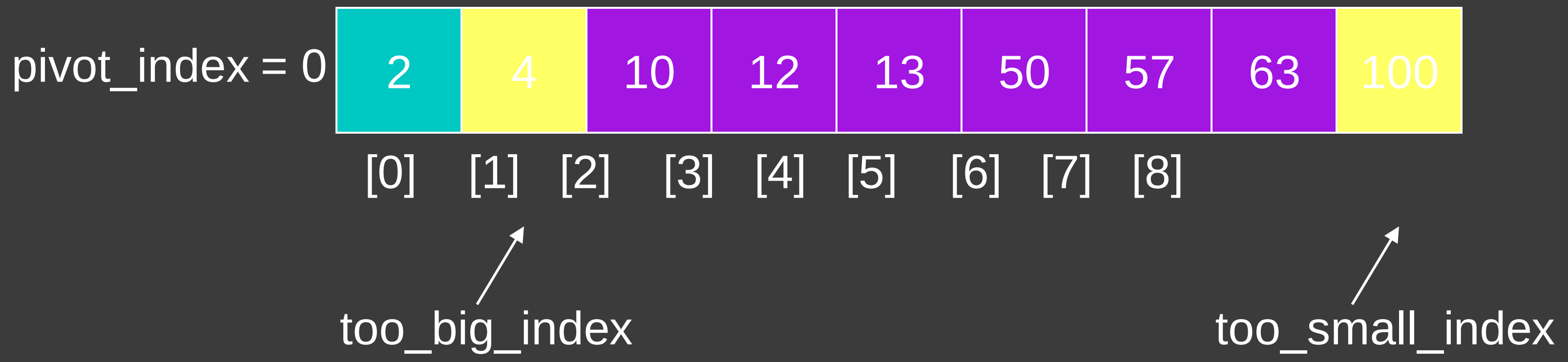
RECURSION: QUICKSORT SUB-ARRAYS

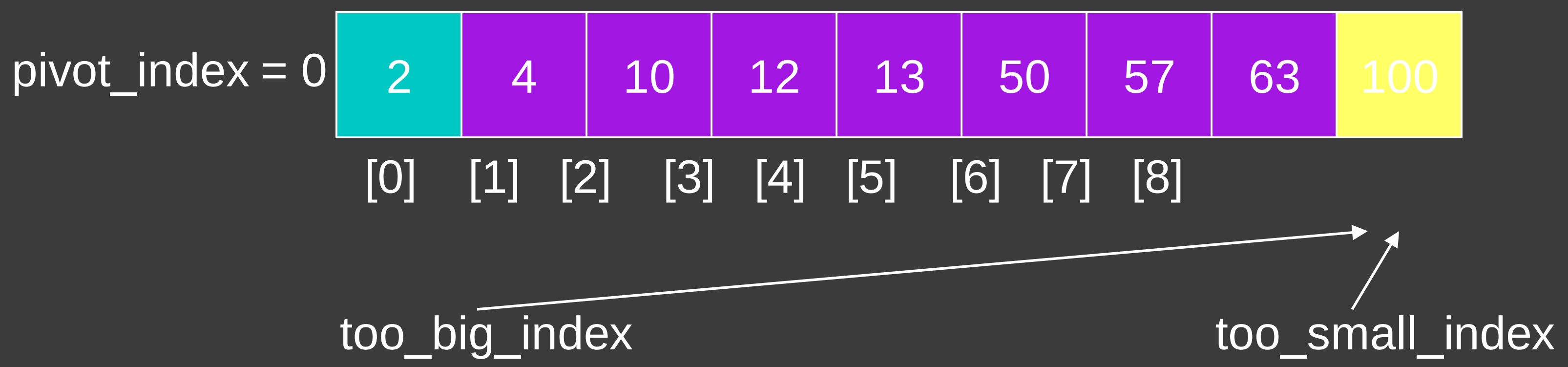


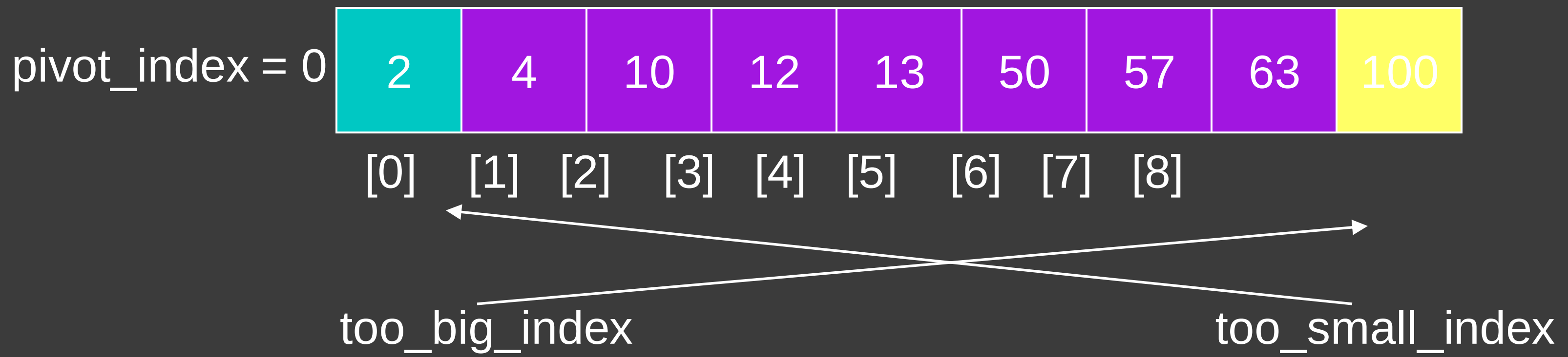
QUICKSORT: WORST CASE

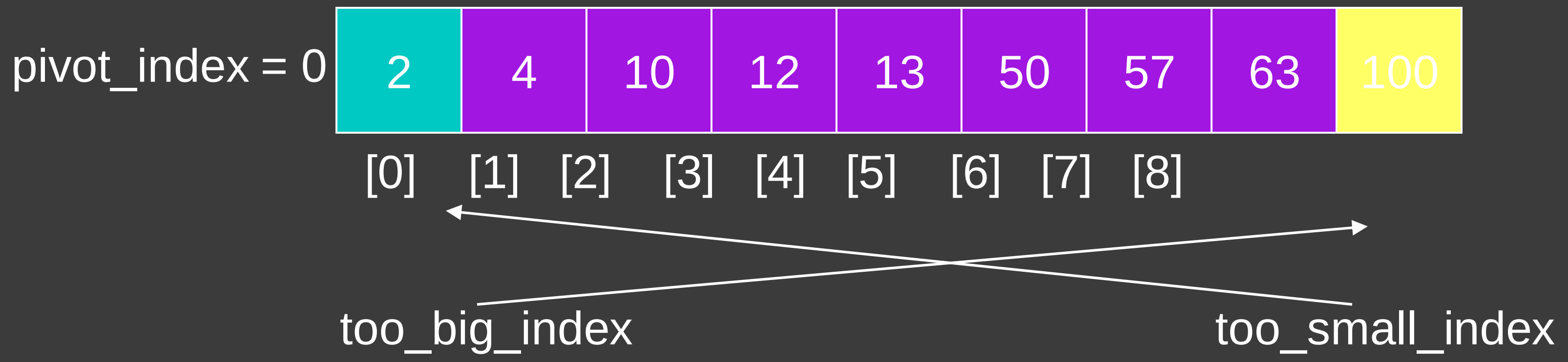
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



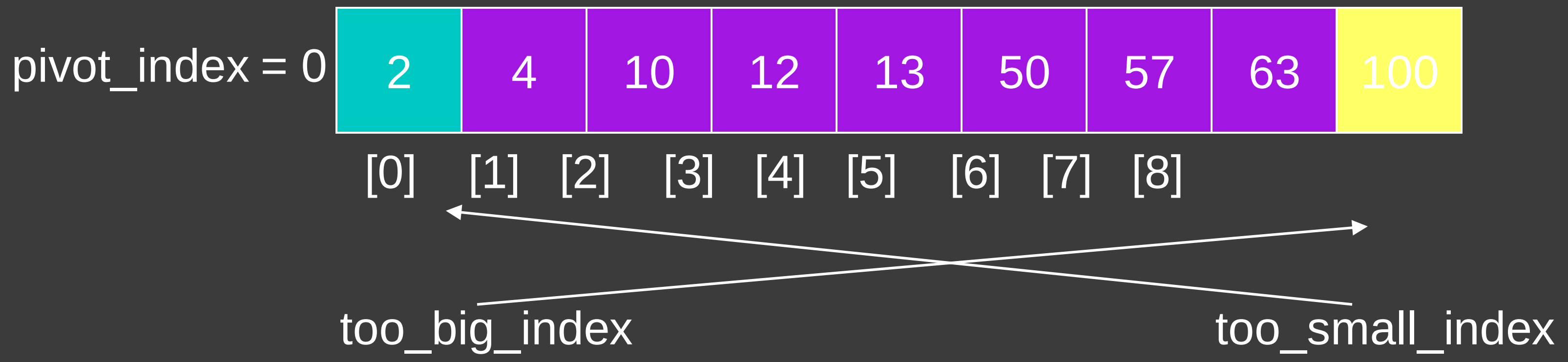

pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

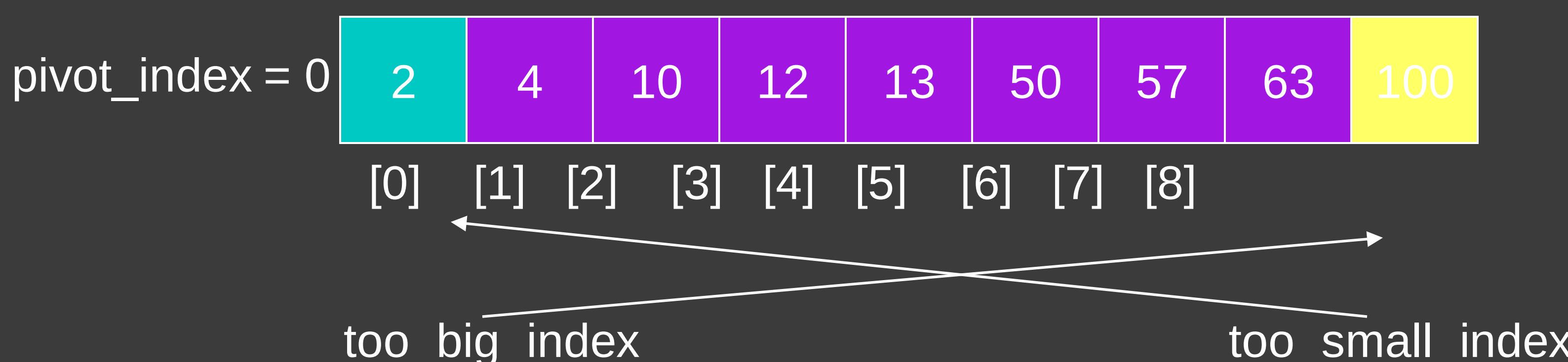

pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

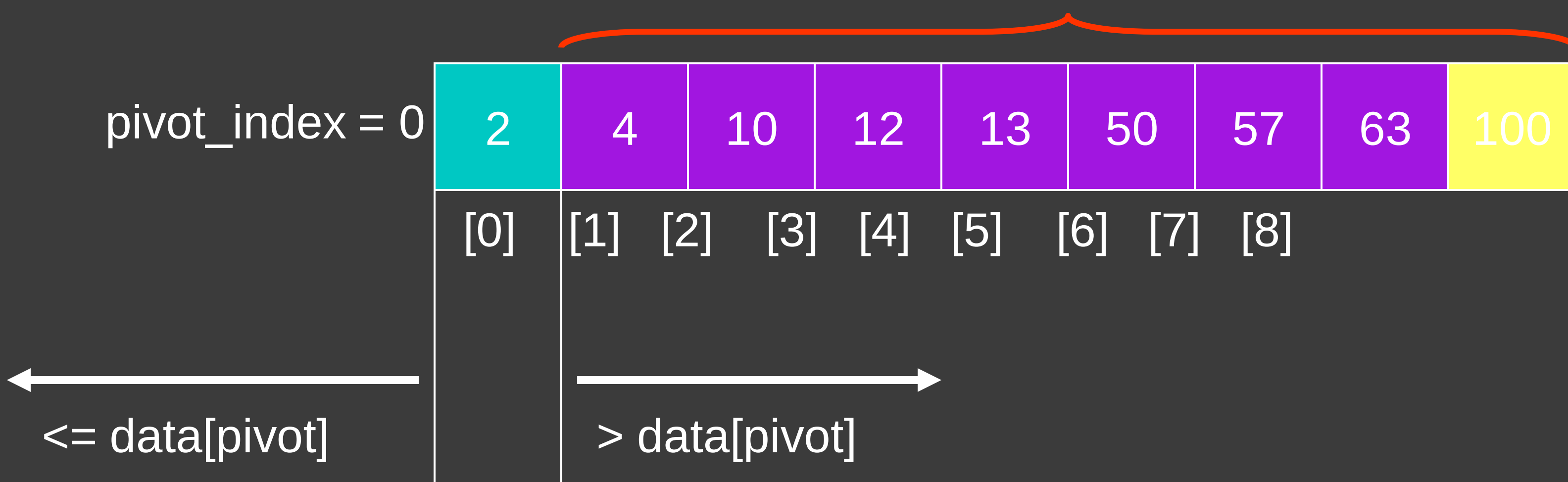

pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index



pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index


pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index


pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index


pivot_index = 0
2
4
10
12
13
50
57
63
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
<= data[pivot]
> data[pivot]

COMPLEXITY OF QUICK SORT

If we have an array of equal elements, the array index will

never increment i or decrement j,

and will do infinite swaps.

i and j will never cross.

COMPLEXITY OF QUICK SORT

Worst Case: $O(N^2)$

This happens when the pivot is the smallest (or the largest) element.

Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

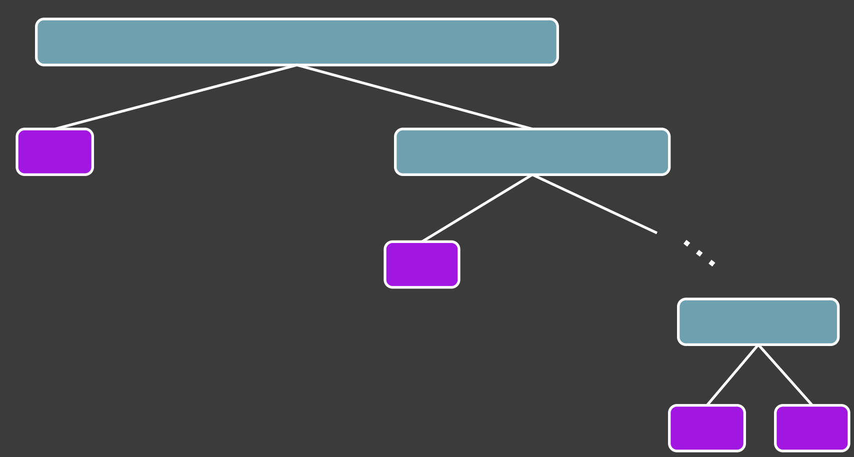
WORST-CASE RUNNING TIME

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



WORST-CASE ANALYSIS

The pivot is the smallest (or the largest) element

T(N) = T(N-1) + cN, N > 1

Telescoping:

$$T(N-1) = T(N-2) + c(N-1)$$

$$T(N-2) = T(N-3) + c(N-2)$$

$$T(N-3) = T(N-4) + c(N-3)$$

…………...

$$T(2) = T(1) + c.2$$

WORST-CASE ANALYSIS

$T(N) + T(N-1) + T(N-2) + \ldots + T(2) =$

$= T(N-1) + T(N-2) + \ldots + T(2) + T(1) +$

$c(N) + c(N-1) + c(N-2) + \ldots + c.2$

$T(N) = T(1) +$

c times (the sum of 2 thru N)

$= T(1) + c\,(N\,(N+1) / 2 - 1) =$ **$O(N^2)$**

COMPLEXITY OF QUICK SORT

Average-case O(N logN)

Best-case O(NlogN)

The pivot is the median of the array, the left and the right parts have same size. There are **logN** partitions, and to obtain each partitions we do **N** comparisons (and not more than **N/2** swaps). Hence the complexity is **O(NlogN)**

BEST CASE ANALYSIS

$T(N) = T(i) + T(N - i - 1) + cN$

The time to sort the file is equal to

- the time to sort the left partition with **i** elements, plus
- the time to sort the right partition with **N-i-1** elements, plus
- the time to build the partitions.

BEST-CASE ANALYSIS

The pivot is in the middle

$$T(N) = 2T(N/2) + cN$$

Divide by N:

$$T(N) / N = T(N/2) / (N/2) + c$$

BEST-CASE ANALYSIS

Telescoping:

$T(N) / N = T(N/2) / (N/2) + c$

$T(N/2) / (N/2) = T(N/4) / (N/4) + c$

$T(N/4) / (N/4) = T(N/8) / (N/8) + c$

……

$T(2) / 2 = T(1) / (1) + c$

BEST-CASE ANALYSIS

Add all equations:

$T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + ... + T(2) / 2 =$

$= \quad (N/2) / (N/2) + T(N/4) / (N/4) + ... + T(1) / (1) + c.\log N$

After crossing the equal terms:

$T(N)/N = T(1) + c*\text{Log}N$

$T(N) = N + N*c*\text{Log}N =$ **$O(N\log N)$**

ADVANTAGES AND DISADVANTAGES

- **Advantages:**
  - **One of the fastest algorithms on average**
  - **Does not need additional memory (the sorting takes place in the array - this is called in-place processing)**
- **Disadvantages:**
  - **The worst-case complexity is $O(N^2)$**

APPLICATIONS

Commercial applications

- QuickSort generally runs fast
- No additional memory
- The above advantages compensate for the rare occasions when it runs with $O(N^2)$

EXERCISE

- Write quicksort tracing
- 26,5,37,1,61,11,59,15,48,19