

CSE214 – Analysis of Algorithms

PhD Furkan Gözükkara, Toros
University

<https://github.com/FurkanGozukara/CSE214> 2018

Lecture 5

Quicksort

Based on Kruse's and Ryba's Lecture Notes

Sorting algorithms

- ➡ Insertion, selection and bubble sort have quadratic worst-case performance
- ➡ The faster comparison based algorithm ?
 $O(n \log n)$
- ➡ Mergesort and Quicksort

Quicksort Algorithm

- Fastest known sorting algorithm in practice
 - Caveats: not stable,
 - Vulnerable to certain attacks
- Average case complexity $\rightarrow O(N \log N)$
- Worst-case complexity $\rightarrow O(N^2)$
 - Rarely happens, if coded correctly

Quicksort Algorithm

Given an array of n elements (e.g., integers):

- ➡ If array only contains one element, return
- ➡ Else
 - ➡ pick one element to use as *pivot*.
 - ➡ Partition elements into two sub-arrays:
 - ➡ Elements less than or equal to pivot
 - ➡ Elements greater than pivot
 - ➡ Quicksort two sub-arrays
 - ➡ Return results

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 \geq 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

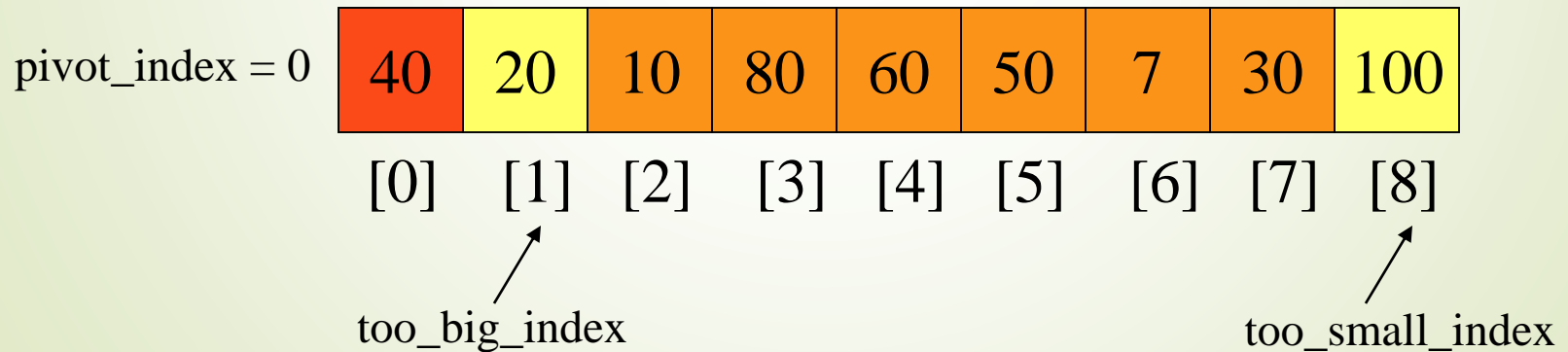
40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

[0] [1] [2] [3] [4] [5] [6] [7] [8]

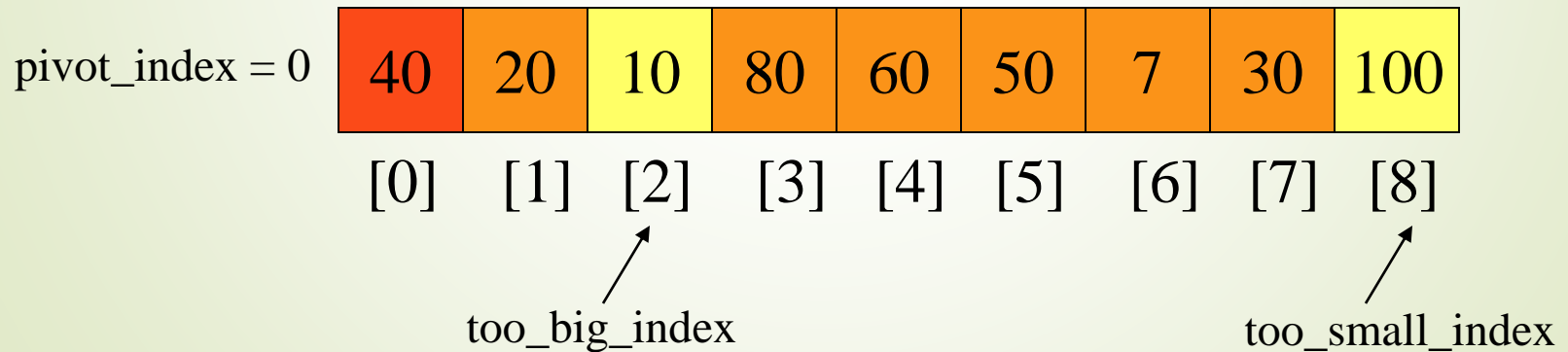
too_big_index

too_small_index

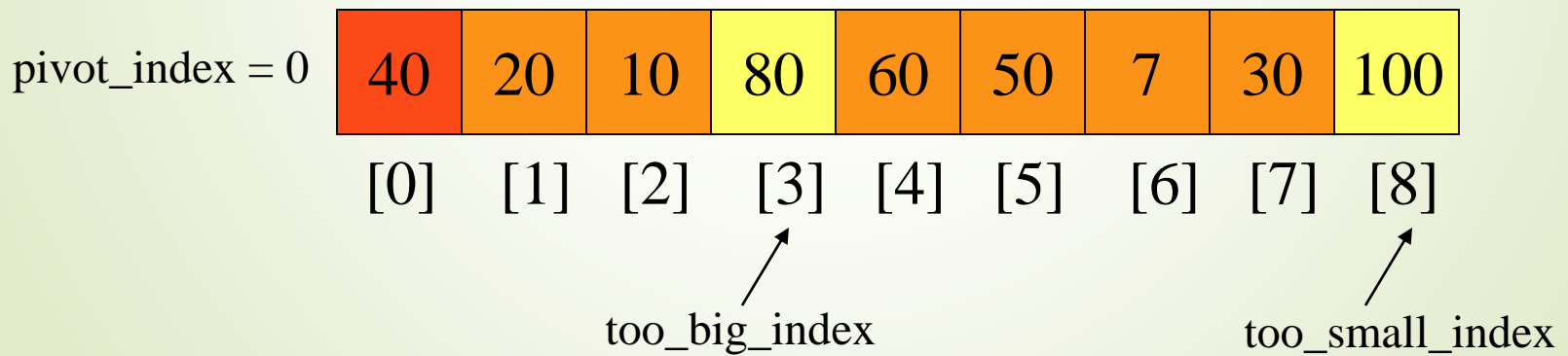
1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`



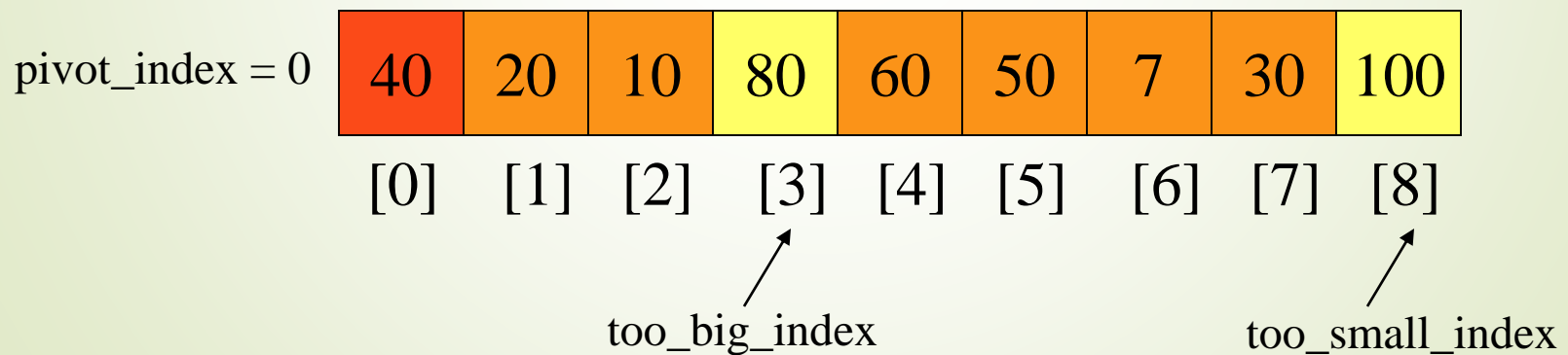
1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`



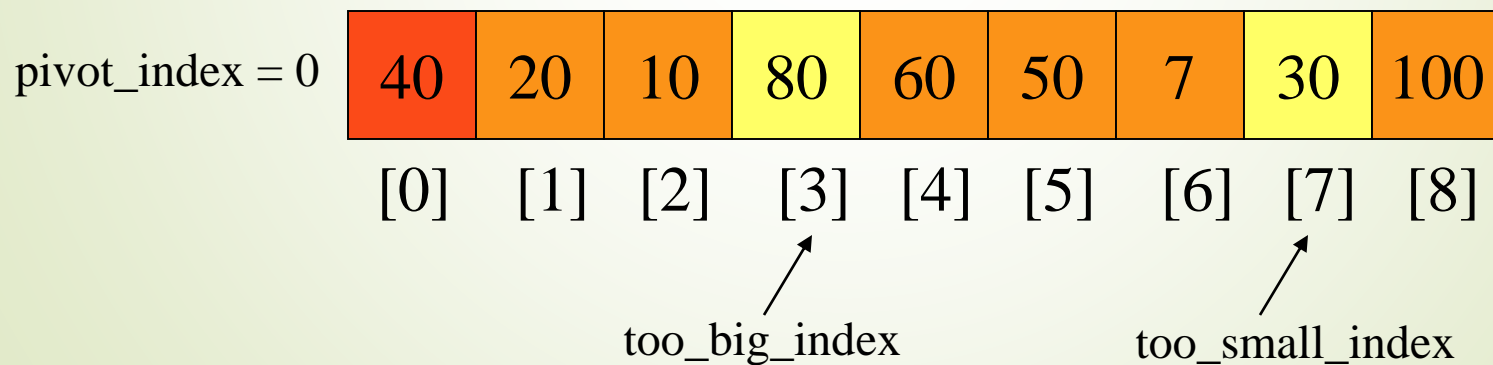
```
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
```



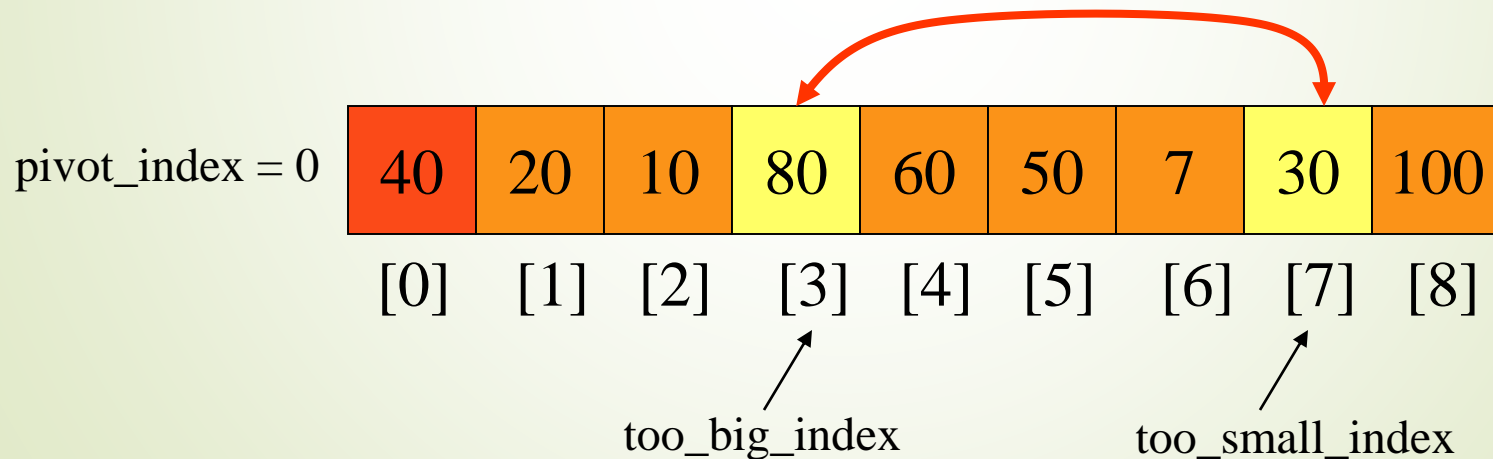
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$



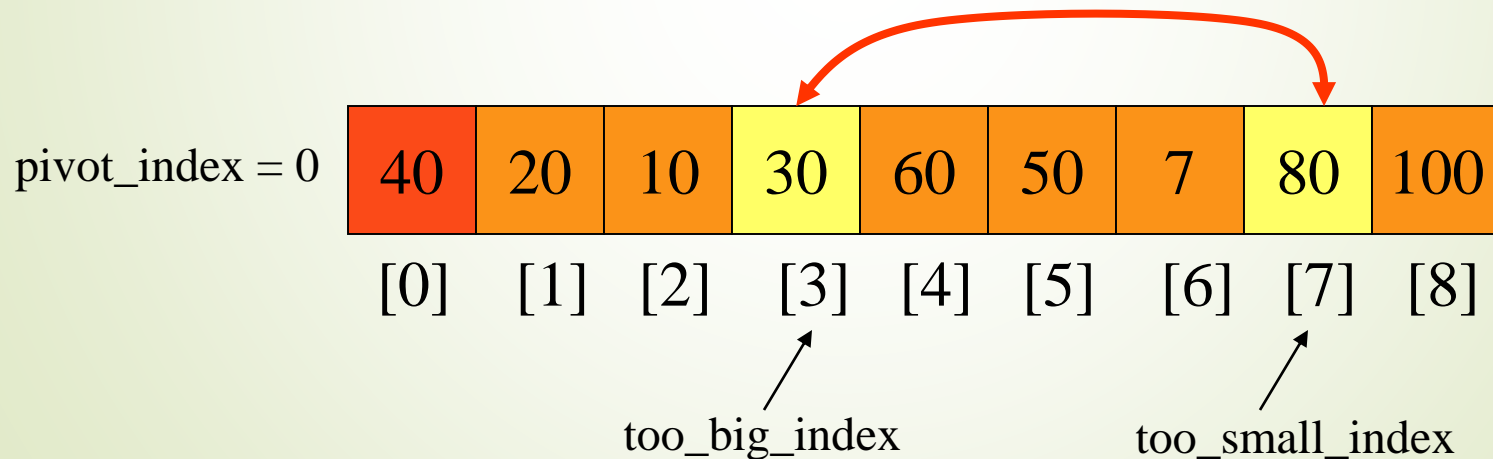
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index



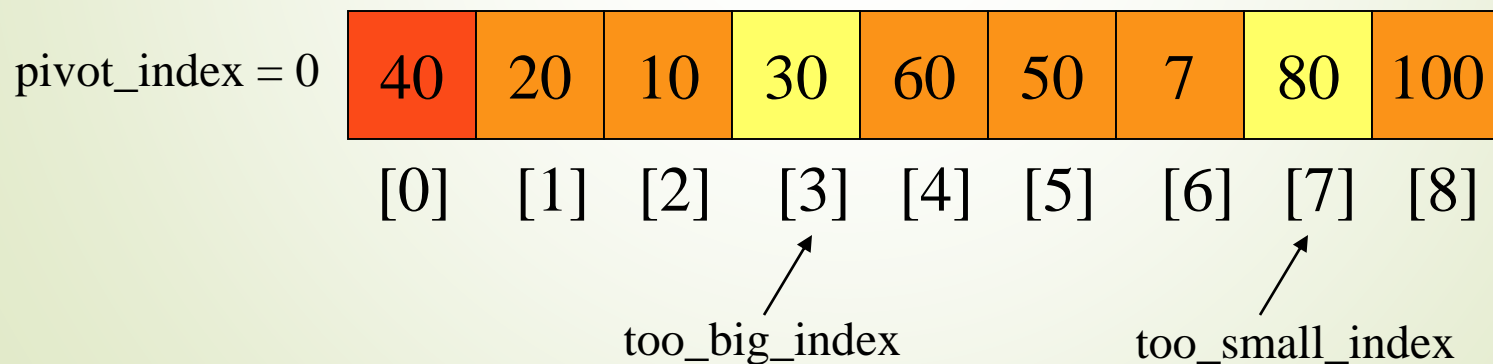
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



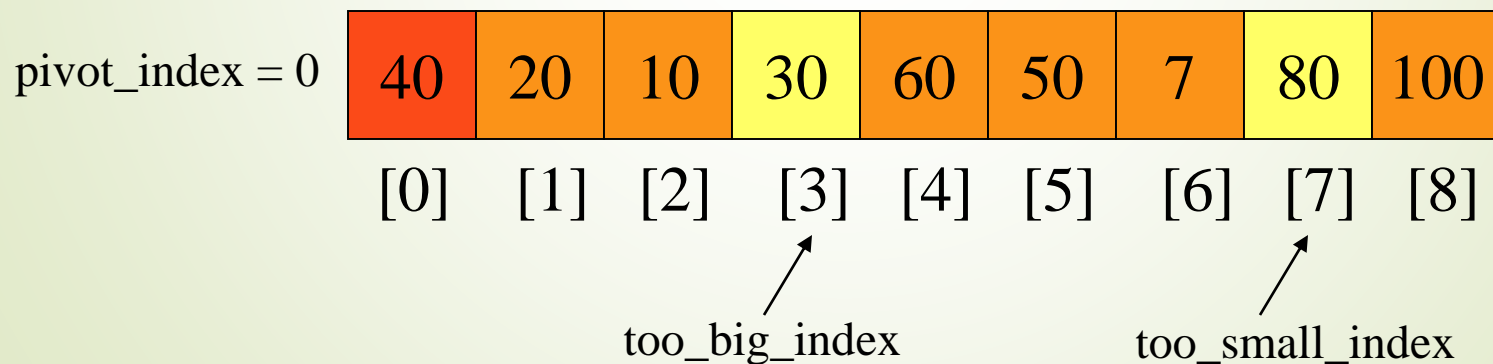
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



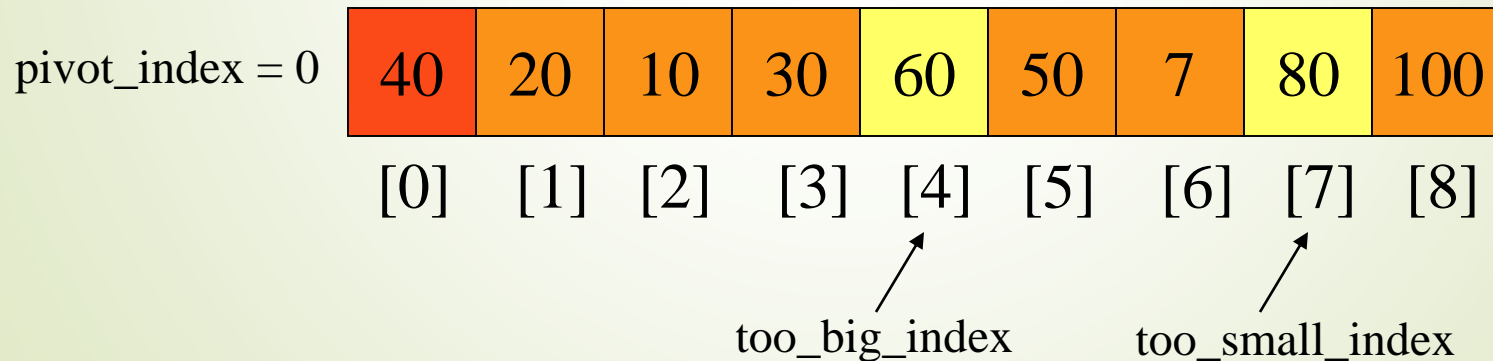
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



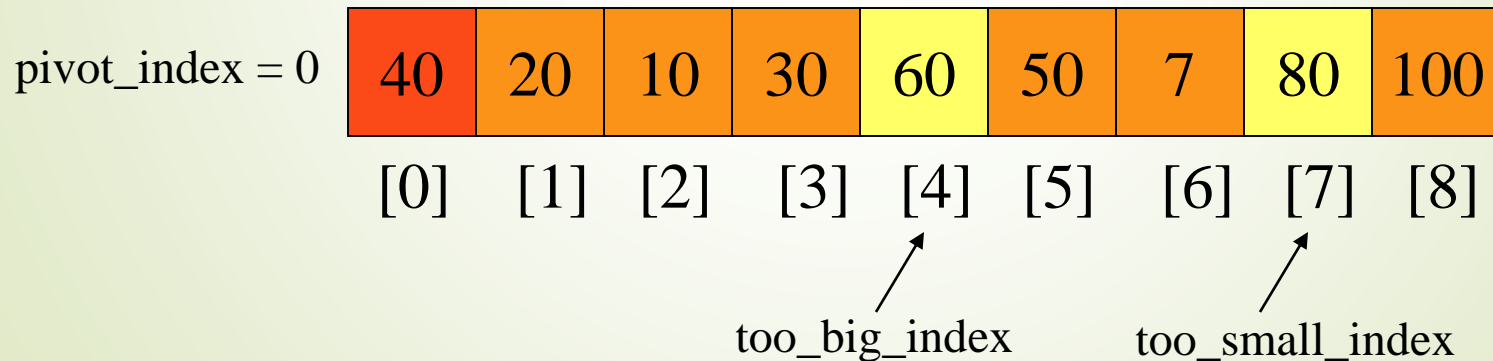
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



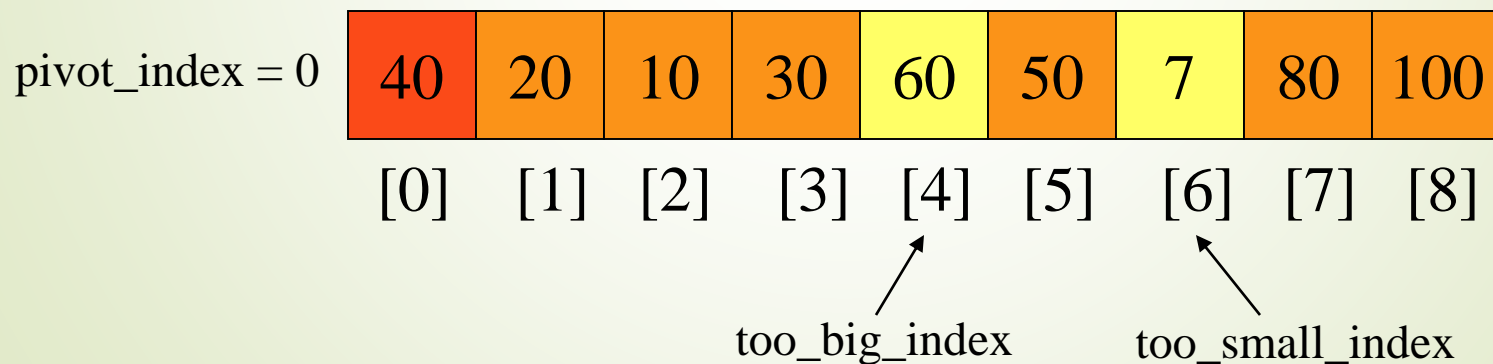
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



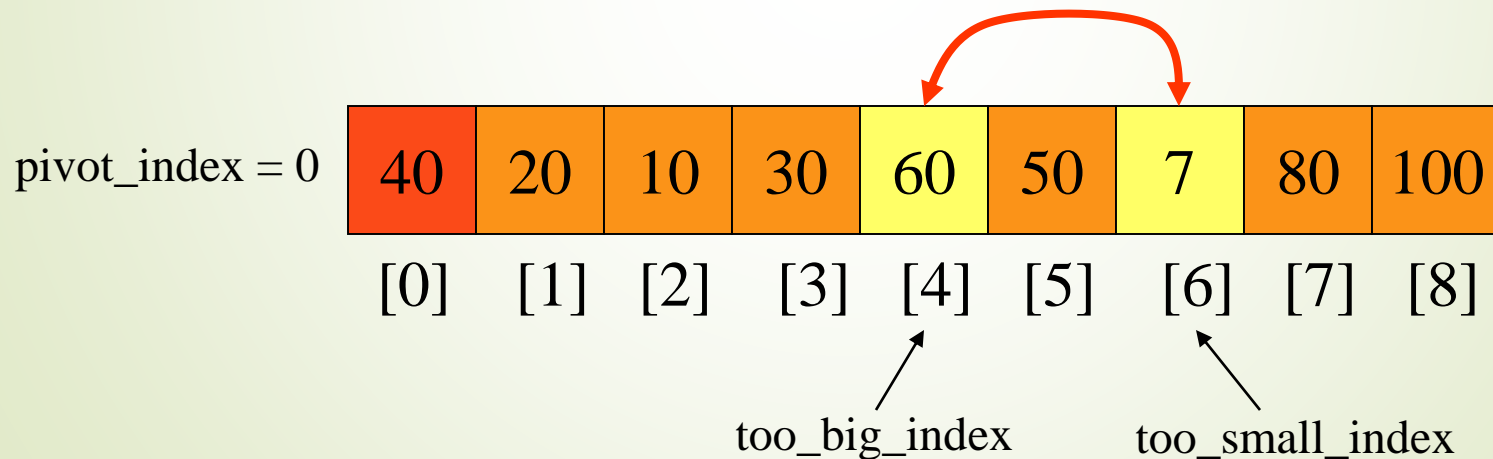
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



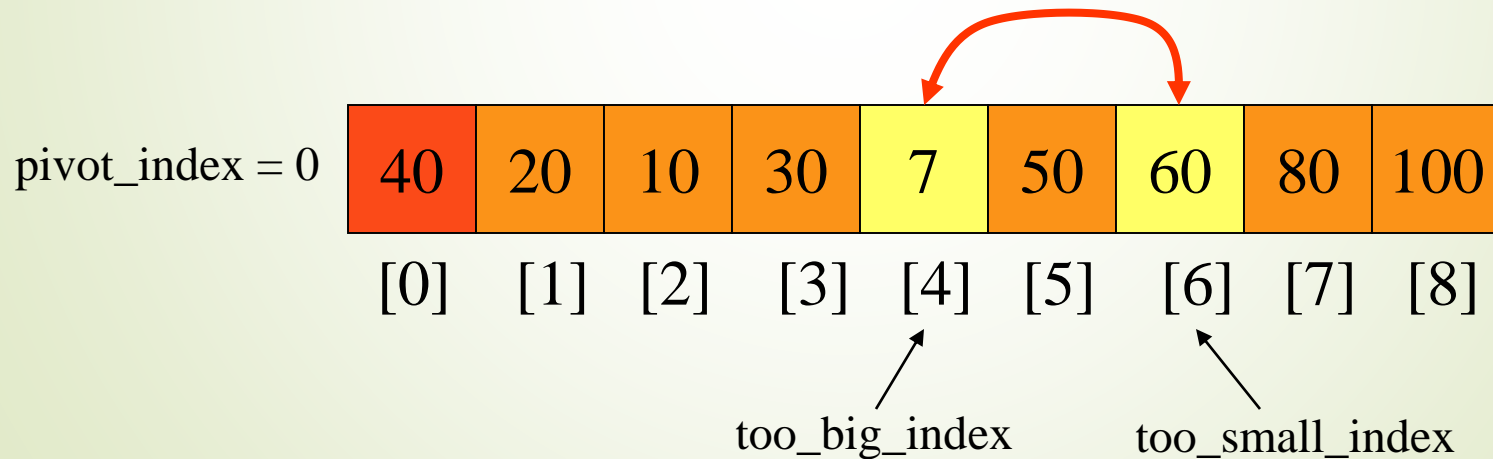
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



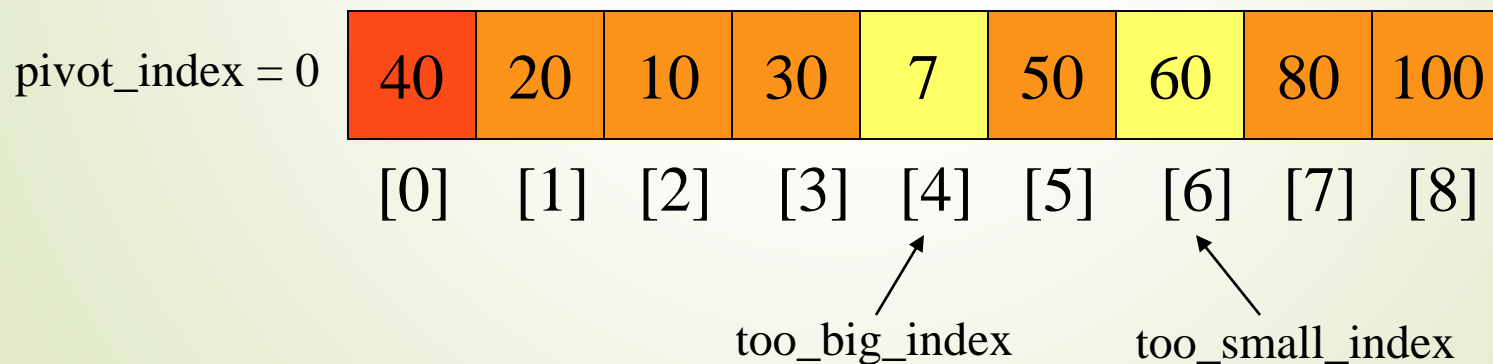
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



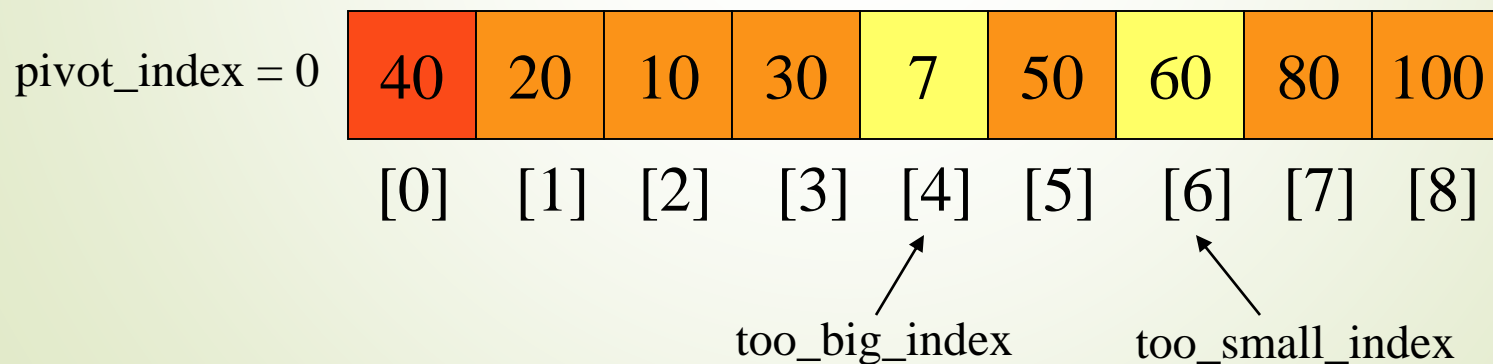
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



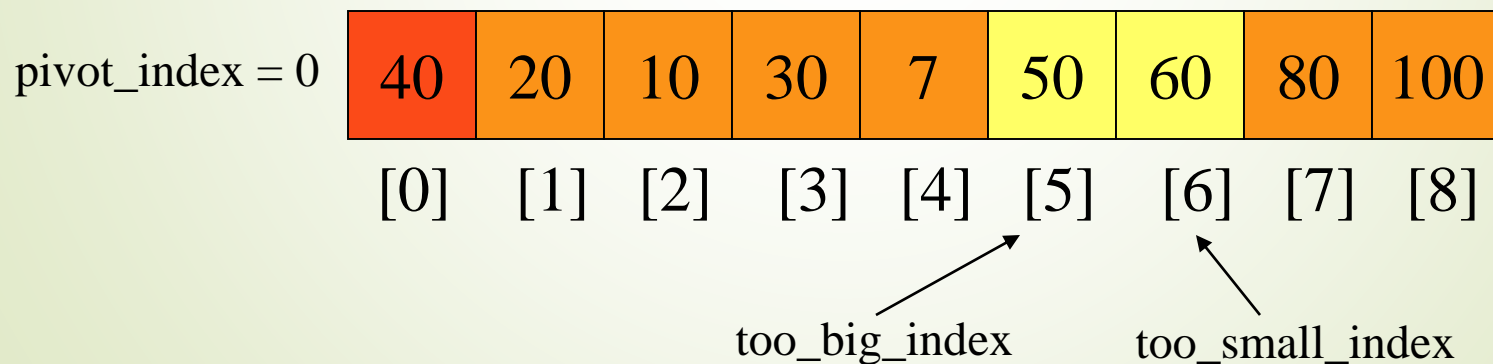
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



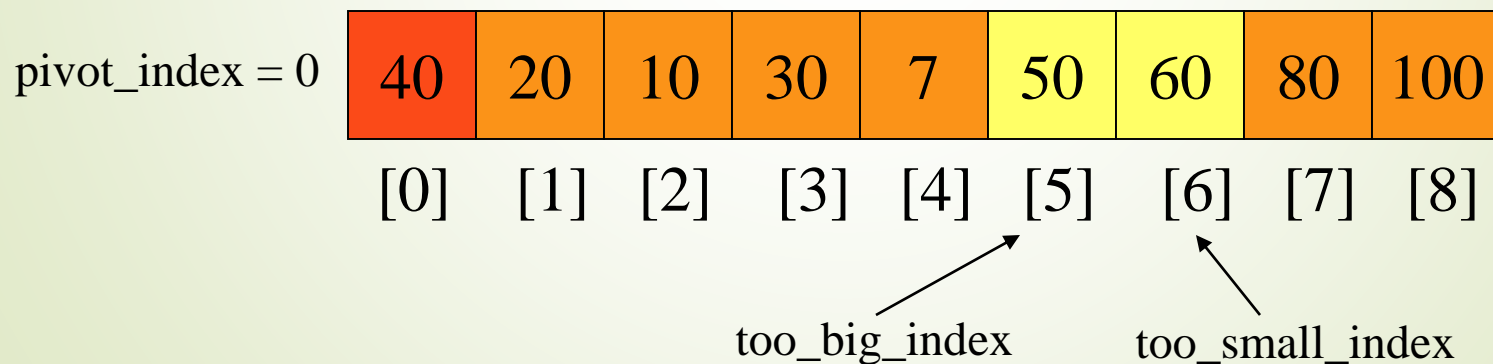
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



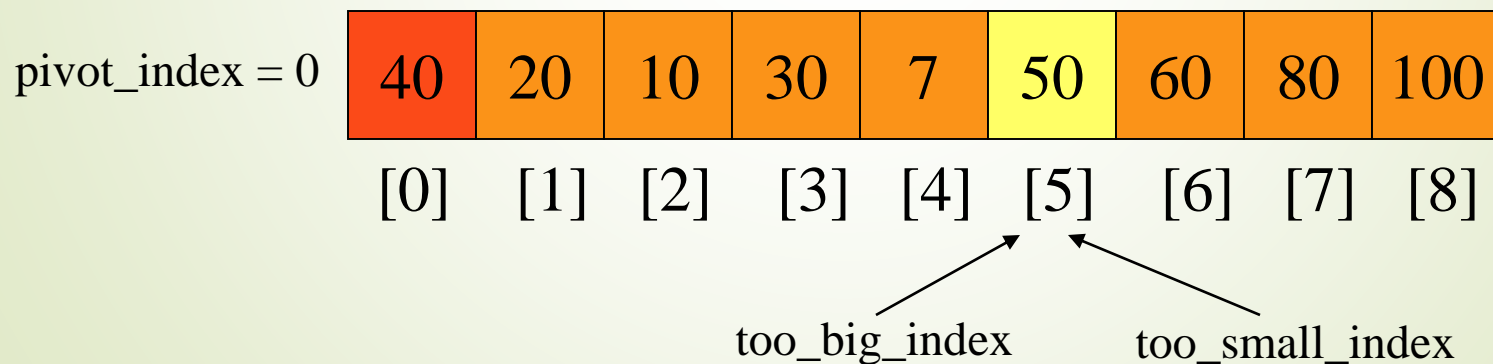
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



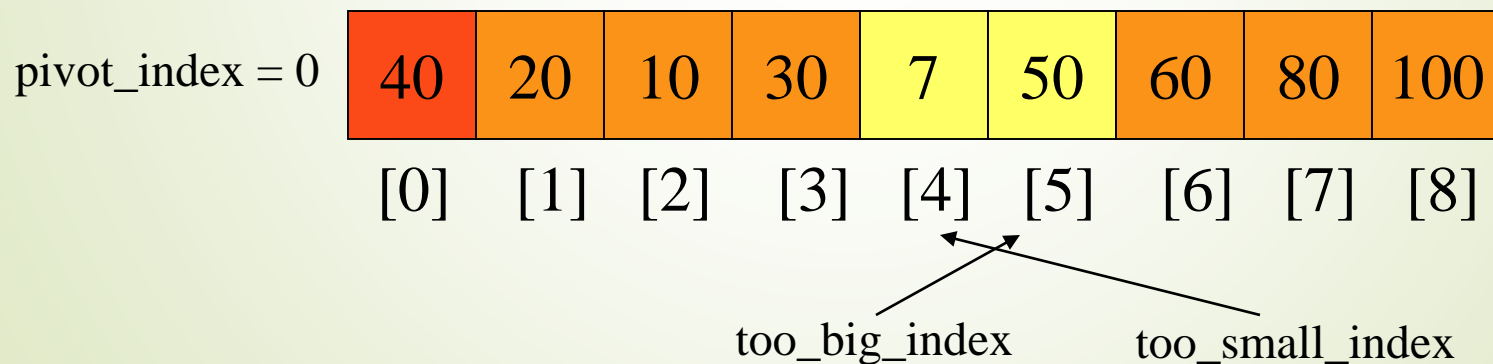
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



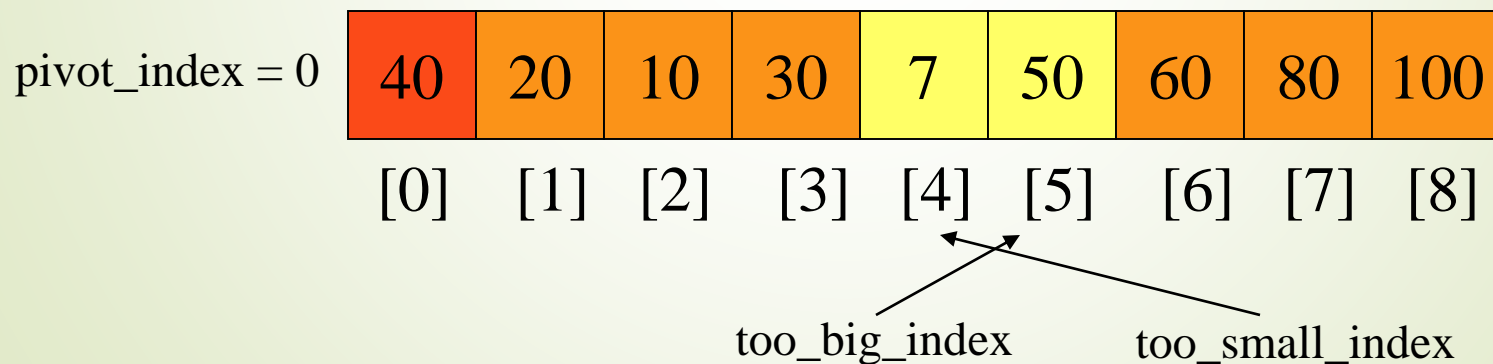
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



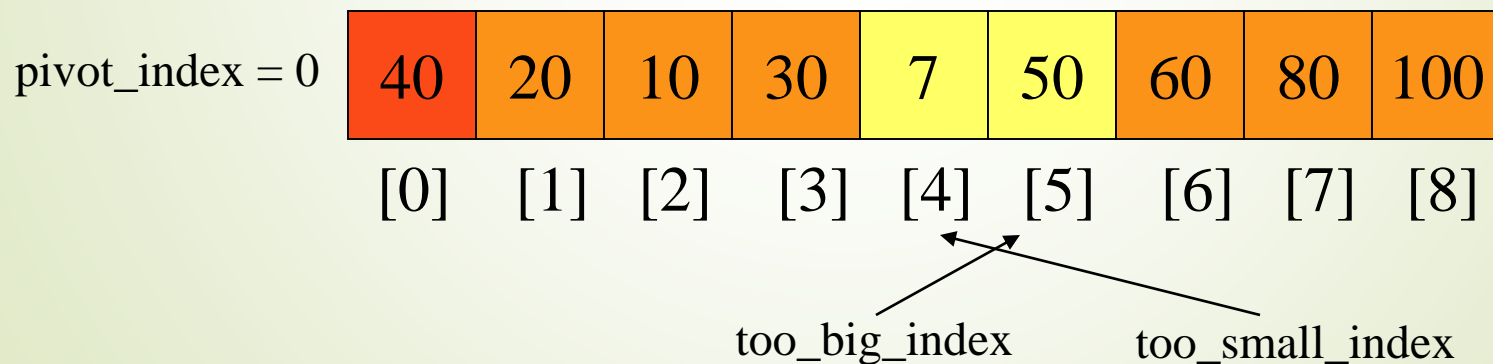
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



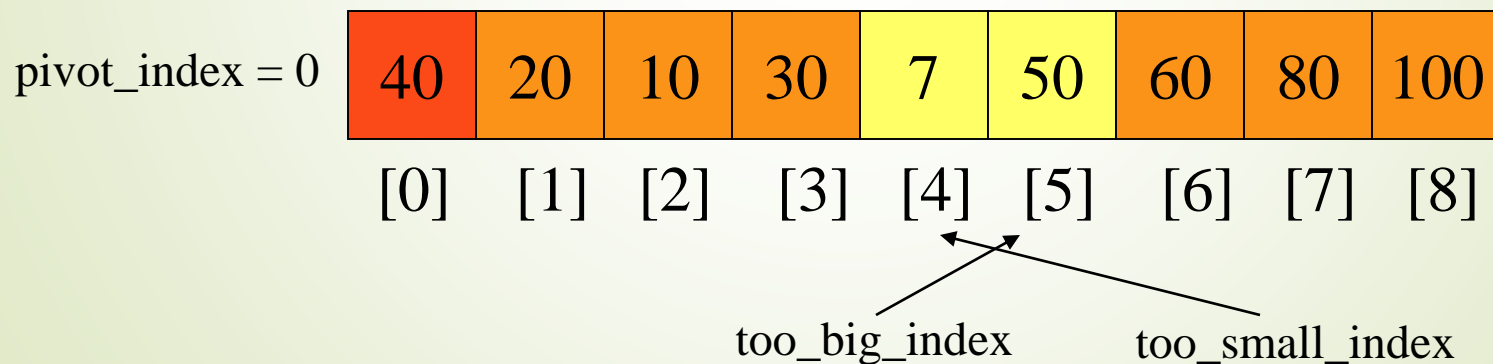
1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
- 3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
4. While `too_small_index > too_big_index`, go to 1.



1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
- 4. While `too_small_index > too_big_index`, go to 1.

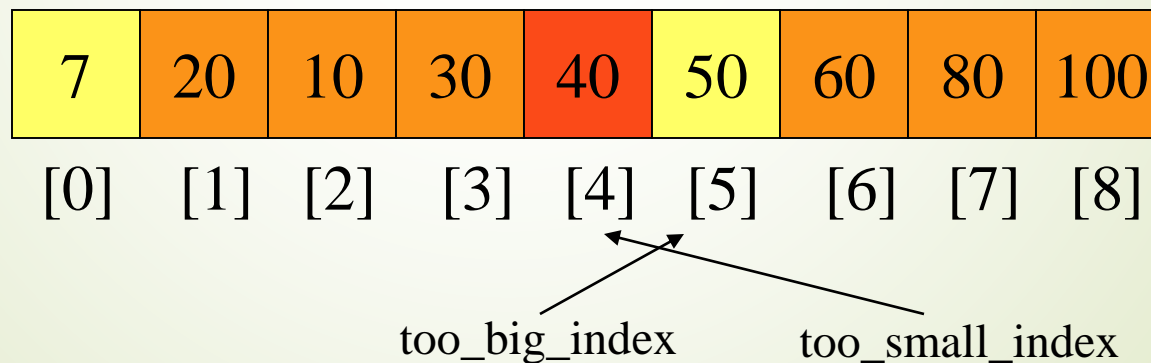


1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
4. While `too_small_index > too_big_index`, go to 1.
- 5. Swap `data[too_small_index]` and `data[pivot_index]`



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$

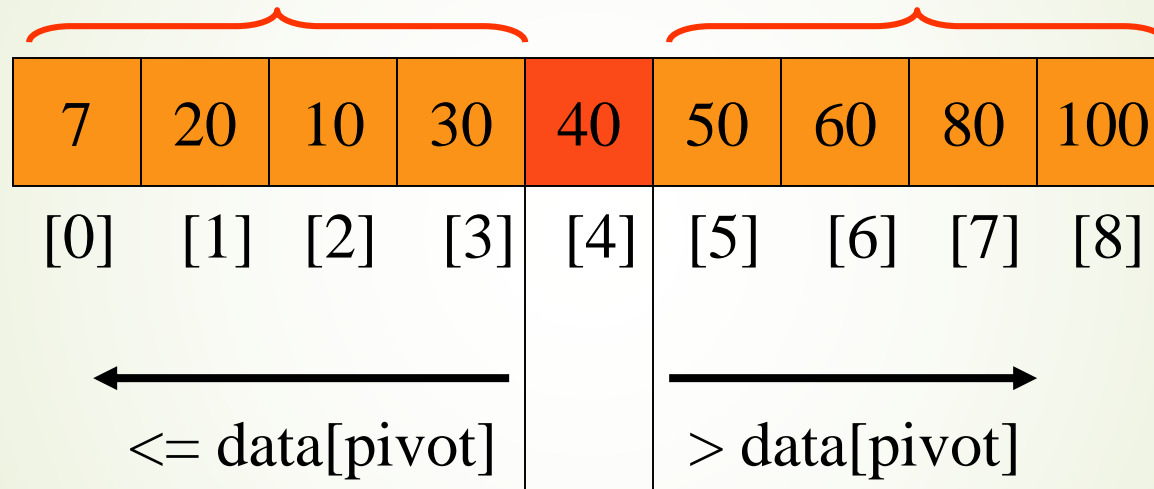
pivot_index = 4



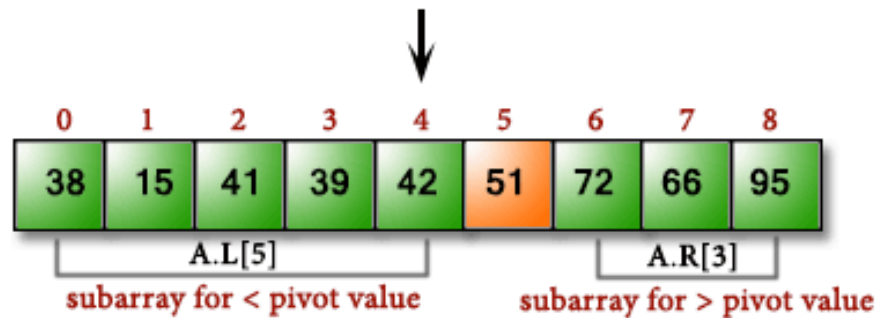
Partition Result

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
←					→			
≤ data[pivot]					> data[pivot]			

Recursion: Quicksort Sub-arrays

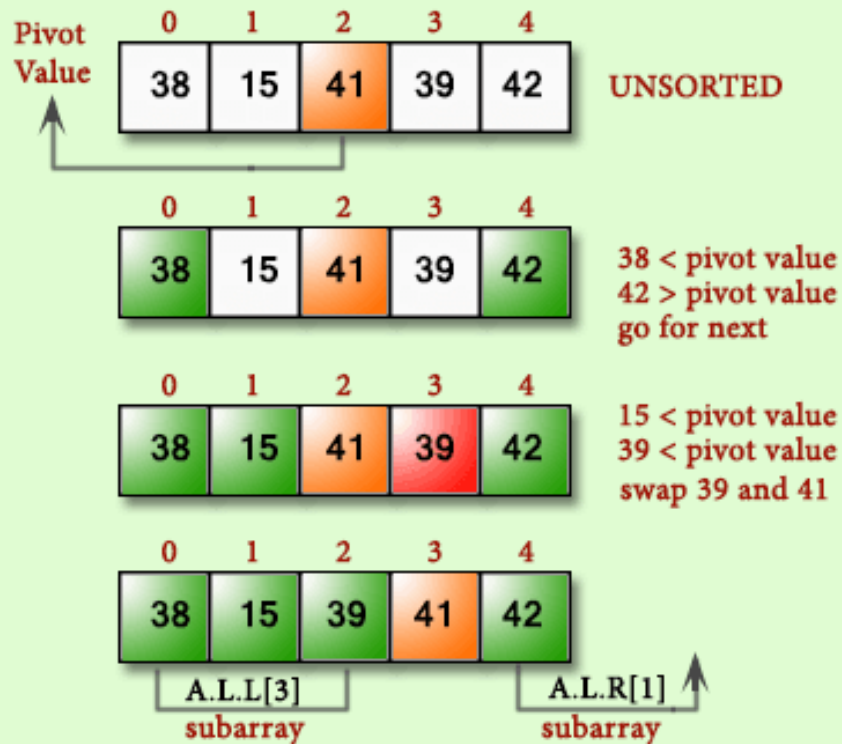






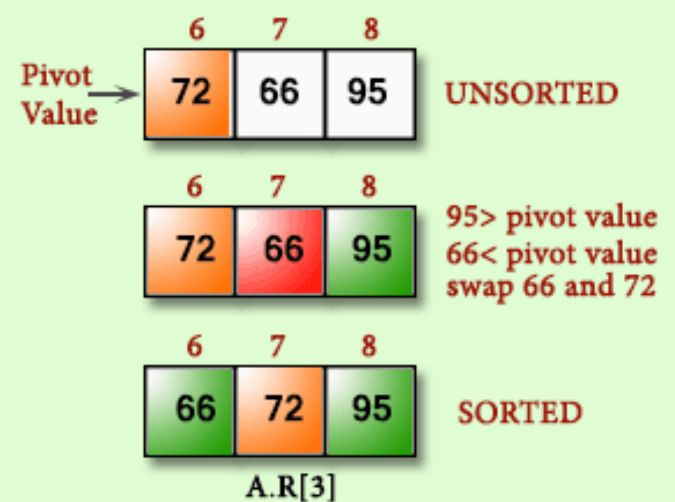
Quick sort recursively

A.L[5]



A.R[3]

Quick sort recursively



A.L.L[3]

Quick sort recursively

Pivot Value →

0	1	2
38	15	39

UNSORTED

0	1	2
38	15	39

39 > pivot value
15 < pivot value
swap 15 and 38

0	1	2
15	38	39

SORTED

A.L.L.L[3]

FINAL SORTING

Split point of A[9]

Split point of A.L[5]

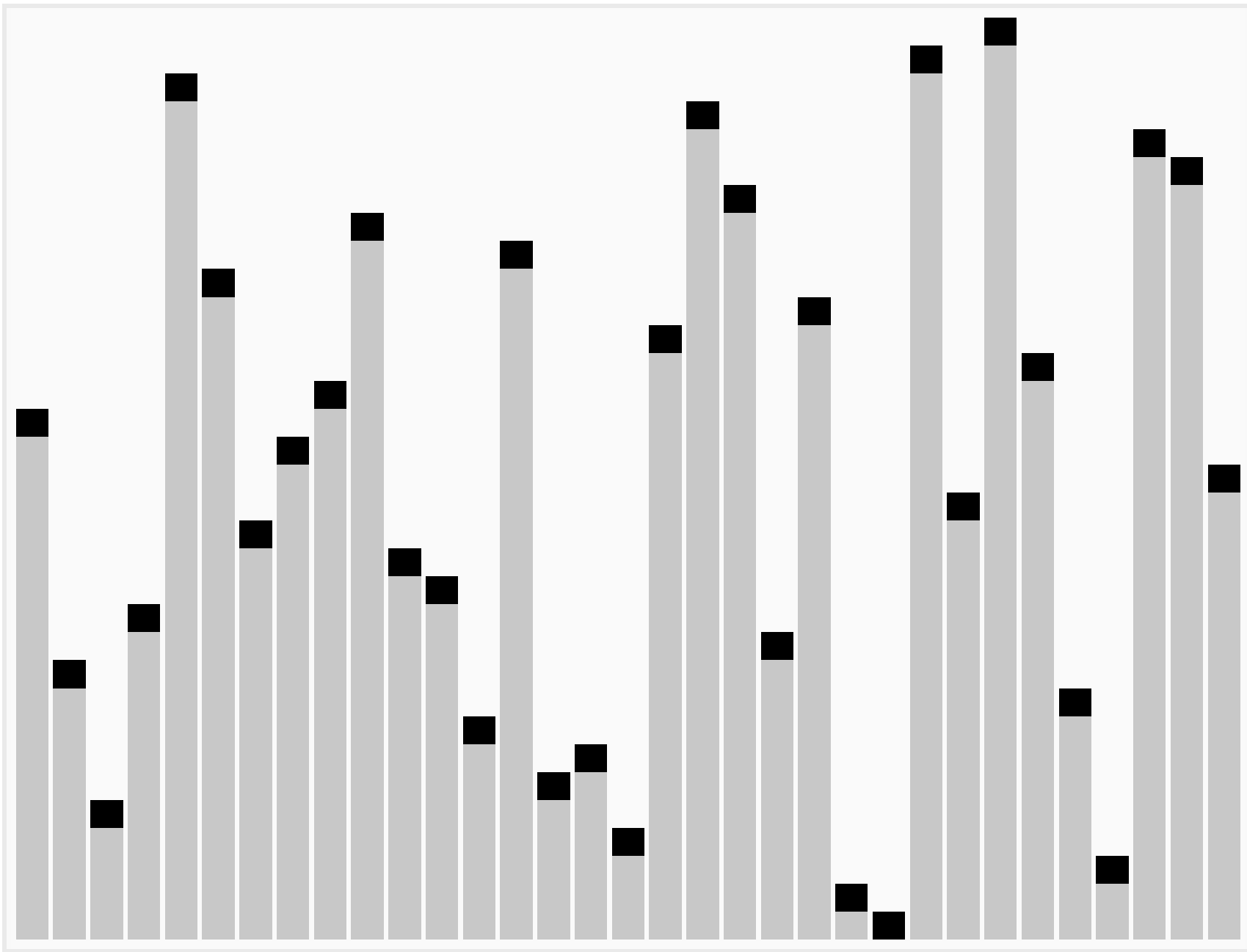
A.L.L.L[3]

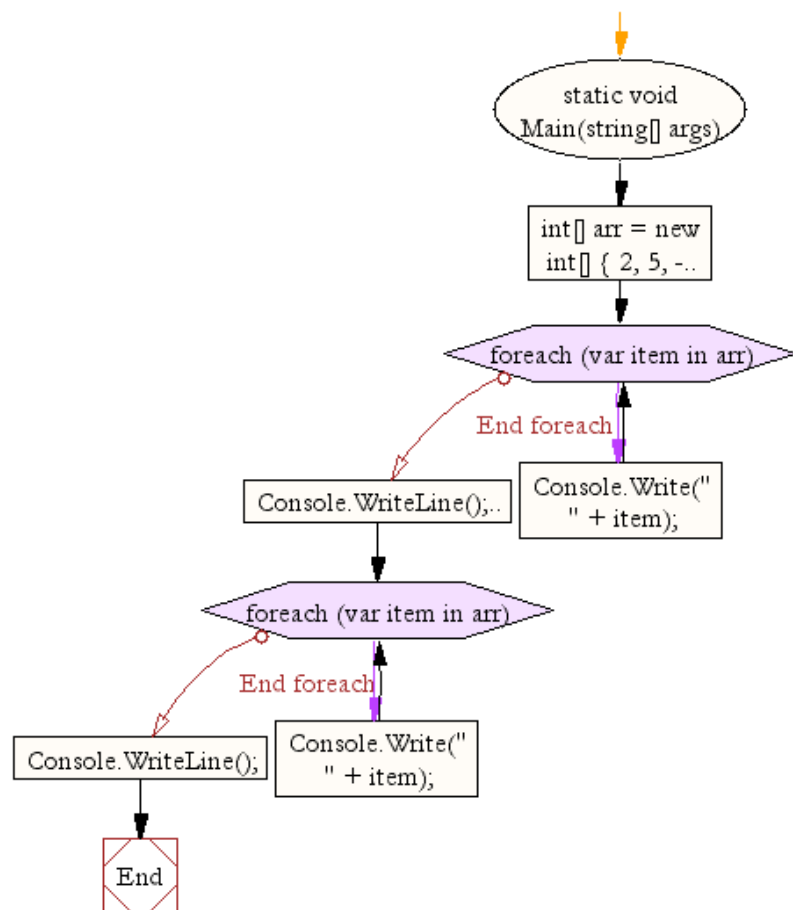
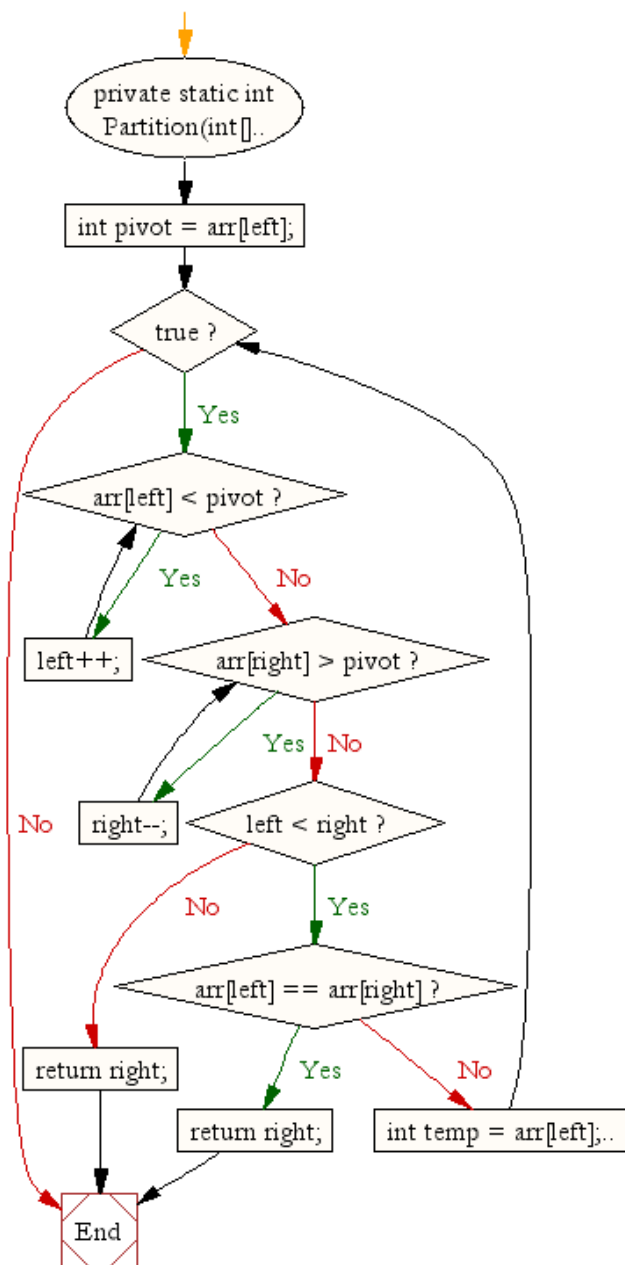
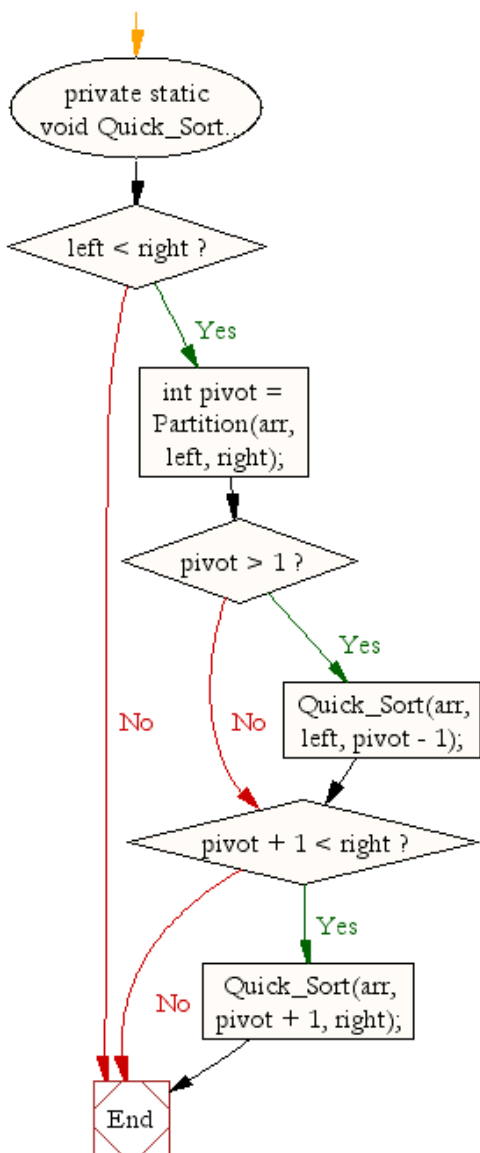
A.L.R[1]

A.R[3]

0	1	2	3	4	5	6	7	8
15	38	39	41	42	51	66	72	95

15	38	39	41	42	51	66	72	95
----	----	----	----	----	----	----	----	----





Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses in partition? $O(n)$

Quicksort Analysis

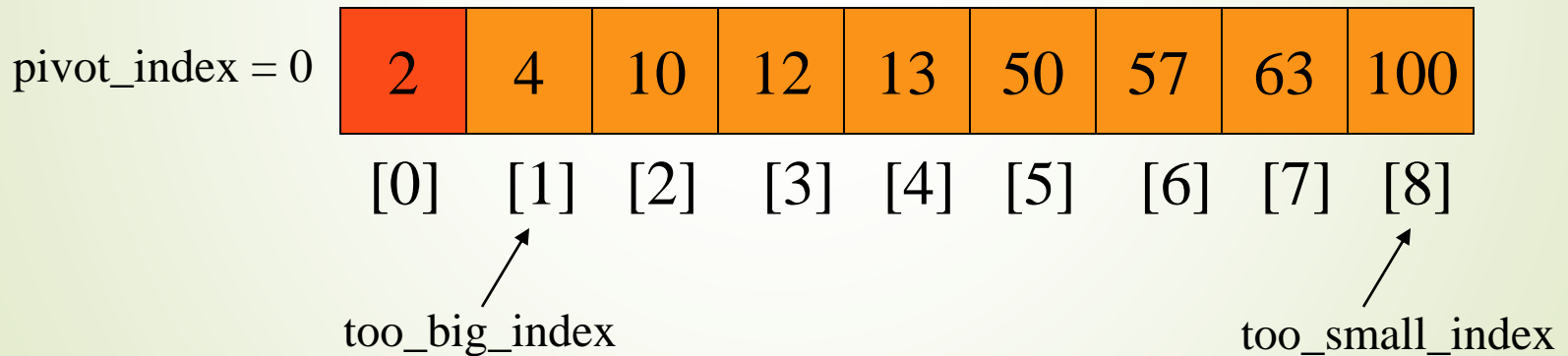
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

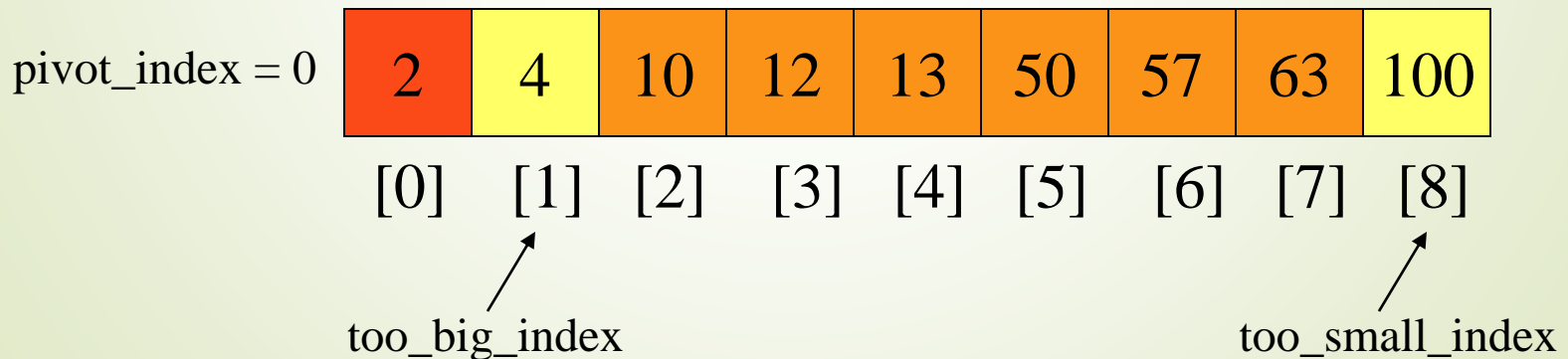
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

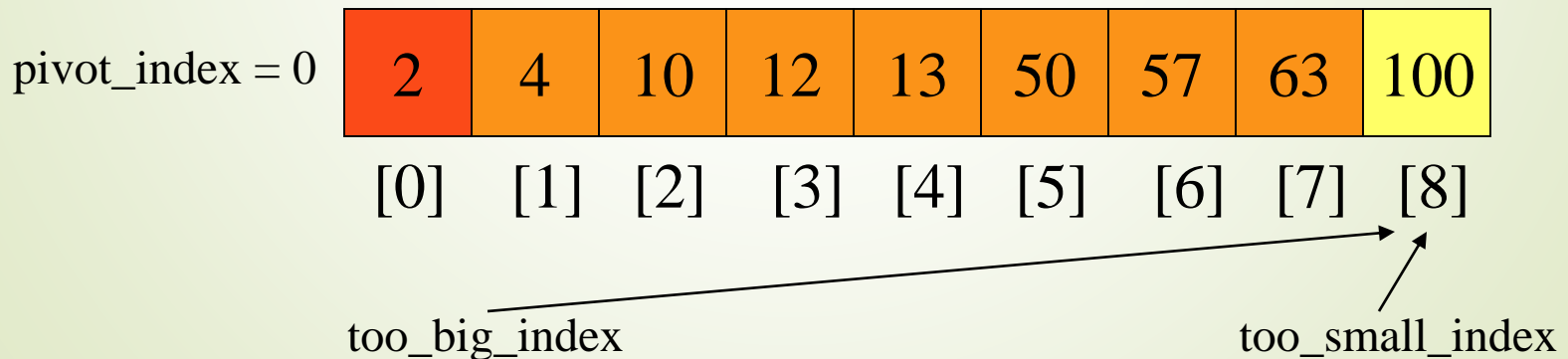
- ➡ Assume first element is chosen as pivot.
- ➡ Assume we get array that is already in order:



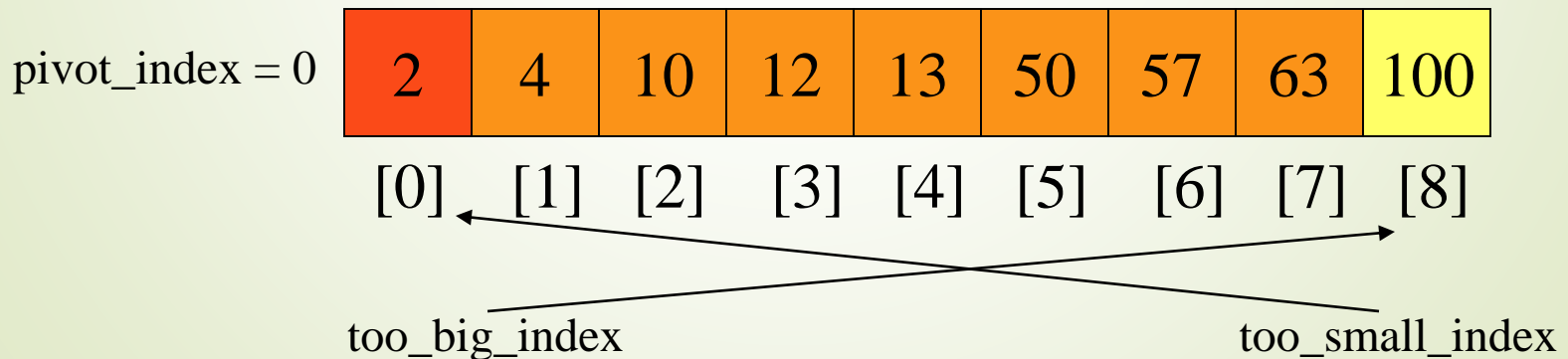
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



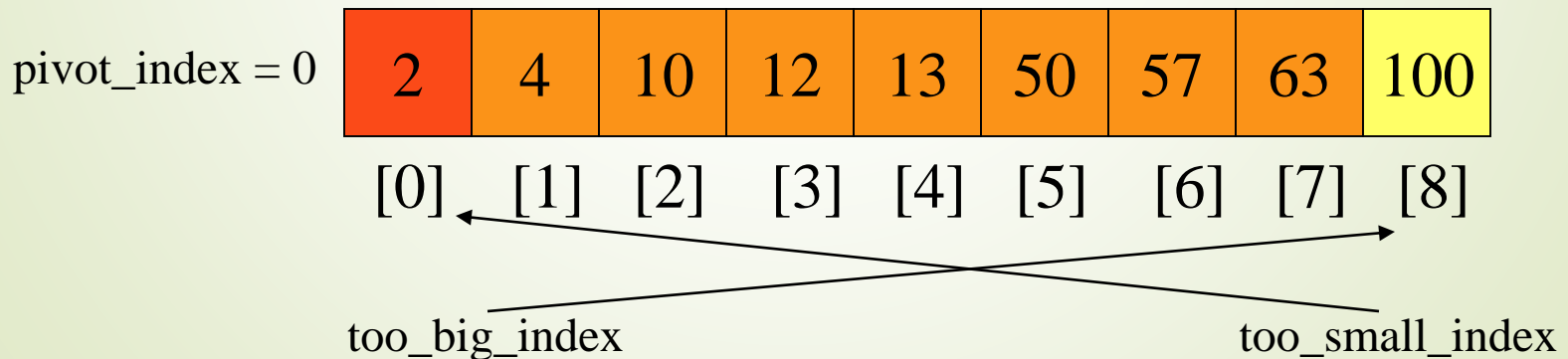
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



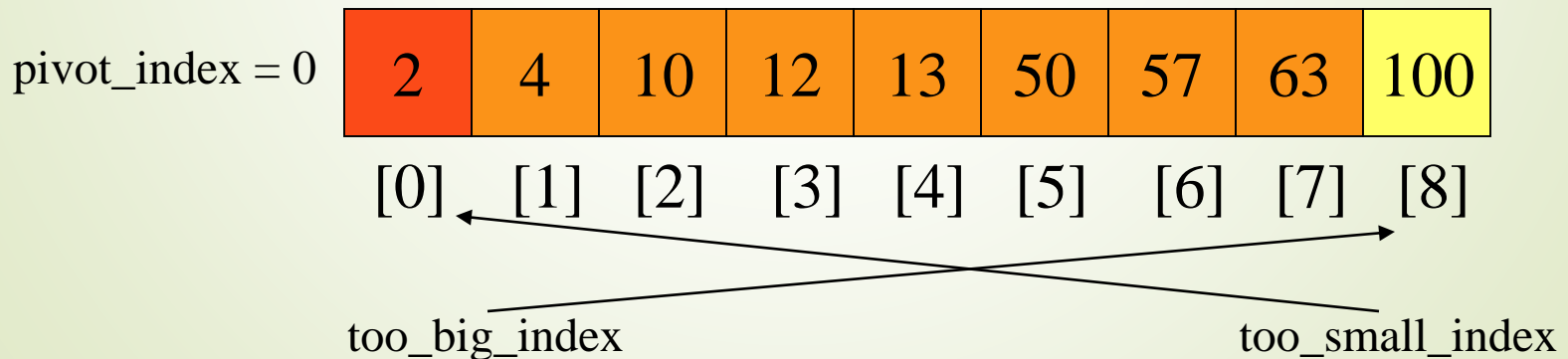
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



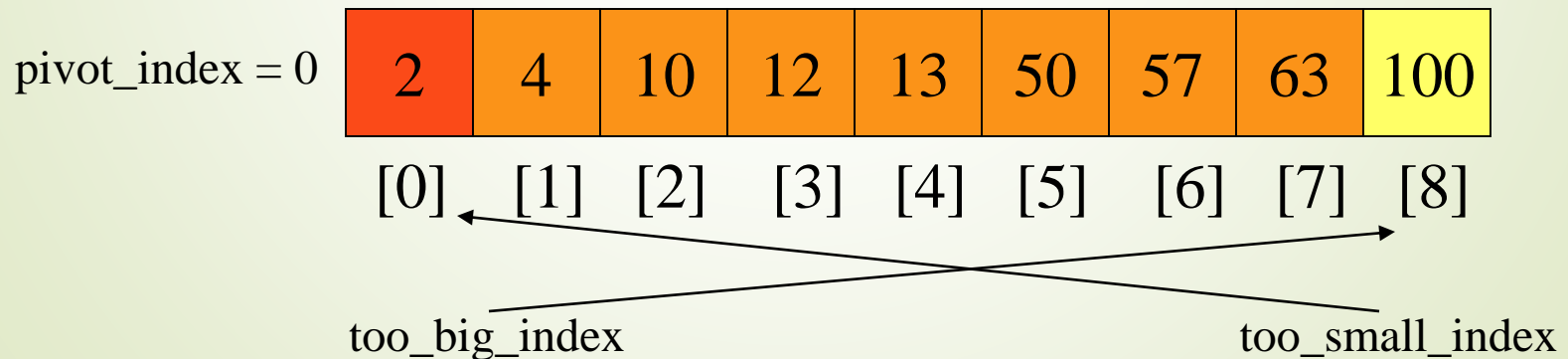
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



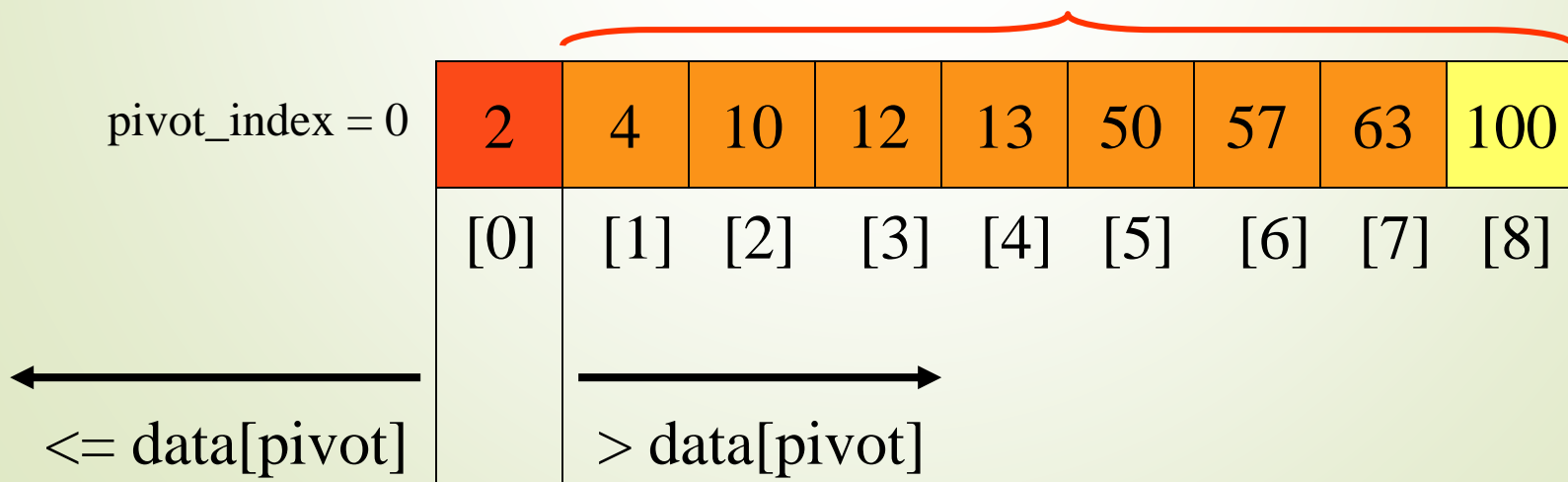
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
- 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)!!!$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)!!!$
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

$\text{data}[0]$, $\text{data}[n/2]$, and $\text{data}[n-1]$.

- ➡ Use this median of the array
 - ➡ Partitioning always cuts the array into roughly half
 - ➡ An optimal quicksort ($O(N \log N)$)
 - ➡ However, hard to find the exact median

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
 - Sub-array of size 1: trivial
 - Sub-array of size 2:
 - if(`data[first] > data[second]`) swap them
 - Sub-array of size 3?

Pivot: median of three

- We will use median of three
 - Compare just three elements: the leftmost, rightmost and center
 - Swap these elements if necessary so that
 - median3** {
 - $A[\text{left}] = \text{Smallest}$
 - $A[\text{right}] = \text{Largest}$
 - $A[\text{center}] = \text{Median of three}$
 - Pick $A[\text{center}]$ as the pivot
 - Swap $A[\text{center}]$ and $A[\text{right} - 1]$ so that pivot is at second last position

Pivot: median of three

- ➡ Code for partitioning with median of three pivot:

```
int center = ( left + right ) / 2;  
if( a[ center ] < a[ left ] )  
    swap( a[ left ], a[ center ] );  
if( a[ right ] < a[ left ] )  
    swap( a[ left ], a[ right ] );  
if( a[ right ] < a[ center ] )  
    swap( a[ center ], a[ right ] );  
  
    // Place pivot at position right - 1  
swap( a[ center ], a[ right - 1 ] );
```

Pivot: median of three



$A[\text{left}] = 2$, $A[\text{center}] = 13$,
 $A[\text{right}] = 6$



Swap $A[\text{center}]$ and $A[\text{right}]$



Choose $A[\text{center}]$ as pivot



pivot



Swap pivot and $A[\text{right} - 1]$



pivot

Note we only need to partition $A[\text{left} + 1, \dots, \text{right} - 2]$

Implementation of partitioning step

- Works only if pivot is picked as median-of-three.
 - $A[\text{left}] \leq \text{pivot}$ and $A[\text{right}] \geq \text{pivot}$
 - Thus, only need to partition $A[\text{left} + 1, \dots, \text{right} - 2]$
- j will not run past the end
 - because $a[\text{left}] \leq \text{pivot}$
- i will not run past the end
 - because $a[\text{right}-1] = \text{pivot}$

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Main Quicksort Routine

```
if( left + 10 <= right )  
{
```

```
    Comparable pivot = median3( a, left, right );
```

Choose pivot

```
        // Begin partitioning
```

```
        int i = left, j = right - 1;  
        for( ; ; )  
        {  
            while( a[ ++i ] < pivot ) { }  
            while( pivot < a[ --j ] ) { }  
            if( i < j )  
                swap( a[ i ], a[ j ] );  
            else  
                break;  
        }
```

Partitioning

```
        swap( a[ i ], a[ right - 1 ] ); // Restore pivot
```

```
        quicksort( a, left, i - 1 ); // Sort small elements  
        quicksort( a, i + 1, right ); // Sort large elements
```

Recursion

```
    }  
    else // Do an insertion sort on the subarray  
        insertionSort( a, left, right );
```

For small arrays

Quicksort Faster than Mergesort

- Both quicksort and mergesort take $O(N \log N)$ in the average case.
- Why is quicksort faster than mergesort?
 - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
 - Mergesort involves a large number of data movements.
 - Quicksort is done in-place.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Performance of quicksort

- ➡ Worst-case: takes $O(n^2)$ time.
- ➡ Average-case: takes $O(n \log n)$ time.
- ➡ On typical inputs, quicksort runs faster than other algorithms.

Further Analysis of Quicksort

- ➡ The analysis is quite tricky.
- ➡ Assume all the input elements are distinct
 - ➡ no duplicate values makes this code faster!
 - ➡ there are better partitioning algorithms when duplicate input elements exist (e.g. Hoare's original code)
- ➡ Let $T(n)$ = worst-case running time on an array of n elements.

Worst-case of quicksort

- QUICKSORT runs very slowly when its input array is already sorted (or is reverse sorted).
 - almost sorted data is quite common in the real-world
- This is caused by the partition using the min (or max) element which means that one side of the partition will have no elements. Therefore:

$$\begin{aligned}T(n) &= T(0) + T(n-1) + \Theta(n) \\&= \Theta(1) + T(n-1) + \Theta(n) \\&= T(n-1) + \Theta(n) \\&= \Theta(n^2) \text{ (arithmetic series)}\end{aligned}$$

no elements

n-1 elements

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

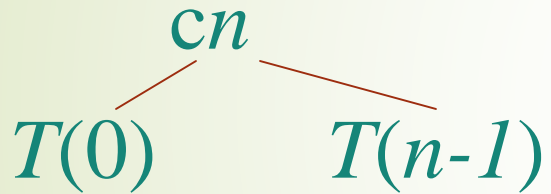
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

$$T(n)$$

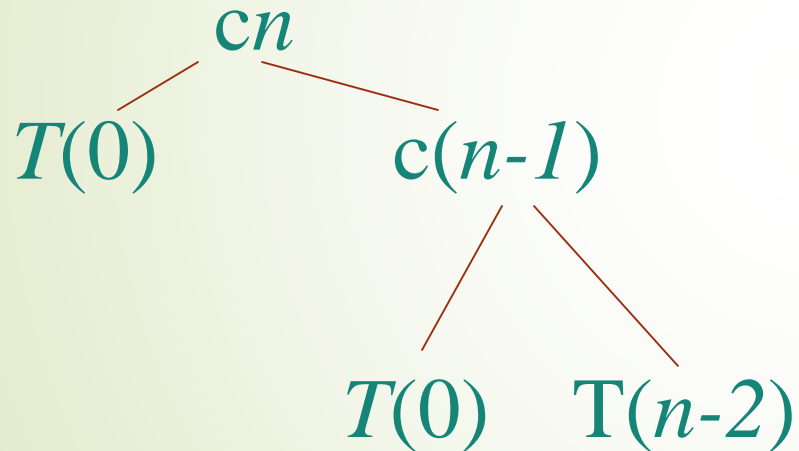
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



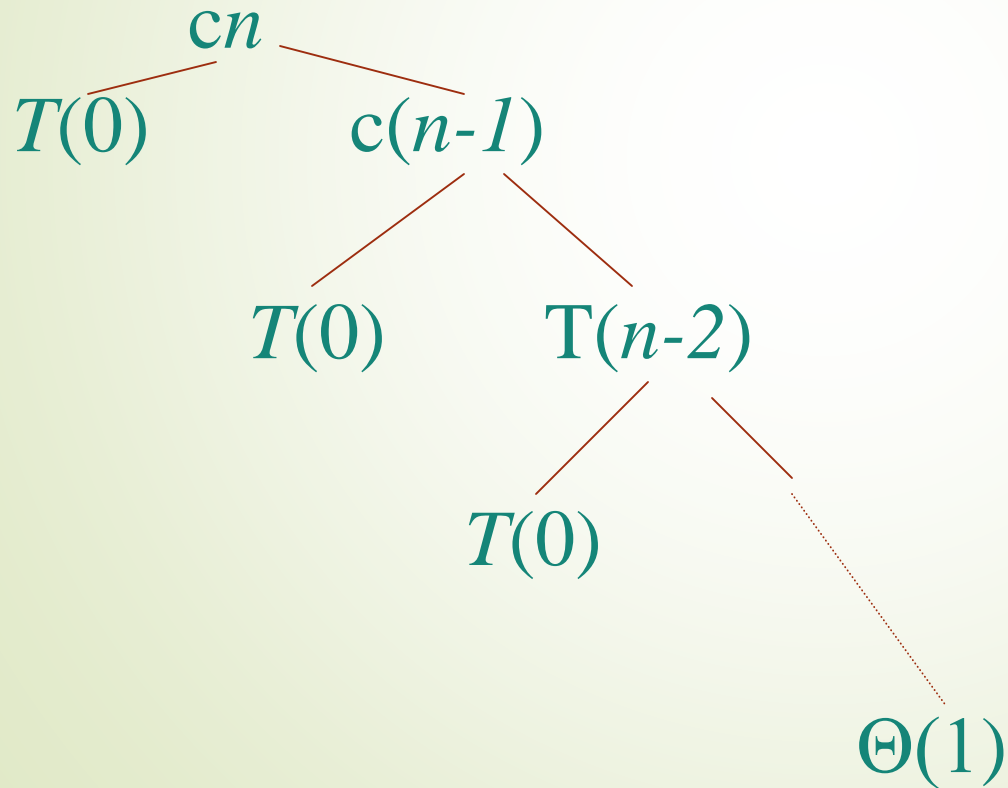
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



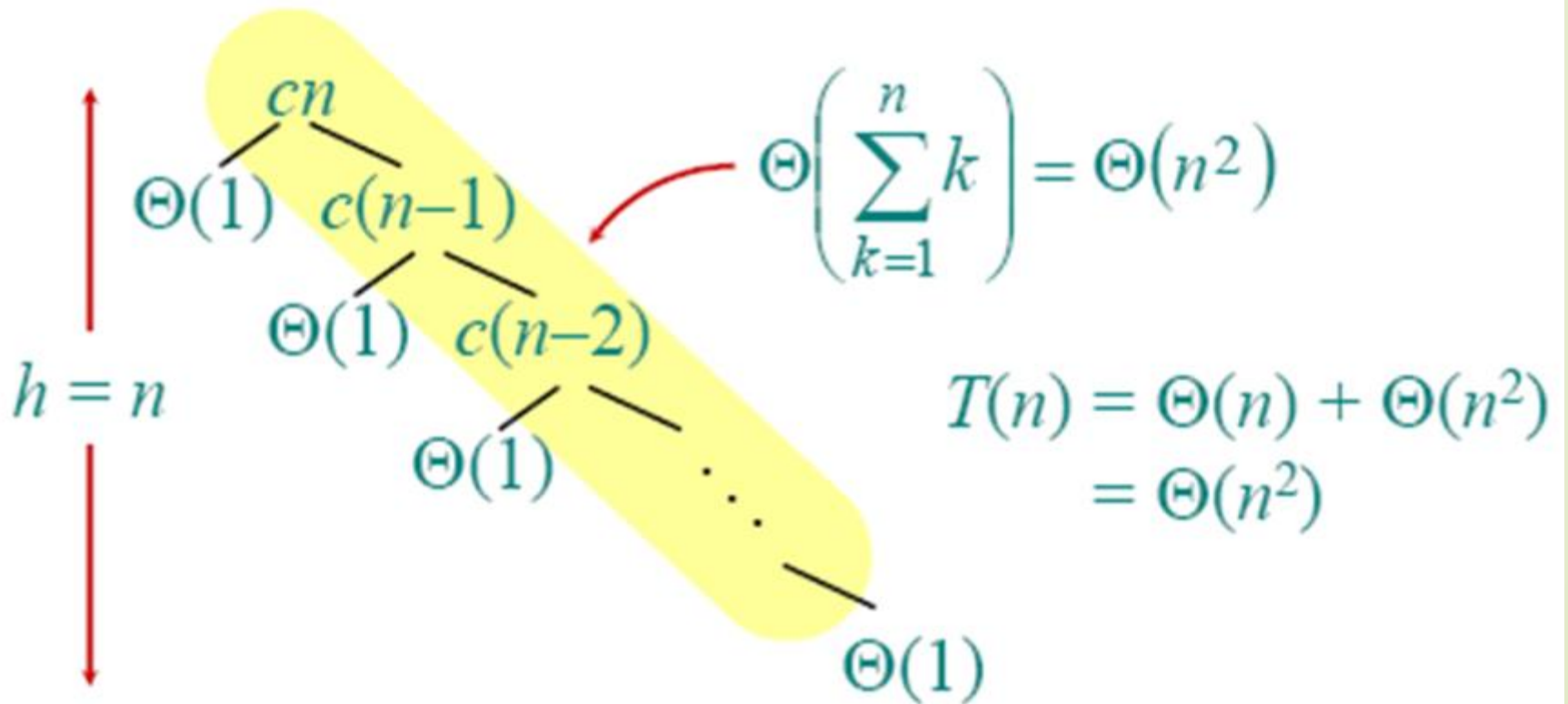
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



Quicksort isn't Quick?

- In the worst case, quicksort isn't any quicker than insertion sort.
- So why bother with quicksort?
- It's average case running time is very good, as we'll see.

Best-case Analysis

- If we're lucky, PARTITION splits the array evenly:

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

Case 2 of the
Master Method

- $= \Theta(n \log n)$ (same as merge sort)

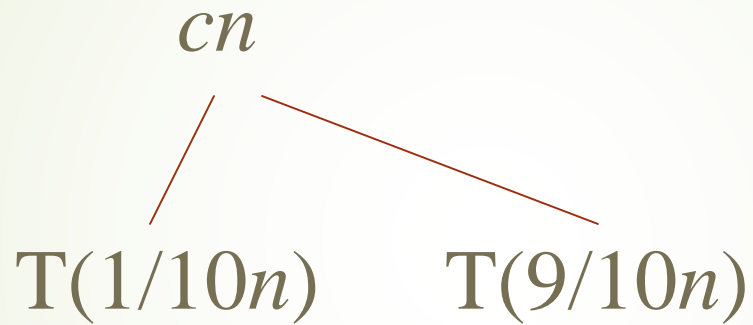
Almost Best-case

- ➡ What if the split is always 1/10 : 9/10?
- ➡ $T(n) = T(1/10n) + T(9/10n) + \Theta(n)$

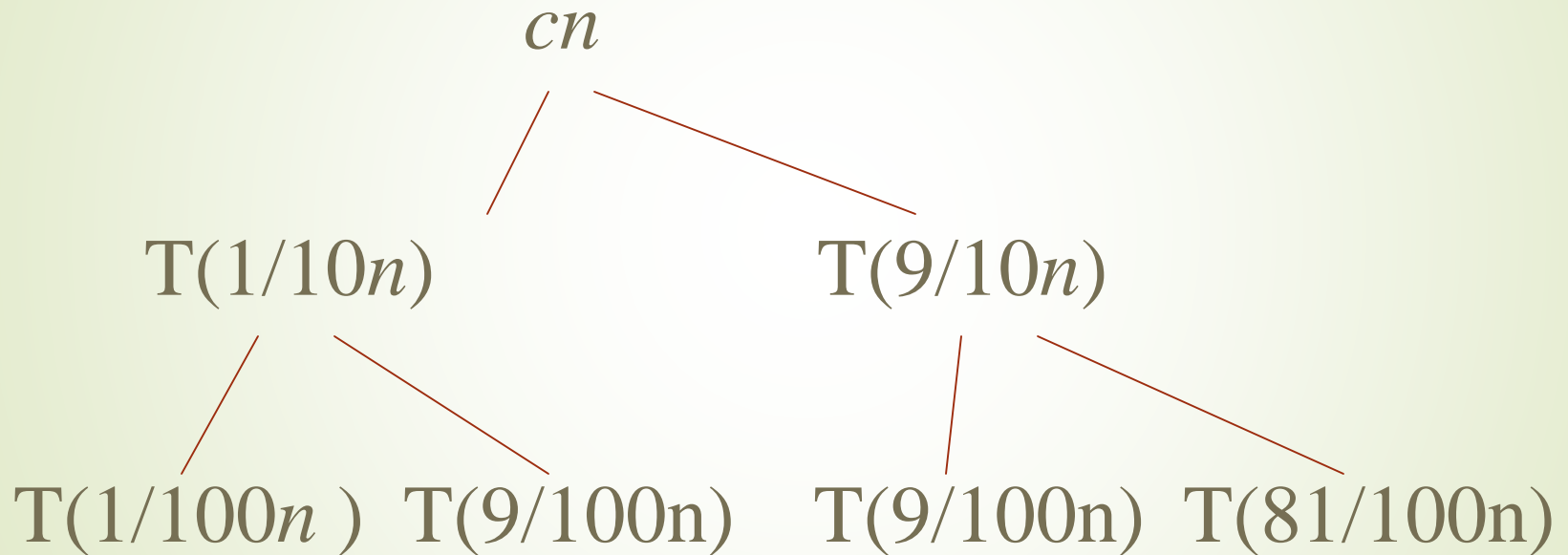
Analysis of “almost-best” case

$$T(n)$$

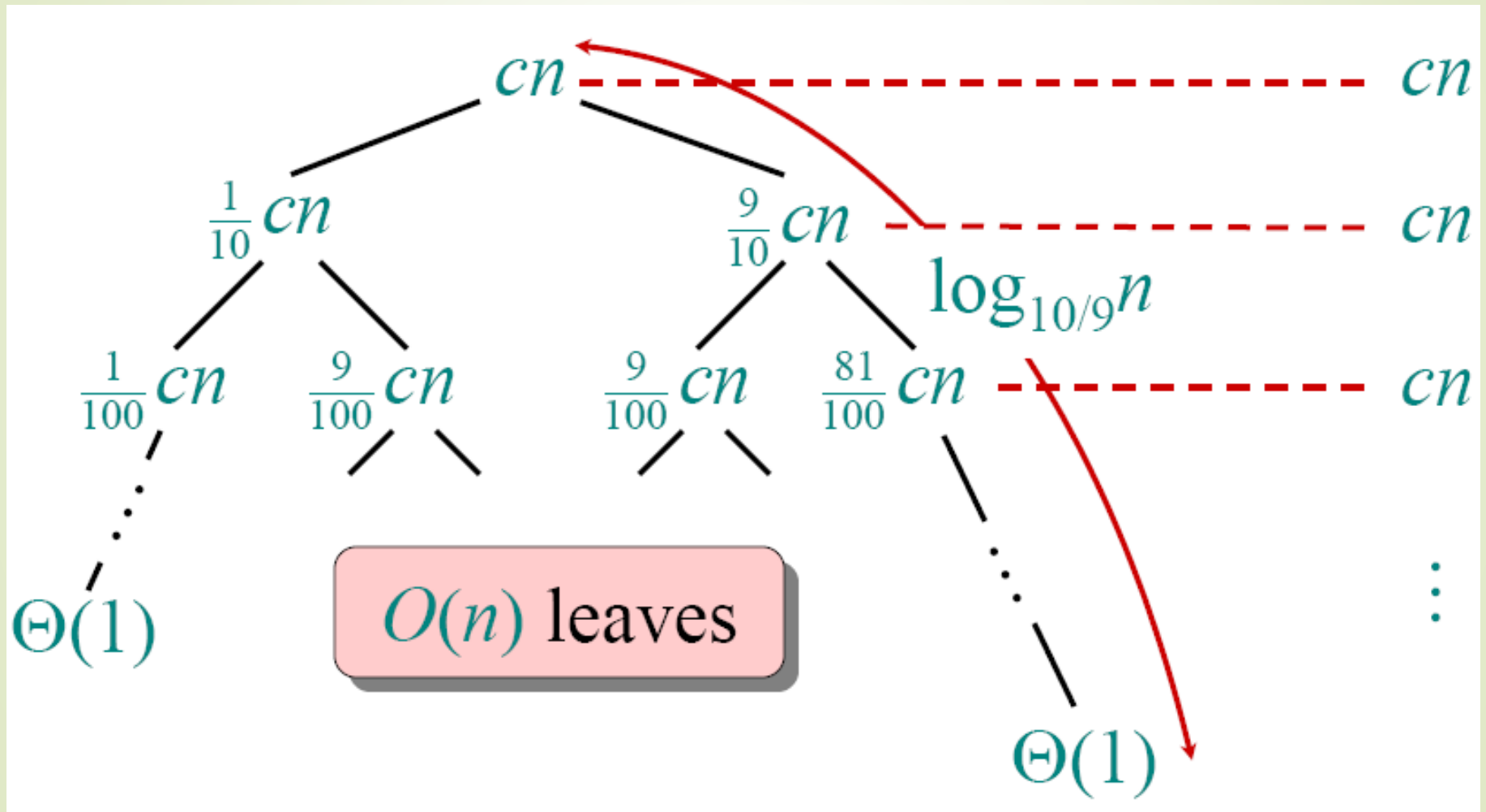
Analysis of “almost-best” case



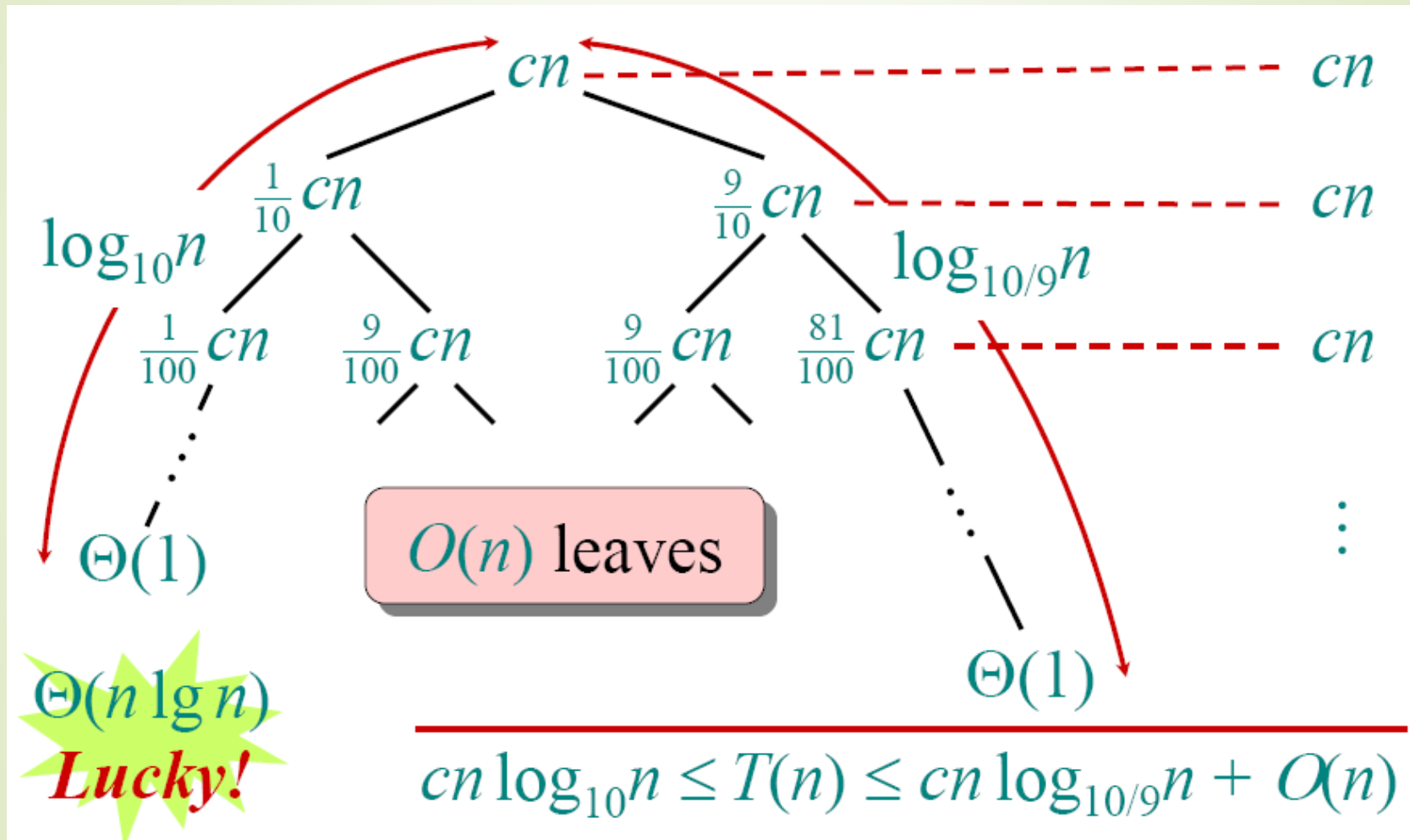
Analysis of “almost-best” case



Analysis of “almost-best” case



Analysis of “almost-best” case



Short and Long Path Heights

- Short path node value:
$$n \rightarrow (1/10)n \rightarrow (1/10)^2n \rightarrow \dots \rightarrow 1$$
- $\therefore n(1/10)^{sp} = 1$
- $\therefore n = 10^{sp}$ // take logs
- $\therefore \log_{10}n = sp$
- Long path node value:
$$n \rightarrow (9/10)n \rightarrow (9/10)^2n \rightarrow \dots \rightarrow 1$$
- $\therefore n(9/10)^{lp} = 1$
- $\therefore n = (10/9)^{lp}$ // take logs
- $\therefore \log_{10/9}n = lp$

Quicksort in Practice

- Quicksort is a great general-purpose sorting algorithm.
 - especially with a randomized pivot
 - Quicksort can benefit substantially from code tuning
 - Quicksort can be over twice as fast as merge sort
- Quicksort behaves well even with caching and virtual memory.

Timing Comparisons

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

- Running time estimates:
- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second

	insertion sort (N^2)			mergesort ($N \log N$)			quicksort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant