CSE214 – Analysis of Algorithms

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/Analysis-of-Algorithms-2019

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(nlogn)

- Mergesort and Quicksort

Quicksort Algorithm

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

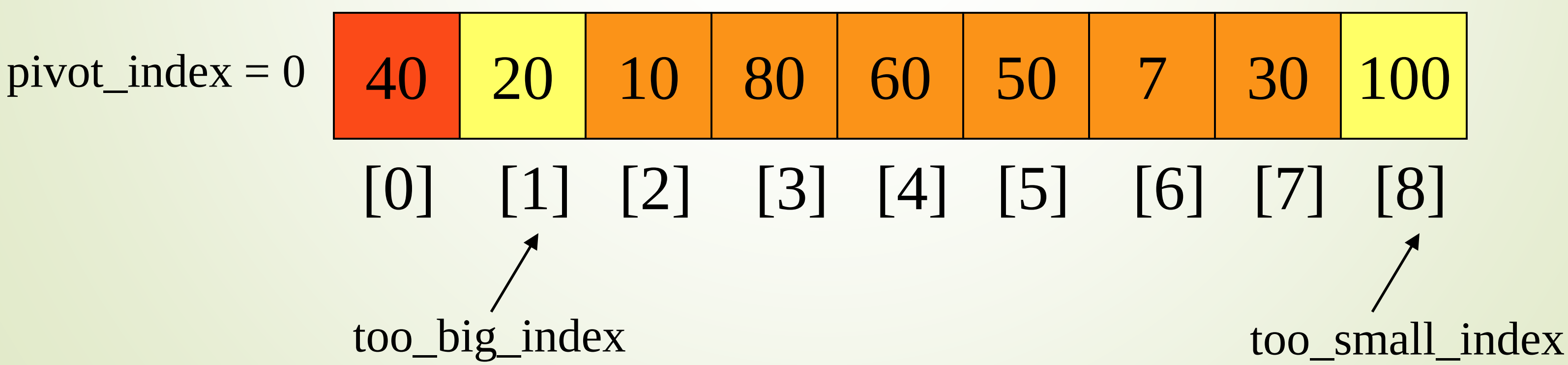
Partitioning Array

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

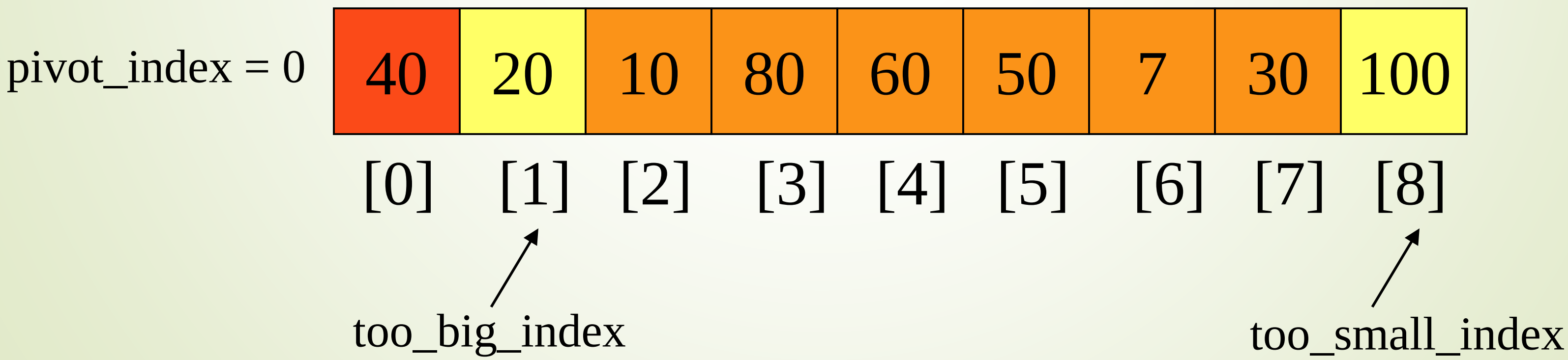
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

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

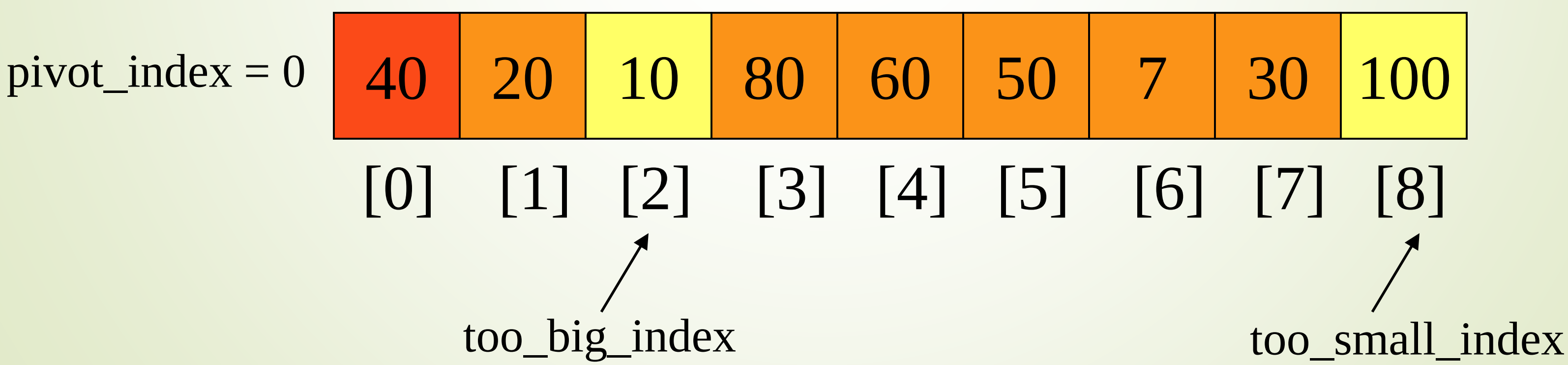
Partitioning loops through, swapping elements below/above pivot.


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

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

2. While data[too_small_index] > data[pivot]

   --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

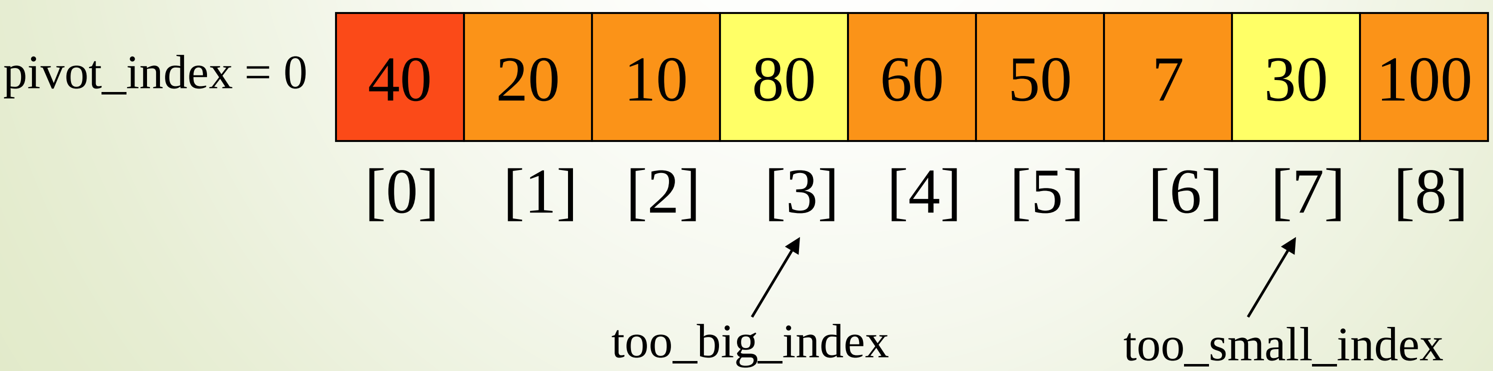
too_small_index

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

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index



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

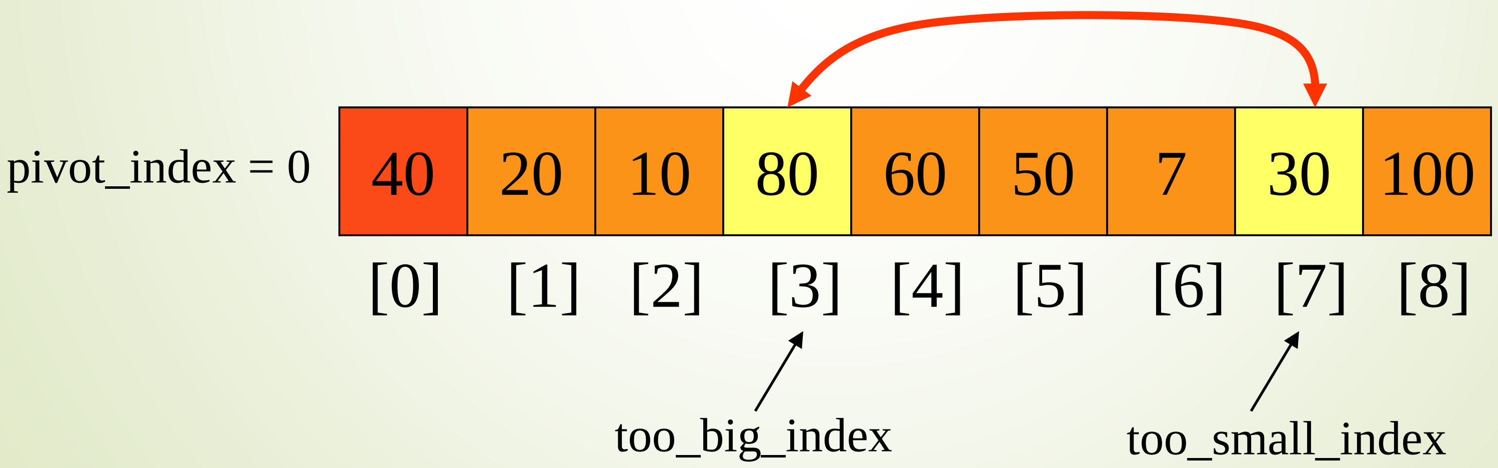
   ++too_big_index

2. While data[too_small_index] > data[pivot]

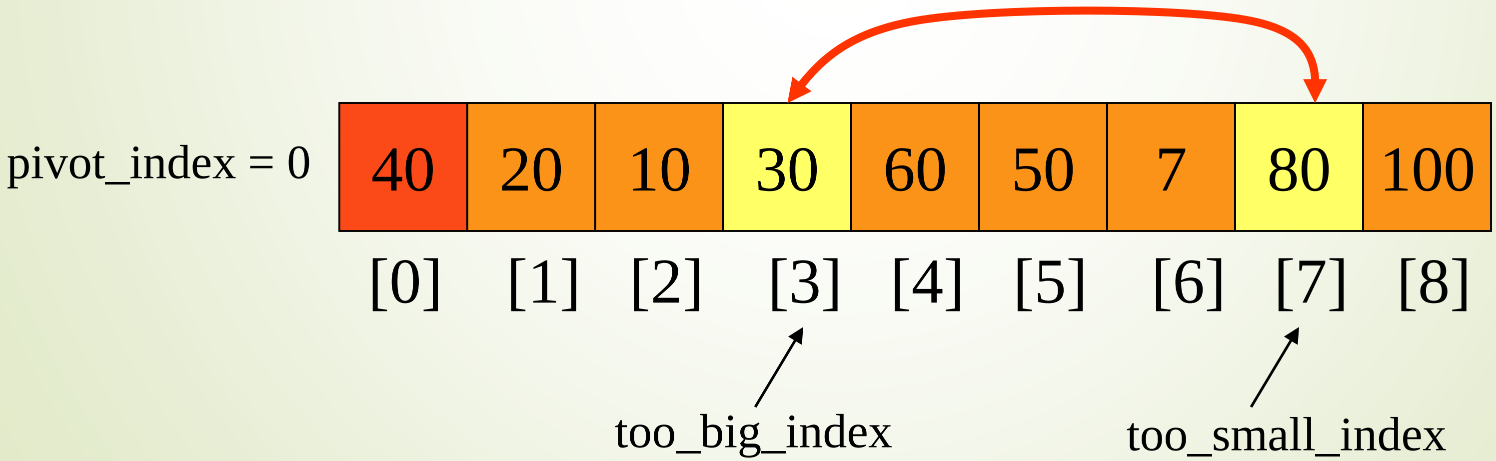
   --too_small_index

3. too_big_index < too_small_index

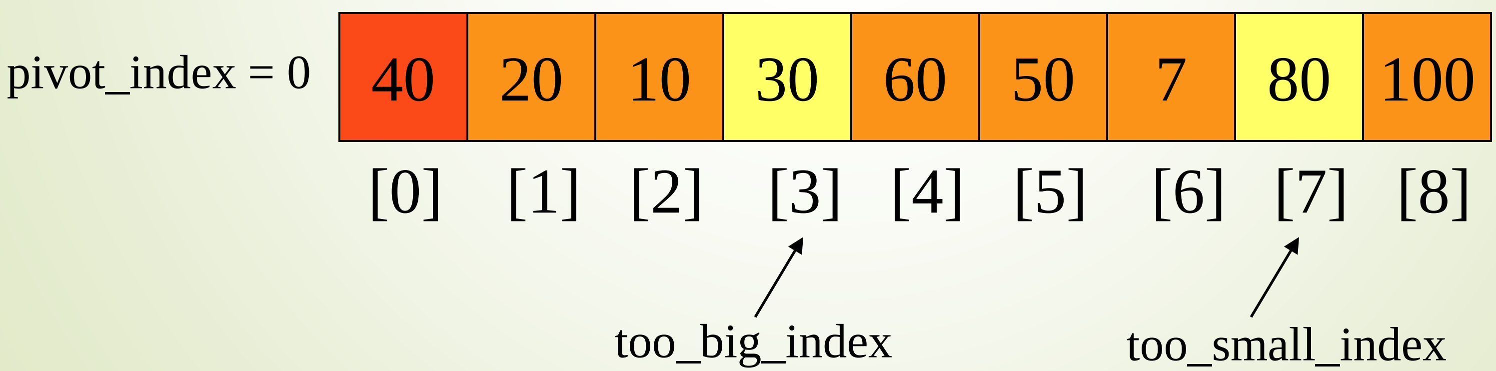
   swap data[too_big_index] and data[too_small_index]



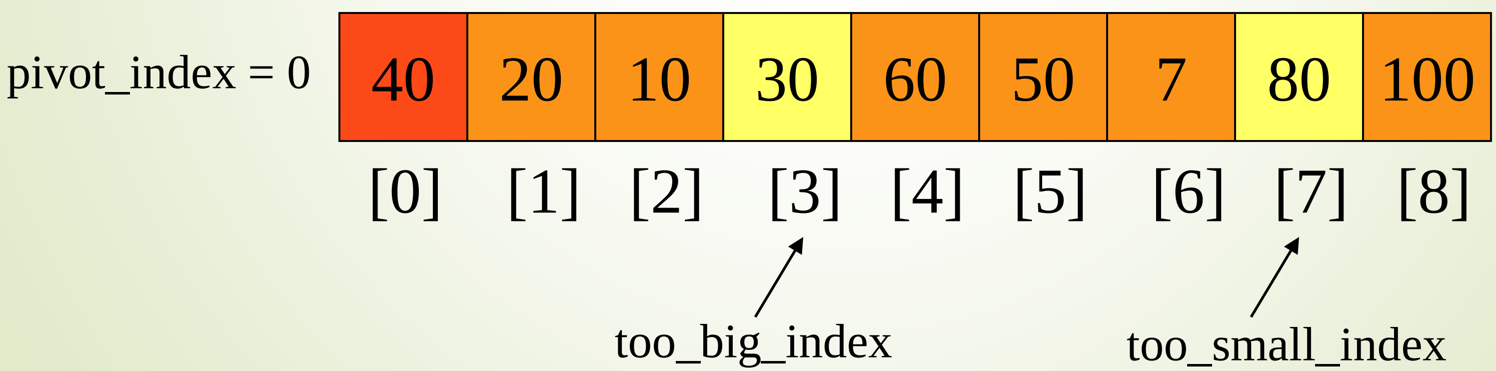
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



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.



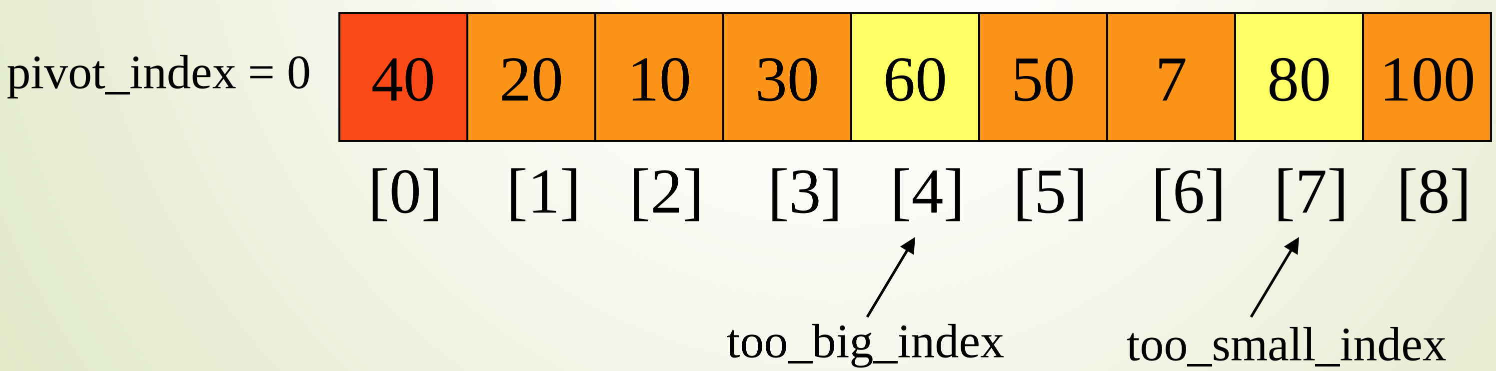
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

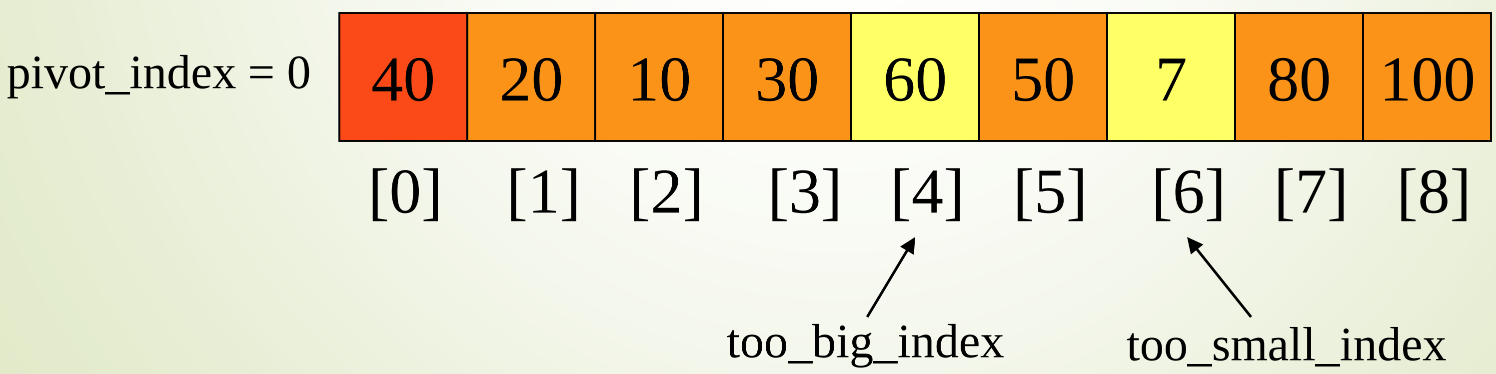
40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

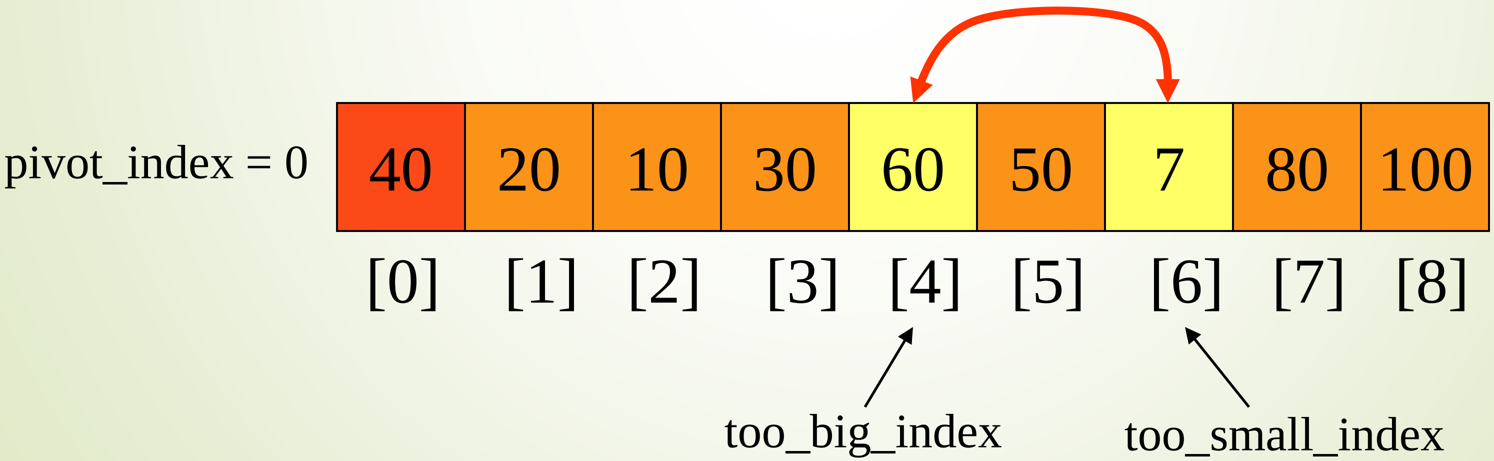
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



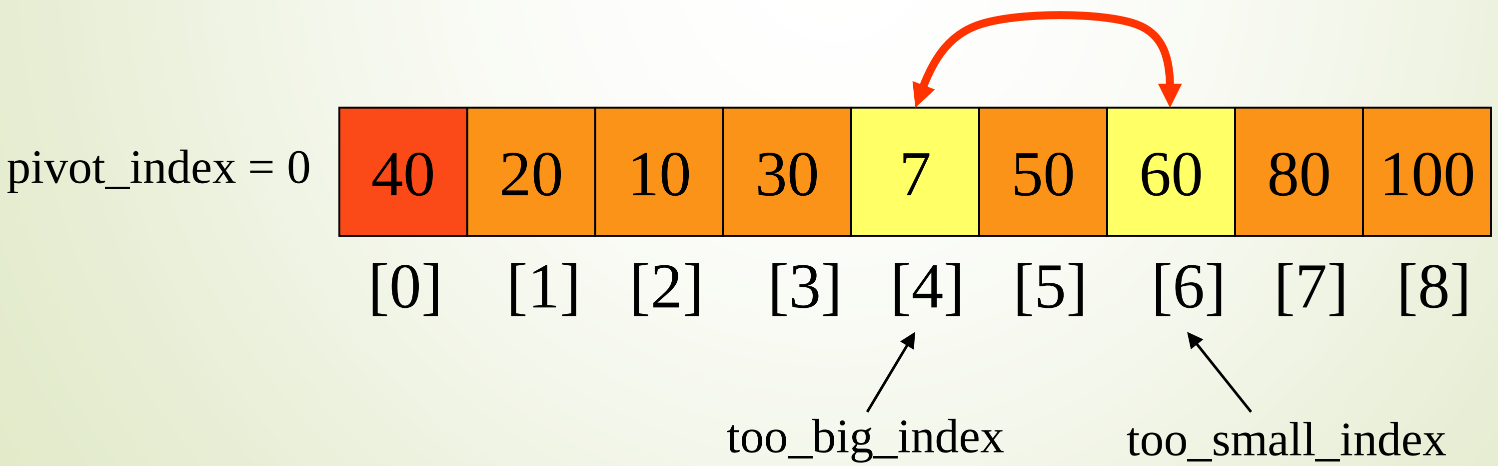
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



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.



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.

pivot_index = 0

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

too_big_index

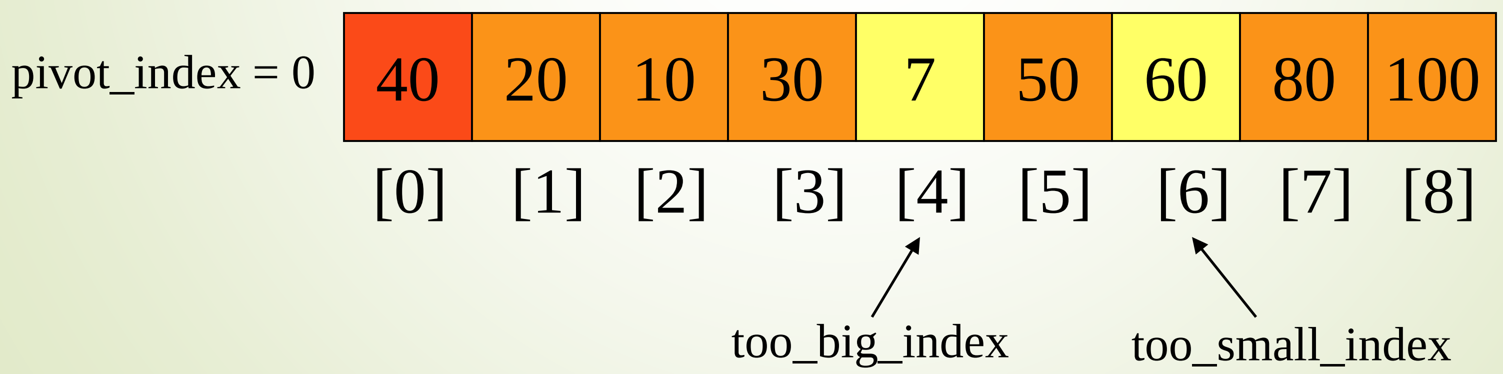
too_small_index

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.

pivot_index = 0

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

too_big_index too_small_index

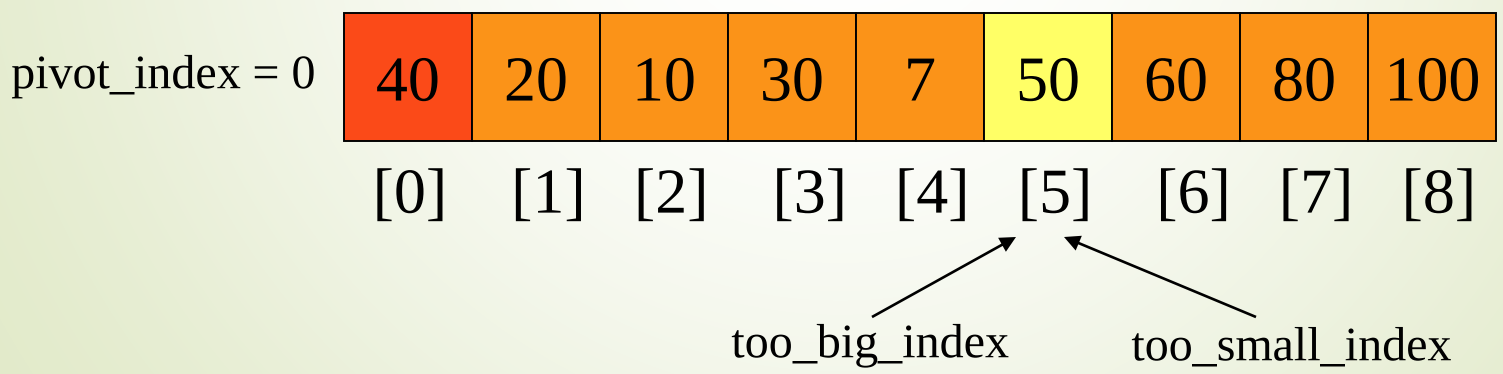
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

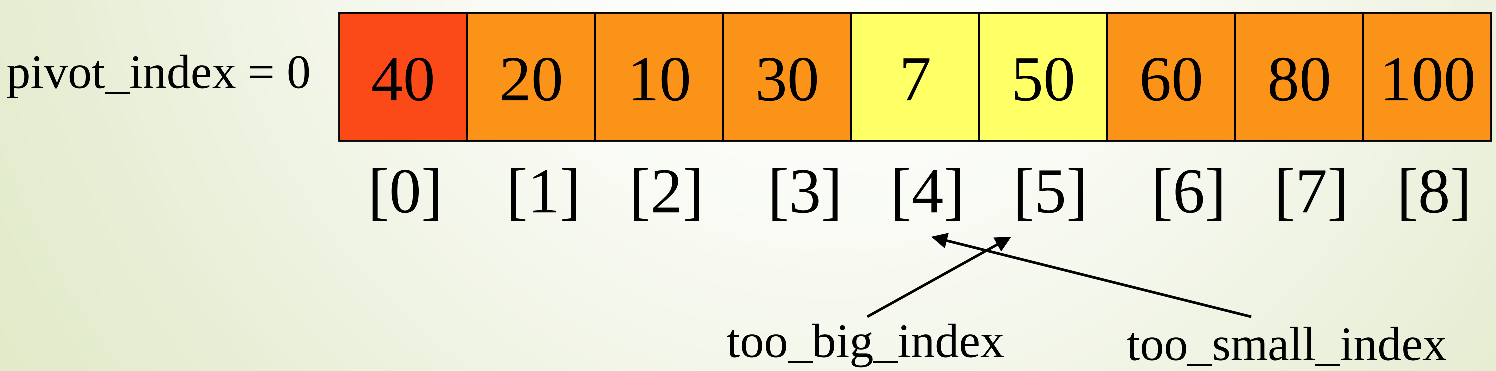
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

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.

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

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.

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

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]

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

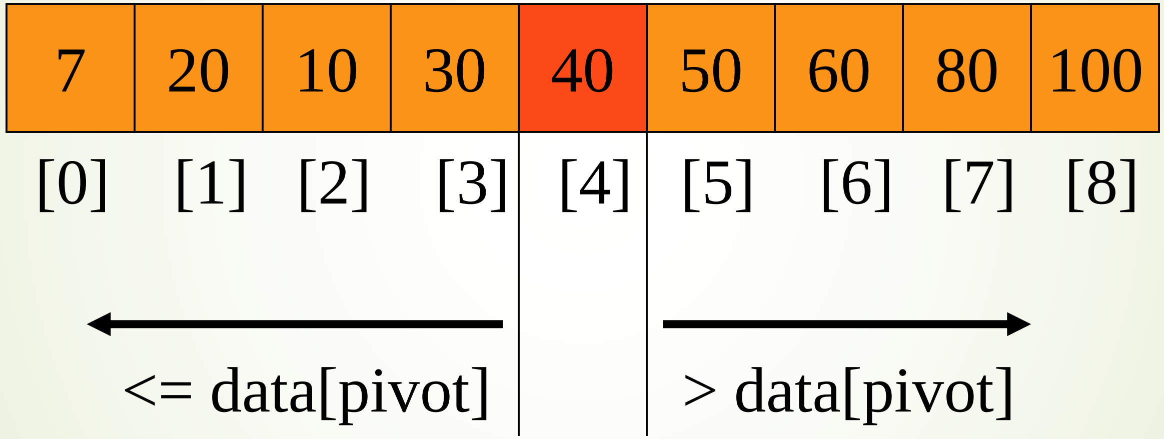
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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

pivot_index = 4

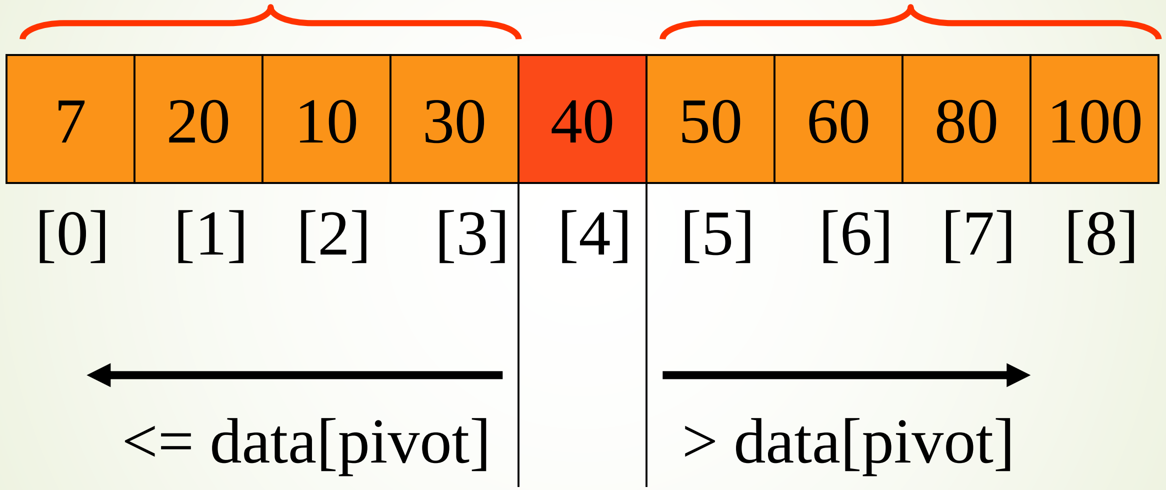
7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

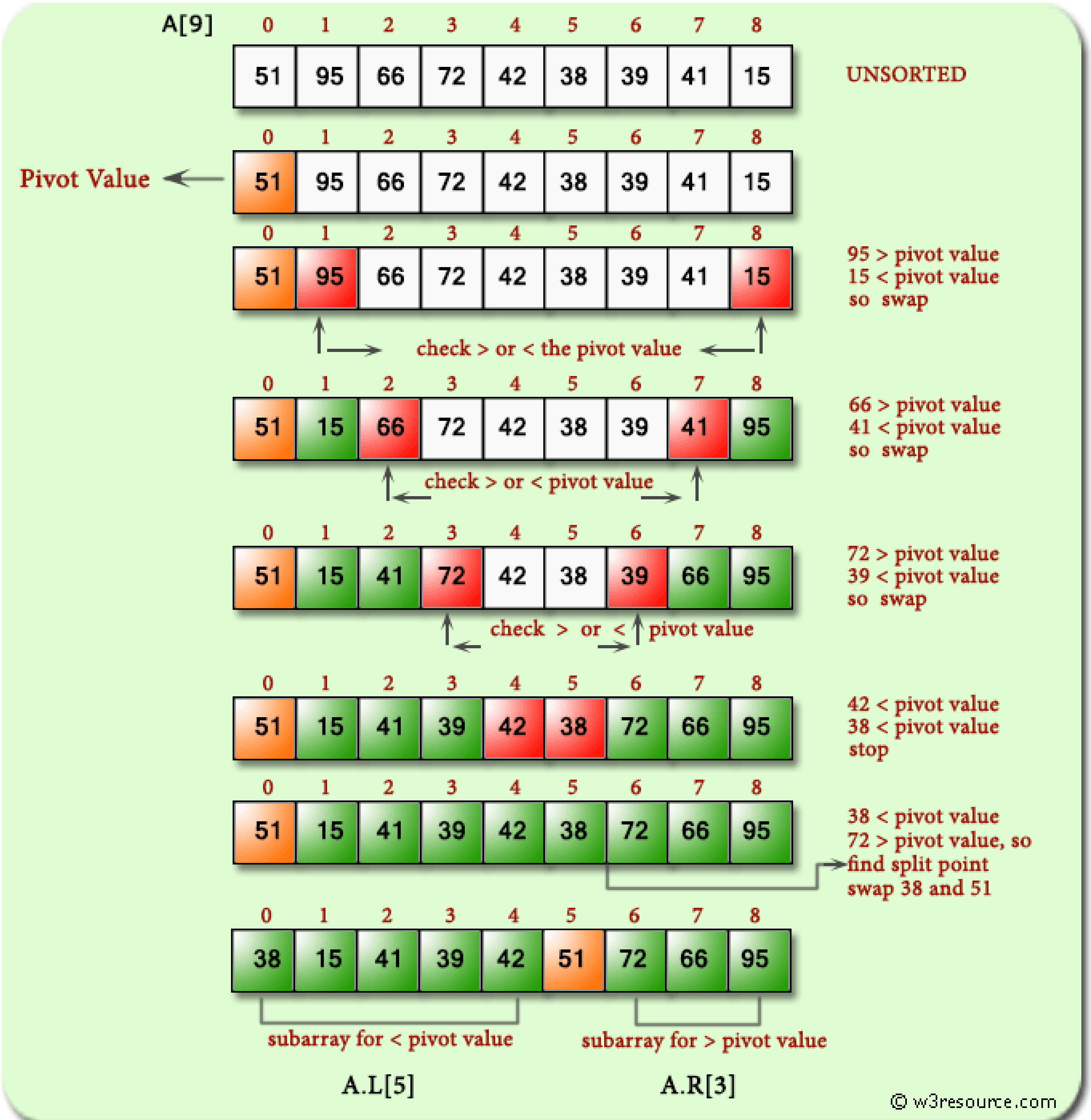
too_big_index too_small_index

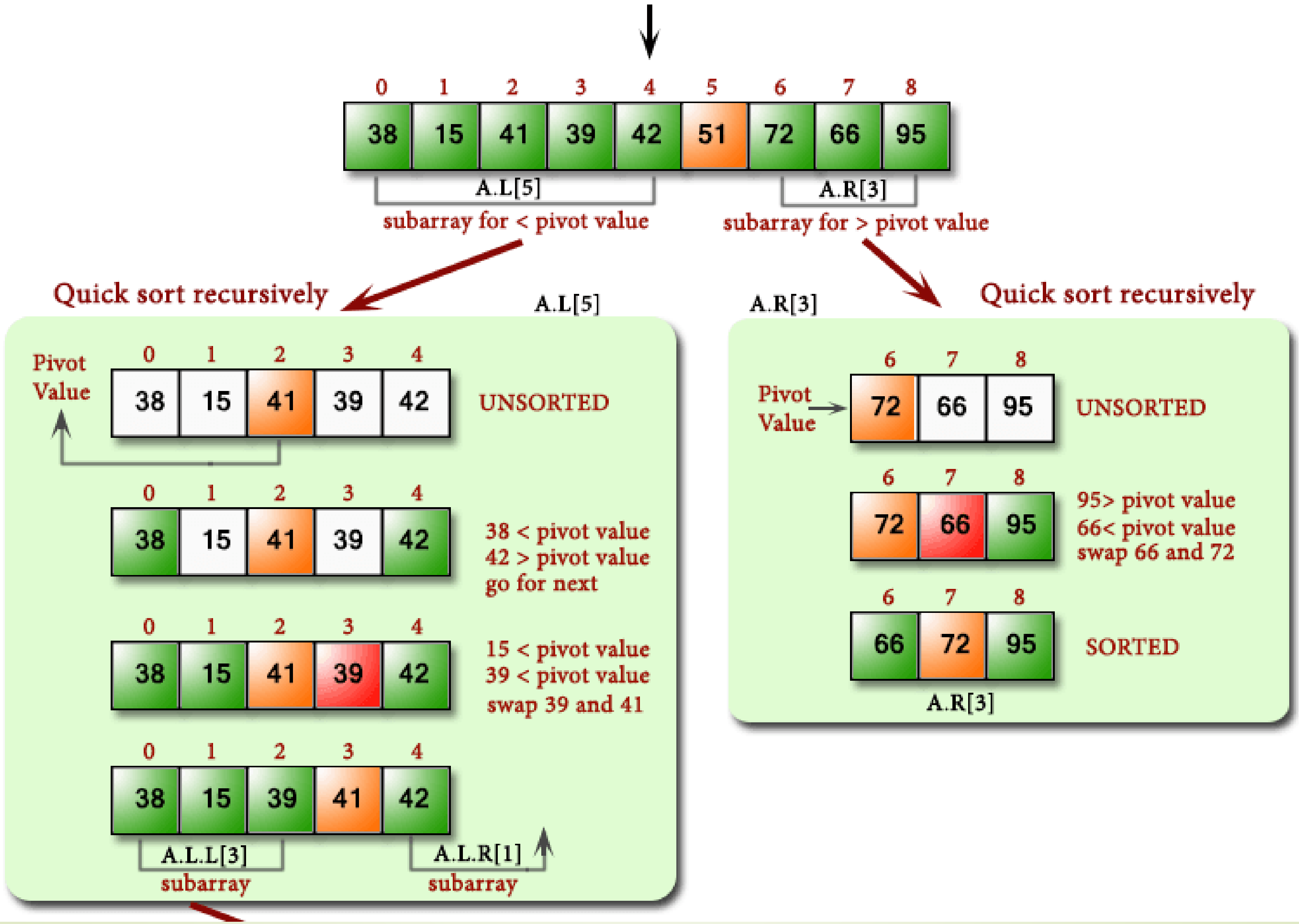
Partition Result



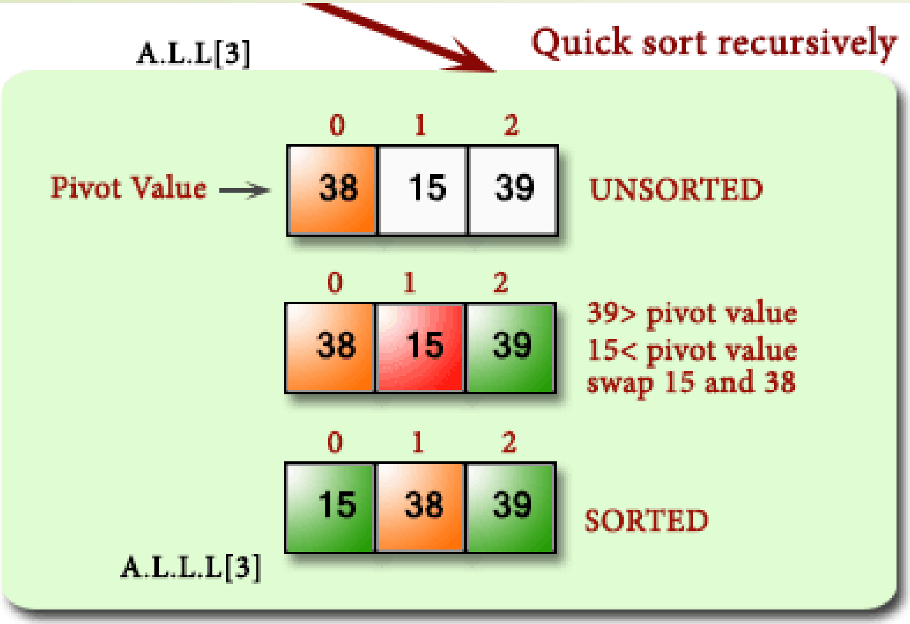
Recursion: Quicksort Sub-arrays



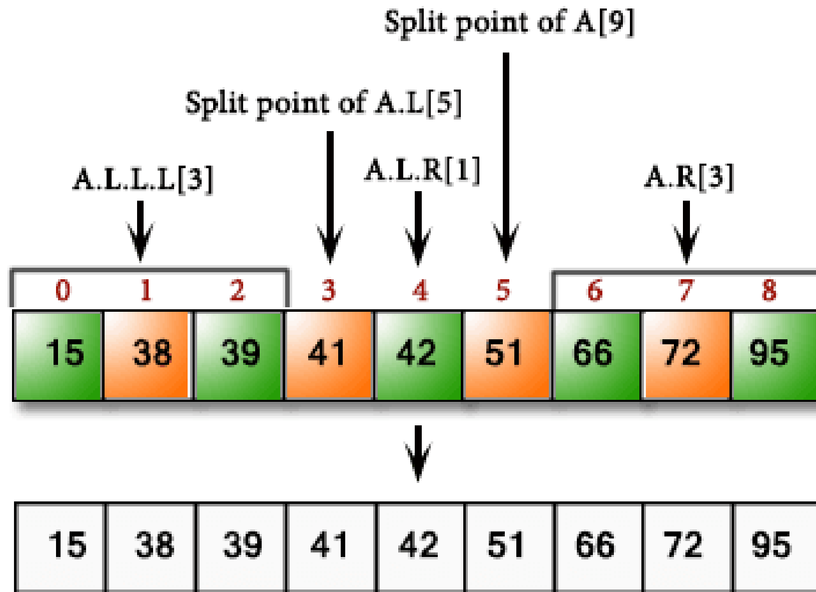

A[9]
0 1 2 3 4 5 6 7 8
51 95 66 72 42 38 39 41 15
UNSORTED
Pivot Value
51 95 66 72 42 38 39 41 15
51 95 66 72 42 38 39 41 15
95 > pivot value
15 < pivot value
so swap
check > or < the pivot value
51 15 66 72 42 38 39 41 95
66 > pivot value
41 < pivot value
so swap
check > or < pivot value
51 15 41 72 42 38 39 66 95
72 > pivot value
39 < pivot value
so swap
check > or < pivot value
51 15 41 39 42 38 72 66 95
42 < pivot value
38 < pivot value
stop
51 15 41 39 42 38 72 66 95
38 < pivot value
72 > pivot value, so
find split point
swap 38 and 51
38 15 41 39 42 51 72 66 95
subarray for < pivot value
subarray for > pivot value
A.L[5]
A.R[3]
© w3resource.com

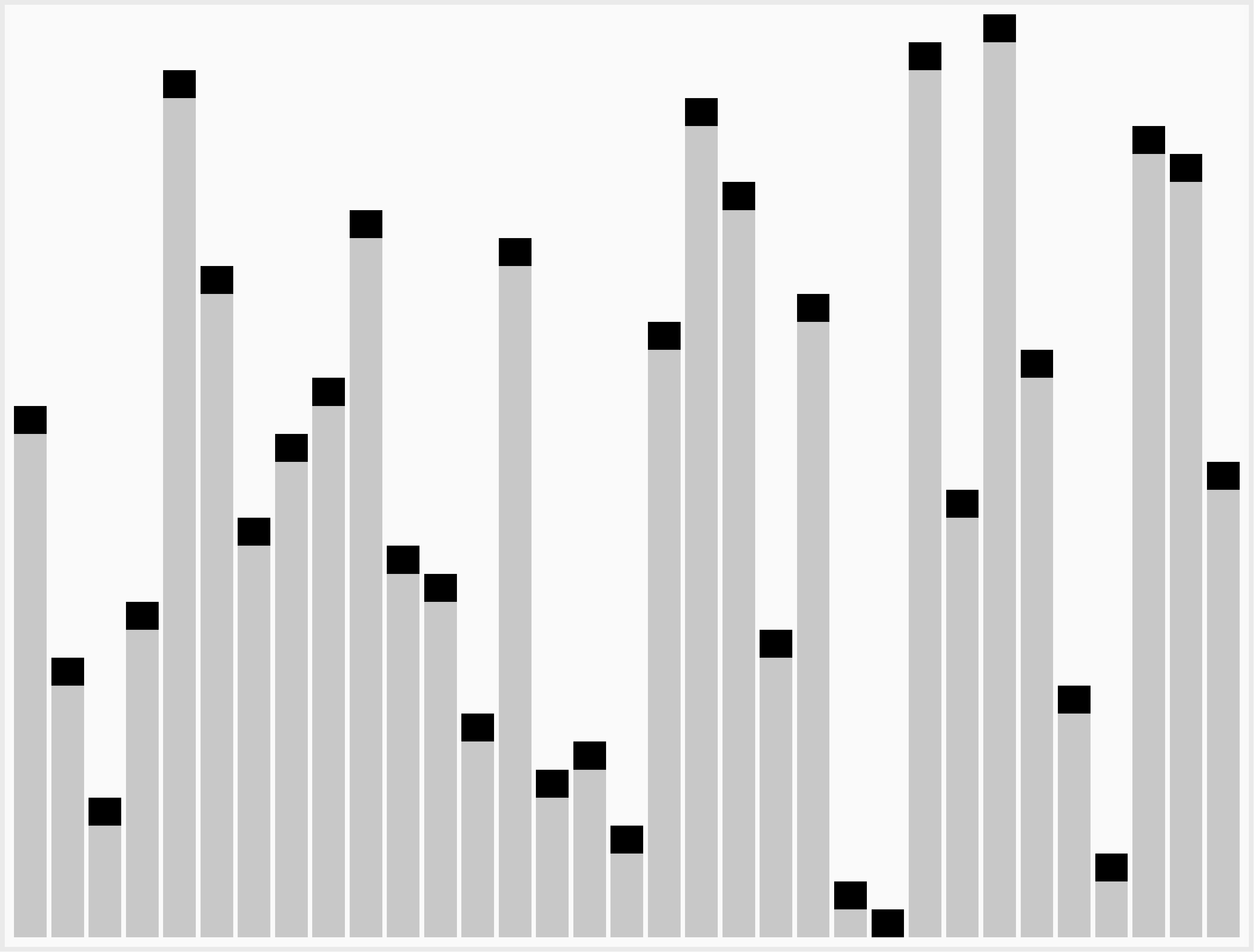


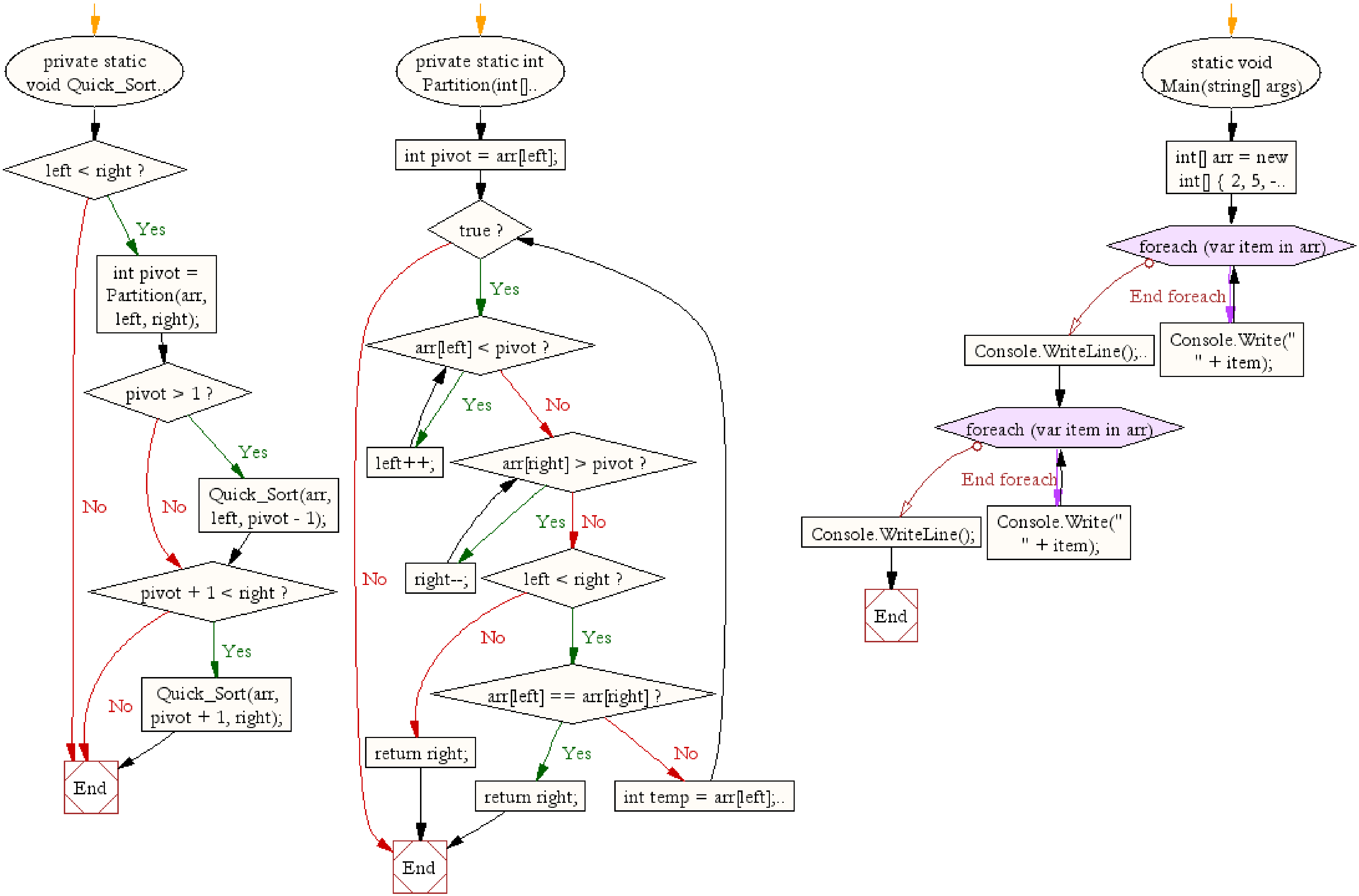
0 1 2 3 4 5 6 7 8
38 15 41 39 42 51 72 66 95
A.L[5]
subarray for < pivot value
A.R[3]
subarray for > pivot value
Quick sort recursively
A.L[5]
Pivot Value
38 15 41 39 42
UNSORTED
38 < pivot value
42 > pivot value
go for next
15 < pivot value
39 < pivot value
swap 39 and 41
38 15 39 41 42
A.L.L[3] subarray
A.L.R[1] subarray
Quick sort recursively
A.R[3]
6 7 8
72 66 95
UNSORTED
95> pivot value
66< pivot value
swap 66 and 72
66 72 95
SORTED
A.R[3]



FINAL SORTING







private static void Quick_Sort..
left < right ?
Yes
int pivot = Partition(arr, left, right);
pivot > 1 ?
Yes
Quick_Sort(arr, left, pivot - 1);
No
No
pivot + 1 < right ?
Yes
Quick_Sort(arr, pivot + 1, right);
No
End
private static int Partition(int[]..
int pivot = arr[left];
true ?
Yes
arr[left] < pivot ?
Yes
No
left++;
arr[right] > pivot ?
Yes
No
right--;
left < right ?
No
No
Yes
arr[left] == arr[right] ?
return right;
Yes
No
return right;
int temp = arr[left];..
End
static void Main(string[] args)
int[] arr = new int[] { 2, 5, -..
foreach (var item in arr)
End foreach
Console.WriteLine();..
Console.Write(" " + item);
foreach (var item in arr)
End foreach
Console.WriteLine();
Console.Write(" " + item);
End

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

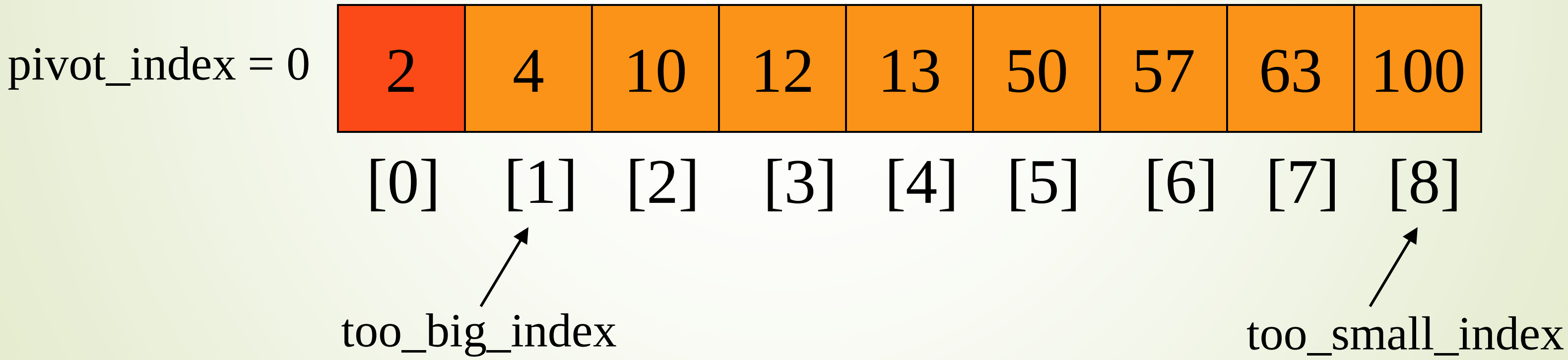
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

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 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]

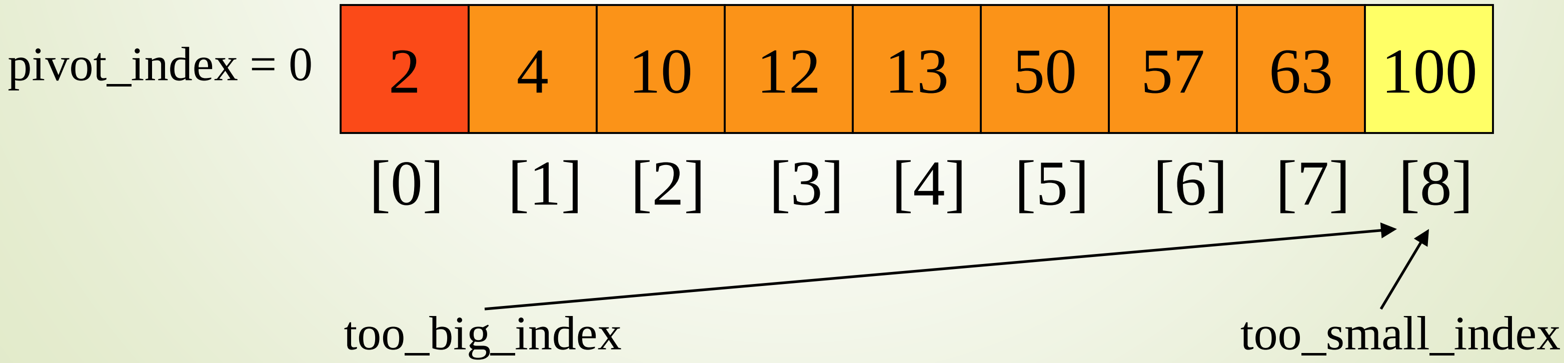
pivot_index = 0

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

too_big_index

too_small_index

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 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]

pivot_index = 0

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

too_big_index

too_small_index

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]

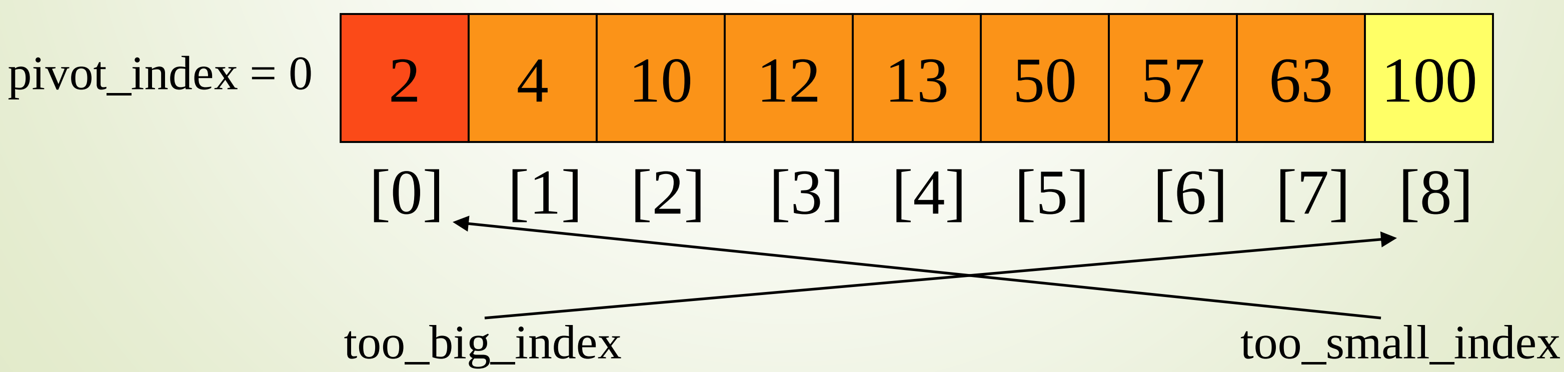
pivot_index = 0

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

too_big_index

too_small_index

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 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]

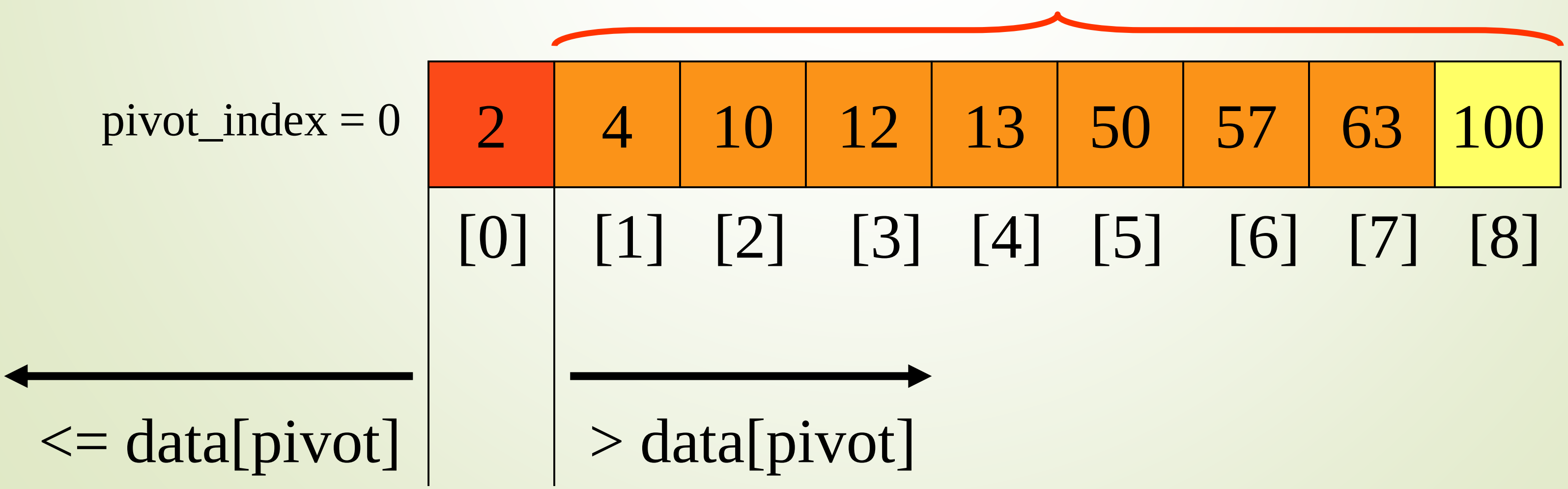
pivot_index = 0

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

too_big_index

too_small_index

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]



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:

data[0], data[n/2], and 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
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three

    median3
  - Pick A[center] as the pivot
  - Swap A[center] and A[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

2	5	6	4	13	3	12	19	6

A[left] = 2, A[center] = 13,
A[right] = 6

2	5	6	4	6	3	12	19	13

Swap A[center] and A[right]

2	5	6	4	6	3	12	19	13

Choose A[center] as pivot

pivot

2	5	6	4	19	3	12	6	13

Swap pivot and A[right – 1]

pivot

Note we only need to partition A[left + 1, …, right – 2]

Implementation of partitioning step

- Works only if pivot is picked as median-of-three.
  - A[left] <= pivot and A[right] >= pivot
  - Thus, only need to partition A[left + 1, …, right – 2]
- j will not run past the end
  - because a[left] <= pivot
- i will not run past the end
  - because a[right-1] = 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 );

        // 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;
    }

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

    quicksort( a, left, i - 1 );      // Sort small elements
    quicksort( a, i + 1, right );     // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

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(n2) 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 has no elements. Therefore:

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

no elements

n-1 elements

Worst-case recursion tree

$$T(n) = T(0) + T(n\text{-}1) + cn$$

Worst-case recursion tree

$T(n) = T(0) + T(n\text{-}1) + cn$

$T(n)$

Worst-case recursion tree

$$T(n) = T(0) + T(n\text{-}1) + cn$$

$$cn$$

$$T(0) \qquad\qquad T(n\text{-}1)$$

Worst-case recursion tree

$T(n) = T(0) + T(n\text{-}1) + cn$

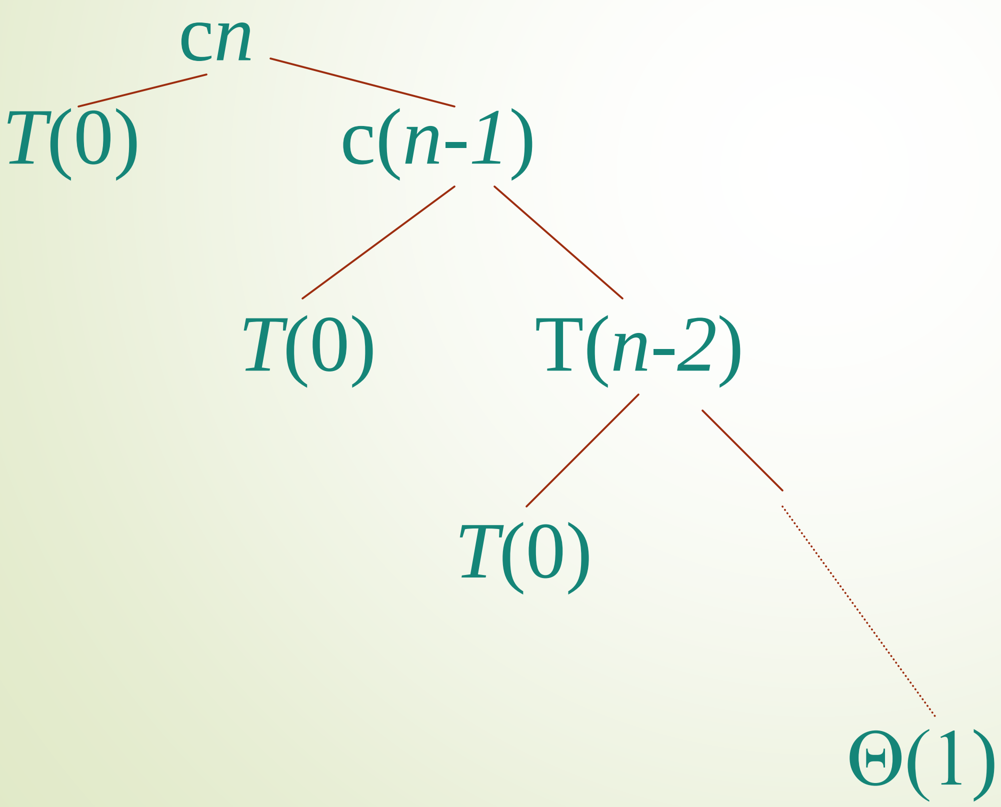
cn

$T(0)$ $\quad\quad$ $c(n\text{-}1)$

$T(0)$ $\quad$ $T(n\text{-}2)$

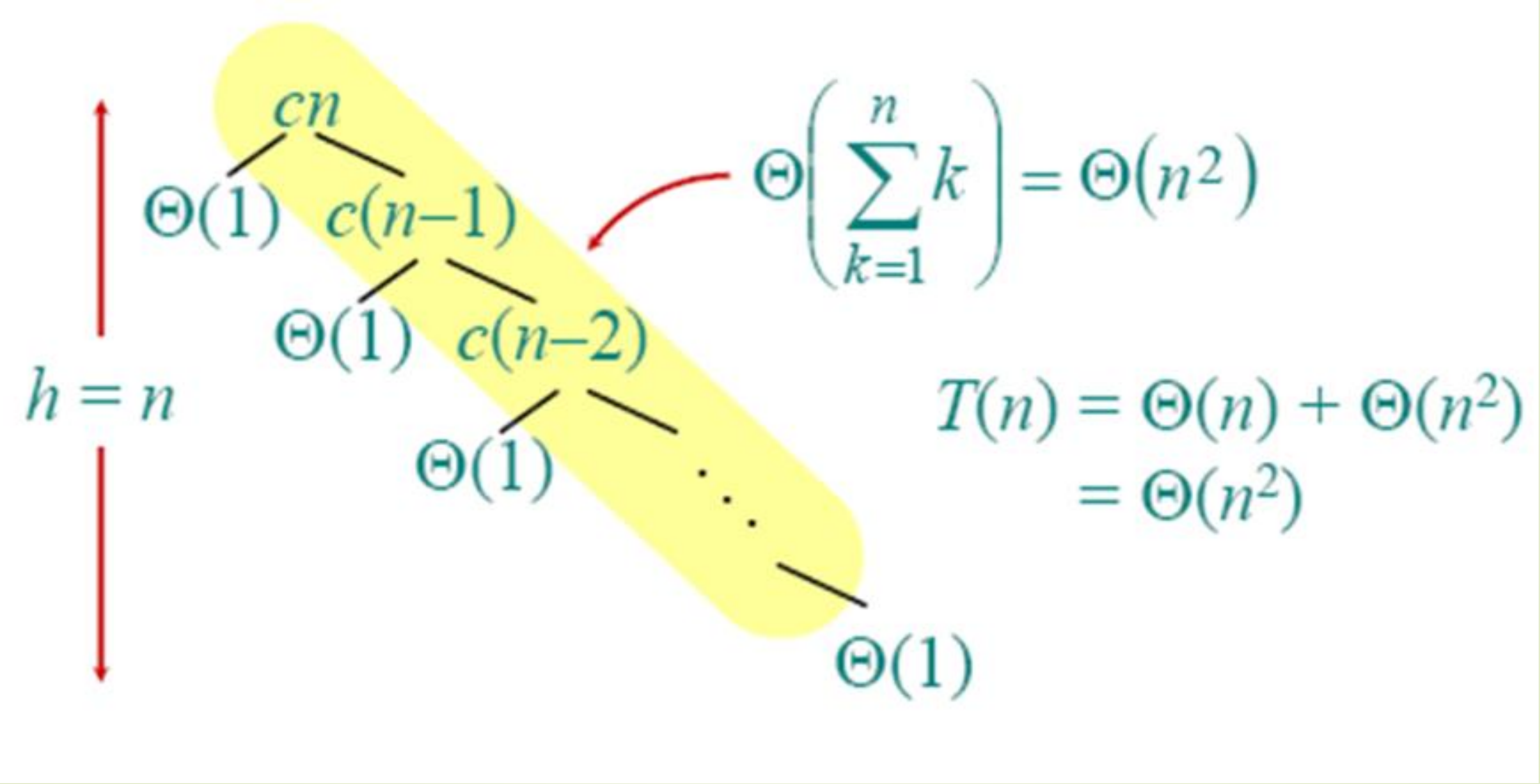
Worst-case recursion tree

$$T(n) = T(0) + T(n\text{-}1) + cn$$



Worst-case recursion tree

$$T(n) = T(0) + T(n\text{-}1) + cn$$



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:
- $T(n) = 2T(n/2) + \Theta(n)$
- $= \Theta(n \log n)$ (same as merge sort)

Case 2 of the Master Method

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

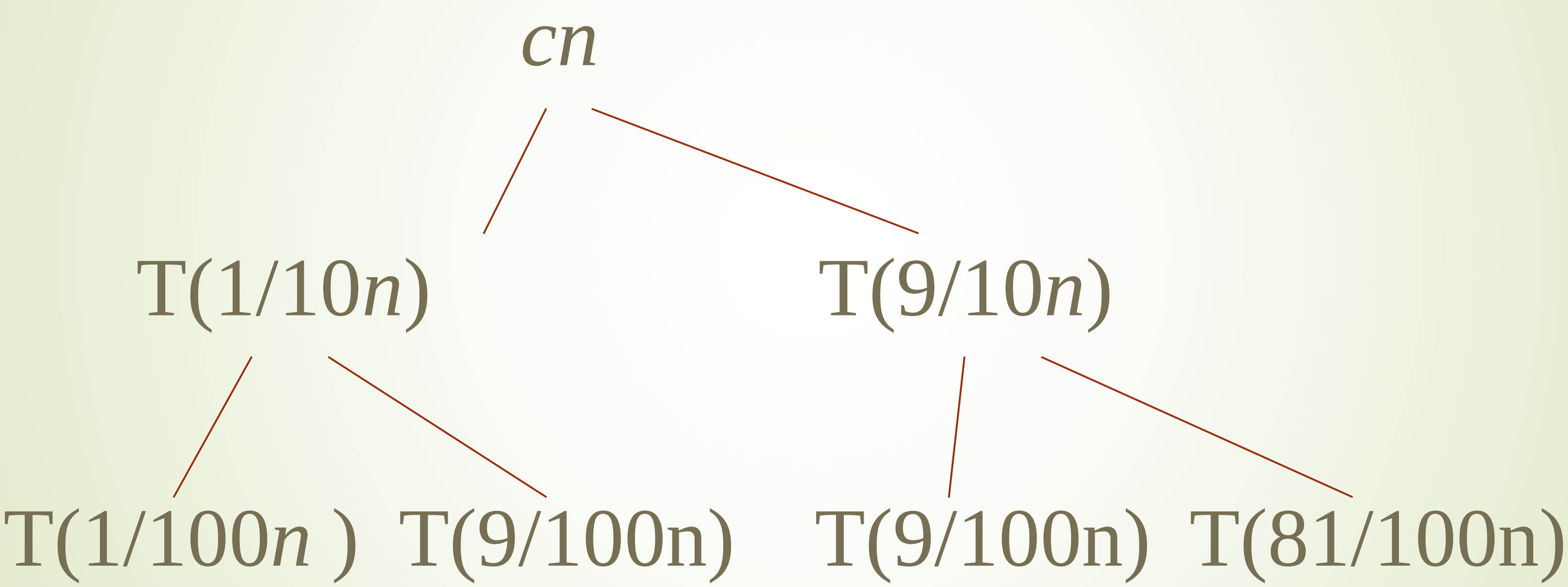
T(n)

Analysis of “almost-best” case

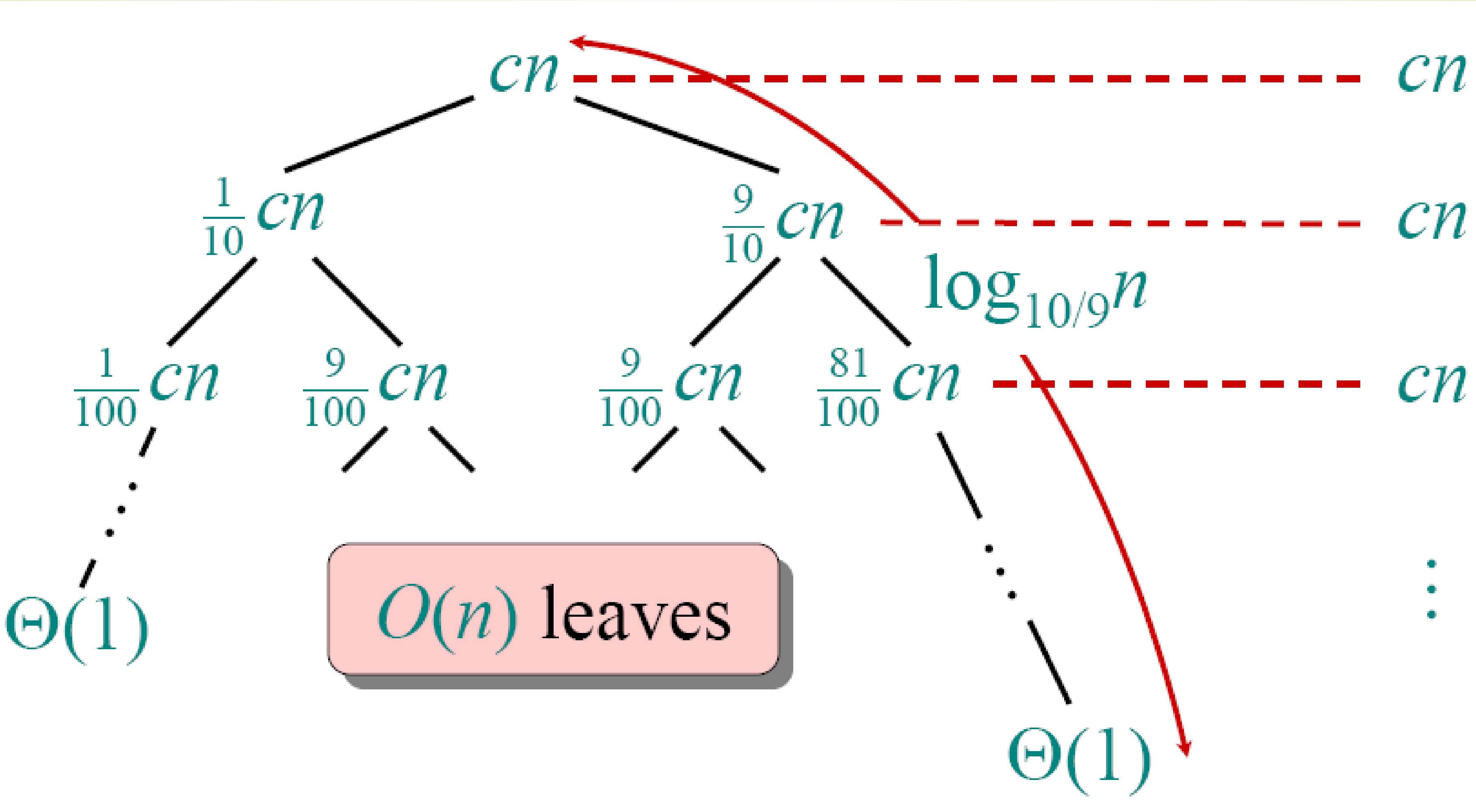
cn

T(1/10n) T(9/10n)

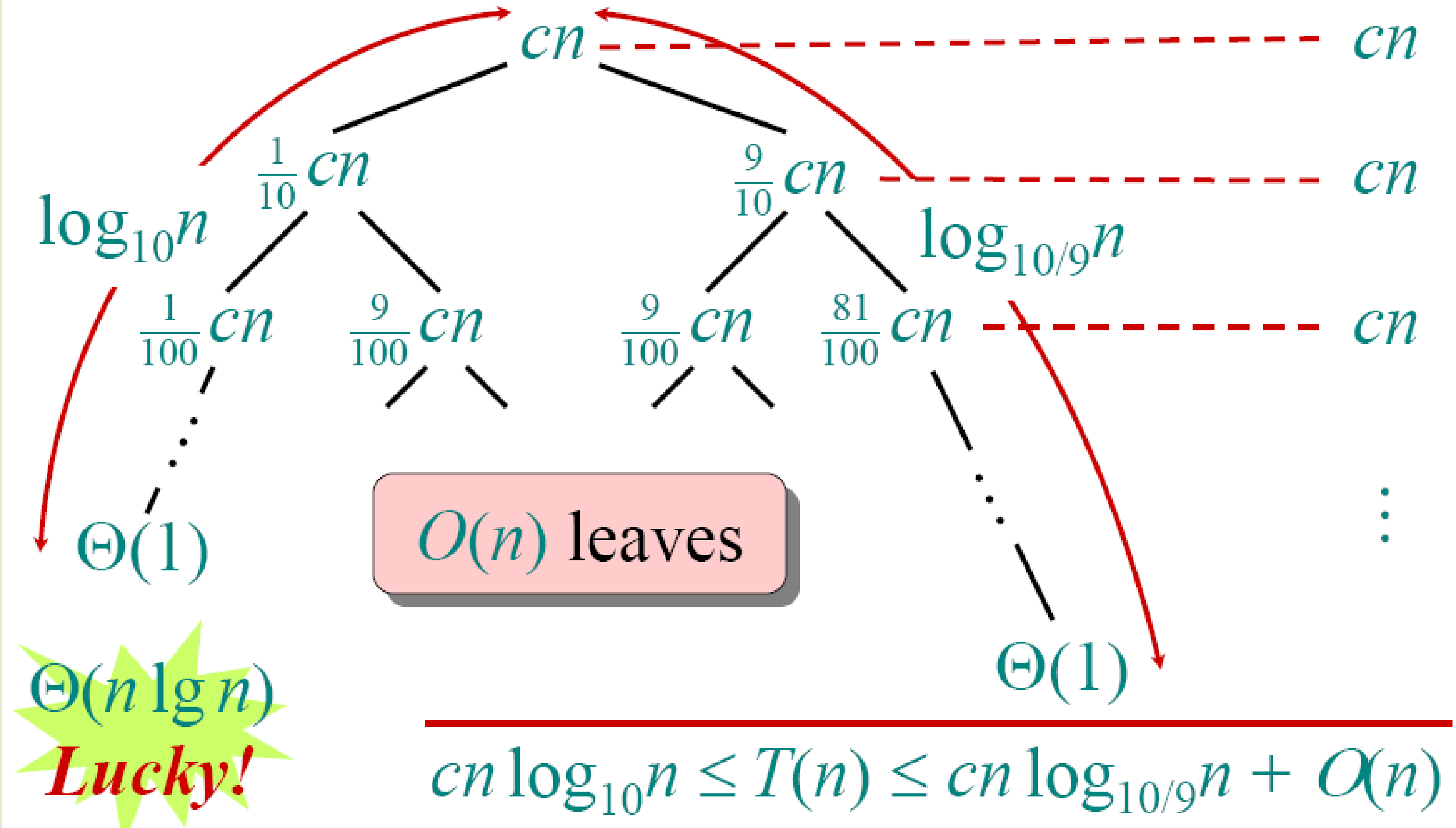
Analysis of “almost-best” case



Analysis of “almost-best” case



Analysis of “almost-best” case



$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

Short and Long Path Heights

- Short path node value:
  $n \to (1/10)n \to (1/10)^2 n \to \dots \to 1$

- $\therefore\ n(1/10)^{sp} = 1$
- $\therefore\ n = 10^{sp}$ // take logs
- $\therefore\ \log_{10} n = sp$

- Long path node value:
  $n \to (9/10)n \to (9/10)^2 n \to \dots \to 1$

- $\therefore\ n(9/10)^{lp} = 1$
- $\therefore\ n = (10/9)^{lp}$ // 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

Quickselect

- Quickselect algorithm is used to find the i-th smallest element in a given unordered array

- Randomized algorithm using divide and conquer
- Similar to randomized quicksort
  - *Like quicksort*: Partitions input array recursively
  - *Unlike quicksort*: Makes a single recursive call
    - *Reminder: Quicksort makes two recursive calls*

- Expected runtime: $\Theta(n)$
    - *Reminder: Expected runtime of quicksort: $\Theta(nlgn)$*

Selection in Expected Linear Time: Example 1

Select the 2nd smallest element:

6	10	13	5	8	3	2	11

$i = 2$

Partition the input array:

2	3	5	13	8	10	6	11

make a recursive call to select the 2nd smallest element in left subarray

Selection in Expected Linear Time: Example 2

Select the 7^{th} smallest element:

6	10	13	5	8	3	2	11

$i = 7$

Partition the input array:

2	3	5	13	8	10	6	11

make a recursive call to select the 4^{th} smallest element in right subarray

Selection in Expected Linear Time

R-SELECT($\mathbf{A}, p, r, i$)

  if $p = r$ then

    return $\mathbf{A}[p]$

$q \leftarrow$ R-PARTITION($\mathbf{A}$, p, r)

k $\leftarrow q - p + 1$

if $i \leq k$ then

  return R-SELECT($\mathbf{A}$, p, q, i)

else

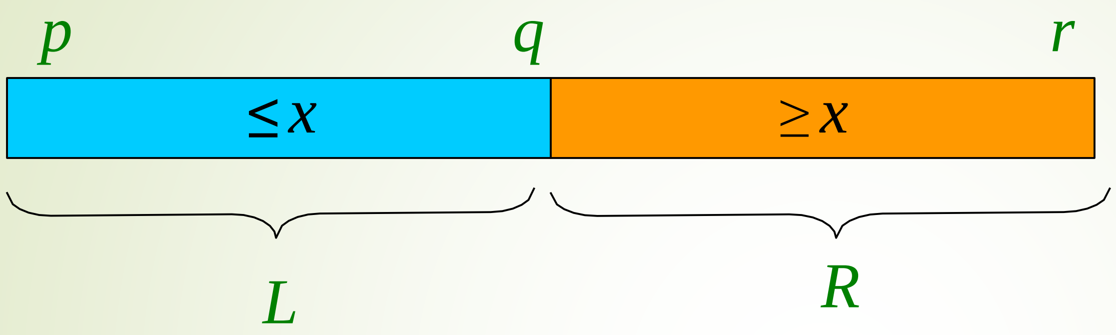
  return R-SELECT($\mathbf{A}$, q+1, r, i-k)

$\leq x$ (k smallest elements)	$\geq x$

p q r

$x = pivot$

Selection in Expected Linear Time



$x = pivot$

- All elements in L $\leq$ all elements in R
- L contains $|L| = q-p+1$ = k smallest elements of A[$p...r$]

if $i \leq |L| = k$ then

search L recursively for its i-th smallest element

else

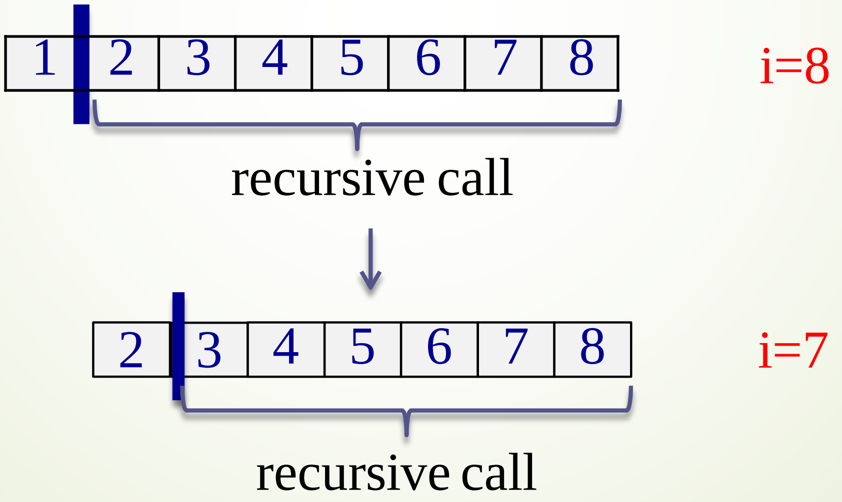
search R recursively for its (i-k)-th smallest element

Runtime Analysis

- ***Worst case:***

Imbalanced partitioning at every level

and the recursive call always to the larger partition



Runtime Analysis

- ***Worst case:***

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

  ➔ $T(n) = \Theta(n^2)$

  Worse than the naïve method (based on sorting)

- ***Best case***: Balanced partitioning at every recursive level

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

  ➔ $T(n) = \Theta(n)$

- ***Avg case***: Expected runtime – need analysis