ECE244

Wael Aboulsaadat

Analysis of Algorithms

Big-Oh
Lecture 2

Acknowledgment: these slides are partially based on slides by; Prof. Schmidt from Drexel U., Prof. Shewchuk from UC Berkely, Kruse & Ryba Data Structure and Program Design in C++, Prof. Savitch Problem Solving in C++ and others

Asymptotic Notation

- **By now you should have an intuitive feel for asymptotic (big-O) notation:**
  - *What does O(n) running time mean? O(n^2)? O(n lg n)?*

Rules for big Oh Analysis

1. We assume an arbitrary time unit.

2. Running of each of the following type of statement takes time 1 or T(1): [*omitting the arithmetic operators*]
   1. assignment statement
   2. I/O statement
   3. Boolean expression evaluation
   4. function return

3. Running time of a selection statement (if, switch) is T(1) for the condition evaluation + the maximum of the running times for the individual clauses in the selection.

More Rules

4. Loop execution time is the time for the loop setup (initialization & setup) + the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations.

   † Always assume that the loop executes the maximum number of iterations possible

5. Running time of a function call is T(1) for function setup + the time required for the execution of the function body.

6. Running time of a sequence of statements is the largest time of any statement in the sequence.

Summing an Array: What is the big Oh?

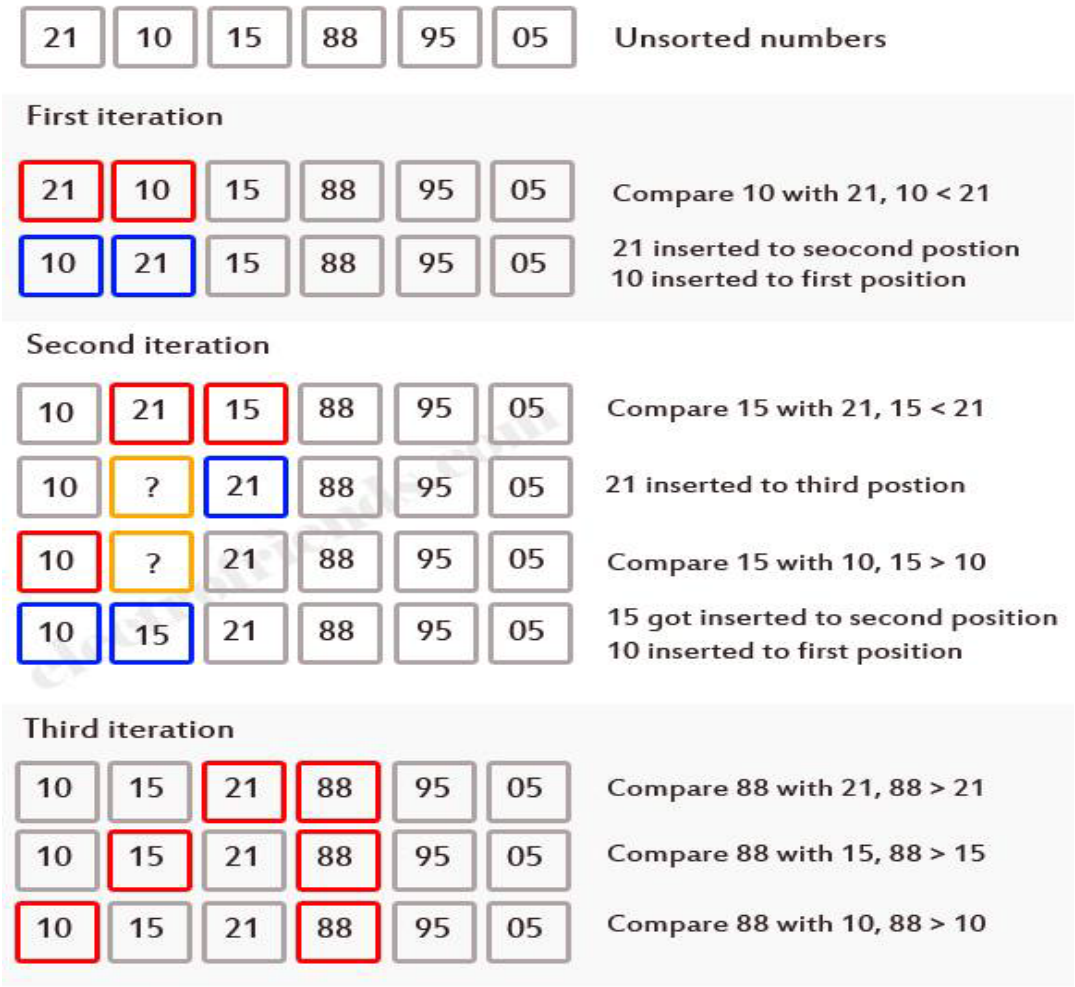
```
Line 1:     sum = 0;
Line 2:     for (i = 0; i < n; i++)
Line 3:         sum += a[i];
Line 4:     cout << sum;
```

How many steps are executed when this code runs?

Line 1:	1	
Line 4:	1	
Line 3:	n	
Line 2:	n +1	
Total:	2n +3	➔ O(n)

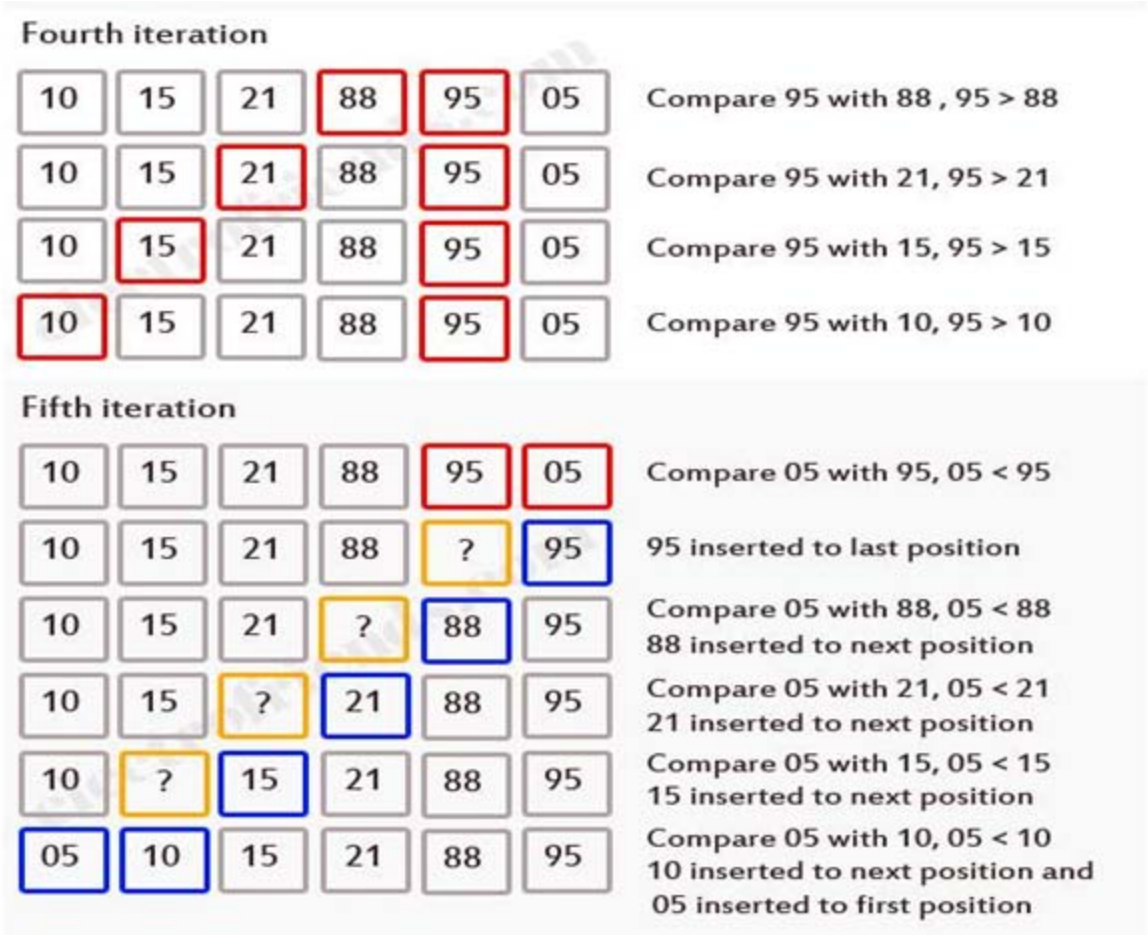
Insertion Sort: What is the big Oh?

- Trace



Insertion Sort: What is the big Oh?

- Trace cont'd



Insertion Sort: What is the worst case?

- Data: in reverse order

```
Line 1:   for (j=1; j<n; ++j) {
Line 2:       key = A[j];
Line 3:       i = j-1;
Line 4:       for (  ; 0≤i && key<A[i]; --i){
Line 5:                 A[i+1] = A[i];
Line 6:                 --i;
              }
Line 7:       A[i+1] = key;
          }
```

Insertion Sort: What is the worst case?

- Data: in reverse order

Line 1:	for (j=1; j<n; ++j) {	n
Line 2:	key = A[j];	n-1
Line 3:	i = j-1;	n-1
Line 4:	for (; 0≤i && key<A[i]; --i){	Sum j=1..(n-1) of t_j
Line 5:	A[i+1] = A[i];	Sum j=1..(n-1) of (tj-1)
Line 6:	--i;	Sum j=1..(n-1) of (tj-1)
	}	
Line 7:	A[i+1] = key;	n-1
	}	

Insertion Sort: What is the worst case?

- Data: in reverse order

Line 1: for (j=1; j<n; ++j) {	n
Line 2: key = A[j];	n-1
Line 3: i = j-1;	n-1
Line 4: for (; 0≤i && key<A[i]; --i){	Sum j=1..(n-1) of t_j
Line 5: A[i+1] = A[i];	Sum j=1..(n-1) of (tj-1)
Line 6: --i;	Sum j=1..(n-1) of (tj-1)
}	
Line 7: A[i+1] = key;	n-1
}	

$T(n) = n + (n-1) + (n-1) +$
$(\text{sum } j=1..(n-1) \text{ of } t_j) + (\text{sum } j=1..(n-1) \text{ of } t_j\text{-}1) + (\text{sum } j=1..(n-1) \text{ of } t_j\text{-}1)$
$+ (n-1)$
$= an^2 + bn + c \quad \rightarrow O(n^2)$

Insertion Sort: What is the best Case?

- Data: already sorted

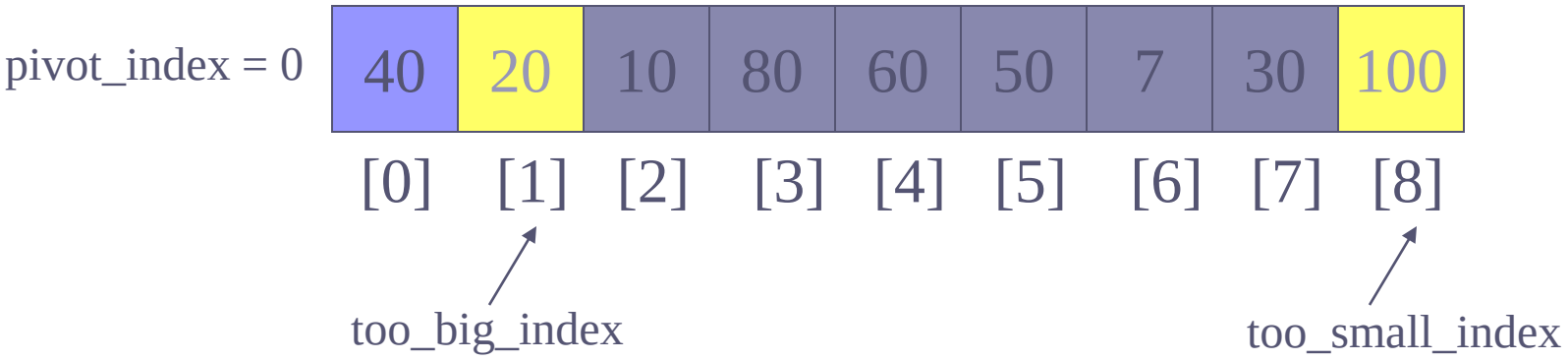
```
Line 1:   for (j=1; j<n; ++j) {                          n
Line 2:       key = A[j];                                n-1
Line 3:       i = j-1;                                   n-1
Line 4:       for (  ; 0≤i && key<A[i]; --i){            1
Line 5:               A[i+1] = A[i];                     0
Line 6:               --i;                               0
              }
Line 7:       A[i+1] = key;                              n-1
          }
```

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

$= bn + c \quad \rightarrow O(n)$

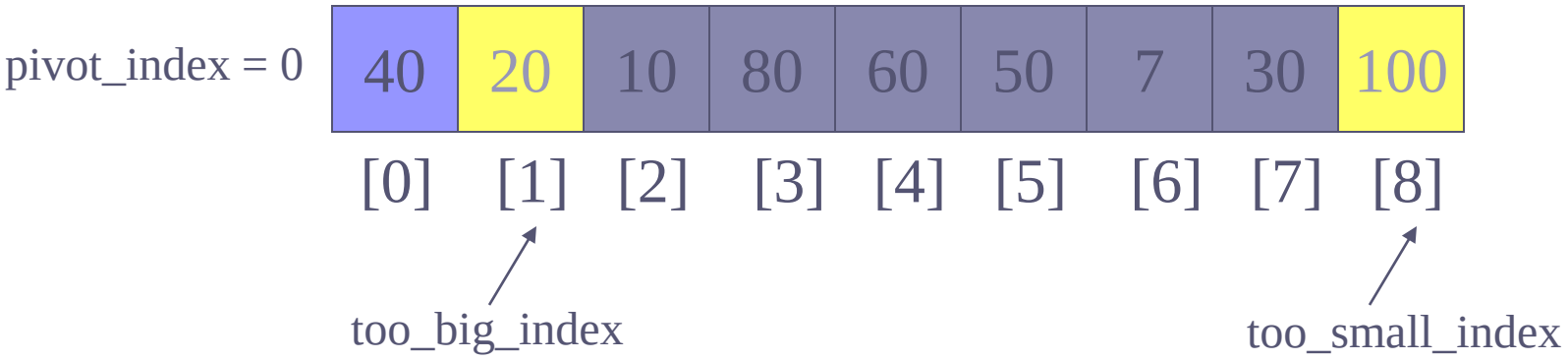
Quick Sort: What is the big Oh?

- Trace



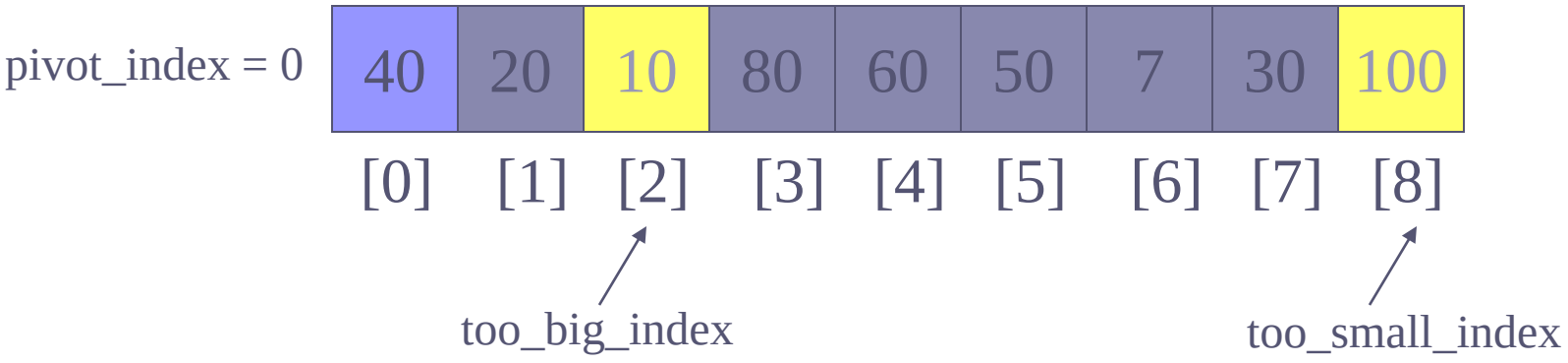
Recall QuickSort

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



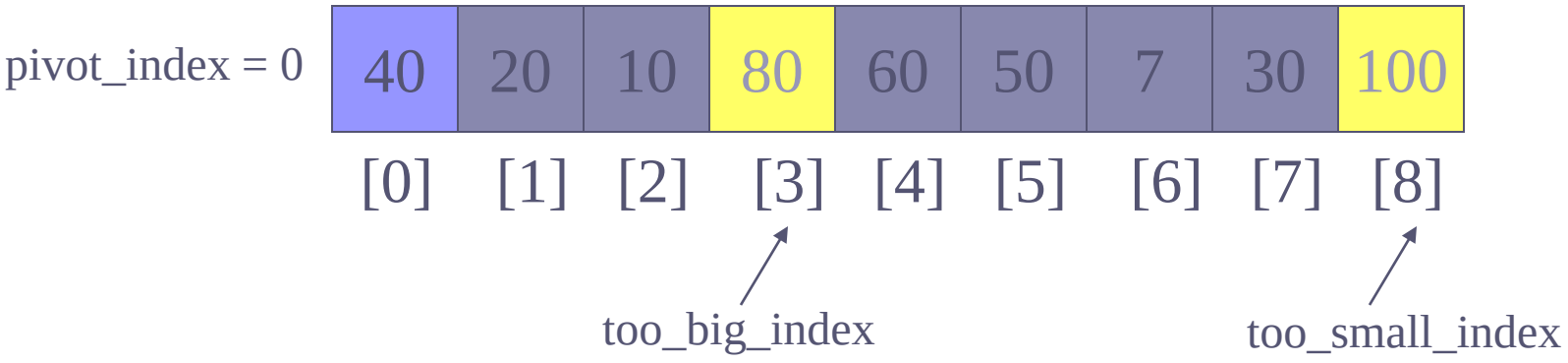
Recall QuickSort

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



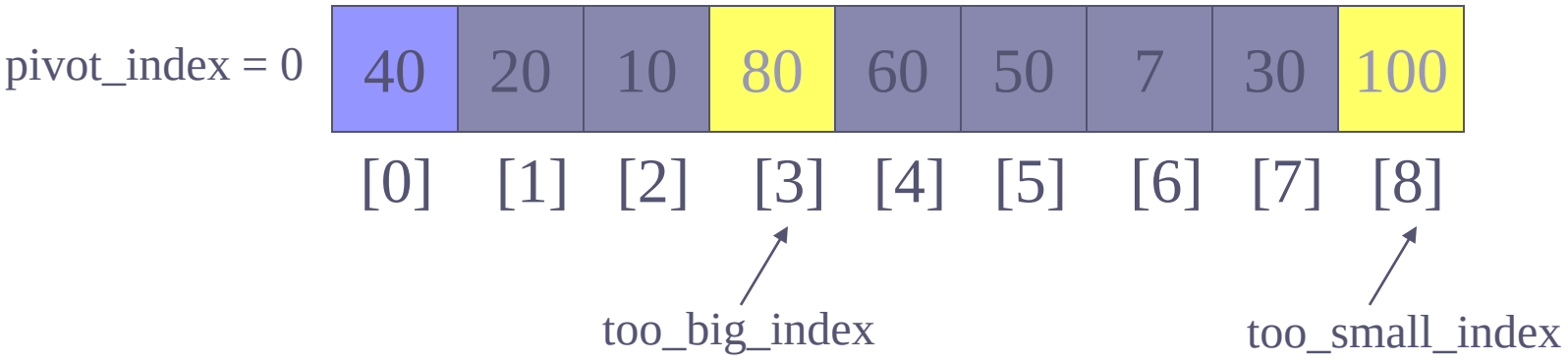
Recall QuickSort

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



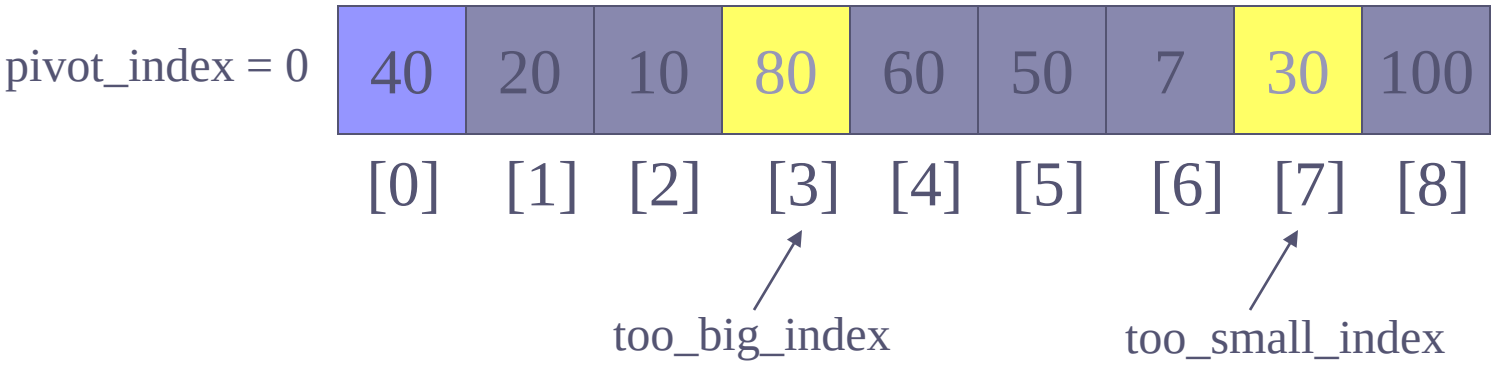
Recall QuickSort

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



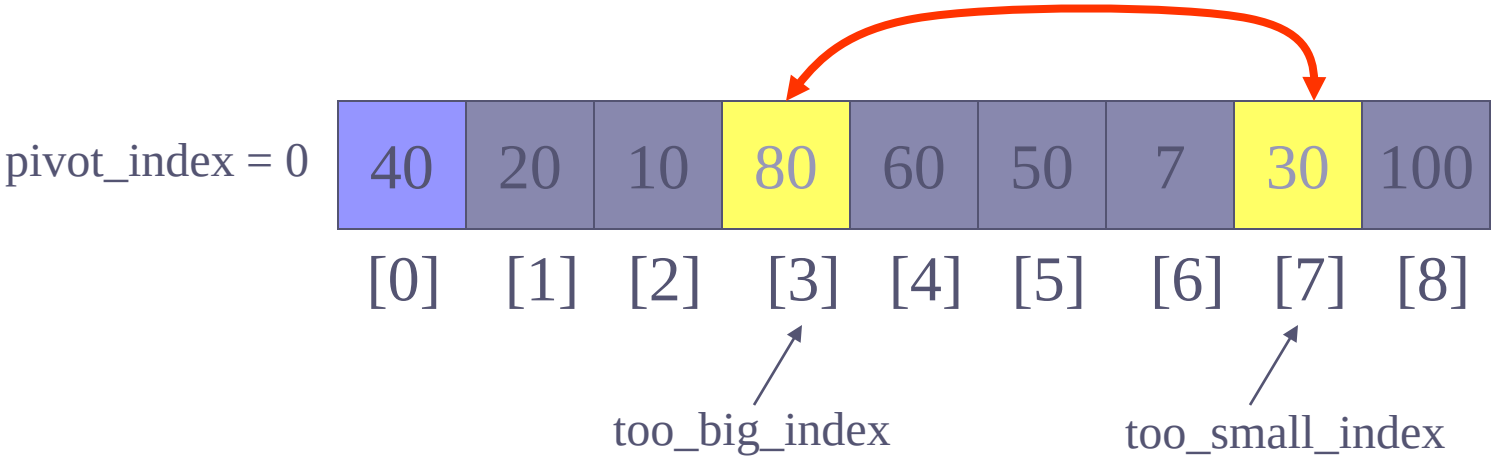
Recall QuickSort

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



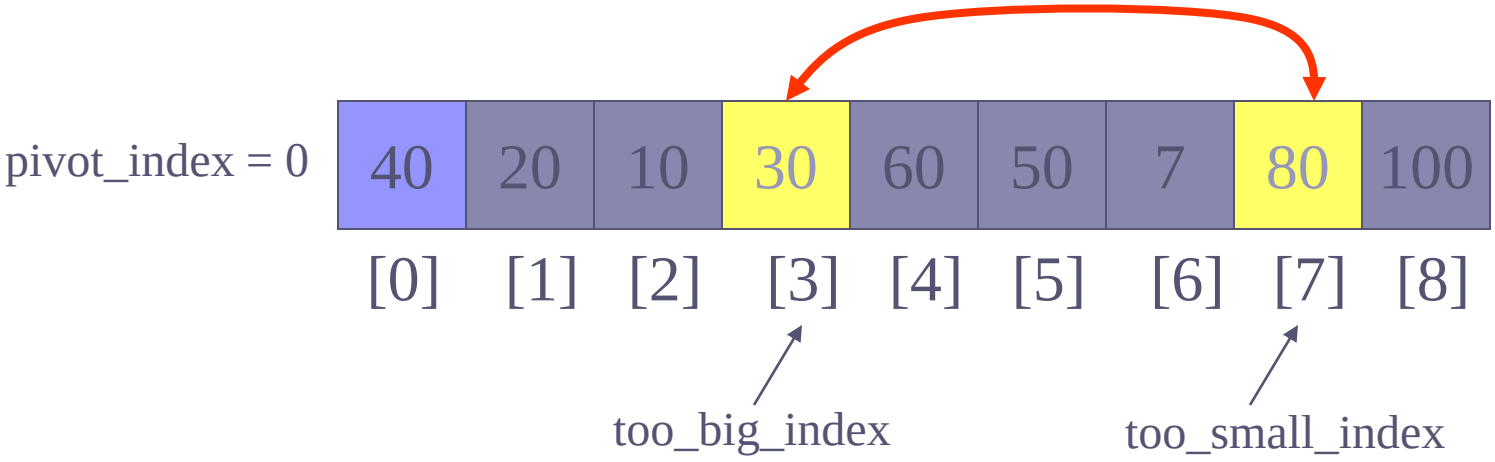
Recall QuickSort

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]



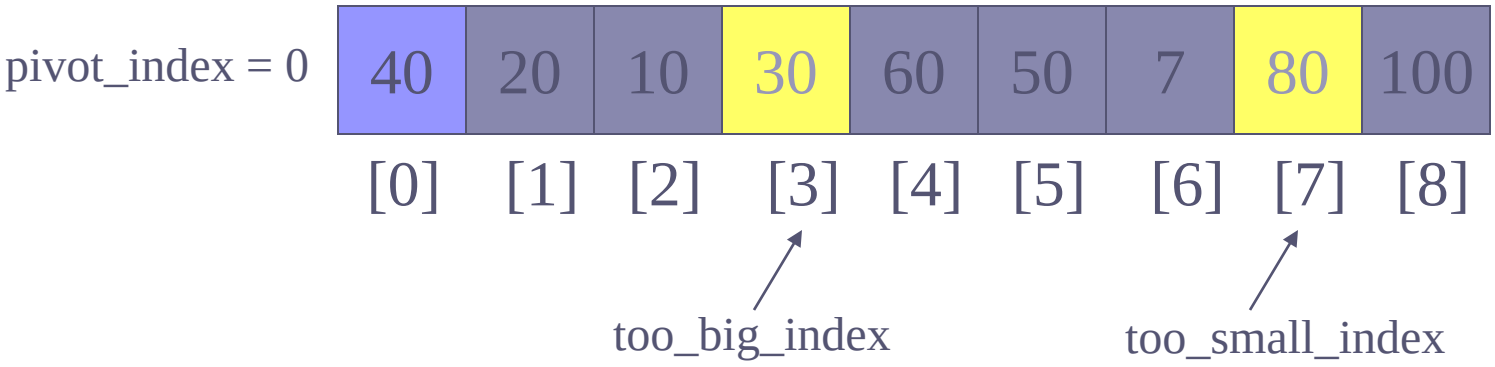
Recall QuickSort

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]



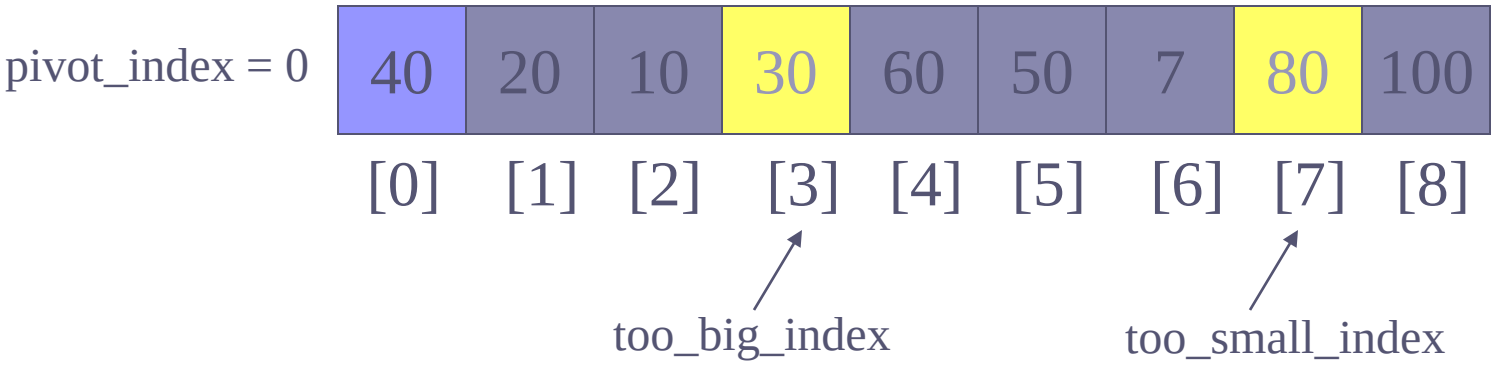
Recall QuickSort

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.



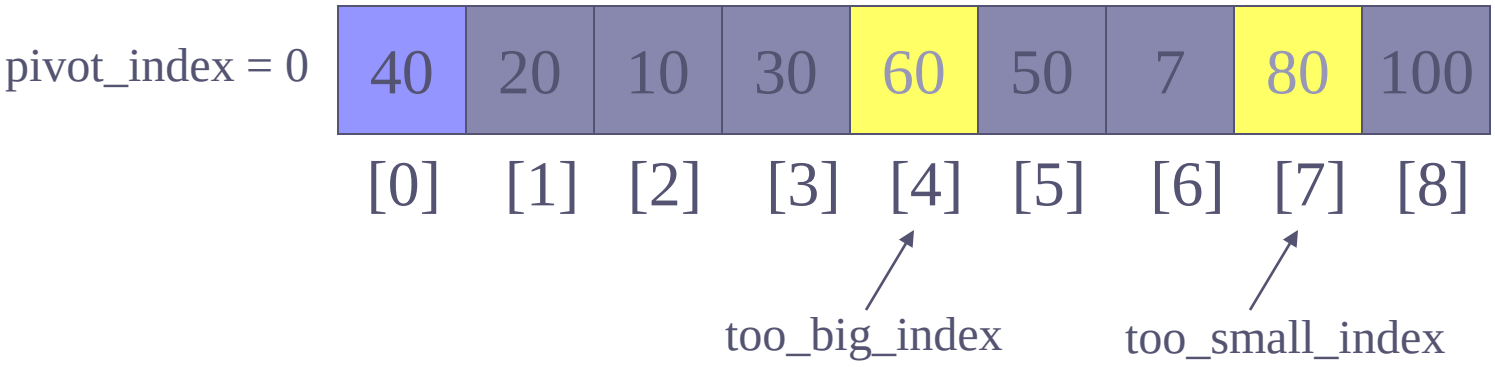
Recall QuickSort

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.



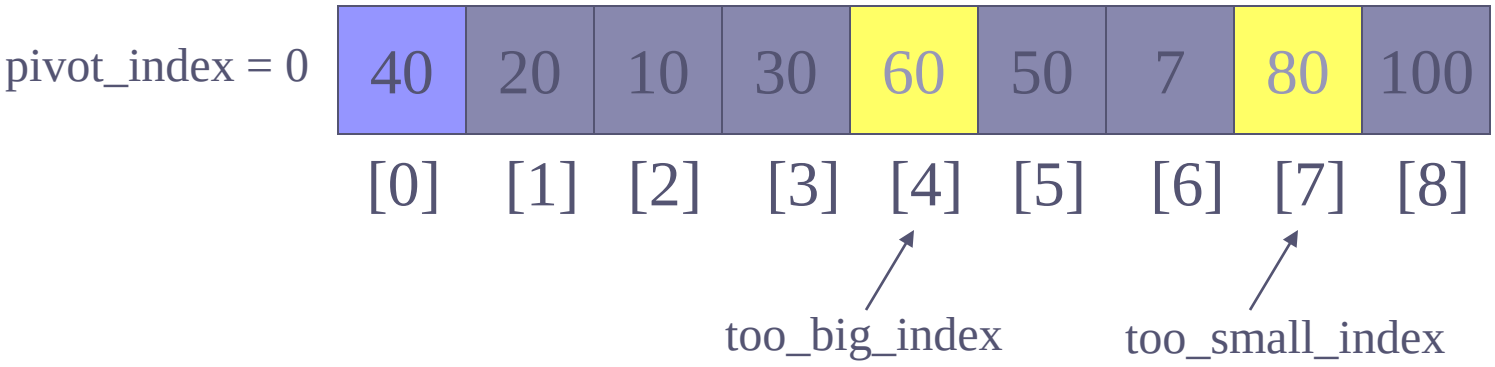
Recall QuickSort

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.



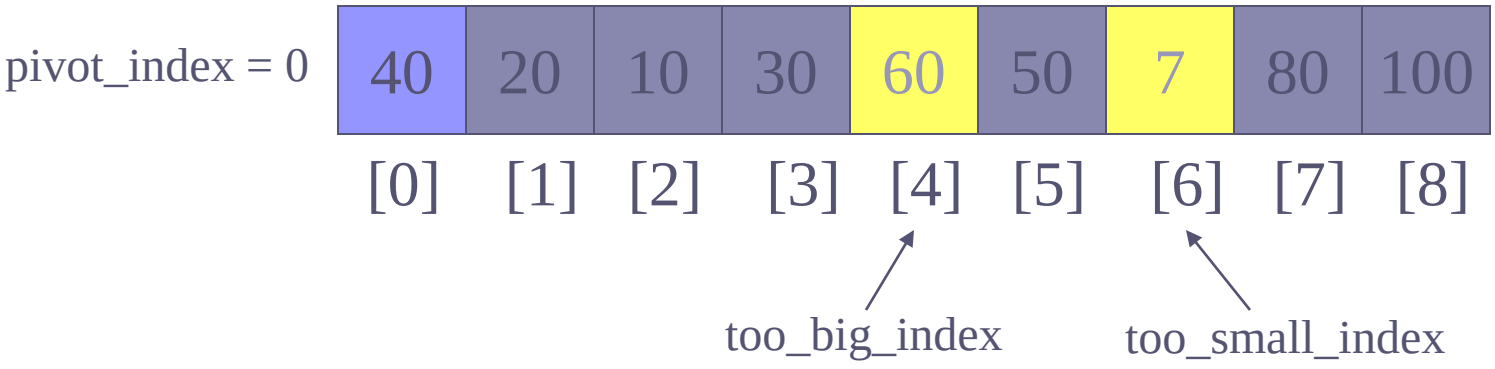
Recall QuickSort

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.



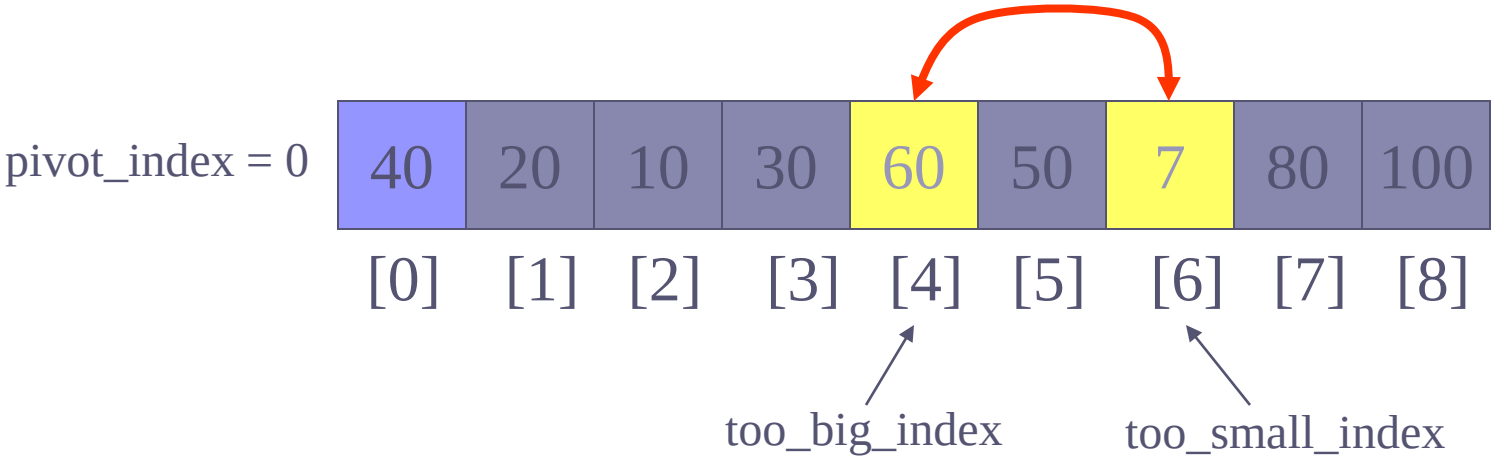
Recall QuickSort

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.



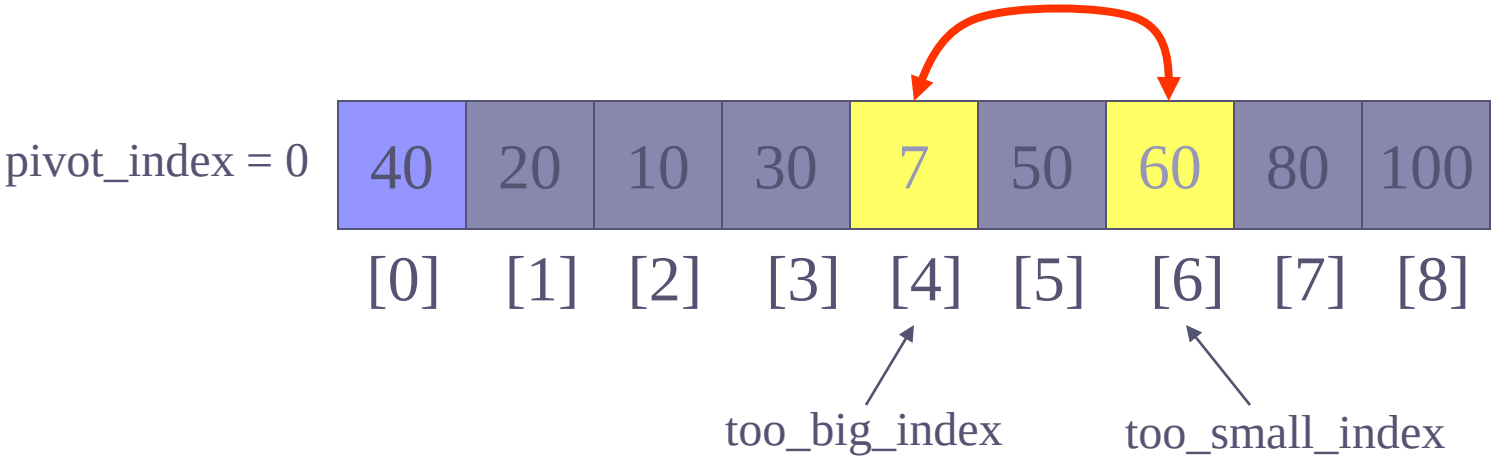
Recall QuickSort

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.



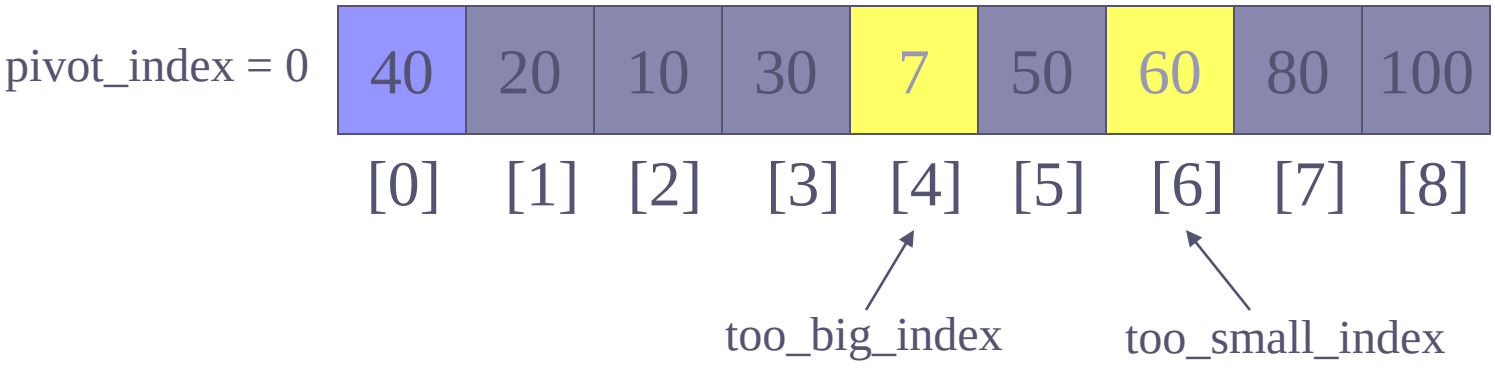
Recall QuickSort

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.



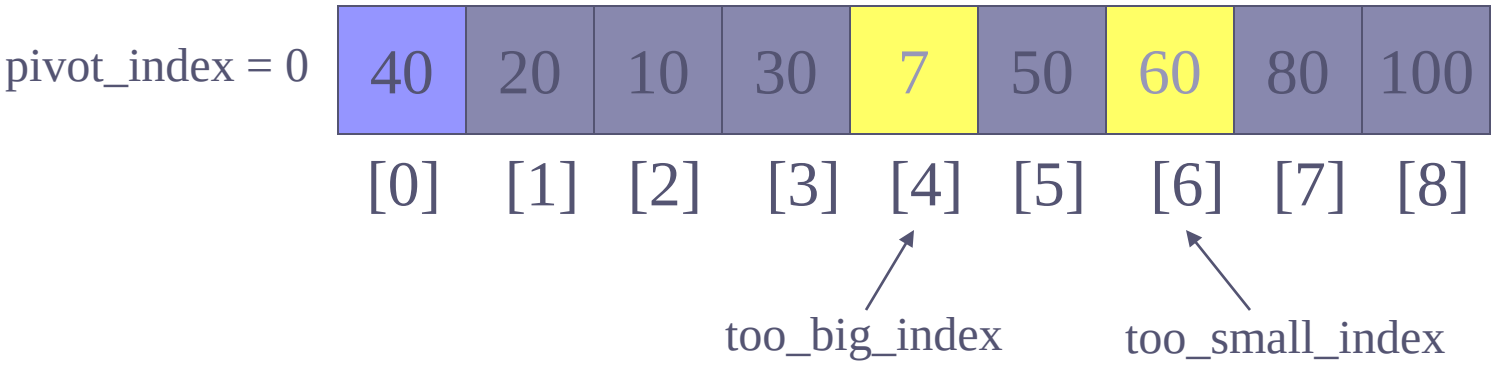
Recall QuickSort

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.



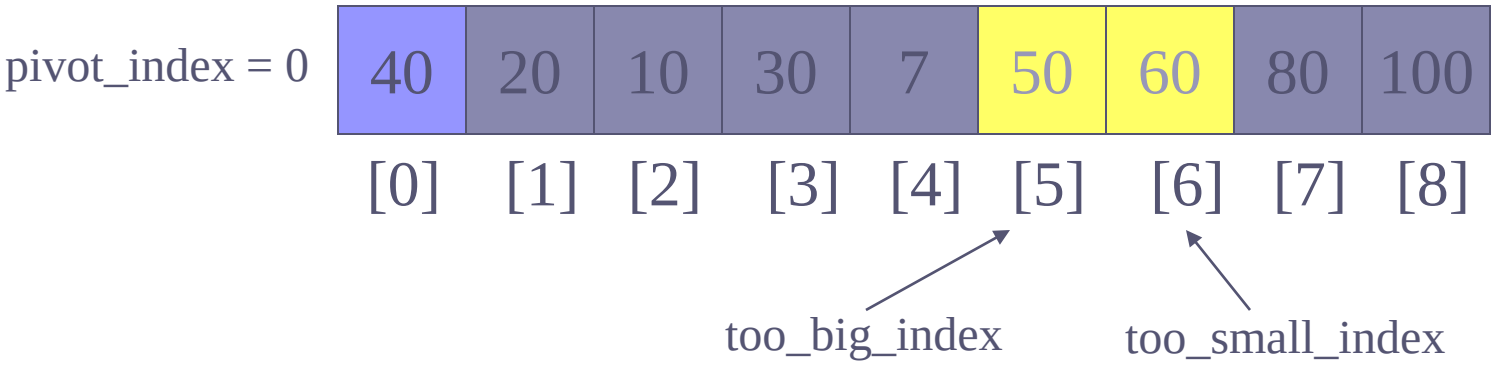
Recall QuickSort

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.



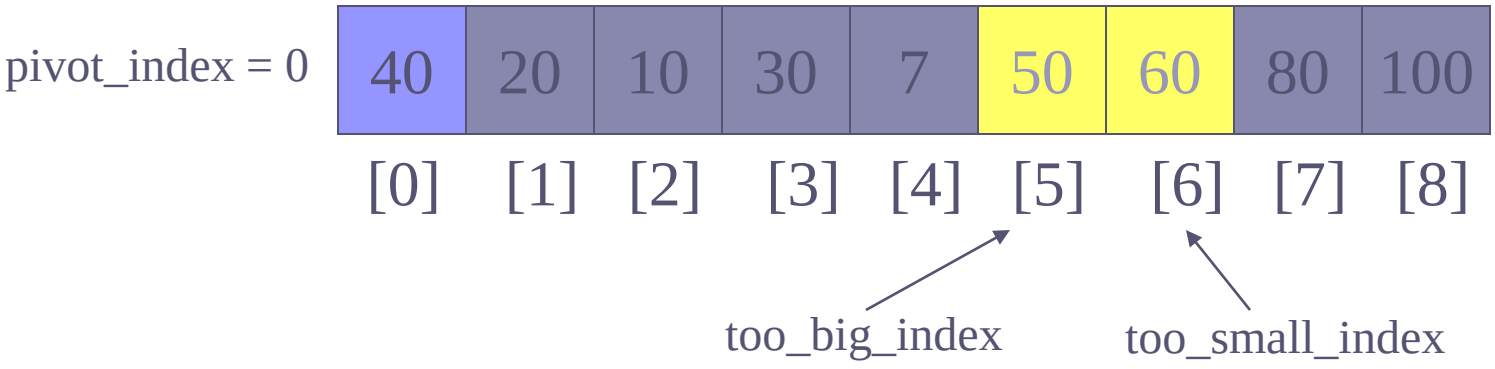
Recall QuickSort

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.



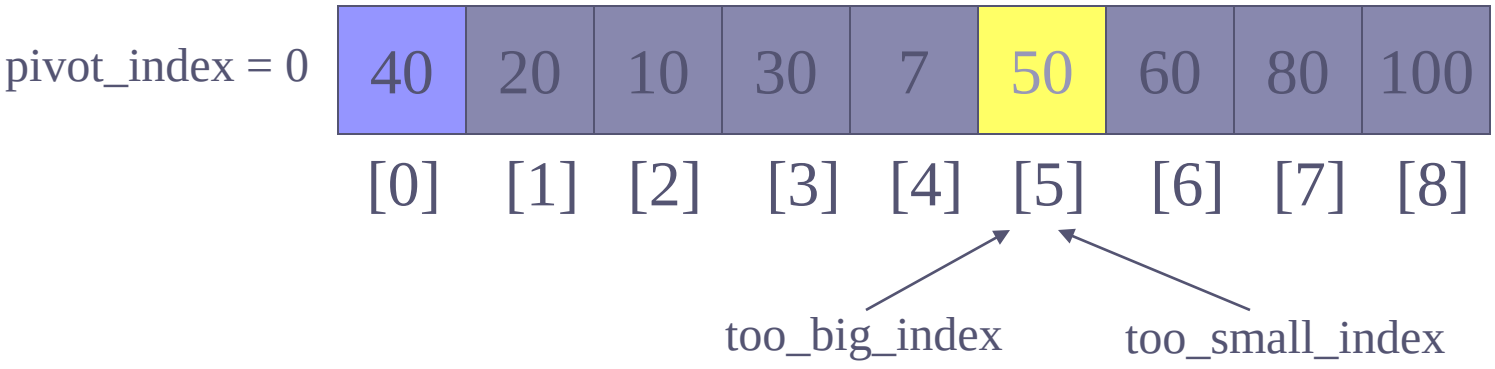
Recall QuickSort

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.



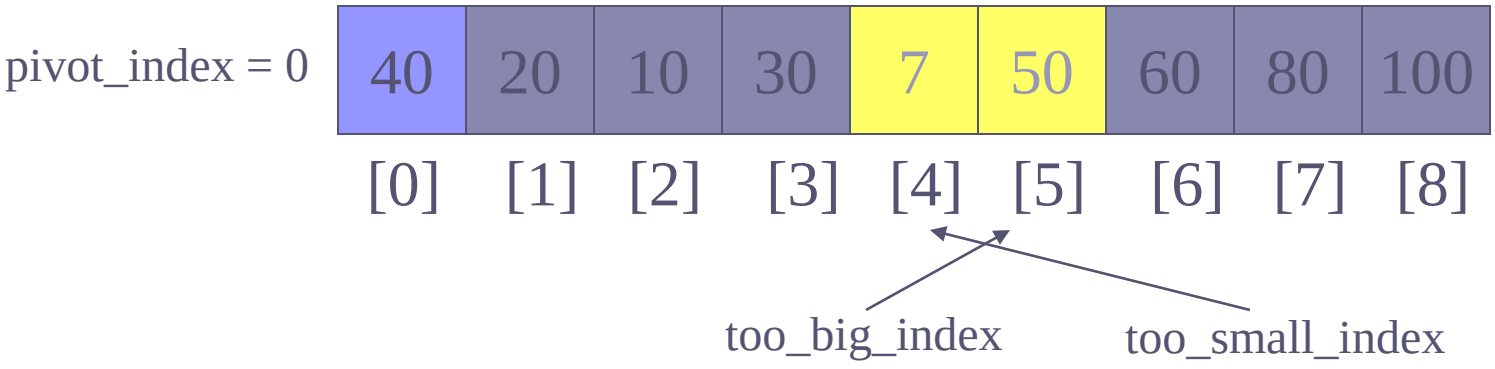
Recall QuickSort

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.



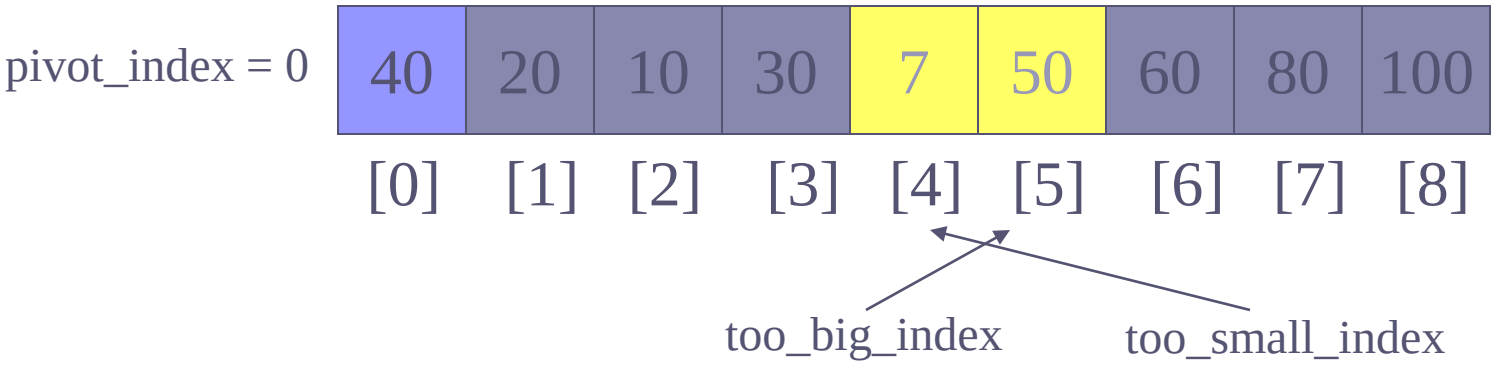
Recall QuickSort

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.



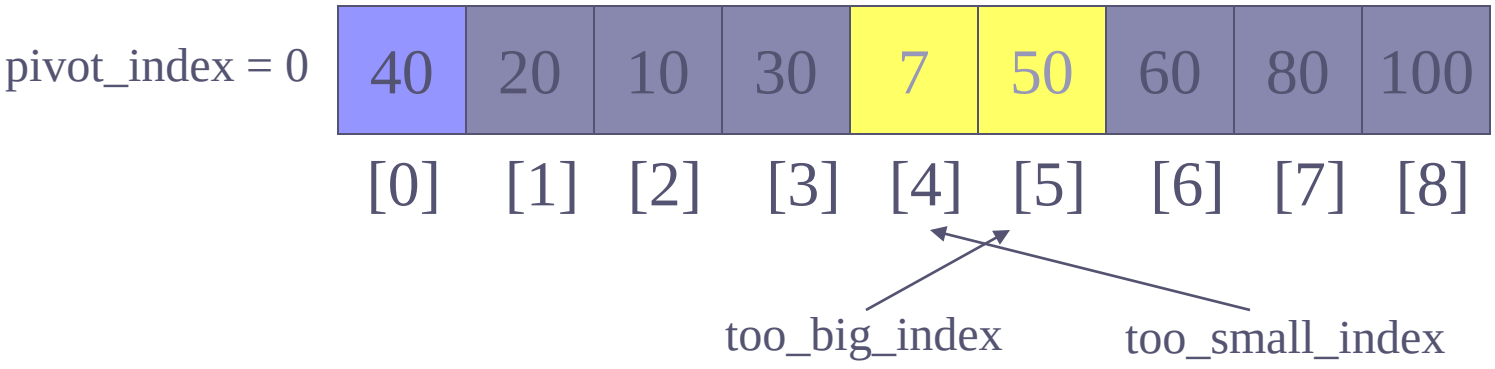
Recall QuickSort

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.



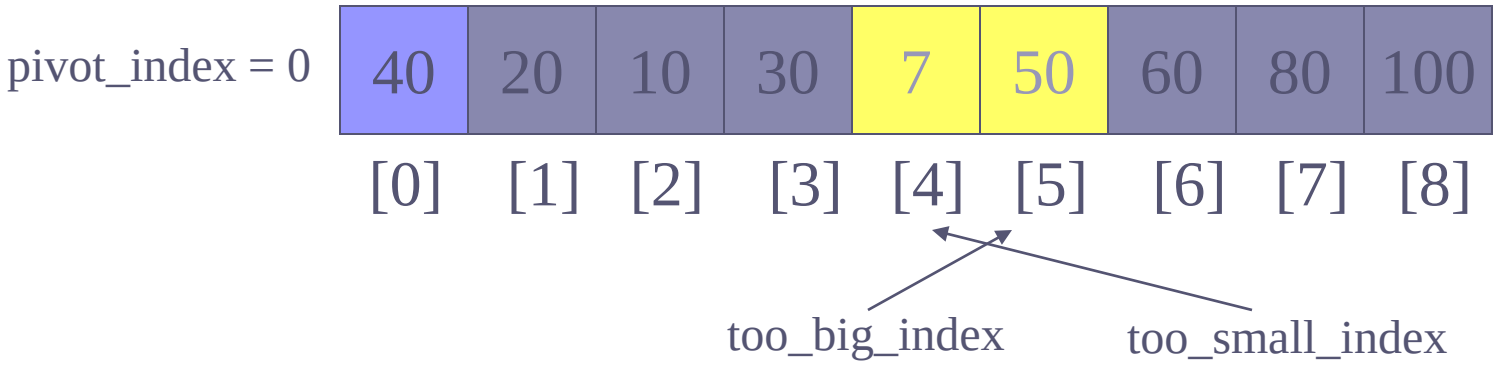
Recall QuickSort

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.



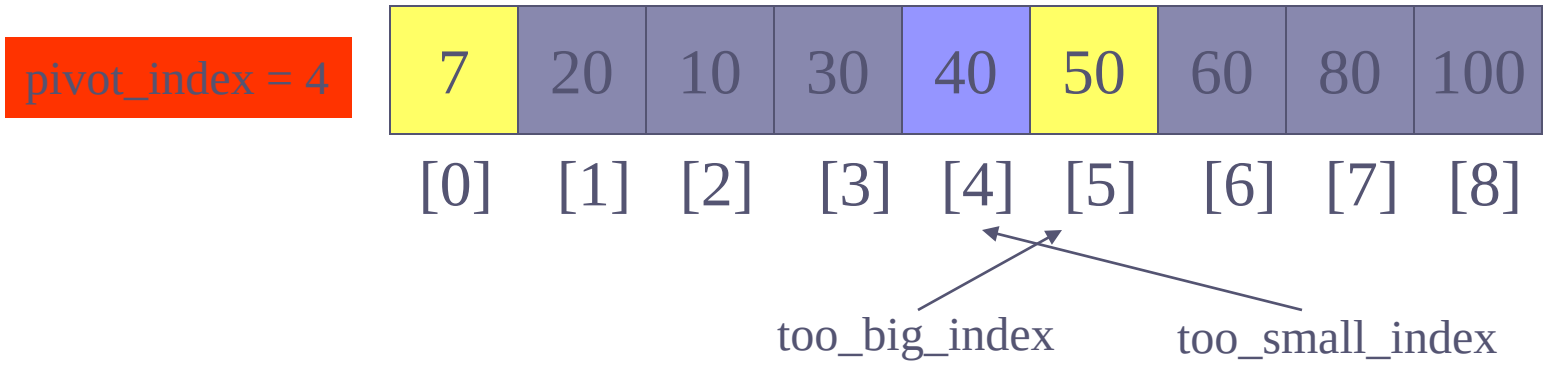
Recall QuickSort

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]



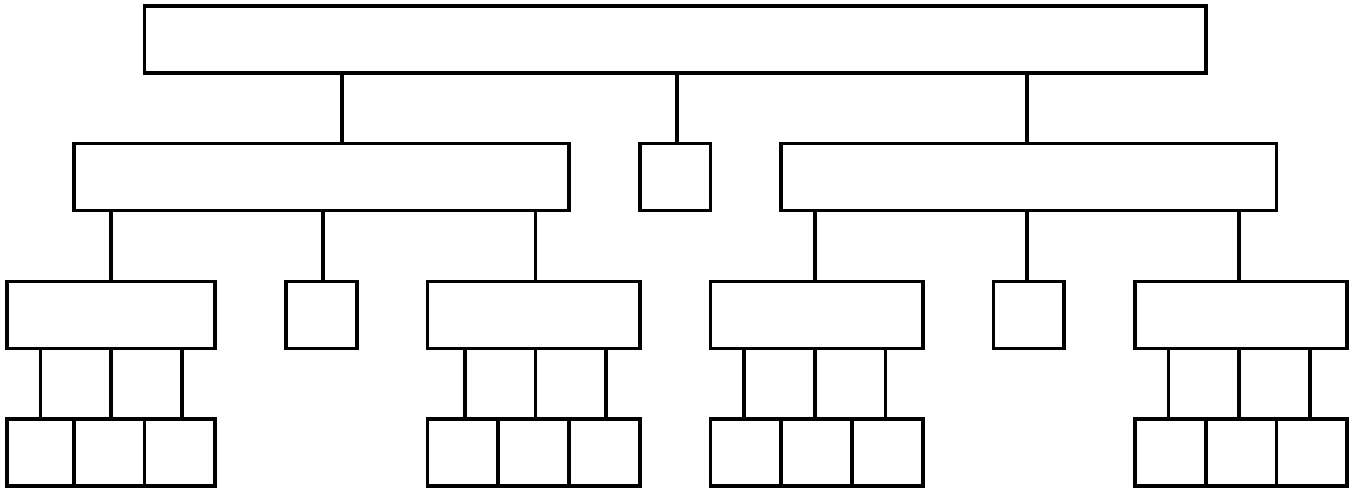
Recall QuickSort

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]



Quick Sort: What is the big Oh?

Best case



in each recursion step the partitioning produces two parts of equal length.

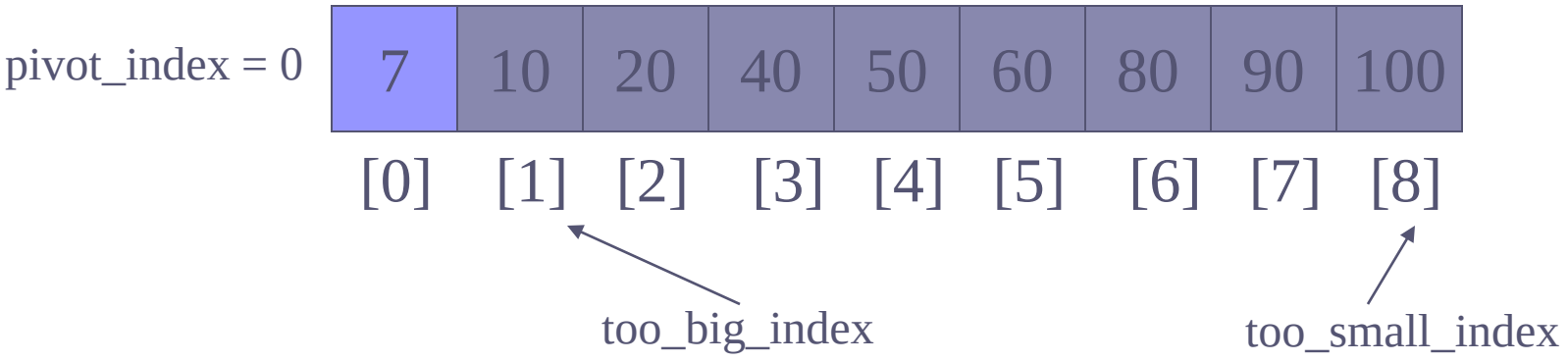
recursion depth is log(*n*) , each level there are *n* elements to be treated => O(n log n)

Quick Sort: What is the big Oh?

What happens when the array is already sorted and the smallest item is chosen as the pivot?

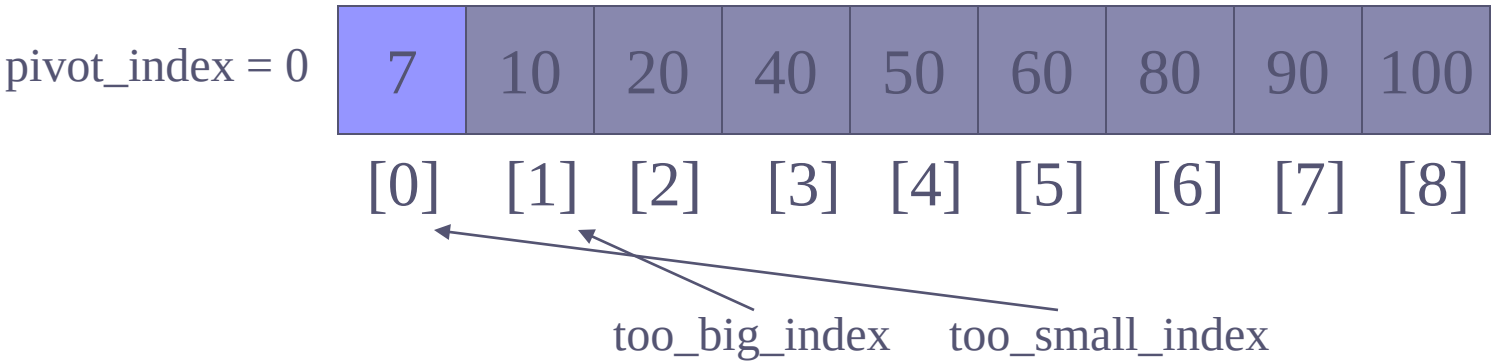
Recall QuickSort

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]



Recall QuickSort

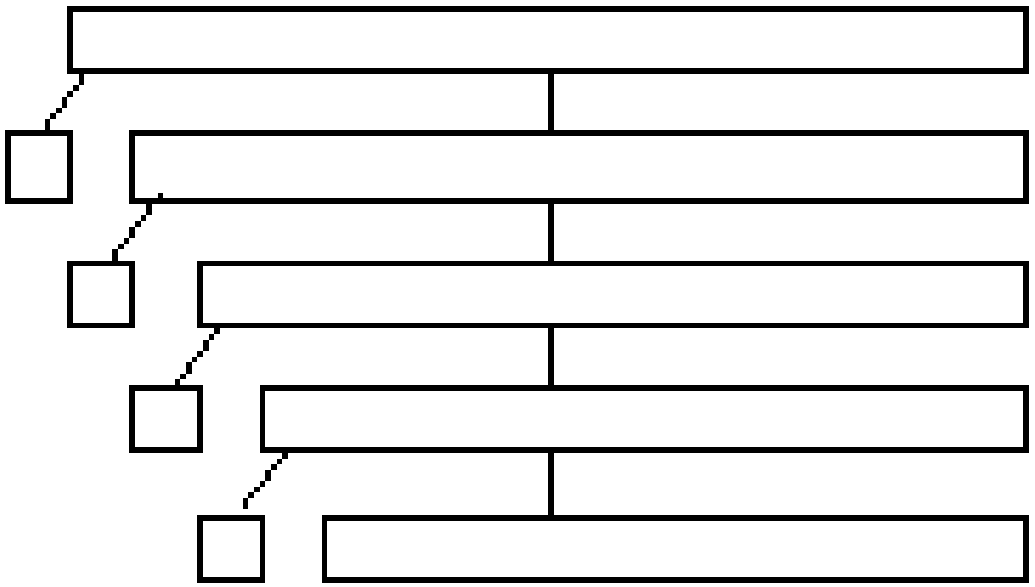
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index]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



Quick Sort: What is the big Oh?

What happens when the array is already sorted and the smallest item is chosen as the pivot?

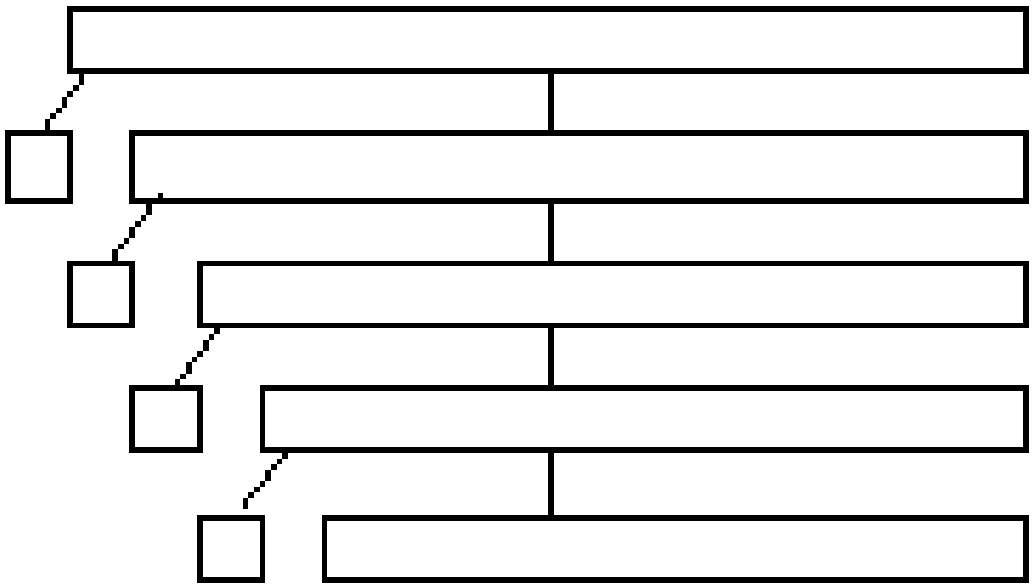
Worst case



Quick Sort: What is the big Oh?

What happens when the array is already sorted and the smallest item is chosen as the pivot?

Worst case



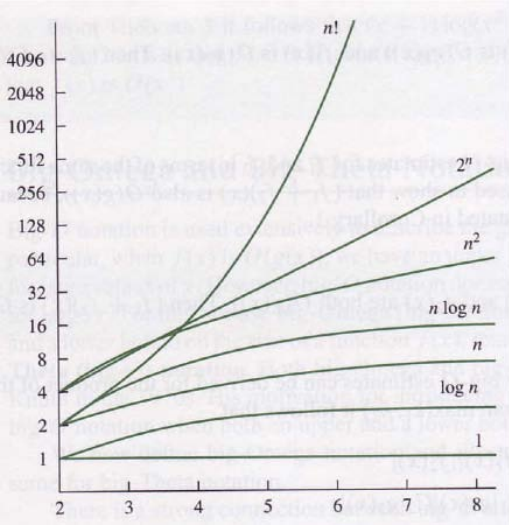
recursion depth is *n*-1, , each level there are *n* elements to be treated => O(n^2)

Quicksort

- **Analysis**
  - choice of the pivot determines which partition is achieved
  - Take the median for optimal partition => worst case is O(n log n)
  - Even if the worst case occurs, `quicksort`'s performance is acceptable for moderately large arrays

Common Orders of Functions
(Slower growing functions first)

- $O(1)$ constant
- $O(\log n)$ logarithmic
- $O((\log n)^c)$ polylogarithmic
- $O(n)$ linear
- $O(n \log n)$ linearithmic, loglinear, quasilinear or supralinear



- $O(n^2)$ quadratic
- $O(n^c)$, $c > 1$ polynomial, sometimes called algebraic
- $O(c^n)$ exponential, sometimes called geometric
- $O(n!)$ factorial, sometimes called combinatorial
- $O(n^n)$ exponential

General Rules

- A normal loop has big Oh, O(n)
- A doubly nested loop generally has big Oh, $O(n^2)$
- A triply nested loop generally has big Oh, $O(n^3)$
- Anytime anything is halved on each iteration, you usually get O(log n)
- Anytime you are generating the permutations or combinations of a pattern (e.g. string, set of roads) it is either O(n!) or $O(c^n)$