Quick Sort: Analysis + Linear Time Sorting

Instructor: Krishna Venkatasubramanian

CSC 212

Partly based on slides from Prof. Carolina Ruiz's CS2223 at WPI, and Prof. Roger Crawfis' CS680 at OSU

Announcements

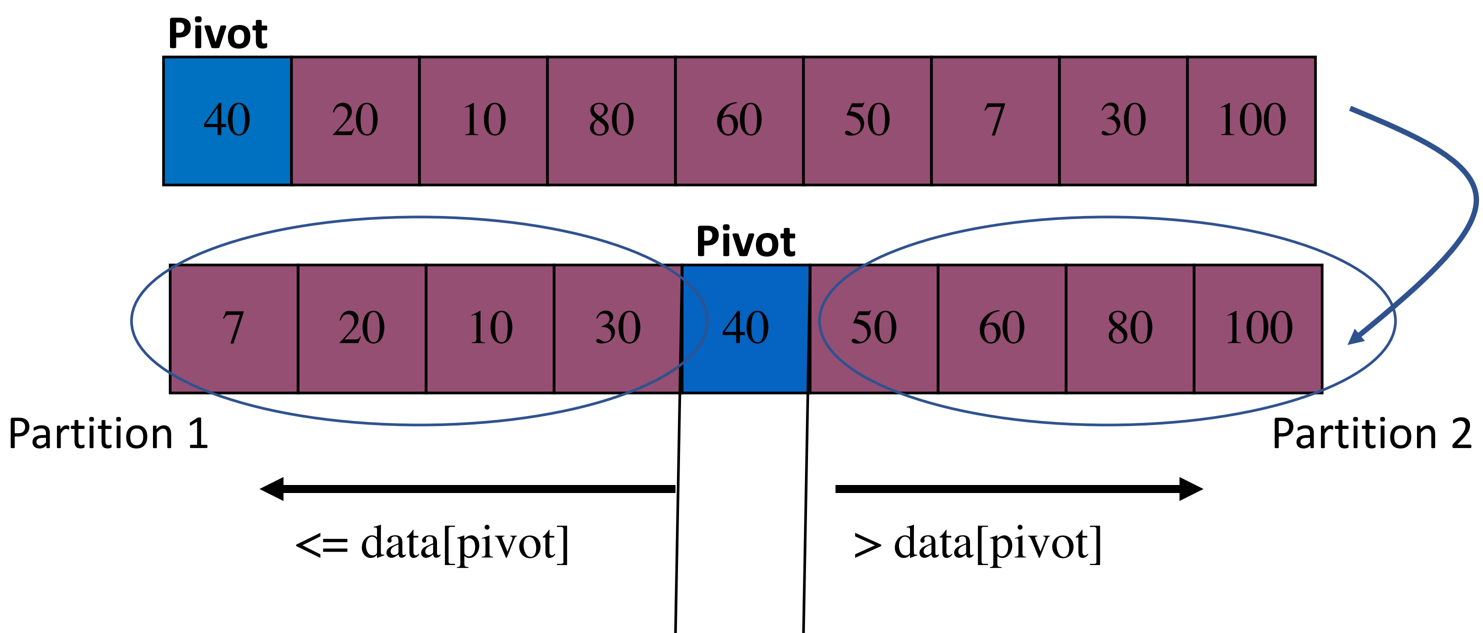
- No class next Tuesday Oct 15
- Quiz 3 will be Tuesday, Oct 22
  - It will cover material from the lectures of Oct 8, Oct 10, and Oct 17.
- There will be no class on Oct 28 and Oct 30, as I will be traveling
  - However, there will be a lab that week.

Quicksort Algorithm (Recap)

```
QuickSort(A,l,r)
  if r-l+1 == 1
    return
  else
    p = Partition(A,l,r)
    QuickSort(A,l,p)
    QuickSort(A,p+1,r)
```

How to Partition (Recap)

- Given an array A
  - Pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot (at index p)
    - Elements greater than pivot (at index p)



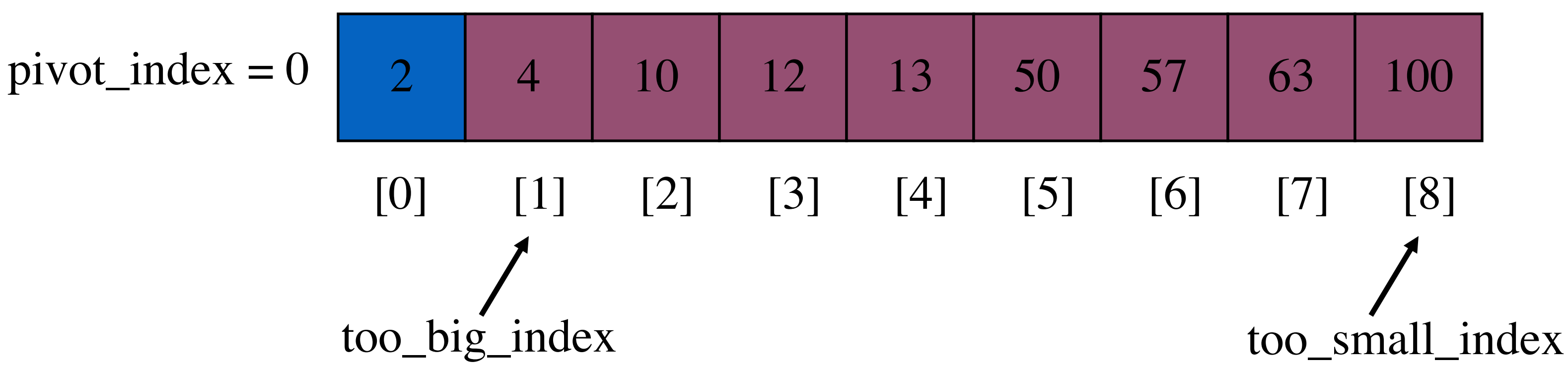
Quicksort Analysis (Recap)

- 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? $O(n)$

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

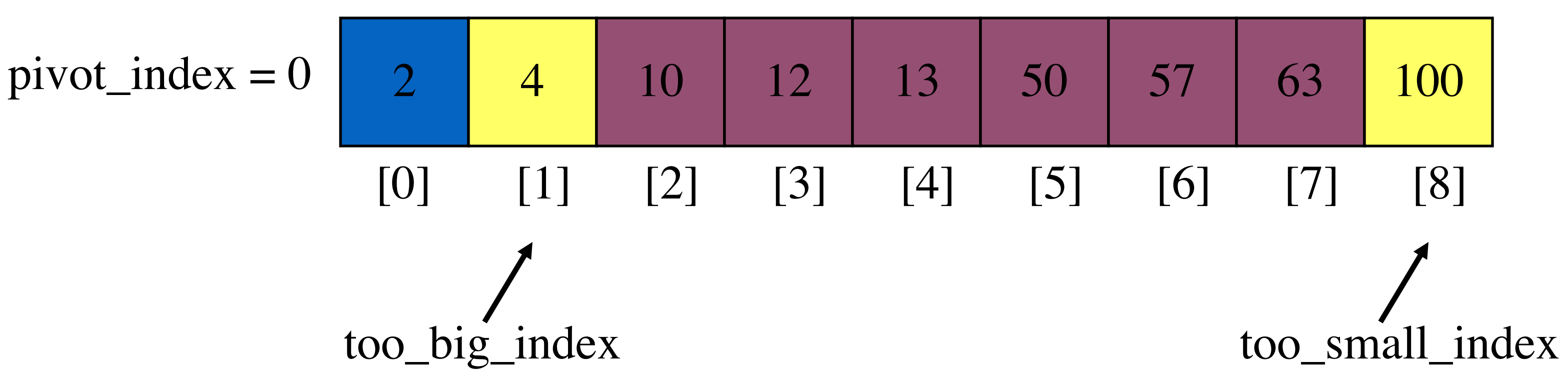
Quicksort: Worst Case

- Assume first element is chosen as pivot.
- Assume we **get array that is already sorted**:



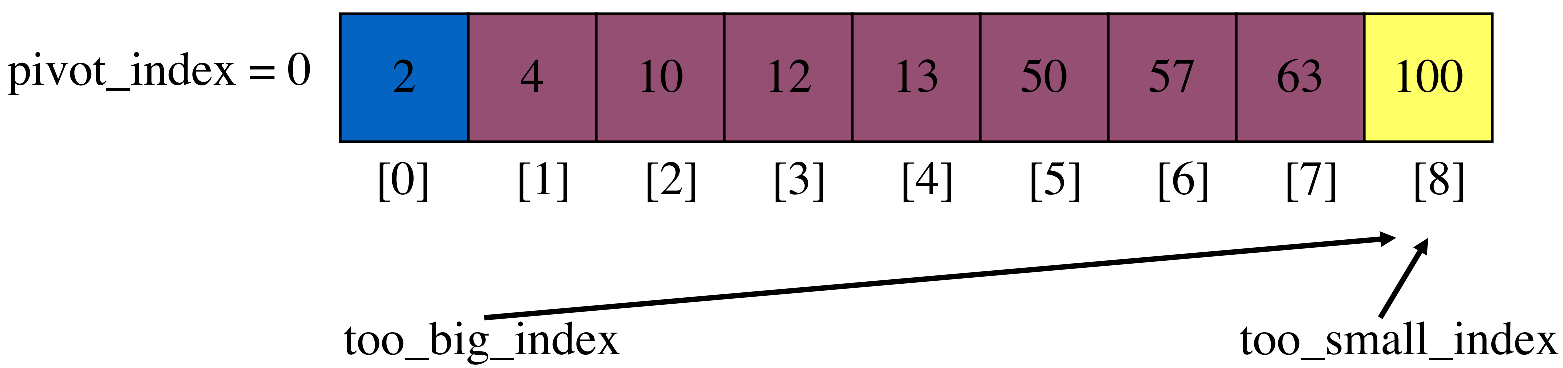
Worst-Case Operation: Example

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



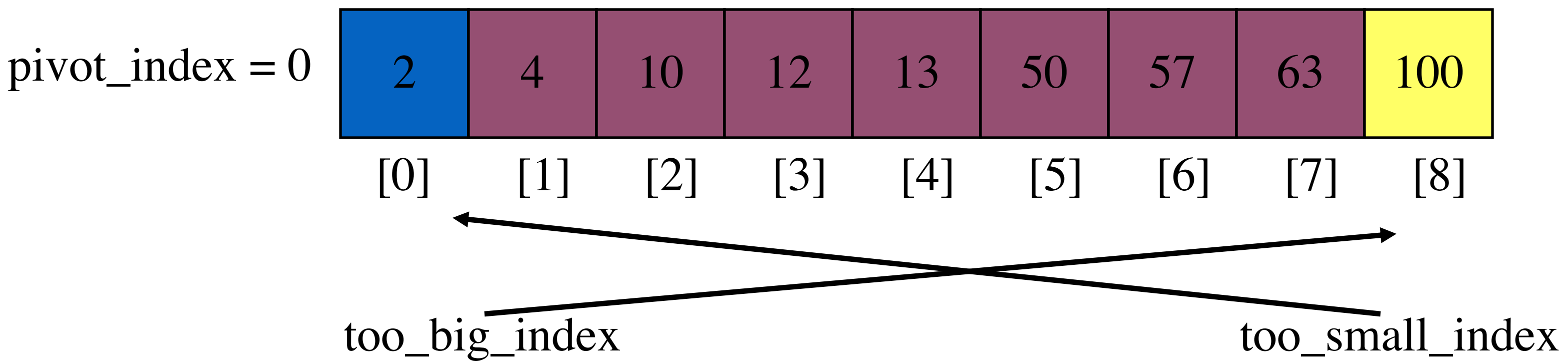
Worst-Case Operation: Example

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



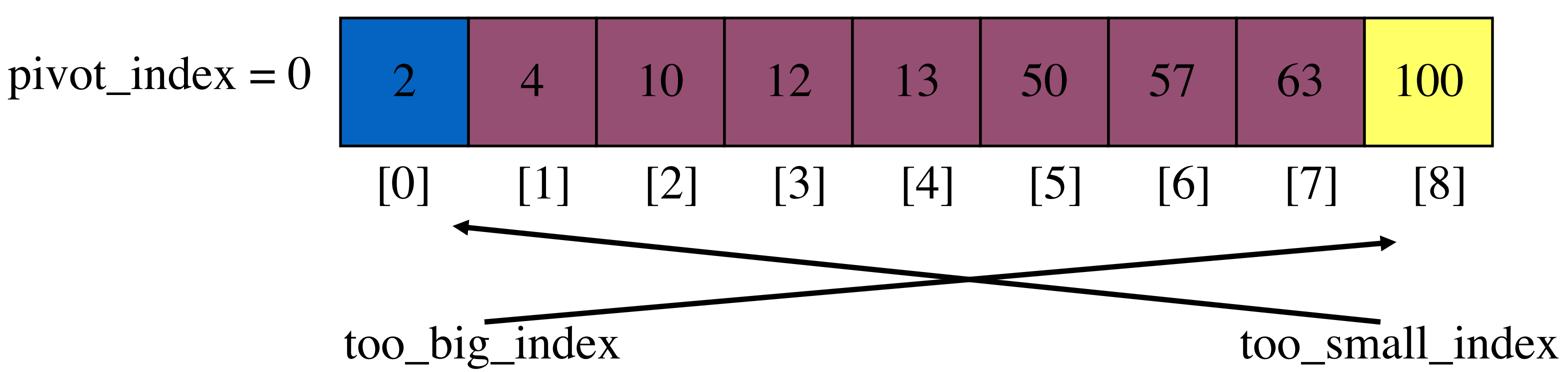
Worst-Case Operation: Example

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



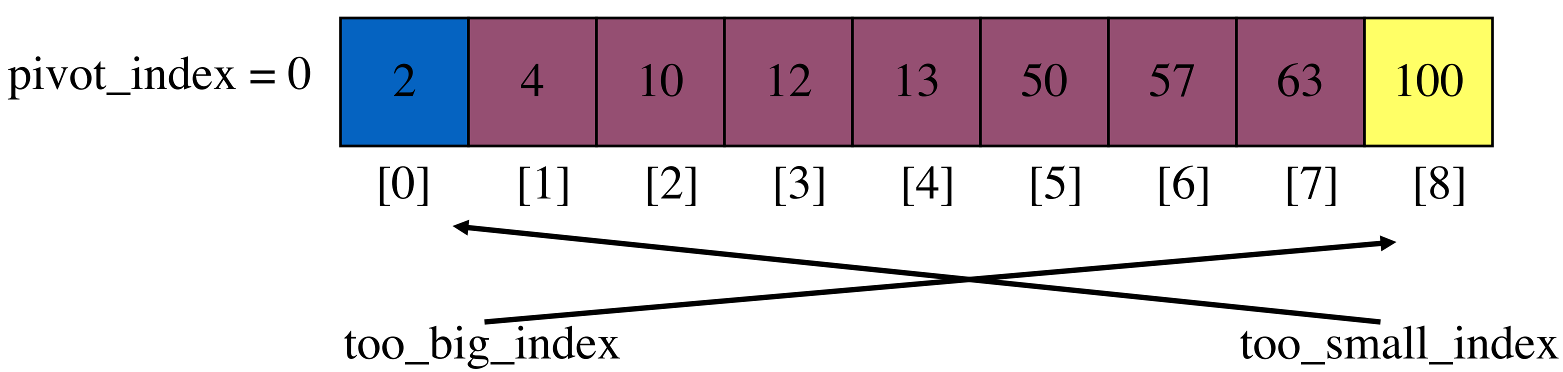
Worst-Case Operation: Example

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



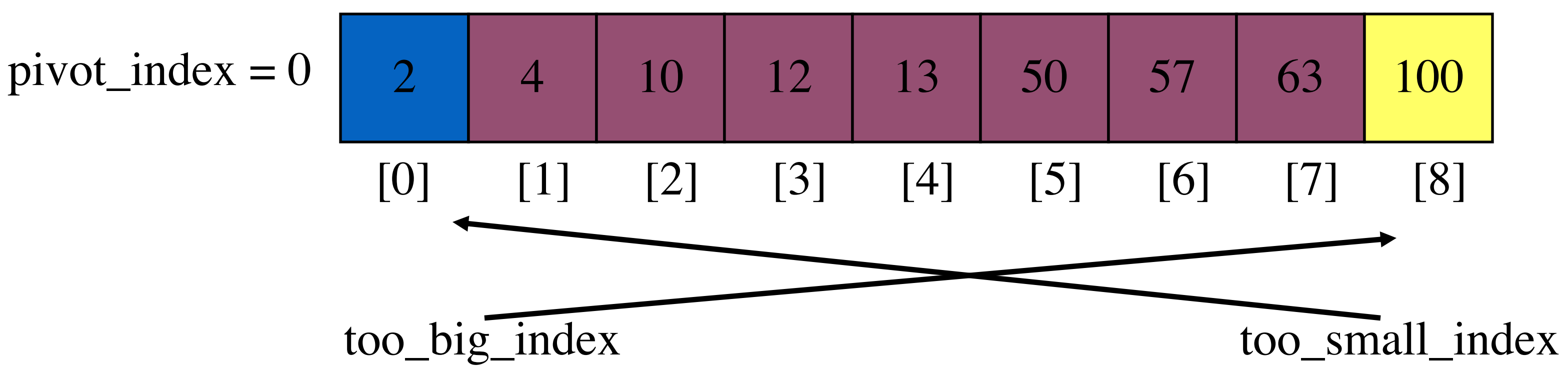
Worst-Case Operation: Example

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



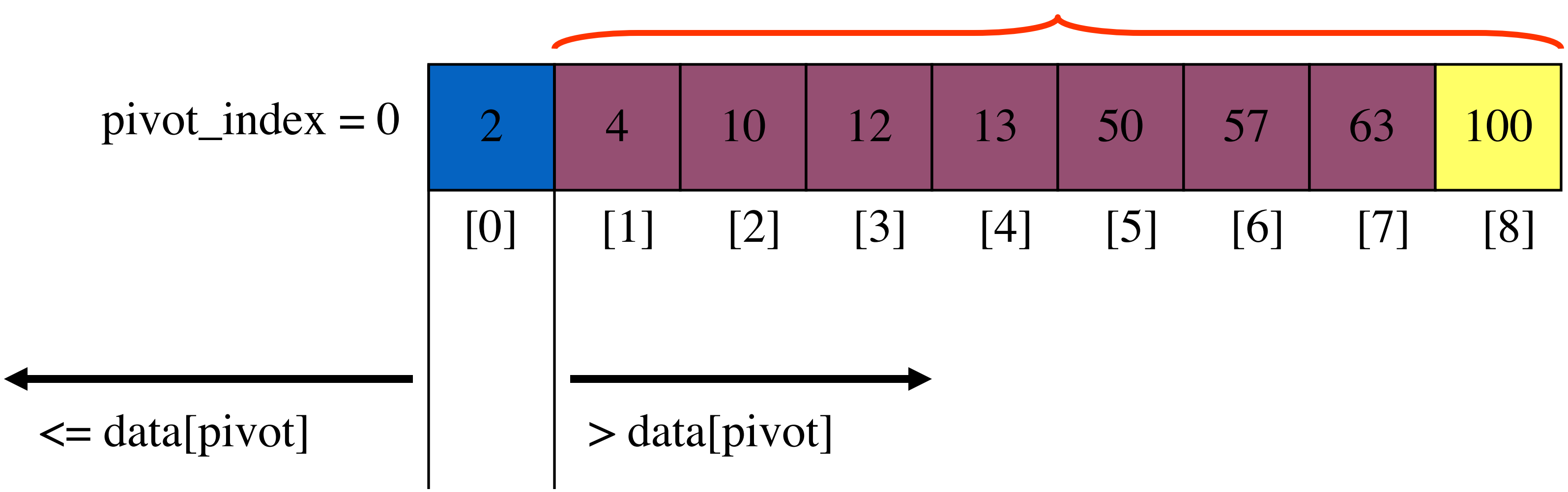
Worst-Case Operation: Example

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



Worst-Case Operation: Example

1. **while** data[too_big_index] <= data[pivot]
   too_big_index = too_big_index+1
2. **while** data[too_small_index] > data[pivot]
   too_small_index = too_small_index-1
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: $O(n \log_2 n)$

Quicksort Analysis

- Worst case running time?
- Recursion:
  - Partition splits array in two sub-arrays:
    - one sub-array of **size 0**
    - the other sub-array of **size n-1**
  - Quicksort each sub-array

- Depth of recursion tree?
  - **O(n)**
- Number of accesses per partition?
  - **O(n)**

Quicksort Analysis

- **Best case running time: $O(n \lg n)$**

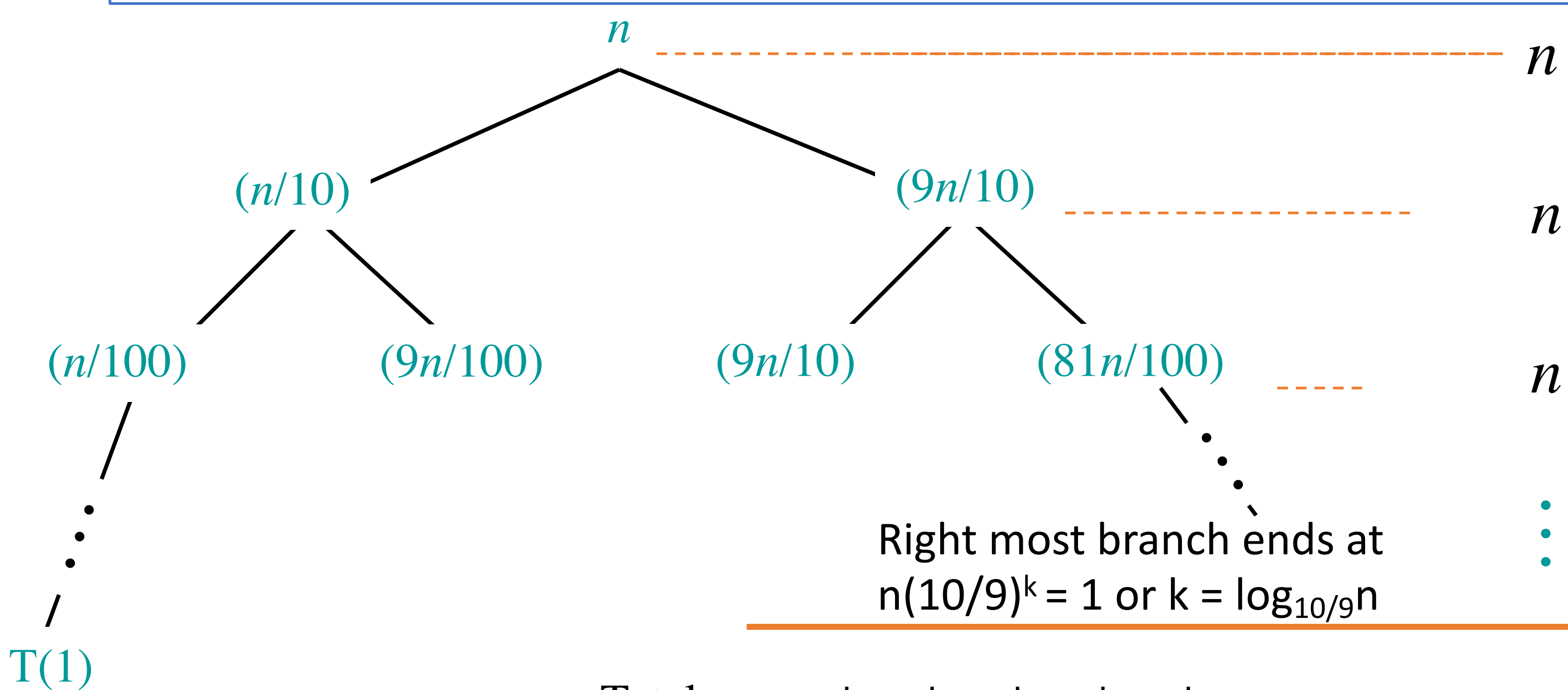
- **Worst case running time: $O(n^2)$**

QuickSort Analysis

- The performance of QuickSort depends on the size of the partitions
- Lop-sided partitions perform worse than balanced partitions
- However, **not all lop-sided partitions are bad**

Remember this from Quiz 2?

Essentially QuickSort's performance when it produces a 9-1 split every time



Total $= \underbrace{n + n + n + n + n + \ \ldots \ldots .}_{f(n) \,=\, \log_{10/9} n}$

Left most branch ends at

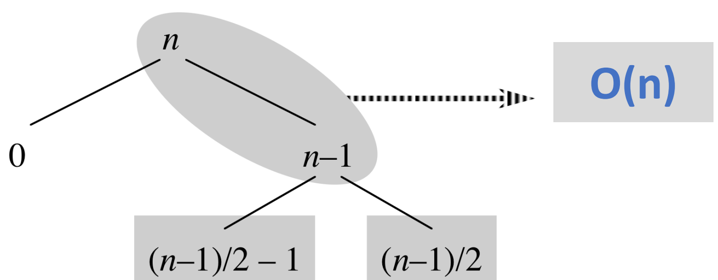
$n/10^k = 1$ or $k = \log_{10} n$

$= O(n \log_{10/9} n) = O(n \lg n)$

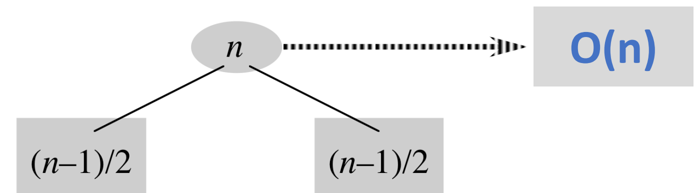
QuickSort Analysis

- **It is the repeated lop-sided partitioning of of an list into sub-lists of size 0 and size n-1 that leads to O(n^2) performance.**

- What happens in the **average case?**
  - That is we alternate between one level with lop-sided partition and then next level we have balanced (back and forth) partition?



Cost of reaching here is **O(n) + O(n-1) ~ O(n)**



Cost of reaching here is **O(n)**

The cost of bad splits absorbed by the good splits

Improved Pivot Selection

- Notice, unlike InsertionSort, **QuickSort performs badly when the list is already sorted**
  - **Same holds true for reverse sorted array**!!
- What can we do to avoid worst case?
  - Pick **median value of three elements** from data array: data[0], data[n/2], and data[n-1].
    - Use this median value as pivot.
  - Randomized QuickSort
    - You will play with this in the lab

QuickSort: Summary

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two sub-arrays, recursively sort
    - All of first sub-array < all of second sub-array
  - Pro's:
    - O(n lg n) average case
    - Sorts in place
    - Fast in practice (*why?*)
  - Con's:
    - O(n^2) worst case
      - Naïve implementation: worst case on sorted input
      - Good partitioning makes this very unlikely.

Non-Comparison Based Sorting

- Many times we have restrictions on our keys
  - Deck of cards: Ace->King and four suites
  - Social Security Numbers
  - Employee ID's

- We will examine an algorithm which under certain conditions can run in **O(*n*)** time.
  - **Counting sort**
  - **Bucket Sort** (you will play with this in assignment 1)

Counting Sort

- Depends on assumption about the numbers being sorted
  - Assume numbers are in the range *1.. k*
- The algorithm:
  - Input: A[1..*n*], where A[j] $\in$ {1, 2, 3, …, *k*}
  - Output: B[1..*n*], sorted (not sorted in place)
  - Also: Array C[1..*k*] for auxiliary storage
  - Therefore needs O(|B|+|C|) extra storage
    - Which is same as O(n+k)

Counting Sort Pseudocode)

Input list

Size of A (and B)

Range of numbers

```
CountingSort(A,n,k)
        B = [], C=[]
        for i=0 to k
                C[i]= 0
        for j=0 to n
                C[A[j]] += 1
        for i=1 to k
                C[i] = C[i] + C[i-1]
        for j=n-1 downto 0
                B[C[A[j]]-1] = A[j]
                C[A[j]] -= 1
        return B
```

This is called a ***histogram***.

Example

A	2	5	3	0	2	3	0	3

C	2	0	2	3	0	1	After **first** for loop

C	2	2	4	7	7	8	After **second** for loop

k = range of numbers = (0-5)

Example

[0] j

A								
	2	5	3	0	2	3	0	3

C						
	2	2	4	7	7	8

[0] [5]

Third loop

A[j] = 3

C[A[j]]-1 = 7 – 1 = 6

[0] [4] [7]

B								
							3	

C						
	2	2	4	6	7	8

[0] [5]

k = range of numbers = (0-5)

Example

Third loop

[0] j

A								
	2	5	3	0	2	3	0	3

C						
	2	2	4	6	7	8

[0] [5]

A[j] = 0

C[A[j]] -1 = 2- 1 = 1

[0] [4] [7]

B								
		0					3	

C						
	1	2	4	6	7	8

[0] [5]

k = range of numbers = (0-5)

Example

	[0]					j		
A	2	5	3	0	2	3	0	3

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

A[j] = 3

C[A[j]] -1 = 6- 1 = 5

Third loop

	[0]				[4]			[7]
B		0				3	3	

C	1	2	4	5	7	8
	[0]					[5]

k = range of numbers = (0-5)

Example

	[0]				j			
A	2	5	3	0	2	3	0	3

C	1	2	4	5	7	8
	[0]					[5]

Third loop

A[j] = 2

C[A[j]] -1 = 4- 1 = 3

	[0]				[4]			[7]
B		0		2		3	3	

C	1	2	3	5	7	8
	[0]					[5]

k = range of numbers = (0-5)

Example

	[0]			j				
A	2	5	3	0	2	3	0	3

C	1	2	3	5	7	8
	[0]					[5]

Third loop

A[j] = 0
C[A[j]] -1 = 1- 1 = 0

	[0]				[4]			[7]
B	0	0		2		3	3	

C	0	2	3	5	7	8
	[0]					[5]

k = range of numbers = (0-5)

Example

	[0]		j					
A	2	5	3	0	2	3	0	3

C	0	2	3	5	7	8
	[0]					[5]

A[j] = 3

C[A[j]] -1 = 5- 1 = 4

Third loop

	[0]				[4]			[7]
B	0	0		2	3	3	3	

C	0	2	3	4	7	8
	[0]					[5]

k = range of numbers = (0-5)

Example

	[0]	j						
A	2	5	3	0	2	3	0	3

C	0	2	3	4	7	8
	[0]					[5]

A[j] = 5
C[A[j]] -1 = 8- 1 = 7

Third loop

	[0]				[4]			[7]
B	0	0		2	3	3	3	5

C	0	2	3	4	7	7
	[0]					[5]

k = range of numbers = (0-5)

Example

j

[0]

A	2	5	3	0	2	3	0	3

C	0	2	3	4	7	7

[0] [5]

Third loop

A[j] = 2

C[A[j]] -1 = 3- 1 = 2

[0] [4] [7]

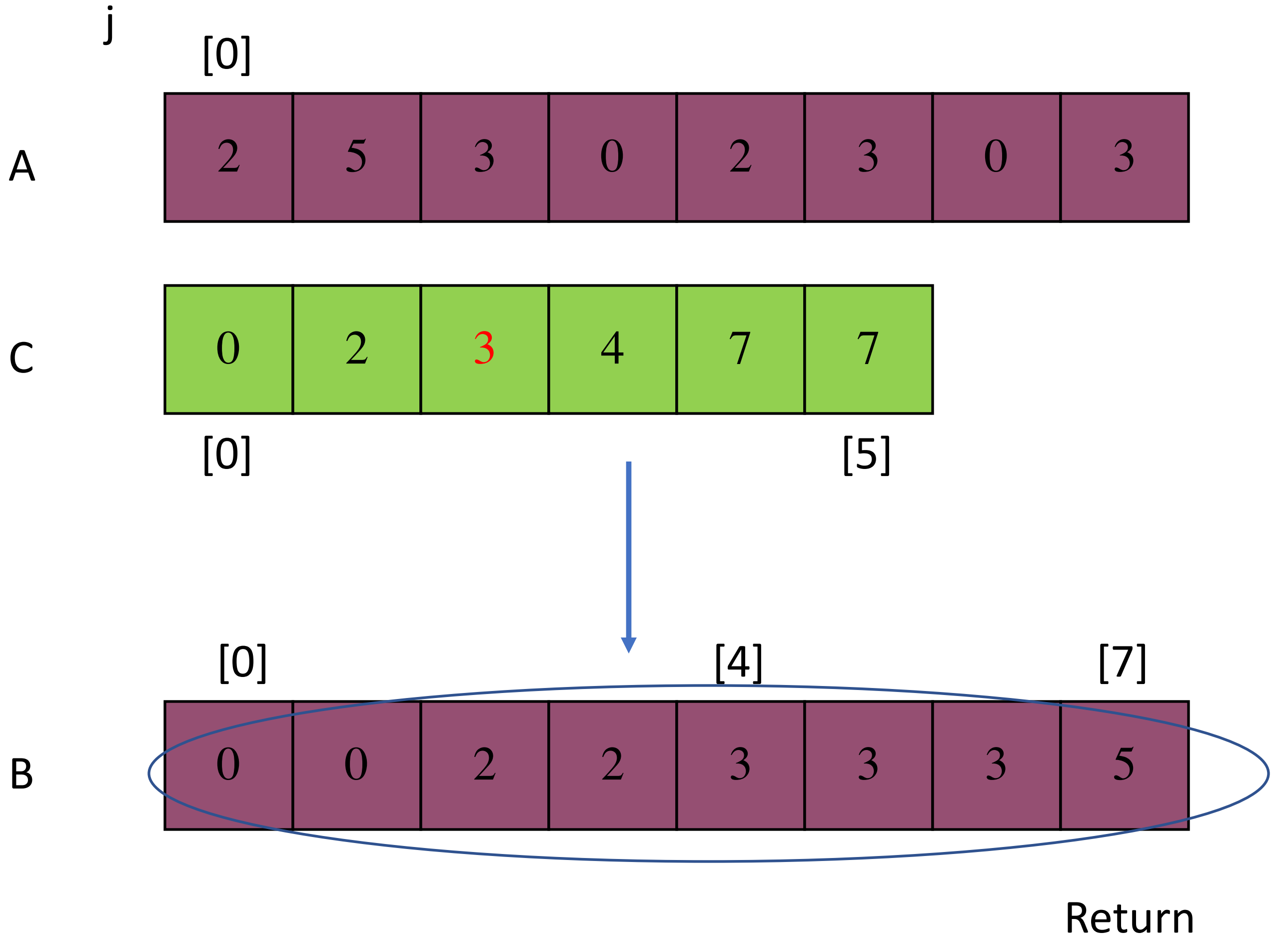
B	0	0	2	2	3	3	3	5

C	0	2	2	4	7	7

[0] [5]

k = range of numbers = (0-5)

Example



Third loop

k = range of numbers = (0-5)

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What is the running time?

Counting Sort

- Total time: O(n + k)
  - Works well if k = O(n) or k = O(1)

- This sorting is *stable.*
  - A sorting algorithm is ***stable*** when numbers with the same values appear in the output array in the same order as they do in the input array.

Counting Sort: Summary

- **Assumption:** input taken from **small** set of **numbers** of size k
- Basic idea:
  - Count number of elements less than you for each element.
  - This gives the position of that number – similar to selection sort.
- Pro's:
  - Fast
  - Asymptotically fast - $O(n+k)$
  - Simple to code
- Con's:
  - Doesn't sort in place.
  - Elements must be integers.
  - Requires $O(n+k)$ extra storage.


That's all Folks!
Any Question?