CS 315 - Day 14, Quicksort

Today we'll discuss Quicksort. It's one of the most efficient and "simple" sorts, ubiquitous, and still an active research area.

Quicksort, like Mergesort, is a ***divide-and-conquer*** (extra-credit for example?) algorithm, and this generally leads to $O(\, n \lg(n)\,)$ asymptotic behavior.

Quicksort is ***in-place*** (no copying into temp arrays, like Mergesort), which contributes to its efficiency.

Description of Quicksort using a deck of cards.

In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Work through the algorithm, "think like a computer," on randomly ordered cards.
4. Try the Quicksort algorithm on already sorted cards.
5. Try the Quicksort algorithm on reverse sorted cards.
6. Could you characterize a ***worst-case*** based on your testing?

The worst case happens when the Partition only moves ("takes care of") one element.

Quicksort pseudo-code:

```
Quicksort(A,p,r)
  if p < r
    then q  = Partition(A , p , r)
         Quicksort(A , p , q-1)
         Quicksort(A , q+1 , r)
```

Partition pseudo-code:

```
Partition(A,p,r)

  x = A[r]
  i = p - 1

  for (j=p; j<=r-1; j++)
  {
     if ( A[j] <= x )
     {
          i = i + 1
          swap A[i] <--> A[j]

     }
   }

  swap A[i+1] <--> A[r]
  return i + 1
```

Partition worksheet, https://jcsites.juniata.edu/faculty/kruse/cs315/partitionExample.pdf

Partition Worksheet

Sample data

Consider the following call:

```
Partition( A , 1 , 6 )
```

Fill in the table below with the values of the variables and Array A, after the loop control statement (in **bold**) is executed. There might be more rows in the table than necessary.

indicies	1	2	3	4	5	6	
Array A	*-5*	*9*	*-1*	*-10*	*2*	*-3*	**variables**
Beginning of Loop 1 (after **bolded** is executed)							x = i = j = p = r =
Beginning of Loop 2 (if entire loop run)							i = j =
Beginning of Loop 3 (if entire loop run)							i = j =
Beginning of Loop 4 (if entire loop run)							i = j =
Beginning of Loop 5 (if entire loop run)							i = j =
Beginning of Loop 6 (if entire loop run)							i = j =
Beginning of Loop 7 (if entire loop run)							i = j =
Complete							return =

```
Partition(A,p,r)
  x = A[r]
  i = p - 1

  for (j=p; j<=r-1; j++)
  {
     if ( A[j] <= x )
     {
          i = i + 1
          swap A[i] <--> A[j]
     }
   }

  swap A[i+1] <--> A[r]

  return i + 1
```

indicies	1	2	3	4	5	6	
Array A	*-5*	*9*	*-1*	*-10*	*2*	*-3*	**variables**
Beginning of Loop 1 (after **bolded** is executed)	*-5*	*9*	*-1*	*-10*	*2*	*-3*	x = *-3* i = *0* j = *1* p = *1* r = *6*
Beginning of Loop 2 (if entire loop run)	*-5*	*9*	*-1*	*-10*	*2*	*-3*	i = *1* j = *2*
Beginning of Loop 3 (if entire loop run)	*-5*	*9*	*-1*	*-10*	*2*	*-3*	i = *1* j = *3*
Beginning of Loop 4 (if entire loop run)	*-5*	*9*	*-1*	*-10*	*2*	*-3*	i = *1* j = *4*
Beginning of Loop 5 (if entire loop run)	*-5*	*-10*	*-1*	*9*	*2*	*-3*	i = *2* j = *5*
Beginning of Loop 6 (if entire loop run)							i = j =
Beginning of Loop 7 (if entire loop run)							i = j =
Complete	*-5*	*-10*	*-3*	*9*	*2*	*-1*	return = *3*

What is the running time/order of growth/amount of work of Partition?

Exercise 7.1-3 asks you to show that the running time of PARTITION on a subarray $A[p:r]$ of $n = r - p + 1$ elements is $\Theta(n)$.

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

Q: what do you think the best-case for Quicksort is?

A: When partitions are balanced (same size) at each level

Q: What about the worst-case?

A: When the partitions are unbalanced.

an already sorted list

(increasing) OR (decreasing)

Best Case

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$ ← HW problem!

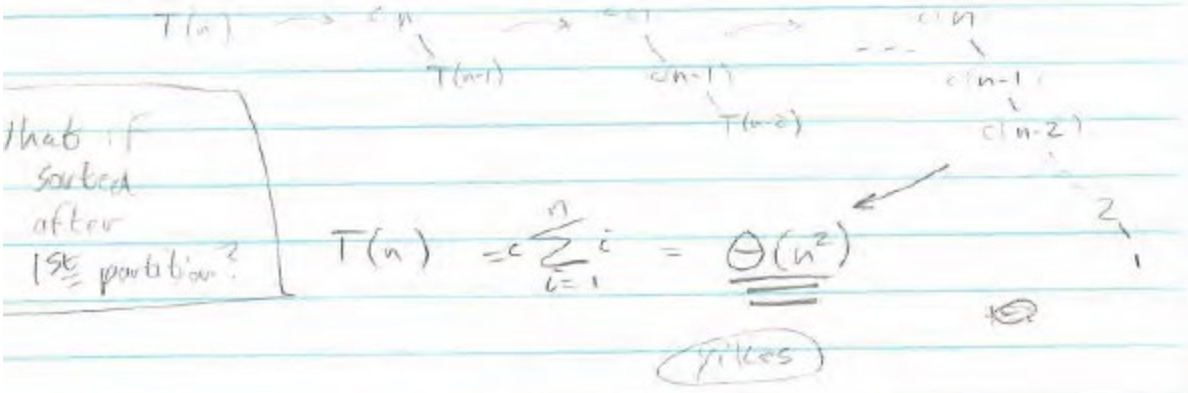
$T(n) = \Theta(n \lg n)$

unlike Merge-Sort, we have a dynamic partition (it's not always $\frac{1}{2}$)

Worst Case (QS(2,r)) (Q(1,0))

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

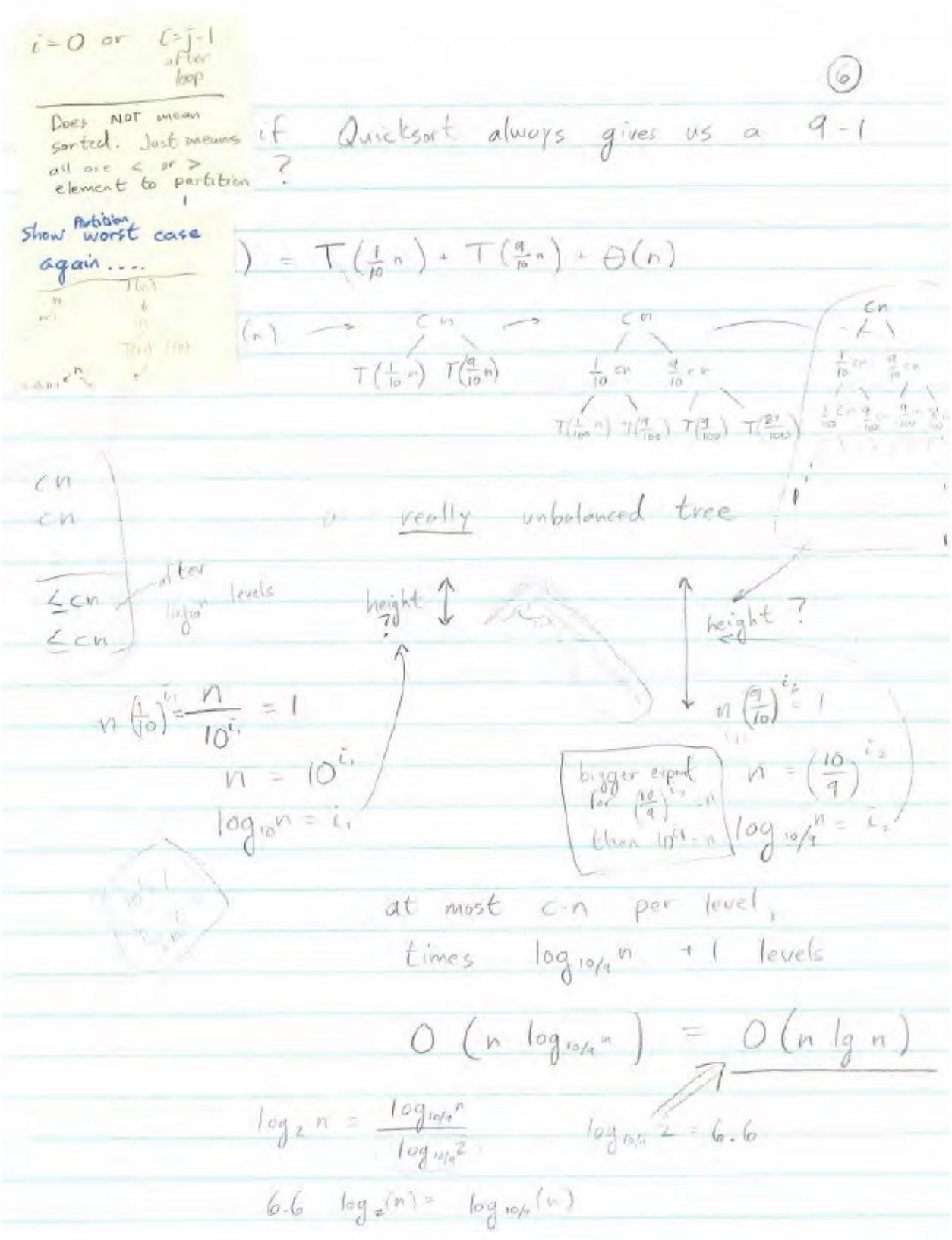
$= T(n-1) + \Theta(n)$



What if sorted after 1st partition?

$T(n) = c\sum_{i=1}^{n} i = \Theta(n^2)$

(yikes)



As long as there is a ***constant*** ratio on the split, Quicksort will be $O(n \cdot \lg(n))$.

Run `QuicksortPoCkey.ipynb`, which includes the pseudocode above, stepping through the code and understanding the output, with the output flag, `print_flag` sent as `True` .

```
N=8
A = list(range(N))
random.shuffle(A)
print(A)
Quicksort(A,0,N-1,True)
print(A)
```

`[7, 1, 6, 0, 5, 4, 2, 3]`	initial list, the 3 will be the first element partitioned around
`[1, 0, 2, 3, 5, 4, 6, 7]`	entire list is partitioned originally, 3 is in sorted order and there are two sublists after partition, the left has all elements ≤ 3 and the right are >3
`[1, 0, 2, 3, 5, 4, 6, 7]`	left sublist is partitioned around 2, 0 is in sorted order, and the resulting left sublist has all elements ≤ 2 and the right is empty, none are > 2
`[0, 1, 2, 3, 5, 4, 6, 7]`	left sublist is partitioned around 0, 0 is in sorted order, and the resulting left sublist is empty, none are ≤ 0 and the right has all elements > 0
`[0, 1, 2, 3, 5, 4, 6, 7]`	original left sublist is partitioned around 7, 7 is in sorted order, and the resulting left sublist after partition has all elements ≤ 7 and the right sublist is empty, no elements > 7
`[0, 1, 2, 3, 5, 4, 6, 7]`	left sublist is partitioned around 6, 6 is in sorted order, and the resulting left sublist has all elements ≤ 6 and the right is empty, none are > 6
`[0, 1, 2, 3, 4, 5, 6, 7]`	left sublist is partitioned around 4, 4 is in sorted order, and the resulting left sublist is empty, none are ≤ 4 and the right has all elements > 4
`[0, 1, 2, 3, 4, 5, 6, 7]`	final, sorted list