Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

QUICKSORT algorithm

- QUICKSORT uses the divide-and-conquer technique
- It's not asymptotically faster than HEAPSORT or MERGESORT
- However, since it is **in-place** and has **low hidden constants**, it's most efficient in practice
- Java's Arrays.sort uses a variation of the QUICKSORT algorithm

QUICKSORT algorithm

- The idea is to pick a pivot x, and divide the array like below, and then recurse on L and G

$$\underbrace{\boxed{\quad\quad \le x \quad\quad}}_{L}\boxed{x}\underbrace{\boxed{\quad\quad > x \quad\quad}}_{G}$$

- This process is done by PARTITION, which at the end returns the index of x, as well.

QUICKSORT algorithm

- After doing the partition, L and G are not necessarily sorted.

- However, if we sort L and G recursively, the whole array will be sorted x since is in its correct position.

Basic QUICKSORT algorithm

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

Divide

Conquer

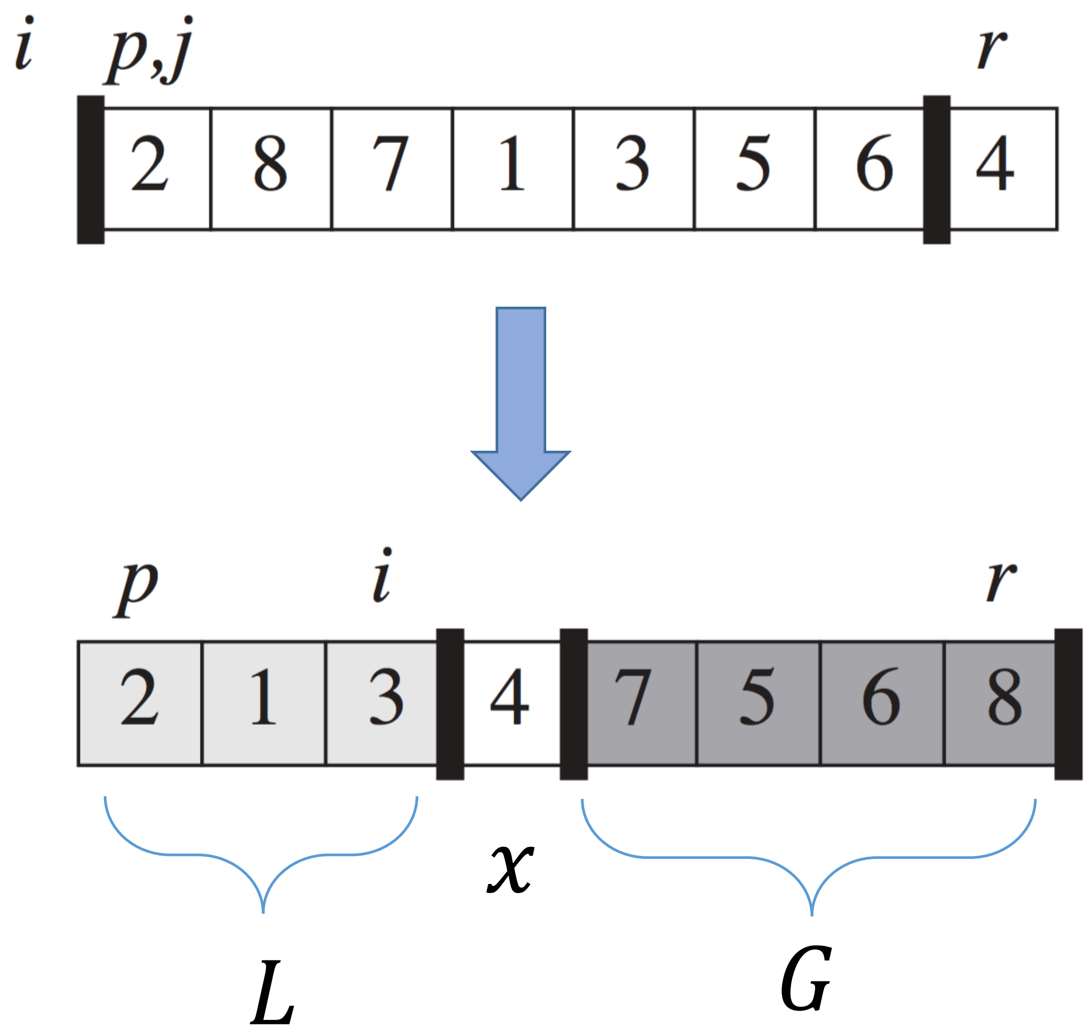
No Combine is necessary

- The initial call is QUICKSORT(A, 1, $A.length$)

Basic QUICKSORT algorithm

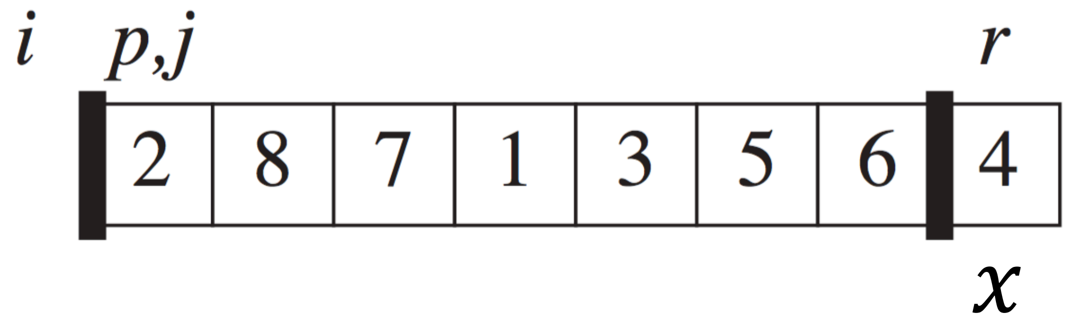
- PARTITION is the important part of the algorithm

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.




```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

$i \quad j$

2	8	7	1	3	5	6	4

x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

i	j						
2	8	7	1	3	5	6	4
							x

```
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

i	j						
2	8	7	1	3	5	6	4
							x

```
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

blue $\leq x$

green $> x$

i		j					
2	8	7	1	3	5	6	4
							x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

i		j					
2	8	7	1	3	5	6	4
							x

```
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

i			j				
2	8	7	1	3	5	6	4
							x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

	i		j				
2	1	7	8	3	5	6	4
							x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

	i			j			
2	1	7	8	3	5	6	4
							x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

		i		j			
2	1	3	8	7	5	6	4
							x

$$
\begin{array}{l}
\quad \textbf{if } A[j] \le x \\
\qquad i = i + 1 \\
\qquad \text{exchange } A[i] \text{ with } A[j]
\end{array}
$$

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

		i			j		
2	1	3	8	7	5	6	4
							x

if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

		i				j	
2	1	3	8	7	5	6	4
							x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

		i				j	
2	1	3	8	7	5	6	4
							x

```
if A[j] ≤ x
    i = i + 1
    exchange A[i] with A[j]
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

		i					j, r
2	1	3	8	7	5	6	4
							x

```
exchange A[i + 1] with A[r]
return i + 1
```

Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

		i					j, r
2	1	3	4	7	5	6	8
							x

```
exchange A[i + 1] with A[r]
return i + 1
```

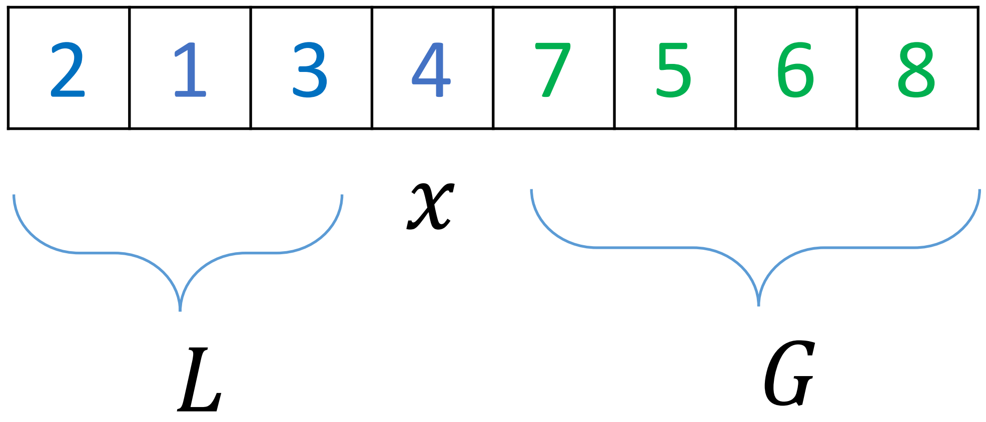
Partition

- For a given array A, we pick $A[r]$ as the pivot.
- Here, $x = 4$ is the pivot.

$$\underbrace{\boxed{2}\,\boxed{1}\,\boxed{3}}_{\le x}\;\underset{x}{\boxed{4}}\;\underbrace{\boxed{7}\,\boxed{5}\,\boxed{6}\,\boxed{8}}_{> x}$$

Partition

- Since x is in its **correct** position, sorting the G and sorting L recursively, will result in a sorted array.



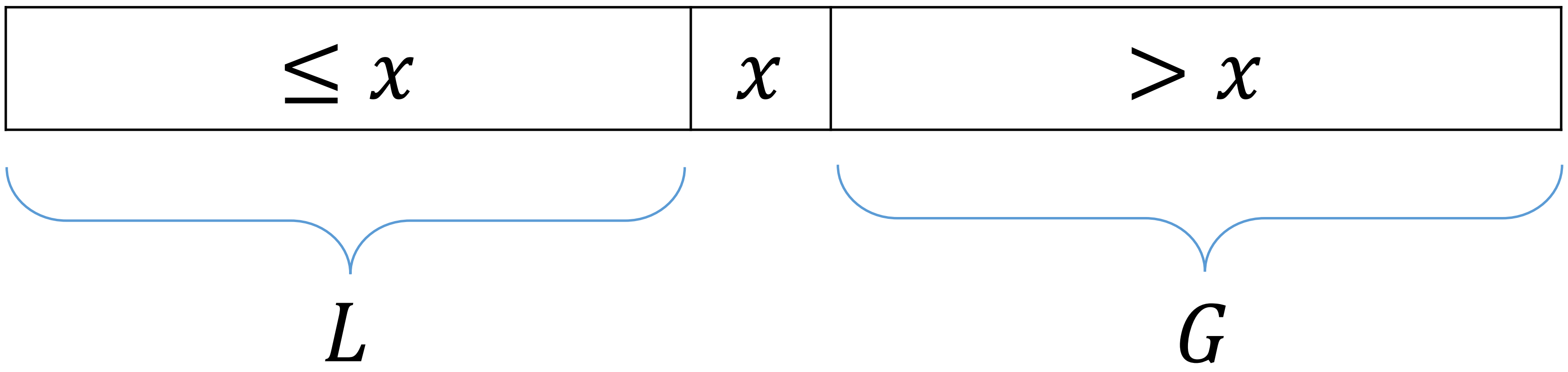
Time complexity

- The running time of Partition is $\Theta(n)$
- What can we say about the running time of Quicksort?

Time complexity

- The running time of PARTITION is $\Theta(n)$
- What can we say about the running time of QUICKSORT?

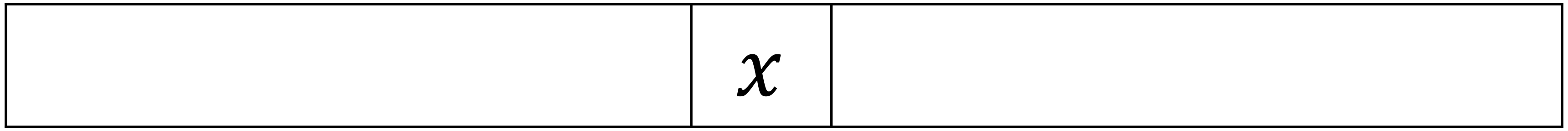
$$T(n) = T(size\ of\ L) + T(size\ of\ G) + \Theta(n)$$



Time complexity

$$T(n) = T(size\ of\ L) + T(size\ of\ G) + \Theta(n)$$

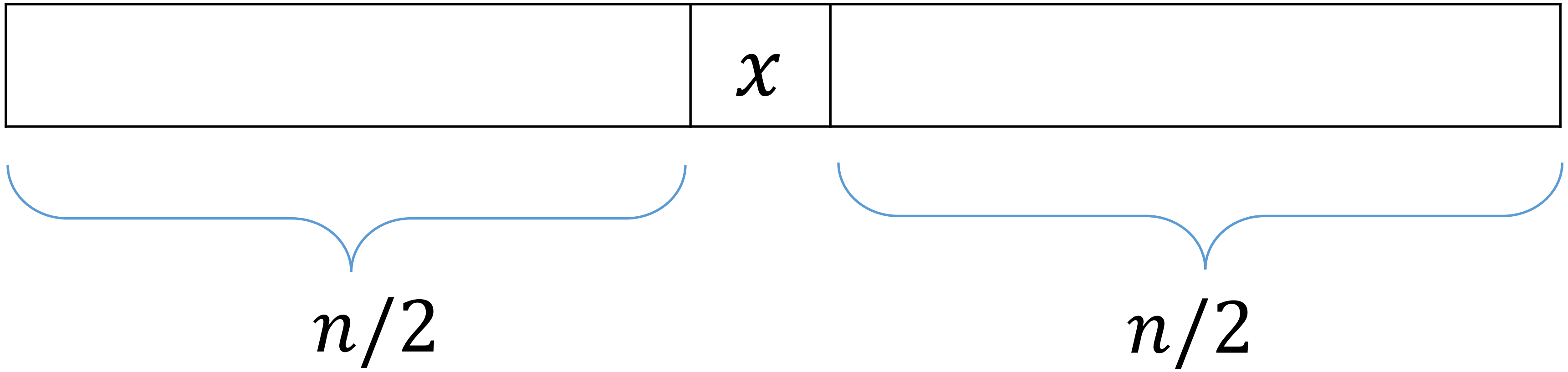
Question: What is the **best partition** that makes $T(n)$ to be as small as possible?



Time complexity

$$T(n) = T(size\ of\ L) + T(size\ of\ G) + \Theta(n)$$

Question: What is the **best partition** that makes $T(n)$ to be as small as possible?



$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \rightarrow T(n) = \boldsymbol{\Theta(n\log n)}$$

Time complexity

- Best case happens if the pivot is the ***median*** of the n elements.

- However, even if the input is randomized, in expectation $A[r]$ will partition the array in ***almost half*** which is good enough.

- **Note:** Median can be found in linear time with a complicated **deterministic algorithm** that we will discuss later. But we avoid to use it in $\textsc{Quicksort}$ since it doesn't work well in practice. But such an algorithm **guarantees** the $\Theta(n \log n)$ time in the worst-case.

Side note on writing recurrences

- To be very exact the recurrence for the best-case of quicksort, and merge-sort is

$$T(n) = \begin{cases} \Theta(1) & if\ n \leq 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & if\ n > 1 \end{cases}$$

- But we ignore **small n's, floors** and **ceilings** for simplicity. And only write

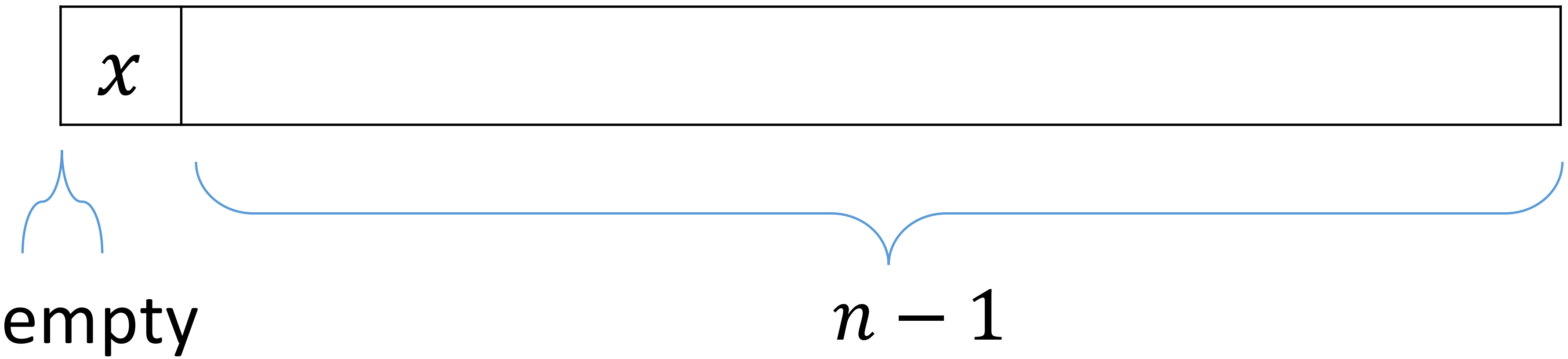
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

- It's because these usually change the solution by only a constant factor.

Time complexity

$$T(n) = T(size\ of\ L) + T(size\ of\ G) + \Theta(n)$$

Question: What is the **worst partition** that makes $T(n)$ to be as large as possible?



$$T(n) = T(0) + T(n-1) + \Theta(n) \rightarrow T(n) = \boldsymbol{\Theta(n^2)}$$

Time complexity

- Worst case is when every time either L is empty or G is empty; hence, $T(n) = \Theta(n^2)$. You can use recursion tree method to solve the recursion in the previous slide.

- This happens when the pivot is the **maximum** or the **minimum** in the current subarray.

- An example of this is when the input is sorted either in an increasing or decreasing order.

Time complexity

- So far, we have an *intuition* that a random input might make the basic QUICKSORT work well. Therefore, **shuffling the input first** and then calling quicksort seems like a good idea.

- So, let's **pick the pivot randomly** instead of randomizing (shuffling) the input.

- Then, we prove that this idea works very well.

RANDOMIZED-QUICKSORT

RANDOMIZED-PARTITION(A, p, r)

$i =$ RANDOM(p, r)
exchange $A[r]$ with $A[i]$
return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)

if $p < r$
    $q =$ RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT$(A, p, q - 1)$
    RANDOMIZED-QUICKSORT$(A, q + 1, r)$

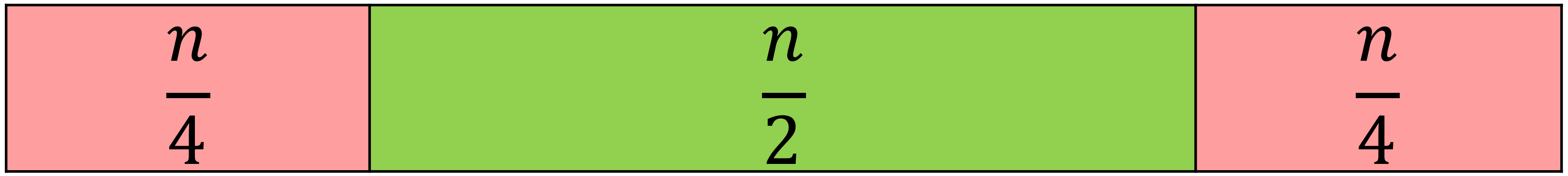
Randomized-Quicksort

The analysis of randomized-quicksort is a bit complicated (optional – CLRS section 7.4.2), so we analyze a simpler version of it which provides **a better intuition** for the $\Theta(n \log n)$ complexity.

Although, you can understand most of that proof since you know the properties of expectation.

Paranoid-Quicksort

- We call it paranoid because it keeps picking a pivot until it gets a good pivot and only then proceeds.
- We are not looking for an ideal pivot.
- We look for one that is **close enough** from the the median, if we consider the sorted order.
- In particular, between the first and the third quartile.



source: https://ocw.mit.edu/courses/

PARANOID-QUICKSORT

The sketch of PARANOID-PARTITION(A, p, r):

1. Pick an i randomly in $[p, r]$
2. Count how many elements are $\leq A[i]$
3. If the count is in $[n/4, 3n/4]$, i.e. it's a good pivot, do the exchange and call PARTITION, otherwise repeat the process.

Paranoid-Quicksort

- We only need to replace Randomized-Partition with Paranoid-Partition to get the Paranoid-Quicksort algorithm.

- **Exercise:** write a pseudocode for Paranoid-partition.

Analysis of Paranoid-Quicksort

- What is the recursion for the worst-case input?

Analysis of Paranoid-Quicksort

- What is the recursion for the worst-case input?
  $T(n) = T(n/4) + T(3n/4) + f(n)$
- Where $f(n)$ is the expected running time of Paranoid-Partition.

Analysis of Paranoid-Quicksort

- What is the recursion for the worst-case input?
  $T(n) = T(n/4) + T(3n/4) + f(n)$
- Where $f(n)$ is the expected running time of Paranoid-Partition.

Wait a second …

1. What is this analysis even called?

Analysis of PARANOID-QUICKSORT

- What is the recursion for the worst-case input?
  $T(n) = T(n/4) + T(3n/4) + f(n)$
- Where $f(n)$ is the expected running time of PARANOID-PARTITION.

Wait a second …

1. What is this analysis even called? **Worst-case expected running time**

Analysis of Paranoid-Quicksort

- What is the recursion for the worst-case input?
  $T(n) = T(n/4) + T(3n/4) + f(n)$
- Where $f(n)$ is the expected running time of Paranoid-Partition.

Wait a second …

2. How is this different from averge-case analysis?

Analysis of Paranoid-Quicksort

- What is the recursion for the worst-case input?
  $T(n) = T(n/4) + T(3n/4) + f(n)$
- Where $f(n)$ is the expected running time of Paranoid-Partition.

Wait a second …

2. How is this different from averge-case analysis?

Averge-case means getting the average time over all inputs. But worst-case means considering only the worst input. On the other hand, expected running time is a **property of randomized algorithms** and determines the time complexity of the algorithm in expectation for some input.

Analysis of Paranoid-Quicksort

- So, we only need to compute $f(n)$ which depends on how many times Paranoid-Partition tries before finding a good pivot.

- Since the probability of picking a good pivot is at least 1/2. 2 is the expected number of trials.

Analysis of PARANOID-QUICKSORT

- Therefore, $f(n) = 2c\,n = \Theta(n)$ in expectation, and we have:

  $$T(n) = T(n/4) + T(3n/4) + \Theta(n)$$

- Since the Master Theorem doesn't apply to this kind of recurrence we use the recursion tree method.

- We obtain $T(n) = \Theta(n \log n)$ as the **worst-case expected running time**.

Side note on randomized algorithms

- Randomized algorithms are also called probabilistic algorithms
- There are two types of randomized algorithms

Las Vegas **Monte Carlo**

Side note on randomized algorithms

- Randomized algorithms are also called probabilistic algorithms
- There are two types of randomized algorithms

Las Vegas

Gambles on **time** but guarantees correctness

Monte Carlo

Gambles on **correctness**, but guarantees time

Side note on randomized algorithms

- Randomized algorithms are also called probabilistic algorithms
- There are two types of randomized algorithms

Las Vegas

Gambles on **time** but guarantees correctness

Example: RANDOMIZED-QUICKSORT

Monte Carlo

Gambles on **correctness**, but guarantees time

Take a randomized algorithms course to find out more about this

Properties of quicksort

- Quicksort **is not stable** because the **exchange of $\boldsymbol{A[i]}$ and $\boldsymbol{A[r]}$** in randomized-partition can change the original order of equal elements.

- However, with $\Theta(n)$ extra space we can turn any unstable algorithm into a stable one.

- But, this extra space will cause the sorting algorithm to not be in-place anymore.

Making a sort stable

- (x, y) is pair of numbers which we call an **ordered pair.**
- Examples: $(1, 2)$, $(6, 7)$

- Since the **order matters** $(1, 2) \neq (2, 1)$

- Two pairs are (x, y) and (w, z) are equal if

$$x = w \textbf{ AND } y = z.$$

Making a sort stable

- When comparing, priority is with the **first number** in the pair.
- We say $(x, y) < (w, z)$ if

  $(x < w)$ **OR**

  $(x = w$ **AND** $y < z)$
- For example,

$$(2,3) < (2,5)$$
$$(2,3) < (7,7)$$
$$(7,8) > (7,1)$$

Making a sort stable

- Convert the input array A, to an array of ordered pairs P

A	5	3	5	6	7	5

$\Downarrow$

P	$(5, 1)$	$(3, 2)$	$(5, 3)$	$(6, 4)$	$(7, 5)$	$(5, 6)$

- Since the second number of the pairs are **unique** indices, all elements of P are **guaranteed to be distinct**.

Making a sort stable

- Now, we can sort P, while preserving the original order of equal numbers.

P	$(\mathbf{5}, 1)$	$(\mathbf{3}, 2)$	$(\mathbf{5}, 3)$	$(\mathbf{6}, 4)$	$(\mathbf{7}, 5)$	$(\mathbf{5}, 6)$

$\Downarrow$

P	$(\mathbf{3}, 2)$	$(\mathbf{5}, 1)$	$(\mathbf{5}, 3)$	$(\mathbf{5}, 6)$	$(\mathbf{6}, 4)$	$(\mathbf{7}, 5)$

A	$\mathbf{3}$	$\mathbf{5}$	$\mathbf{5}$	$\mathbf{5}$	$\mathbf{6}$	$\mathbf{7}$

Making a sort stable

- After sorting P, because the priority is with a_i's, the pairs are sorted based on a_i's and if there are equal a_i's the one whose original index was lower appears first.
- Then, we can traverse the array P and put the first elements back into A.
- Since we are using $\Theta(n)$ extra variables to make P, the soring will not be in-place anymore.

Making a sort stable

- Making an ordered pair in Java

```java
class Pair implements Comparable<Pair>{
    int first;
    int second;

    public Pair(int first, int second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public int compareTo(Pair p) {
        if(first == p.first && second == p.second)
            return 0; //means equal
        if(first < p.first || (first == p.first && second < p.second))
            return -1; //this is less than p
        return 1; //this is greater than p
    }
}
```

Properties of quicksort - optional from here :)

- The simplifying assumption made in randomized and paranoid quicksort is that all elements are **distinct**.
- However, if there are duplicate elements we need to partition as follows, and then recurse on only G and L.

$$\underbrace{\boxed{\quad < x \quad}}_{L}\underbrace{\boxed{\quad = x \quad}}_{E}\underbrace{\boxed{\quad > x \quad}}_{G}$$

Properties of quicksort

- If interested, you can take a look at **Problem 7.2.b** in CLRS.
- Also, an idea mentioned in **Exercise 7.1.2** can be used the make the paranoid partition work when equal elements are present.

Properties of quicksort

- We say that quicksort is an **in-place** sorting; however, the extra space required is expected to be $\Theta(\log n)$ due to recursive calls.

$\textsc{Quicksort}(A, p, r)$

if $p < r$
  $q = \textsc{Partition}(A, p, r)$
  $\textsc{Quicksort}(A, p, q - 1)$
  $\textsc{Quicksort}(A, q + 1, r)$

- But still we consider it an in-place soring since the extra memory beside the input array is much less than $\Theta(n)$.