Chapter 7
Quicksort

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, McGraw-Hill, 2001.

- Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2001.
- These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 7 Topics

- What is quicksort?
- How does it work?
- Performance of quicksort
- Randomized version of quicksort

Description of Quicksort

- Quicksort is another divide-and-conquer algorithm.
- Basically, what we do is divide the array into two subarrays, so that all the values on the left are smaller than the values on the right.
- We repeat this process until our subarrays have only 1 element in them.
- When we return from the series of recursive calls, our array is sorted.

Description of Quicksort

- **Divide:** Partition A[p..r] into two (possibly empty) subarrays A[p..q-1] and A[q+1 .. r] such each element of A[p..q-1] $\leq$ A[q] and A[q] $\leq$ each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: A[p..r] is now sorted.

The Quicksort Algorithm

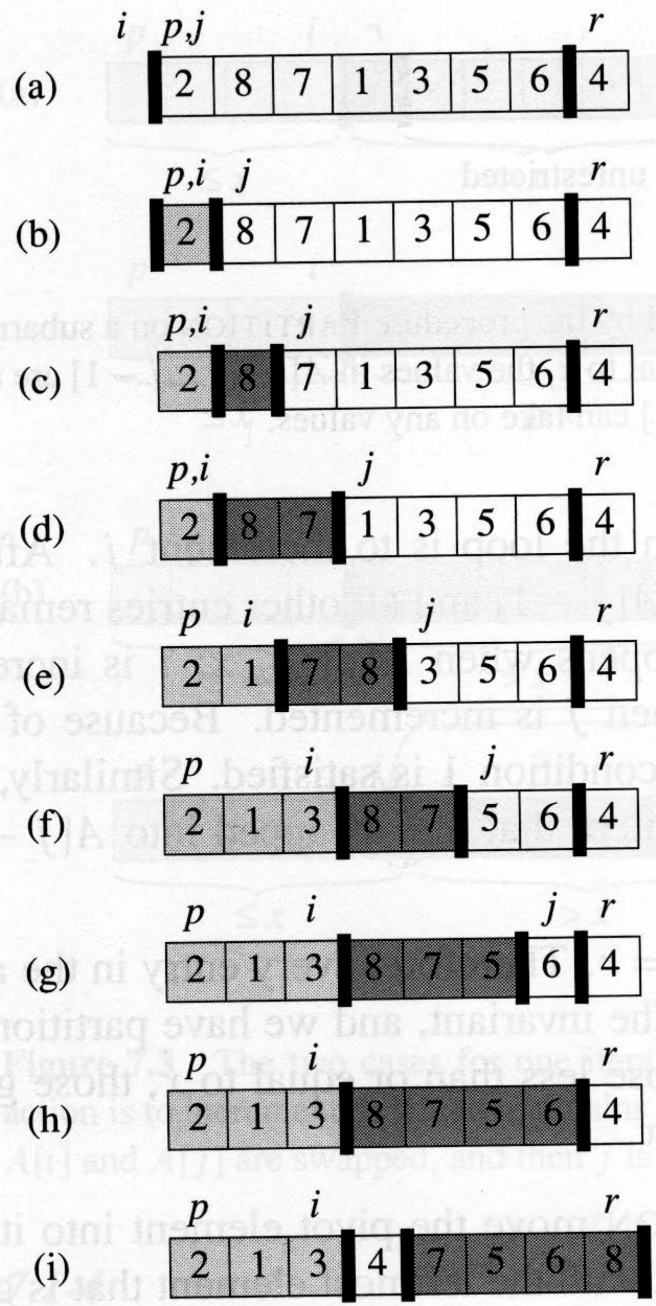
```
QUICKSORT(A,p,r)
  if p < r
     then q ← PARTITION(A,p,r)
          QUICKSORT(A,p,q-1)
          QUICKSORT(A,q+1,r)
```

Initial call:

QUICKSORT(A,1, length[A])

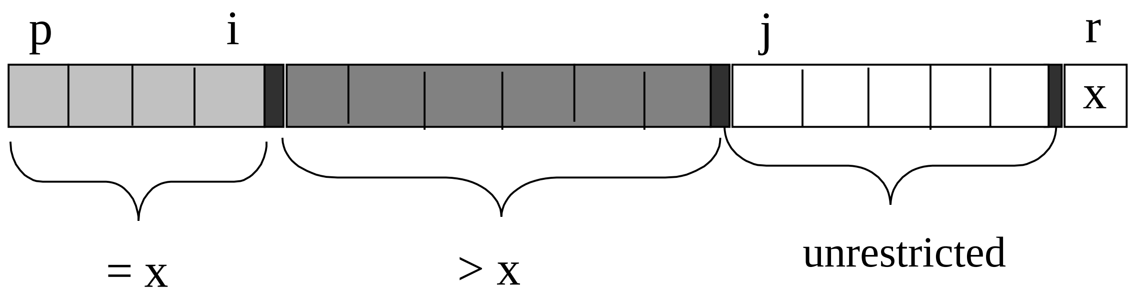
The Partition Algorithm

```
PARTITION(A,p,r)
x ← A[r]
i ← p - 1
for j ← p to r-1
  do if A[j] ≤ x
    then i ← i + 1
         exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```




```
PARTITION(A,p,r)
x ← A[r]
i ← p - 1
for j ← p to r-1
 do if A[j] ≤ x
  then i ← i + 1
    exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

Regions of Subarray Maintained by PARTITION



Each value in A[p..i] $\leq$ x.

Each value in A[i+1..j-1] > x.

A[r] = x.

A[j..r-1] can take on any values.

Loop Invariant for Partition

We can prove the correctness of the Partition algorithm by an analysis of its loop invariant conditions:

At the beginning of each iteration of the loop in lines 3-6, for any array index k,

1. if $p \le k \le i$, then $A[k] \le x$.
2. if $i + 1 \le k \le j-1$, then $A[k] > x$.
3. if $k = r$, then $A[k] = x$.

Loop Invariant Correctness

Initialization:

- Prior to the first iteration of the loop, $i = p - 1$, and $j = p$. There are no values between p and i, and no values between i+1 and j-1, so the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Loop Invariant Correctness

Maintenance:

- There are two cases to consider depending on the outcome of the test in line 4:
- When A[j] > x, the only action in the loop is to increment j. After j is incremented, condition 2 holds for all A[j-1] and all other entries remain unchanged.
- When A[j] $\leq$ x, i is incremented, A[i] and A[j] are swapped, and then j is incremented. Because of the swap, we now have that A[i] $\leq$ x, and condition 1 is satisfied. Similarly, we also have that A[j-1]>x, since the item that was swapped into A[j-1] is, by the loop invariant, greater than x.

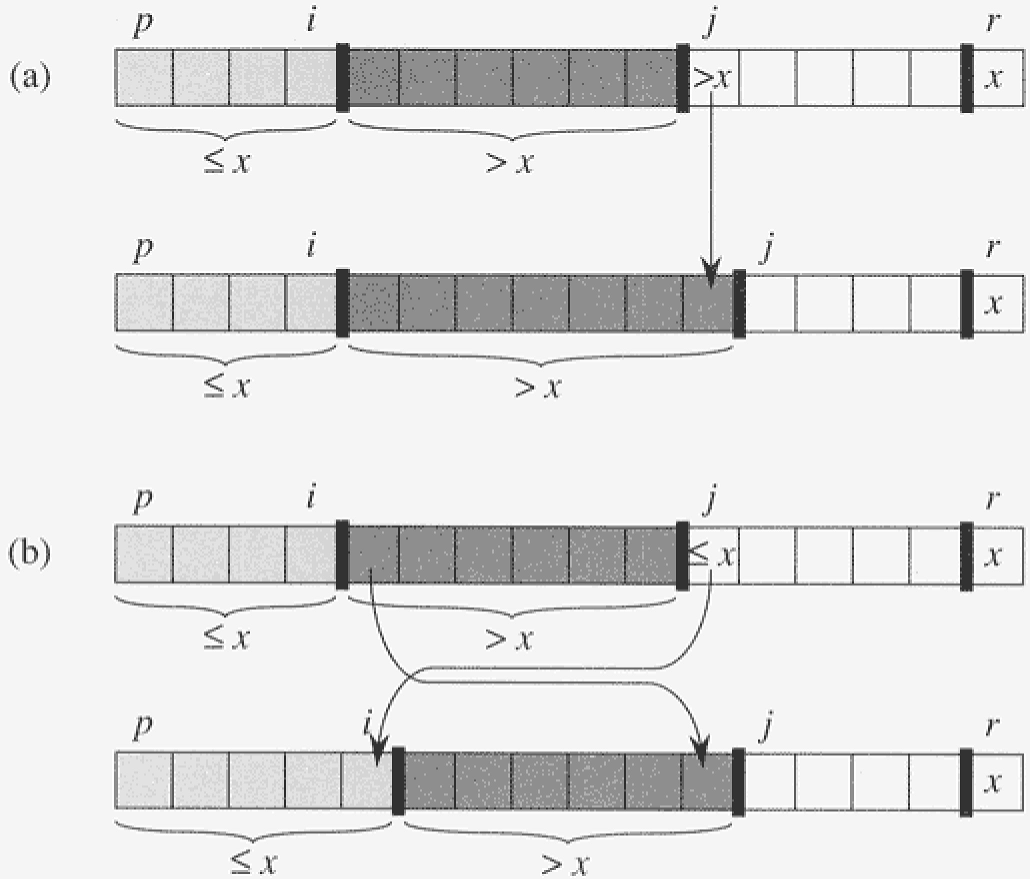


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Loop Invariant Correctness

Termination:

At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets:

those less than or equal to x,

those greater than x, and

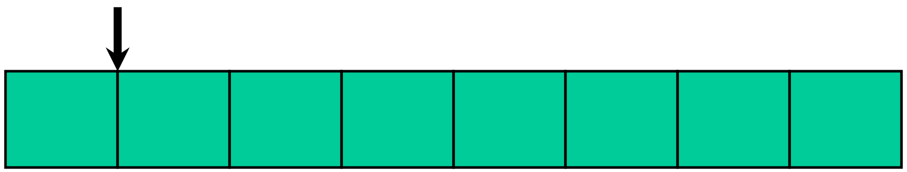
a singleton set containing x.

Performance of Quicksort

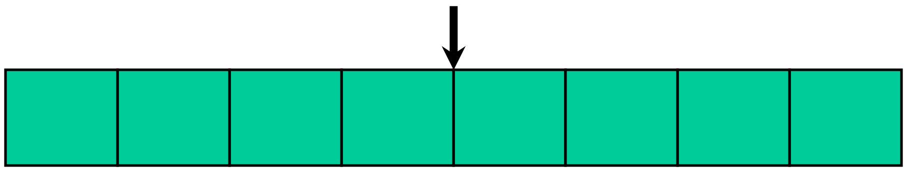
- Depends on whether the partitioning is balanced or unbalanced:
  - Balance of partition depends on location of pivot
  - If balanced, runs as fast as Merge sort
  - If unbalanced, runs as slowly as Insertion sort

Worst/Best case partitioning

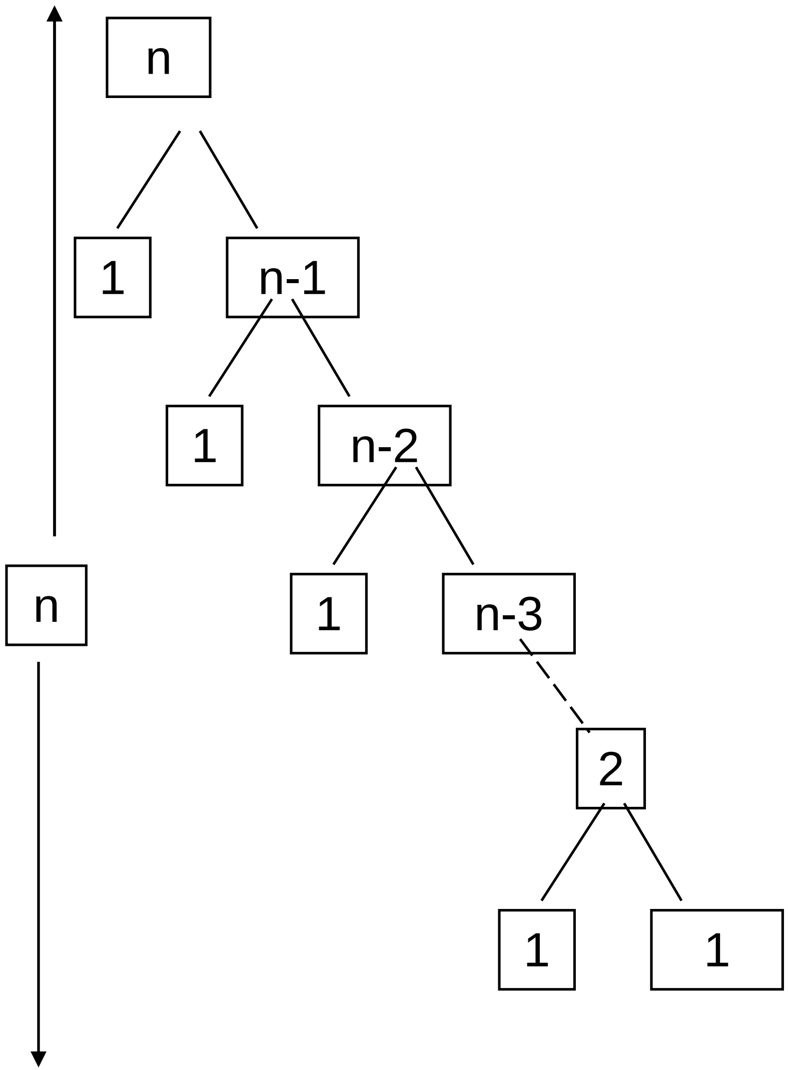
- **Worst case:**
  - One partition contains n – 1 elements
  - The other partition contains 1 element



- **Best case:**
  - Both partitions are of equal size



Worst case partitioning



n

n-1

n-2

n-3

. . .

2

1

____

Total of $\Theta(n^2)$

Worst Case Performance

Assume we have a maximally unbalanced partition at each step, splitting off just 1 element from the rest each time. This means we will have to call Partition n-1 times.

The cost of Partition is: $\Theta(n)$

So the recurrence for Quicksort is:

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n)
\end{aligned}
$$

Worst Case Performance

We can solve the recurrence by iteration:

$$
\begin{aligned}
T(n) &= \Theta(n) + T(n-1) \\
&= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \dots \quad + \Theta(1) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta\left(\sum_{k=1}^{n} k\right) \\
&= \Theta(n^2)
\end{aligned}
$$

Best Case

Best case: Each time the partitioning is done, it splits the array into two regions of equal size. After each call to Partition, each subarray contains n/2 of the elements from the previous call. If we halve the remaining elements each time, we will have to call Partition $\log_2 n$ times.

Best Case Performance

Best case: Call Partition, which splits the array into two equal-size subarrays. For each of the 2 subarrays, call Partition, which splits ...

Recurrence for Quicksort:

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

This matches Master Method case 2. Solving the recurrence we get:

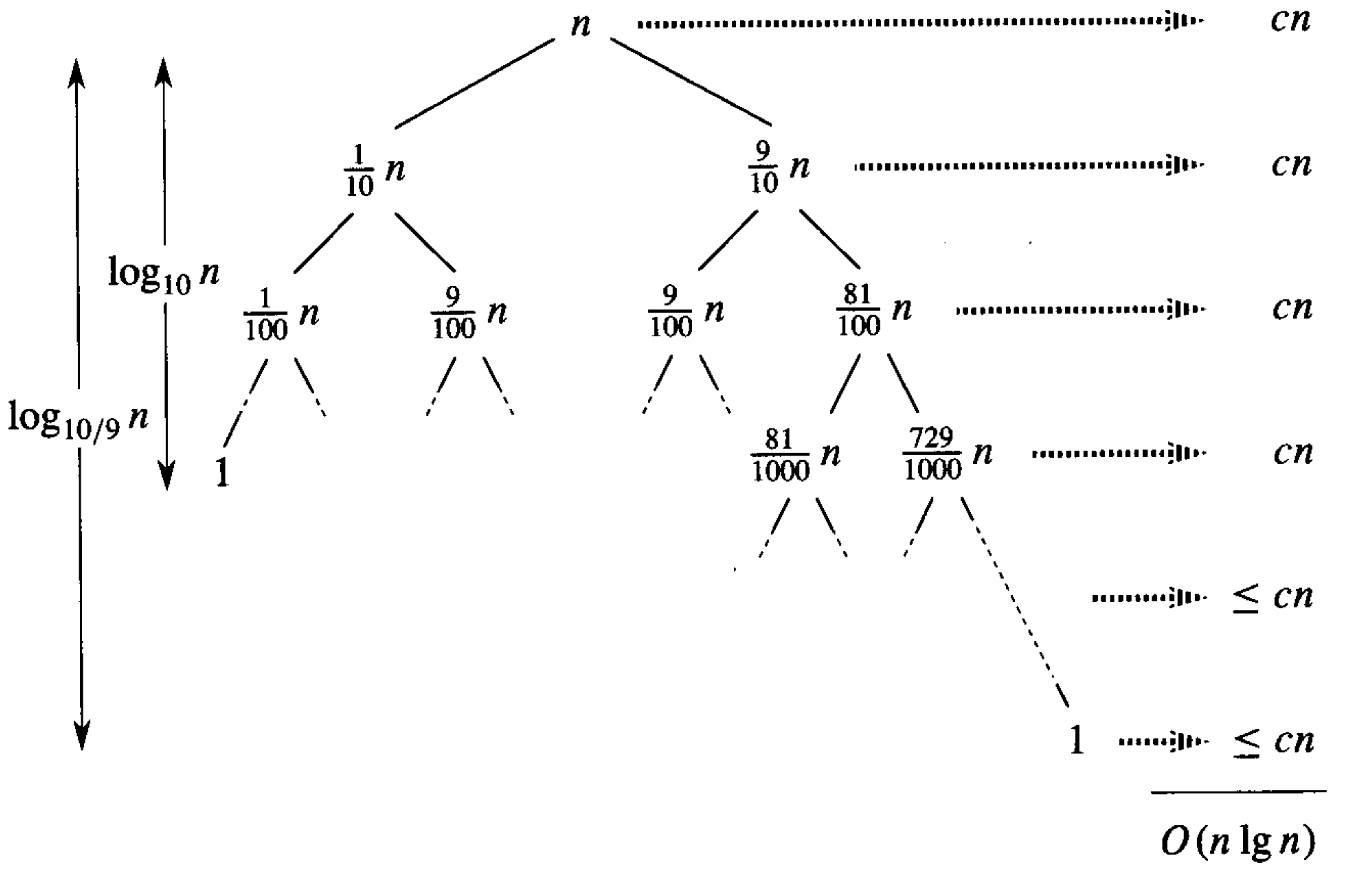
$$T(n) = O(n \lg n)$$

Average Case

- Average case analysis is complex and difficult.
- However, we can observe that average-case performance is much closer to best-case than worst case.
- Suppose split is always 9-to-1
- Recurrence:

$$
\begin{aligned}
T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\
&= T(9n/10) + T(n/10) + cn \\
&= \log_{10/9} n * n = O(n \lg n)
\end{aligned}
$$

Average Case Analysis



Average Case Analysis

- What if we have a 99-1 split?
- We still have a running time of O(n lg n)
- Any split of *constant proportionality* yields a recursion tree of depth Θ(lg n), where the cost at each level is O(n).
- So whenever the split is of constant proportionality, Quicksort performs on the order of O(n lg n).

Average Case

- Best case:

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

- Average case example:

  $T(9n/10) + T(n/10) + cn$

- Worst case:

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

Randomized Version of Quicksort

- When an algorithm has an average case performance and worst case performance that are very different, we can try to minimize the odds of encountering the worst case.
- We can:
  - Randomize the input
  - Randomize the algorithm

Randomized Version of Quicksort

- **Randomizing the input**

  With a given set of input numbers, there are very few permutations that produce the worst-case performance in Quicksort.

  We can randomly permute the numbers in a n-element array in O(n) time.

  For Quicksort, add an initial step to randomize the input array.

  Running time is now independent of input ordering.

Randomized Version of Quicksort

- **Randomizing the algorithm:**

  In standard Quicksort, the worst case is encountered when we choose a bad pivot.

  If the input array is already sorted (or inverse sorted), we will always pick a bad pivot.

  But if we pick our pivot randomly, we will rarely get a bad pivot.

  So, randomly choose a pivot element in A[p..r].

  Running time is now independent of input ordering.

Randomized Partition

```
RANDOMIZED-PARTITION (A, p, r)
i ← RANDOM (p, r)
exchange A[r] ↔ A[i]
return PARTITION (A, p, r)
```

Randomized Quicksort

```
RANDOMIZED-QUICKSORT (A, p, r)
   if p < r
   then q ← RANDOMIZED-PARTITION (A, p, r)
        RANDOMIZED-QUICKSORT (A, p, q-1)
        RANDOMIZED-QUICKSORT (A, q+1, r)
```

Conclusion

Quicksort runs O(n lg n) in the best and average case, but O(n^2) in the worst case.

Worst case scenarios for Quicksort occur when the array is already sorted, in either ascending or descending order.

We can increase the probability of obtaining average-case performance from Quicksort by using Randomized-partition.