Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- **Divide**: pick a random element \( x \) (called pivot) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal \( x \)
  - \( G \) elements greater than \( x \)
- **Recur**: sort \( L \) and \( G \)
- **Conquer**: join \( L \), \( E \) and \( G \)
Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm \texttt{partition}(S, p)
\begin{description}
\item Input\ sequence $S$, position $p$ of pivot\n\item Output\ subsequence $L$, $E$, $G$ of the elements of $S$ less than, equal to, or greater than the pivot, resp.
\end{description}
\begin{verbatim}
L, E, G \leftarrow \text{empty sequences}
x \leftarrow S.remove(p)
while \neg S.isEmpty()
  \begin{cases}
    y \leftarrow S.remove(S.first()) & \text{if } y < x \\
    L.addLast(y) & \text{if } y = x \\
    E.addLast(y) & \text{if } y > x\end{cases}
return L, E, G
\end{verbatim}
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
- Each node represents a recursive call of quick-sort and stores
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1

Execution Example

- Pivot selection

© 2014 Goodrich, Tamassia, Goldwasser
Execution Example (cont.)

Partition, recursive call, pivot selection

 Execution Example (cont.)

Partition, recursive call, base case
Execution Example (cont.)

Recursive call, ..., base case, join

```
7 2 9 4 3 7 6 1
2 4 3 1 → 1 2 3 4
1 → 1
4 3 → 3 4
4 → 4
```

Execution Example (cont.)

Recursive call, pivot selection

```
7 2 9 4 3 7 6 1
2 4 3 1 → 1 2 3 4
1 → 1
4 3 → 3 4
4 → 4
7 9 7
```

© 2014 Goodrich, Tamassia, Goldwasser
Execution Example (cont.)

Partition, ..., recursive call, base case

Join, join
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of $L$ and $G$ has size $n - 1$ and the other has size 0.
- The running time is proportional to the sum $n + (n - 1) + \ldots + 2 + 1$.
- Thus, the worst-case running time of quick-sort is $O(n^2)$.

---

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size $s$.
  - **Good call**: the sizes of $L$ and $G$ are each less than $3s/4$.
  - **Bad call**: one of $L$ and $G$ has size greater than $3s/4$.

- A call is **good** with probability $1/2$.
  - $1/2$ of the possible pivots cause good calls.
Expected Running Time, Part 2

- **Probabilistic Fact:** The expected number of coin tosses required in order to get \( k \) heads is \( 2^k \).
- For a node of depth \( i \), we expect:
  - \( i/2 \) ancestors are good calls
  - The size of the input sequence for the current call is at most \( (3/4)^{i/2} n \)
- Therefore, we have:
  - For a node of depth \( 2 \log_{4/3} n \), the expected input size is one.
  - The expected height of the quick-sort tree is \( O(\log n) \).
- The amount of work done at the nodes of the same depth is \( O(n) \).
- Thus, the expected running time of quick-sort is \( O(n \log n) \).

In-Place Quick-Sort

- Quick-sort can be implemented to run in-place.
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that:
  - The elements less than the pivot have rank less than \( h \).
  - The elements equal to the pivot have rank between \( h \) and \( k \).
  - The elements greater than the pivot have rank greater than \( k \).
- The recursive calls consider:
  - Elements with rank less than \( h \).
  - Elements with rank greater than \( k \).

**Algorithm** inPlaceQuickSort\((S, I, R)\)

- **Input** sequence \( S \), ranks \( I \) and \( R \)
- **Output** sequence \( S \) with the elements of rank between \( I \) and \( R \) rearranged in increasing order.
- If \( I \geq R \) return
- \( i \leftarrow \) a random integer between \( I \) and \( R \)
- \( x \leftarrow S.\text{elemAtRank}(i) \)
- \( (h, k) \leftarrow \text{inPlacePartition}(x) \)
- inPlaceQuickSort\((S, I, h - 1)\)
- inPlaceQuickSort\((S, k + 1, R)\)
In-Place Partitioning

Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

\[ j \quad k \]

(pivot = 6)

Repeat until j and k cross:

- Scan j to the right until finding an element \( > x \).
- Scan k to the left until finding an element \( < x \).
- Swap elements at indices j and k.

```
3 2 5 1 0 7 3 5 9 2 7 9 8 9 7
```

```
3 2 5 1 0 7 3 5 9 2 7 9 8 9 7
```

Java Implementation

```java
/** Sort the subarray S[a..b] inclusive. */
private static <K> void quickSortInPlace(K[] S, Comparator<K> comp,
                                       int a, int b) {
    if (a >= b) return;        // subarray is trivially sorted
    int left = a;
    int right = b - 1;
    K pivot = S[b];
    K temp;                   // temp object used for swapping
    while (left <= right) {    // scan until reaching value equal or larger than pivot (or right marker)
        while (left <= right && comp.compare(S[left], pivot) < 0) left++;
        // scan until reaching value equal or smaller than pivot (or left marker)
        while (left <= right && comp.compare(S[right], pivot) > 0) right--;
        // indices did not strictly cross
        if (left <= right) {    // swap values and shrink range
            temp = S[left];    S[left] = S[right];    S[right] = temp;
            left++; right--;
        }
    }
    // put pivot into its final place (currently marked by left index)
    temp = S[left];    S[left] = S[b];    S[b] = temp;
    // make recursive calls
    quickSortInPlace(S, comp, a, left - 1);
    quickSortInPlace(S, comp, left + 1, b);
}
```

© 2014 Goodrich, Tamassia, Goldwasser
### Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
</table>
| selection-sort | $O(n^2)$ | - in-place  
                   - slow (good for small inputs) |
| insertion-sort | $O(n^2)$ | - in-place  
                   - slow (good for small inputs) |
| quick-sort   | $O(n \log n)$ | - in-place, randomized  
                   - fastest (good for large inputs) |
| heap-sort    | $O(n \log n)$ | - in-place  
                   - fast (good for large inputs) |
| merge-sort   | $O(n \log n)$ | - sequential data access  
                   - fast (good for huge inputs) |