Merge and Quick Sort†

- Merge Sort‡
- Merge Sort Tree
- Implementation
- Quick Sort‡
- Pivot Item
- Randomized Quick Sort

†Adapted from: Goodrich and Tamassia, Data Structures and Algorithms in Java, John Wiley & Son (1998).
‡Running time assumes no duplicate elements.
Merge Sort
Divide and Conquer

- **Merge sort** is based on a method of algorithm design called *divide and conquer*.

- In general:
  1. **Divide** - If the input size is below a certain threshold, solve the problem directly. Otherwise, divide the input data into two or more **disjoint** set.

  2. **Recurse** - Recursively solve the subproblems associated with the subsets.

  3. **Conquer** - **Merge** the solutions to the subproblems into a solution of the original problem.
Merge Sort
Algorithm

- Consider a Sequence, $S$, with $n$ elements:

- In general:
  1. **Divide** - Remove the elements in $S$ and place into two sequences $S_1$ and $S_2$, each containing about half of the elements.

    $$S_1 = \{x : x \in \text{first } \lfloor n/2 \rfloor \text{ elements of } S\}$$
    $$S_2 = \{x : x \in \text{last } \lceil n/2 \rceil \text{ elements of } S\}$$

  2. **Recurse** - Recursively sort sequences $S_1$ and $S_2$.

  3. **Conquer** - **Merge** the elements of the sorted subsequences, $S_1$ and $S_2$, into a unique sorted sequence.
**Merge Sort Tree**

- Visualize a merge sort by means of a binary tree, $T$:
  
  - Each node of $T$ represents a recursive call of the merge-sort algorithm.
  
  - Associate the root of the $T$ with the sequence $S$.
  
  - Associate each node $v$ of $T$ as the subset sequences, $S_1$ and $S_2$, associated with the recursive calls.
  
  - The external nodes of $T$ are associated with the individual elements of $S$. 

Merge Sort

85  24  63  45  17  31  96  50

(17  31  96  50)

63  45

85  24

17  31  96  50

63  45

85

25 September, 1999
Merge Sort, Cont.
Merge Sort, Cont.

Merge and Quick Sort
Lawrence M. Brown
Merge Sort, Cont.

25 September, 1999
Merge Sort, Cont.
Merge Sort, Cont.
Merging Two Sequences

Algorithm: \( \text{merge}( S_1, S_2, S ) \)

Input: \( S_1, S_2 \) sorted in ascending order, and \( S \), an empty Sequence

Output: Sorted Sequence, \( S = S_1 \cup S_2 \).

\[
\begin{align*}
\text{while } & S_1 \text{ is not empty and } S_2 \text{ is not empty } \textbf{do} \\
& \text{if } S_1.\text{first().element()} \leq S_2.\text{first().element()} \textbf{ then} \\
& \quad S.\text{insertLast}( S_1.\text{remove}( S_1.\text{first()} ) ) \quad \text{// move first element to end of } S \\
& \text{else} \\
& \quad S.\text{insertLast}( S_2.\text{remove}( S_2.\text{first()} ) ) \quad \text{// move first element to end of } S \\
\end{align*}
\]

\[
\begin{align*}
\text{while } & S_1 \text{ is not empty } \textbf{do} \\
& S.\text{insertLast}( S_1.\text{remove}( S_1.\text{first()} ) ) \quad \text{// move remaining elements of } S_1 \text{ to } S \\
\end{align*}
\]

\[
\begin{align*}
\text{while } & S_2 \text{ is not empty } \textbf{do} \\
& S.\text{insertLast}( S_2.\text{remove}( S_2.\text{first()} ) ) \quad \text{// move remaining elements of } S_2 \text{ to } S \\
\end{align*}
\]
Merging Two Sequences

\[ S_1: 24 \rightarrow 45 \rightarrow 63 \rightarrow 85 \]
\[ S_2: 17 \rightarrow 31 \rightarrow 50 \rightarrow 96 \]
\[ S: 17 \rightarrow 24 \rightarrow 31 \rightarrow 45 \rightarrow 50 \]

\[ S_1: 45 \rightarrow 63 \rightarrow 85 \]
\[ S_2: 31 \rightarrow 50 \rightarrow 96 \]
\[ S: 17 \rightarrow 24 \rightarrow 31 \rightarrow 45 \rightarrow 50 \]

\[ S_1: 85 \]
\[ S_2: 96 \]
\[ S: 17 \rightarrow 24 \rightarrow 31 \rightarrow 45 \rightarrow 50 \rightarrow 85 \]

\[ S_1 \]
\[ S_2 \]
\[ S: 17 \rightarrow 24 \rightarrow 31 \rightarrow 45 \rightarrow 50 \rightarrow 85 \rightarrow 96 \]
Implementation

• SortObject Interface

    public interface SortObject
    {
        // sort sequence S in nondecreasing order using comparator c
        public void sort( Sequence S, Comparator c );
    }

• Sequence in merge sort implementation supports newContainer()

    // instantiates another container of the same class
    public Container newContainer()
    {
        return (Container)( new Sequence() );
    }
public class **ListMergeSort** implements **SortObject**
{
    public void **sort** ( **Sequence** S, **Comparator** c )
    {
        int n = S.size();

        if (n < 2) return;  // a sequence with 0 or 1 element is already sorted

        **Sequence** S1 = (Sequence) S.newContainer();  // divide
        for (int i=1; i <= (n+1)/2; i++)
            S1.insertLast( S.remove( S.first() ) );

        **Sequence** S2 = (Sequence) S.newContainer();
        for (int i=1; i <= n/2; i++)
            S2.insertLast( S.remove( S.first() ) );

        **sort** ( S1, c );  // recursion
        **sort** ( S2, c );

        **merge** ( S1, S2, c, S );  // conquer
public void merge( Sequence S1, Sequence S2, Comparator c, Sequence S )
{
    while(! S1.isEmpty() && ! S2.isEmpty())
    {
        if( c.isLessThanOrEqualTo( S1.first().element(), S2.first().element() ) )
            S.insertLast( S1.remove( S1.first() ) );       // move first element to end of S
        else
            S.insertLast( S2.remove( S2.first() ) );       // move first element to end of S
    }

    if( S1.isEmpty() )
    {
        while( ! S2.isEmpty() )
        {
            S.insertLast( S2.remove( S2.first() ) );         // move remaining elements of S2 to S
        }
    }

    if( S2.isEmpty() )
    {
        while( ! S1.isEmpty() )
        {
            S.insertLast( S1.remove( S1.first() ) );         // move remaining elements of S1 to S
        }
    }
}
Running-Time of Merge Sort

• Proposition 1: The merge-sort tree associated with the execution of a merge-sort on a sequence of \( n \) elements has a height of \( \lceil \log n \rceil \).

• Proposition 2: A merge sort algorithm sorts a sequence of size \( n \) in \( O(n \log n) \) time.

• Assume that an access, insert, and delete from the first and last nodes of \( S \) (and subsequences) run in \( O(1) \) time.
Running-Time of Merge Sort

• Let the *time spent* at node $v$ (of merge-sort tree, $T$) be the running time of the recursive call associated with $v$, excluding the recursive calls sent to $v$’s children (remember each node $v$ holds a sequence of numbers).

• Let $i$ represent the depth of the node $v$ in the merge-sort tree, the *time spent* at node $v$ is $O(\frac{n}{2^i})$ since the size of the sequence associated with $v$ in $n/2^i$.

• Observe that $T$ has exactly $2^i$ nodes at depth $i$. The total time spent at depth $i$ in the tree is then

$$O(\text{# of nodes at the level } \cdot \text{ time spend at node } ) = O( 2^i n/2^i ),$$

which is $O(\ n\ )$. The tree has height $\lceil \log n \rceil$.

• Time complexity of merge sort $\rightarrow O(\ n \log n\ )$. 
Quick Sort
Divide and Conquer

- Quick sort is also based on a divide and conquer algorithm similar to merge sort; however, a pivot element defines the subsets.

- In general:
  1. Divide - If the Sequence $S$ has 2 or more elements, select any element in $x$ as a pivot element. Divide $S$ into 3 subsequences:
     - $L$ holds elements $< x$.
     - $E$ holds elements equal to $x$.
     - $G$ holds elements $> x$.
  
  2. Recurse - Recursively sort $L$ and $G$.

  3. Conquer - Merge the three sets together. First insert elements of $L$, then elements of $E$, and lastly, insert elements of $G$. 
Quick Sort

25 September, 1999
Quick Sort

25 September, 1999
Quick Sort
Quick Sort

Lawrence M. Brown

25 September, 1999
Quick Sort
Selection of Pivot Element

- Select the \textit{last} element is the Sequence as the \textit{pivot element}, $q$.

\begin{array}{cccccccc}
S & 85 & 24 & 63 & 45 & 17 & 31 & 96 & 50 \\
\end{array}

- Then, rearrange the Sequence into (implied) subsets with all elements $< q$ to the left and all elements $> q$ to the right.

\begin{array}{cccccccc}
S & 31 & 24 & 17 & 45 & 50 & 85 & 96 & 63 \\
\end{array}
Running Time of Quick Sort

- Let $s_i(n)$ denote the sum of the input sizes of the nodes at depth $i$ in $T$.

- $s_0(n) = n$, the entire input set is at $s_0$.

- $s_1(n) = n - 1$, the pivot element is not propagated.

- $s_2(n) = n - 2$ or $n - 3$, depending on whether one of the nodes has zero elements.
Running Time of Quick Sort

Worst Case

- The worst case running time of quick sort is

\[ O\left(\sum_{i=0}^{n-1} s_i(n)\right) \]

\[ O\left(\sum_{i=0}^{n-1} (n-i)\right) = O(n + (n-1) + (n-2) + \ldots + 2 + 1) \]

\[ O\left(\sum_{i=1}^{n} i\right) = O(n^2) \]

- In the worst case, quick sort is \( O(n^2) \).
Running Time of Quick Sort

Best Case

- **Best** performance of quick sort occurs when the size of $L$ and $G$ are equal.

  $$s_0(n) = n$$
  $$s_1(n) = n - 1$$
  $$s_2(n) = n - (1 + 2) = n - 3$$
  $$s_3(n) = n - (1 + 2 + 2^2) = n - 7$$
  $$\vdots$$
  $$s_i(n) = n - (1 + 2 + 2^2 + \cdots + 2^{i-1}) = n - 2^i + 1$$
  $$\vdots$$

- Implies that the sort tree has a **height** of $O(\log n)$, $n > 1$.

  $$O\left(\sum_{i=0}^{\lfloor \log n \rfloor} s_i(n)\right) = O\left(\sum_{i=0}^{\lfloor \log n \rfloor} (n - 2^i + 1)\right) = O(n \log n)$$

- In the **best case**, quick sort is $O(\ n \log \ n \ )$\footnote{Assuming no duplicate elements.}
Randomized Quick Sort

Best \( O(n \log n) \) Subsets Equal Size Unordered (random) Sequence

Worst \( O(n^2) \) One Subset Zero Size Sorted Sequence

- **Problem:** Quick sort algorithm performs poorly on sorted and “nearly” sorted sequences.

- Modify algorithm to select the pivot element *randomly* from the sequence.

\[ E[\text{running time}] = O(n \log n) \]
Summary

- **Merge sort** recursively divides the Sequence into halves, sorting the elements on the return.

- **Merge sort** runs in $O( n \log n )$.

- **Quick sort** follow same idea as merge sort except that a pivot element determines the subsets. Sorting is performed prior to creating each subset.

- In **quick sort**, the last element of the Sequence is typically selected as the pivot element.

- For **quick sort**, best case running time is $O( n \log n )$, worst case is $O(n^2)$. 