SORTING

• Review of Sorting
• Merge Sort
• Sets
Sorting Algorithms

- **Selection Sort** uses a priority queue P implemented with an unsorted sequence:
  - **Phase 1**: the insertion of an item into P takes $O(1)$ time; overall $O(n)$
  - **Phase 2**: removing an item takes time proportional to the number of elements in P $O(n)$: overall $O(n^2)$
  - Time Complexity: $O(n^2)$

- **Insertion Sort** is performed on a priority queue P which is a sorted sequence:
  - **Phase 1**: the first `insertItem` takes $O(1)$, the second $O(2)$, until the last `insertItem` takes $O(n)$: overall $O(n^2)$
  - **Phase 2**: removing an item takes $O(1)$ time; overall $O(n)$.
  - Time Complexity: $O(n^2)$

- **Heap Sort** uses a priority queue K which is a heap.
  - `insertItem` and `removeMinElement` each take $O(\log k)$, $k$ being the number of elements in the heap at a given time.
  - **Phase 1**: $n$ elements inserted: $O(n\log n)$ time
  - **Phase 2**: $n$ elements removed: $O(n \log n)$ time.
  - Time Complexity: $O(n\log n)$
Divide-and-Conquer

• *Divide and Conquer* is more than just a military strategy, it is also a method of algorithm design that has created such efficient algorithms as *Merge Sort*.

• In terms or algorithms, this method has three distinct steps:

  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.

  - **Recurse**: Use divide and conquer to solve the subproblems associated with the data subsets.

  - **Conquer**: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem.
Merge-Sort

• Algorithm:

- **Divide**: If $S$ has at least two elements (nothing needs to be done if $S$ has zero or one elements), remove all the elements from $S$ and put them into two sequences, $S_1$ and $S_2$, each containing about half of the elements of $S$. (i.e. $S_1$ contains the first $\left\lfloor \frac{n}{2} \right\rfloor$ elements and $S_2$ contains the remaining $\left\lceil \frac{n}{2} \right\rceil$ elements.

- **Recurse**: Recursive sort sequences $S_1$ and $S_2$.

- **Conquer**: Put back the elements into $S$ by merging the sorted sequences $S_1$ and $S_2$ into a unique sorted sequence.

• Merge Sort Tree:

- Take a binary tree $T$
- Each node of $T$ represents a recursive call of the merge sort algorithm.
- We associate with each node $v$ of $T$ a the set of input passed to the invocation $v$ represents.
- The external nodes are associated with individual elements of $S$, upon which no recursion is called.
Merge-Sort

85 24 63 45 17 31 96 50

85 24 63 45

17 31 96 50

85 24 63 45

sorting
Merge-Sort (cont.)

85  24

63  45

17  31  96  50
Merge-Sort (cont.)
Merge-Sort (cont.)

24 85 63 45

17 31 96 50

24 85

63 45

17 31 96 50

24 85

63 45

17 31 96 50

24 85

63 45

17 31 96 50

24 85

63 45

17 31 96 50

24 85

63 45

17 31 96 50

24 85

63 45

17 31 96 50
Merge-Sort (cont.)

24  85

45

63

17  31  96  50

24  85

63  45

sort
Merge-Sort (cont.)
Merge-Sort (cont.)

24  85

45  63

17  31  96  50
Merge-Sort (cont.)
## Merge-Sort (cont.)

<table>
<thead>
<tr>
<th>24</th>
<th>45</th>
<th>64</th>
<th>85</th>
<th>17</th>
<th>31</th>
<th>50</th>
<th>96</th>
</tr>
</thead>
</table>

The diagram visually represents the process of merging two sorted sublists into a single sorted list. Each step shows the progression of sorting algorithms, possibly indicating the use of merge sort. The left and right halves of the list are divided into smaller segments, which are then sorted and merged back together, resulting in a fully sorted list.
Merging Two Sequences

- Pseudo-code for merging two sorted sequences into a unique sorted sequence

**Algorithm** `merge (S1, S2, S):`

**Input:** Sequence `S1` and `S2` (on whose elements a total order relation is defined) sorted in nondecreasing order, and an empty sequence `S`.

**Output:** Sequence `S` containing the union of the elements from `S1` and `S2` sorted in nondecreasing order; sequence `S1` and `S2` become empty at the end of the execution.

**while** `S1` is not empty and `S2` is not empty **do**
  **if** `S1.first().element() ≤ S2.first().element()` **then**
    `{ move the first element of `S1` at the end of `S`}
    `S.insertLast(S1.remove(S1.first()))`
  **else**
    `{ move the first element of `S2` at the end of `S`}
    `S.insertLast(S2.remove(S2.first()))`

**while** `S1` is not empty **do**
  `S.insertLast(S1.remove(S1.first()))`
  `{ move the remaining elements of `S2` to `S`}

**while** `S2` is not empty **do**
  `S.insertLast(S2.remove(S2.first()))`
Merging Two Sequences (cont.)

- Some pictures:
  a)
  \[ S_1 \quad 24 \quad 45 \quad 63 \quad 85 \]
  \[ S_2 \quad 17 \quad 31 \quad 50 \quad 96 \]
  \[ S \]
  b)
  \[ S_1 \quad 24 \quad 45 \quad 63 \quad 85 \]
  \[ S_2 \quad 31 \quad 50 \quad 96 \]
  \[ S \quad 17 \]
Merging Two Sequences (cont.)

c) 

\[ S_1 \rightarrow 45 \rightarrow 63 \rightarrow 85 \]

\[ S_2 \rightarrow 31 \rightarrow 50 \rightarrow 96 \]

\[ S \rightarrow 17 \rightarrow 24 \]

d) 

\[ S_1 \rightarrow 45 \rightarrow 63 \rightarrow 85 \]

\[ S_2 \rightarrow 50 \rightarrow 96 \]

\[ S \rightarrow 17 \rightarrow 24 \rightarrow 31 \]
Merging Two Sequences (cont.)

e) $S_1 \overset{63}{\longrightarrow} 85$

$S_2 \overset{50}{\longrightarrow} 96$

$S \overset{17}{\longrightarrow} 24 \overset{31}{\longrightarrow} 45$

f) $S_1 \overset{63}{\longrightarrow} 85$

$S_2 \overset{96}{\longrightarrow} \overset{}{\rightarrow}$

$S \overset{17}{\longrightarrow} 24 \overset{31}{\longrightarrow} 45 \overset{50}{\longrightarrow}$
Merging Two Sequences (cont.)

g) $S_1$ 85

$S_2$ 96

S 17 24 31 45 50 63 85

h) $S_1$

$S_2$ 96

S 17 24 31 45 50 63 85

sorting
Merging Two Sequences (cont.)

\[ S_1 \]

\[ S_2 \]

\[ S \]

\[ 17 \quad 24 \quad 31 \quad 45 \quad 50 \quad 63 \quad 85 \quad 96 \]
Java Implementation

• Interface SortObject

    public interface SortObject {
        //sort sequence S in nondecreasing order using comparator c
        public void sort (Sequence S, Comparator c);
    }
public class ListMergeSort implements SortObject {
    public void sort(Sequence S, Comparator c) {
        int n = S.size();
        // a sequence with 0 or 1 element is already sorted
        if (n < 2) return;
        // divide
        Sequence S1 = (Sequence) S.newContainer();
        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()));
        }
        // recur
        sort(S1, c);
        sort(S2, c);
        // conquer
        merge(S1,S2,c,S);
    }
}
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()));
        } else {
            S.insertLast(S2.remove(S2.first()));
        }
    }

    if(S1.isEmpty()) {
        while(!S2.isEmpty()) {
            S.insertLast(S2.remove(S2.first()));
        }
    }

    if(S2.isEmpty()) {
        while(!S1.isEmpty()) {
            S.insertLast(S1.remove(S1.first()));
        }
    }
}
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.

• We assume only that the input sequence \( S \) and each of the sub-sequences created by each recursive call of the algorithm can access, insert to, and delete from the first and last nodes in \( O(1) \) time.

• We call the time spent at node \( v \) of merge-sort tree \( T \) the running time of the recursive call associated with \( v \), excluding the recursive calls sent to \( v \)'s children.

• If we let \( i \) represent the depth of node \( v \) in the merge-sort tree, the time spent at node \( v \) is \( O(n/2^i) \) since the size of the sequence associated with \( v \) is \( n/2^i \).

• Observe that \( T \) has exactly \( 2^i \) nodes at depth \( i \). The total time spent at depth \( i \) in the tree is then \( O(2^i n/2^i) \), which is \( O(n) \). We know the tree has height \( \lceil \log n \rceil \).

• Therefore, the time complexity is \( O(n \log n) \).
Set ADT

- A **Set** is a data structure modeled after the mathematical notation of a set. The fundamental set operations are *union*, *intersection*, and *subtraction*.

- A brief aside on mathematical set notation:
  - $A \cup B = \{ x: x \in A \text{ or } x \in B \}$
  - $A \cap B = \{ x: x \in A \text{ and } x \in B \}$
  - $A - B = \{ x: x \in A \text{ and } x \notin B \}$

- The specific methods for a Set $A$ include the following:
  - **size()**: Return the number of elements in set $A$
    - **Input**: None; **Output**: integer.
  - **isEmpty()**: Return if the set $A$ is empty or not.
    - **Input**: None; **Output**: boolean.
  - **insertElement($e$)**: Insert the element $e$ into the set $A$, unless $e$ is already in $A$.
    - **Input**: Object; **Output**: None.
Set ADT (contd.)

- **elements()**:  
  Return an enumeration of the elements in set A.  
  **Input**: None; **Output**: Enumeration.

- **isMember(e)**:  
  Determine if \( e \) is in A.  
  **Input**: Object; **Output**: Boolean.

- **union(B)**:  
  Return \( A \cup B \).  
  **Input**: Set; **Output**: Set.

- **intersect(B)**:  
  Return \( A \cap B \).  
  **Input**: Set; **Output**: Set.

- **subtract(B)**:  
  Return \( A - B \).  
  **Input**: Set; **Output**: Set.

- **isEqual(B)**:  
  Return true if and only if \( A = B \).  
  **Input**: Set; **Output**: boolean.
Generic Merging

**Algorithm** genericMerge\((A, B)\):

**Input:** Sorted sequences \(A\) and \(B\)

**Output:** Sorted sequence \(C\)

let \(A'\) be a copy of \(A\) { We won’t destroy \(A\) and \(B\) }
let \(B'\) be a copy of \(B\)

while \(A'\) and \(B'\) are not empty do
  \(a \leftarrow A'.\text{first}()\)
  \(b \leftarrow B'.\text{first}()\)
  if \(a < b\) then
    firstIsLess\((a, C)\)
    \(A'.\text{removeFirst}()\)
  else if \(a = b\) then
    bothAreEqual\((a, b, C)\)
    \(A'.\text{removeFirst}()\)
    \(B'.\text{removeFirst}()\)
  else
    firstIsGreater\((b, C)\)
    \(B'.\text{removeFirst}()\)

while \(A'\) is not empty do
  \(a \leftarrow A'.\text{first}()\)
  firstIsLess\((a, C)\)
  \(A'.\text{removeFirst}()\)

while \(B'\) is not empty do
  \(b \leftarrow B'.\text{first}()\)
  firstIsGreater\((b, C)\)
  \(B'.\text{removeFirst}()\)
Set Operations

• We can specialize the generic merge algorithm to perform set operations like union, intersection, and subtraction.

• The generic merge algorithm examines and compare the current elements of $A$ and $B$.

• Based upon the outcome of the comparison, it determines if it should copy one or none of the elements $a$ and $b$ into $C$.

• This decision is based upon the particular operation we are performing, i.e. union, intersection or subtraction.

• For example, if our operation is union, we copy the smaller of $a$ and $b$ to $C$ and if $a=b$ then it copies either one (say $a$).

• We define our copy actions in `firstIsLess`, `bothAreEqual`, and `firstIsGreater`.

• Let’s see how this is done ...
Set Operations (cont.)

• For union

```
public class UnionMerger extends Merger {
    protected void firstIsLess(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
    protected void bothAreEqual(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
    protected void firstIsGreater(Object b, Sequence C) {
    }
}
```

• For intersect

```
public class IntersectMerger extends Merger {
    protected void firstIsLess(Object a, Object b, Sequence C) {} // null method
    protected void bothAreEqual(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
}
```
Set Operations (cont.)

protected void firstIsGreater(Object b, Sequence C) {}  // null method

• For subtraction

public class SubtractMerger extends Merger {
    protected void firstIsLess(Object a, Object b, Sequence C) {
        C.insertLast(a);
    }
    protected void bothAreEqual(Object a, Object b, Sequence C) {}  // null method
    protected void firstIsGreater(Object b, Sequence C) {
    }  // null method