MERGE SORT

• Review of Sorting

• Merge Sort
• **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**: $O(n^2)$
Sorting Algorithms (cont.)

- **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 `removeMin` 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.

  - **Recur**: 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 \( \lfloor n/2 \rfloor \) elements and \( S_2 \) contains the remaining \( \lceil n/2 \rceil \) elements.
- **Recur**: 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) (17 31) (96 50)

(85) (24) (63 45) (17 31) (96) (50)
Merge-Sort (cont.)

85  24

63  45

17  31  96  50

85  24

63  45

17  31  96  50

85  24
Merge-Sort (cont.)

24 85

63 45

17 31 96 50

Merge Sort
Merge-Sort (cont.)

```
24  85
63  45
17  31  96  50
24  85
63
45
```

Merge Sort
Merge-Sort (cont.)

24 85

45 63

17 31 96 50

24 85

45 63

17 31 96 50

Merge Sort
Merge-Sort (cont.)

[Diagram of a binary tree structure with numbers 24, 45, 63, 85 on the left and 17, 31, 96, 50 on the right, showing the merge sort process.]
Merge-Sort (cont.)

24 45 63 85

17 31 96 50

24 45 63 85

17 31 50 96

Merge Sort
Merge-Sort (cont.)

(24 45 63 85) (17 31 50 96)

17 24 31 45 50 63 85 96

Merge Sort
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.

```java
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()))
```

---

**Merge Sort**

7.17
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)  
\[ S_1 \: 45 \rightarrow 63 \rightarrow 85 \]  
\[ S_2 \: 31 \rightarrow 50 \rightarrow 96 \]  
\[ S \: 17 \rightarrow 24 \]  
d)  
\[ S_1 \: 45 \rightarrow 63 \rightarrow 85 \]  
\[ S_2 \: 50 \rightarrow 96 \]  
\[ S \: 17 \rightarrow 24 \rightarrow 31 \]
Merging Two Sequences (cont.)

e)  

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

f)  

\[ S_1 \quad 63 \quad 85 \]
\[ S_2 \quad 96 \]
\[ S \quad 17 \quad 24 \quad 31 \quad 45 \quad 50 \]
Merging Two Sequences (cont.)

g)

\[ S_1 \quad 85 \]

\[ S_2 \quad 96 \]

\[ S \quad 17 \quad 24 \quad 31 \quad 45 \quad 50 \quad 63 \]

h)

\[ S_1 \]

\[ S_2 \quad 96 \]

\[ S \quad 17 \quad 24 \quad 31 \quad 45 \quad 50 \quad 63 \quad 85 \]
Merging Two Sequences (cont.)

i)

\[ S_1 \]

\[ S_2 \]

\[ S \]

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