## Comparison of Quadratic Sorts

<table>
<thead>
<tr>
<th></th>
<th>Number of Comparisons</th>
<th>Number of Exchanges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Worst</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Merge Sort

Section 8.7

Merge

- A merge is a common data processing operation performed on two ordered sequences of data.
  - The result is a third ordered sequence containing all the data from the first two sequences.
Merge Algorithm

1. Access the first item from both sequences.
2. while not finished with either sequence
3. Compare the current items from the two sequences, copy the smaller current item to the output sequence, and access the next item from the input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.

Analysis of Merge

- For two input sequences each containing \( n \) elements, each element needs to move from its input sequence to the output sequence
- Merge time is \( O(n) \)
- Space requirements
  - The array cannot be merged in place
  - Additional space usage is \( O(n) \)
Code for Merge

```java
private static void merge(T[] out, T[] left, T[] right) {
    // merge left and right into out
    // Access first item from all sequences
    int i = 0; // left
    int j = 0; // right
    int k = 0; // out
    // while there is data in both left and right
    while (i < left.length && j < right.length) {
        // find smaller and insert into out
        if (left[i].compareTo(right[j]) < 0)
            out[k++] = left[i++];
        else
            out[k++] = right[j++];
    }
    // Copy remaining items from left into out
    while (i < left.length)
        out[k++] = left[i++];
    // Copy remaining items from right into out
    while (j < right.length)
        out[k++] = right[j++];
} // merge()
```

Merge Sort

- We can modify merging to sort a single, unsorted array
  1. Split the array into two halves
  2. Sort the left half
  3. Sort the right half
  4. Merge the two

- This algorithm can be written with a recursive step
(recursive) Algorithm for Merge Sort

Algorithm for Merge Sort
1. if the tableSize is > 1
2. set halfSize to tableSize divided by 2.
3. allocate a table called leftTable of size halfSize.
4. allocate a table called rightTable of size tableSize - halfSize.
5. copy the elements from table[0 ... halfSize - 1] into leftTable.
6. copy the elements from table[halfSize ... tableSize] into rightTable.
7. recursively apply the merge sort algorithm to leftTable.
8. recursively apply the merge sort algorithm to rightTable.
9. apply the merge method using leftTable and rightTable as the input and the original table as the output.

Trace of Merge Sort (cont.)

```
15 20 30 45 50 60 80 90
```

```
30 45 50 60
    15 20 80 90
```
Analysis of Merge Sort

- Each backward step requires a movement of \( n \) elements from smaller-size arrays to larger arrays; the effort is \( O(n) \)
- The number of steps which require merging is \( \log n \) because each recursive call splits the array in half
- The total effort to reconstruct the sorted array through merging is \( O(n \log n) \)
- Requires a total of \( n \) additional storage locations.

Code for Merge Sort

```java
public static void sort(T[] table) {
    // A table with 1 element is already sorted
    if (table.length > 1) {
        // Split table into halves
        int halfSize = table.length/2;
        T[] left = new Comparable[halfSize];
        T[] right = new Comparable[table.length - halfSize];
        System.arraycopy(table, 0, left, 0, halfSize);
        System.arraycopy(table, halfSize, right, 0, table.length-halfSize);
        // sort the halves
        sort(left);
        sort(right);
        // merge the halves
        merge(table, left, right);
    }
} // sort()
```
Heapsort

Section 8.8

- Merge sort time is $O(n \log n)$ but still requires, temporarily, $n$ extra storage locations
- Heapsort does not require any additional storage
- As its name implies, heapsort uses a heap to store the array
First Version of a Heapsort Algorithm

- When used as a priority queue, a heap maintains the smallest value at the top
- The following algorithm
  - places an array’s data into a heap,
  - then removes each heap item (O(n log n)) and moves it back into the array
- This version of the algorithm requires \( n \) extra storage locations

Heapsort Algorithm: First Version

1. Insert each value from the array to be sorted into a priority queue (heap).
2. Set \( i \) to 0
3. while the priority queue is not empty
4. Remove an item from the queue and insert it back into the array at position \( i \)
5. Increment \( i \)

Revising the Heapsort Algorithm

- Instead of using a Min Heap, use a Max heap
- The root contains the largest element
- Then,
  - move the root item to the bottom of the heap
  - reheap, ignoring the item moved to the bottom
Trace of Heapsort

Trace of Heapsort (cont.)
Revising the Heapsort Algorithm

- If we implement the heap as an array
  - each element removed will be placed at the end of the array, and
  - the heap part of the array decreases by one element

Algorithm for In-Place Heapsort

Algorithm for In-Place Heapsort

1. Build a heap by rearranging the elements in an unsorted array
2. while the heap is not empty
3. Remove the first item from the heap by swapping it with the last item in the heap and restoring the heap property
Algorithm to Build a Heap

- Start with an array `table` of length `table.length`
- Consider the first item to be a heap of one item
- Next, consider the general case where the items in `array table` from 0 through `n-1` form a heap and the items from `n` through `table.length - 1` are not in the heap

Algorithm to Build a Heap (cont.)

Refinement of Step 1 for In-Place Heapsort

1.1 while `n` is less than `table.length`
1.2 Increment `n` by 1. This inserts a new item into the heap
1.3 Restore the heap property
Analysis of Heapsort

- Because a heap is a complete binary tree, it has \( \log n \) levels.
- Building a heap of size \( n \) requires finding the correct location for an item in a heap with \( \log n \) levels.
- Each insert (or remove) is \( O(\log n) \).
- With \( n \) items, building a heap is \( O(n \log n) \).
- No extra storage is needed.

Quicksort

Section 8.9
Quicksort

- Developed in 1962
- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called *partitioning*)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
  - The pivot is placed between the two subarrays
- The process is repeated until the array is sorted

Trace of Quicksort

```
44 75 23 43 55 12 64 77 33
```
Arbitrarily select the first element as the pivot

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right.
Trace of Quicksort (cont.)

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right.

Quicksort Example (cont.)

44 is now in its correct position.
Trace of Quicksort (cont.)

Now apply quicksort recursively to the two subarrays

Algorithm for Quicksort

- We describe how to do the partitioning later
- The indexes first and last are the end points of the array being sorted
- The index of the pivot after partitioning is pivIndex

Algorithm for Quicksort
1. if first < last then
2. Partition the elements in the subarray first . . . last so that the pivot value is in its correct place (subscript pivIndex)
3. Recursively apply quicksort to the subarray first . . . pivIndex - 1
4. Recursively apply quicksort to the subarray pivIndex + 1 . . . last
Analysis of Quicksort

- If the pivot value is a random value selected from the current subarray,
  - then statistically half of the items in the subarray will be less than the pivot and half will be greater

- If both subarrays have the same number of elements (best case), there will be $\log n$ levels of recursion

- At each recursion level, the partitioning process involves moving every element to its correct position—$n$ moves

- Quicksort is $O(n \log n)$, just like merge sort

Analysis of Quicksort (cont.)

- The array split may not be the best case, i.e. 50-50

- An exact analysis is difficult (and beyond the scope of this class), but, the running time will be bounded by a constant $\times n \log n$
Analysis of Quicksort (cont.)

- A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty.
- In that case, the sort will be $O(n^2)$
- Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts
  - We'll discuss a solution later

Code for Quicksort

```java
public static void sort(T[] table, int first, int last) {
    if (first < last) {
        // partition the table at pivotIndex
        int pivotIndex = partition(table, first, last);

        // sort the left half
        sort(table, first, pivotIndex-1);

        // sort the right half
        sort(table, pivotIndex+1, last);
    }
} // sort()
```
Algorithm for Partitioning

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first.

Trace of Partitioning (cont.)

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first.
Trace of Partitioning (cont.)

Search for the first value at the left end of the array that is greater than the pivot value.

Then search for the first value at the right end of the array that is less than or equal to the pivot value.
Trace of Partitioning (cont.)

Exchange these values

Trace of Partitioning (cont.)

Exchange these values
Trace of Partitioning (cont.)

Algorithm for Partitioning

**Algorithm for partition Method**

1. Define the pivot value as the contents of table[first].
2. Initialize up to first and down to last.
3. do
4. Increment up until up selects the first element greater than the pivot value or up has reached last.
5. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first.
6. if up < down then
7. Exchange table[up] and table[down].
8. while up is to the left of down
9. Exchange table[first] and table[down].
10. Return the value of down to pivotindex.
Code for partition when Pivot is the largest or smallest value

Code for partition (cont.)

```java
public static void partition(T[] table, int first, int last) {
    // select first element as pivot value
    // Initialize up to first and down to last
    do {
        // Increment up until it selects first element >= pivot or it reaches last
        while ((up < last) && (pivot.compareTo(table[up]) >= 0))
            up++;
        // Decrement down until it select first element < pivot or it reaches first
        while ((down > first) && (pivot.compareTo(table[down]) < 0))
            down--;
        if (up < down) {
            // exchange table[up] and table[down]
            T temp = table[up];
            table[up] = table[down];
            table[down] = temp;
        }
    } while (up < down);
    // exchange table[first] and table[down]
    T temp = table[first];
    table[first] = table[down];
    table[down] = temp;
    // return value of down a pivotIndex
    return down;
} // partition()
```
Revised Partition Algorithm

- Quicksort is $O(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted.
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split.
  - Use three references: first, middle, last.
  - Select the median of these items as the pivot.

Trace of Revised Partitioning

```
44  75  23  43  55  12  64  77  33
```
Trace of Revised Partitioning (cont.)

Sort these values.
Trace of Revised Partitioning (cont.)

Sort these values

Trace of Revised Partitioning (cont.)

Exchange middle with first
Exchange middle with first

Run the partition algorithm using the first element as the pivot
Algorithm for Revised partition Method

Algorithm for Revised partition Method
1. Sort table[first], table[middle], and table[last]
2. Move the median value to table[first] (the pivot value) by exchanging table[first] and table[middle].
3. Initialize up to first and down to last
4. do
5. Increment up until up selects the first element greater than the pivot value or up has reached last
6. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first
7. if up < down then
8. Exchange table[up] and table[down]
9. while up is to the left of down
10. Exchange table[first] and table[down]
11. Return the value of down to pivIndex

Comparison of Sort Algorithms

Summary
## Sort Review

<table>
<thead>
<tr>
<th>Sort Method</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n^{7/5})$</td>
<td>$O(n^{5/4})$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>