# Data Structures

## Sorting

CS284

# Objectives

- To learn how to implement the following sorting algorithms:
  - selection sort
  - bubble sort
  - insertion sort
  - shell sort
  - merge sort
  - heapsort
  - quicksort
- To understand the differences in performance of these algorithms, and which to use for small, medium arrays, and large arrays

# Shell Sort: A Better Insertion Sort

- A type of insertion sort, but with $\mathcal{O}(n^{3/2})$ or better performance than the $\mathcal{O}(n^2)$ sorts
- It is named after its discoverer, Donald Shell
- Can be thought of as a divide-and-conquer approach to insertion sort
- Instead of sorting the entire array, sorts many smaller subarrays using insertion sort before sorting the entire array

# Algorithm – Array `table` of size `n`

```
gap = n/2
while (gap > 0) {
  for each e $\in$ array table from position gap to n-1 {
     Insert e where it belongs in its subarray.
  }
  if (gap is 2)
    then gap = 1
    else gap = gap/2.2  // chosen by experimentation
}
```

- We shall refine line 4 in the next slide

Tracing an example

# Refinement of Step 4, the Insertion Step

```
gap = n/2
while (gap > 0) {
 for each e $\in$ array table from position gap to n-1 {
  nextPos is the position of e
  nextVal = e
  while (nextPos>gap && table[nextPos-gap]>nextVal){
    Shift the element at nextPos-gap to position nextPos
    nextPos = nextPost-gap
  }
  Insert nextVal at nextPos
 }
 if (gap is 2)
   then gap = 1
   else gap = gap/2.2  // chosen by experimentation
}
```

# Analysis of Shell Sort

- Because the behavior of insertion sort is closer to $\mathcal{O}(n)$ than $\mathcal{O}(n^2)$ when an array is nearly sorted, presorting speeds up later sorting
- This is critical when sorting large arrays where the $\mathcal{O}(n^2)$ performance becomes significant
- General analysis is open research problem
  - Performance depends on selection of (decreasing) gap
  - Our algorithm initially sets gap to $n/2$ and then divides by 2.2 and truncates the result
  - Empirical studies show that this approach yields performance $\mathcal{O}(^{5/4})$ or even $\mathcal{O}(n^{7/6})$, but there is no theoretical basis for the result

# Analysis of Shell Sort (cont.)

- If successive powers of 2 used for gap, performance is $\mathcal{O}(n^2)$
- If successive values for gap are based on Hibbard's sequence,

  $$2k - 1 \text{ (i.e. 31, 15, 7, 3, 1)}$$

  it can be proven that the performance is $\mathcal{O}(n^{3/2})$
- Other sequences give similar or better performance

# Code for Shell Sort

```java
public class ShellSort {
  public static <T extends Comparable <T>> void sort(T[] table) {
     // Gap between adjacent elements.
     int gap = table.length / 2;
     while (gap > 0) {
      for (int nextPos = gap; nextPos<table.length; nextPos++) {
         // Insert element at nextPos in its subarray.
         insert(table, nextPos, gap);
       }
       // Reset gap for next pass.
       if (gap == 2)
         { gap = 1; }
       else
         { gap = (int) (gap / 2.2); }
     } // End while.
  }
```

# Code for Shell Sort

```
private static <T extends Comparable <T>>
  void insert(T[] table, int nextPos, int gap) {
    T nextVal = table[nextPos]; // Element to insert.

    // Shift all values>nextVal in subarray down by gap.
    while ((nextPos>gap-1)
         && (nextVal.compareTo(table[nextPos-gap]) < 0)) {
      table[nextPos] = table[nextPos-gap]; // Shift down.
      nextPos -= gap; // Check next position in subarray.
    }
    table[nextPos] = nextVal; // Insert nextVal.
}
```

# Merge

- A merge is a common data processing operation performed on two sequences of data with the following characteristics
  - Both sequences contain items with a common `compareTo` method
  - The objects in both sequences are ordered in accordance with this `compareTo` method
- The result is a third sequence containing all the data from the first two sequences

# Merge Algorithm – leftSeq and rightSeq

```
Access the first item from both sequences.
while (not finished with either sequence) {
  Compare the current items from the two sequences
  Copy the smaller current item to the output sequence, and access the
}
Copy any remaining items from leftSeq to the output sequence.
Copy any remaining items from rightSeq to the output sequence.
```

# Trace of Merge Algorithm

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 50 | 60 | 90 | 30 |

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 45 | 20 | 80 | 15 | 33 |

# Trace of Merge Algorithm

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 50 | 60 | 90 | 30 |

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 45 | 20 | 80 | 15 | 33 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 45 | 50 | 20 | 60 | 80 | 15 | 30 | 33 | 90 |

# Analysis of Merge

- For two input sequences containing $n$ and $m$ elements resp., each element needs to move from its input sequence to the output sequence
- Merge time is $\mathcal{O}(n + m)$

# Code for Merge

```
private static <T extends Comparable<T>>  void merge(T[]
outputSeq, T[] leftSeq, T[] rightSeq) {
    int i = 0; // Index into the left input sequence.
    int j = 0; // Index into the right input sequence.
    int k = 0; // Index into the output sequence.

    while (i < leftSeq.length && j < rightSeq.length) {
     // Find smaller one insert into the output sequ.
     if (leftSeq[i].compareTo(rightSeq[j])<0){
         outputSeq[k++] = leftSeq[i++];
     } else
        { outputSeq[k++] = rightSeq[j++]; }
    }
    // Copy remaining input from left seq. into output.
    while (i < leftSeq.length) {
        outputSeq[k++] = leftSeq[i++];
    }
    // Copy remaining input from right seq. into output.
    while (j < rightSeq.length) {
        outputSeq[k++] = rightSeq[j++];
    }
}
```

# 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

```
if (tableSize>1) {
   halfsize = tableSize/2
   Allocate a table leftTable of size halfSize
   Allocate a table rightTable of size tableSize-halfSize
   Copy elements from table[0..halfSize-1] to leftTable
   Copy elements from table[halfSize..tableSize] to rightTable
   Recursively apply merge sort to leftTable
   Recursively apply merge sort to rightTable
   Apply merge algorithm to leftTable and rightTable
}
```

- Tracing an example

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 45 | 50 | 20 | 60 | 80 | 15 | 30 | 33 | 90 |

# Complexity of Merge Sort

- Merge sort time is $\mathcal{O}(n \log n)$
  - $n$ for the total time for merging, per level
- But it requires, temporarily, $n$ extra storage locations

# Code for Merge Sort

```java
public class MergeSort {
public static <T extends Comparable <T>> void sort(T[] table) {
  // A table with one element is sorted already.
  if (table.length > 1) {
    // Split table into halves.
    int halfSize = table.length / 2;
    T[] leftTable  = (T[])new Comparable[halfSize];
    T[] rightTable = (T[])new Comparable[table.length-halfSize];
    System.arraycopy(table, 0, leftTable, 0, halfSize);
    System.arraycopy(table, halfSize, rightTable, 0,
                     table.length - halfSize);
          //Sort the halves.
    sort(leftTable);
    sort(rightTable);

    // Merge the halves.
    merge(table, leftTable, rightTable);
  }
}}
```

# Heapsort

- Heapsort has the same complexity as Mergesort
- In contrast to Mergesort, Heapsort does not require any additional storage
- As its name implies, heapsort uses a heap to store the array
  - When used as a priority queue, a heap maintains a smallest value at the top
  - Naive heapsort:
    - place an array's data into a heap,
    - then remove each heap item and move it back into the array

# Naive Version of a Heapsort Algorithm

- This version of the algorithm requires $n$ extra storage locations

```
Insert each value from table into a priority queue (heap).
i=0
while (priority queue is not empty) {
   Remove next item from the queue
   Insert it back into the array at position i
   i++
}
```

- Tracing an example

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 15 | 20 | 30 | 45 | 50 | 60 | 80 | 90 |

# Revising the Heapsort Algorithm

- We can do better in terms of space usage
- In heaps we've used so far, each parent node value was not greater than the values of its children (minHeap)
- We can build a heap so that each parent node value is not less than its children (maxHeap)
- Then,
  - move the top item to the bottom of the heap
  - reheap, ignoring the item moved to the bottom
- If we implement the heap as an array,
  - each element removed will be placed at end of the array, and
  - the heap part of the array decreases by one element

# Algorithm for In-Place Heapsort

```
Build a maxHeap h by rearranging the elements in table
while (h is not empty) {
  Remove the first item h by swapping it with the last item in h
  Restore the heap property on h
}
```

- Tracing an example

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 74 | 66 | 89 | 6 | 39 | 29 | 76 | 32 | 18 | 28 | 37 | 26 | 20 |

# 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 $\mathcal{O}(\log n)$
- With $n$ items, building a heap is $\mathcal{O}(n \log n)$
- No extra storage is needed

# Code for Heap Sort

```
public class HeapSort {
 public static <T extends Comparable <T>> void sort(T[] table) {
   buildHeap(table);  // build maxHeap
   shrinkHeap(table); // transform heap into a sorted array.
 }

 private static <T extends Comparable <T>> void buildHeap(T[] table) {
   int n = 1;
   while (n < table.length) {
     n++; // Add a new item to the heap and reheap.
     int child = n - 1;
     int parent = (child - 1) / 2; // Find parent.
     while (parent >= 0
        && table[parent].compareTo(table[child]) < 0) {
       swap(table, parent, child);
       child = parent;
       parent = (child - 1) / 2;
     }
   }
 }
```

# Code for Heap Sort

```
private static <T extends Comparable <T>> void shrinkHeap(T[] table) {
  int n = table.length;

  // Invariant: table[0...n - 1] forms a heap.
  // table[n...table.length - 1] is sorted.
  while (n > 0) {
    n--;
    swap(table, 0, n);
    // table[1...n - 1] form a heap.
    // table[n...table.length - 1] is sorted.
    int parent = 0;
    while (true) {
      int leftChild = 2 * parent + 1;
      if (leftChild >= n) {
        break; // No more children.
      }

    // continued
```

# Code for Heap Sort

```
    int rightChild = leftChild + 1;
    // Find the larger of the two children.
    int maxChild = leftChild;
    if (rightChild<n // There is a right child.
      && table[leftChild].compareTo(table[rightChild])<0) {
      maxChild = rightChild;
    }

    // If the parent is smaller than the larger child,
    if (table[parent].compareTo(table[maxChild]) < 0) {
      // Swap the parent and child.
      swap(table, parent, maxChild);
      // Continue at the child level.
      parent = maxChild;
    }
    else { // Heap property is restored.
      break; // Exit the loop.
    }
   }
  }}
```

# Code for Heap Sort

```
  /** Swap the items in table[i] and table[j].
      @param table The array that contains the items
      @param i The index of one item
      @param j The index of the other item
   */
   private static <T extends Comparable <T>>
      void swap(T[] table, int i, int j) {
     T temp = table[i];
     table[i] = table[j];
     table[j] = temp;
  }
}
```