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

Quicksort

- Developed in 1962
- 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

Quicksort by Example

- We'll look at an example
- We will use a simple partitioning strategy
  - the pivot is chosen to be the first element in the current subarray to be partitioned

0	1	2	3	4	5	6	7	8
44	75	23	43	55	12	64	77	33

Algorithm for Partitioning – `partition(table,first,last)`

- Indices `first` and `last` are the end points of the array being sorted

```
pivot = table[first];
up = first;
down = last;
do {
  Increment up until it selects the first element greater
     than pivot or up == last
  Decrement down until it selects the first element less
     than or equal to pivot
  if (up<down)
   { swap(table[up],table[down]) }
} while (up < down)
swap(table[first],table[down]);
return down;
```

Algorithm for Quicksort – `quicksort(table,first,last)`

- The index of the pivot after partitioning is `pivIndex`

```
quicksort(table,first,last) {
  if (first < last) {
   Partition the elements in the subarray first...last
     so that the pivot value is in its correct
     place (subscript pivIndex)
   quicksort(table, first, pivIndex - 1);
   quicksort(table, pivIndex + 1, last);
  }
}
```

Analysis of Quicksort

```
quicksort(table,first,last) {
  if (first < last) {
   Partition the elements in the subarray first...last
     so that the pivot value is in its correct
     place (subscript pivIndex)
   quicksort(table, first, pivIndex - 1);
   quicksort(table, pivIndex + 1, last);
  }
}
```

- 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

Analysis of Quicksort

```
quicksort(table,first,last) {
  if (first < last) {
   Partition the elements in the subarray first...last
     so that the pivot value is in its correct
     place (subscript pivIndex)
   quicksort(table, first, pivIndex - 1);
   quicksort(table, pivIndex + 1, last);
  }
}
```

- At each recursion level, the partitioning process involves moving every element to its correct position (n moves)
- Quicksort is $\mathcal{O}(n \log n)$, just like merge sort
- 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 * $n \log n$

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 $\mathcal{O}(n^2)$
  - Prove this with an example
- 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

```
public class QuickSort {
  public static <T extends Comparable <T>>
     void sort(T[] table) {
        // Sort the whole table.
        quickSort(table, 0, table.length - 1);
  }

  private static <T extends Comparable <T>>
    void quickSort(T[] table, int first, int last) {
      if (first < last) { // There is data to be sorted.
        int pivIndex = partition(table, first, last);
        quickSort(table, first, pivIndex - 1);
        quickSort(table, pivIndex + 1, last);
      }
  }
}
```

Code for Partitioning

```
private static <T extends Comparable <T>>
  int partition(T[] table, int first, int last) {
    T pivot = table[first];
    int up = first;
    int down = last;
    do {
        while ((up<last) && (pivot.compareTo(table[up])>=0)) {
           up++; }
        while (pivot.compareTo(table[down]) < 0) {
          down--;
        }
         if (up < down) { // if up is to the left of down.
          swap(table, up, down);
        }
      } while (up < down); // Repeat while up is left of down.

    swap(table, first, down);
    return down;
}
```

Revised Partition Algorithm

- Quicksort is $\mathcal{O}(n^2)$ 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 the these items as the pivot

Trace of Revised Partition Algorithm

- Trace

Algorithm for Revised partition Method

```
Sort table[first], table[middle], and table[last]
swap(table[first],table[middle])
up = first
down = last
do {
  Increment up until up selects the first element greater
   than the pivot value or up has reached last
  Decrement down until down selects the first element
    less than or equal to the pivot value or down has reached
  if (up < down)
     { Exchange table[up] and table[down] }
  } while up is to the left of down
swap(table[first],table[down])
Return the value of down to pivIndex
```

Comparison of Sorting

	Number of comparisons		
	Best	Average	Worst
Selection sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Bubble sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Shell sort	$\mathcal{O}(n^{7/6})$	$\mathcal{O}(n^{5/4})$	$\mathcal{O}(n^2)$
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$