1.204 Lecture 9

Divide and conquer:
binary search, quicksort, selection

Divide and conquer

- **Divide-and-conquer (or divide-and-combine) approach to solving problems:**

```
method DivideAndConquer(Arguments)
    if (SmallEnough(Arguments))                    // Termination
        return Answer
    else                                           // "Divide"
        Identity= Combine( SomeFunc(Arguments),
                  DivideAndConquer(SmallerArguments))
    return Identity                                // "Combine"
```

- **Divide and conquer solves a large problem as the combination of solutions of smaller problems**
- **We implement divide and conquer either with recursion or iteration**

Binary search

```
public class BinarySearch {
   public static int binSearch(int a[], int x) { // a is sorted
        int low = 0, high = a.length - 1;
        while (low <= high) {
                int mid = (low + high) / 2;
                if (x < a[mid])
                        high = mid - 1;
                else if (x > a[mid])
                        low = mid + 1;
                else
                        return mid;
        }
        return Integer.MIN_VALUE;
   }    // Easy to write recursively too (2 more arguments)

   Example: -55 -9 -7 -5 -3 -1 2 3 4 6 9 98 309
```

Binary search example

```
   public static void main(String[] args) {
     int[] a= {-1, -3, -5, -7, -9, 2, 6, 9, 3, 4, 98, 309, -55};
     Arrays.sort(a);          // Quicksort
     for (int i : a)
       System.out.print(" " + i);
     System.out.println();
     System.out.println("Location of -1 is " + binSearch(a, -1));
     System.out.println("Location of -55 is "+ binSearch(a,-55));
     System.out.println("Location of 98 is " + binSearch(a, 98));
     System.out.println("Location of -7 is " + binSearch(a, -7));
     System.out.println("Location of  8 is " + binSearch(a, 8));
   }

// Output
 -55 -9 -7 -5 -3 -1 2 3 4 6 9 98 309
BinSrch location of -1 is 5
BinSrch location of -55 is 0
BinSrch location of 98 is 11
BinSrch location of -7 is 2
BinSrch location of  8 is -2147483648
```

Binary search performance

- Each iteration cuts search space in half
  - Analogous to tree search
- Maximum number of steps is O(lg n)
  - There are $n/2^k$ values left to search after each step k
- Successful searches take between 1 and ~lg n steps
- Unsuccessful searches take ~lg n steps every time
- We have to sort the array before searching it
  - Quicksort takes O(n lg n) steps
  - This is the bottleneck step
    - If we have to sort before each search, this is too slow
    - Use binary search tree instead: O(lg n) add, O(lg n) find
  - Binary search used on data that doesn't change (or that arrives sorted)
    - Sort once, search many times

Quicksort overview

- Most efficient general purpose sort, O(n lg n)
  - Simple quicksort has worst case of $O(n^2)$, which can be avoided
- Basic strategy
  - Split array (or list) of data to be sorted into 2 subarrays so that:
    - Everything in first subarray is smaller than a known value
    - Everything in second subarray is larger than that value
  - Technique is called 'partitioning'
    - Known value is called the 'pivot element'
  - Once we've partitioned, pivot element will be located in its final position
  - Then we continue splitting the subarrays into smaller subarrays, until the resulting pieces have only one element (using recursion)

Quicksort algorithm

1. Choose an element as pivot. We use right element
2. Start indexes at left and (right-1) elements
3. Move left index until we find an element> pivot
4. Move right index until we find an element < pivot
5. If indexes haven't crossed, swap values and repeat steps 3 and 4
6. If indexes have crossed, swap pivot and left index values
7. Call quicksort on the subarrays to the left and right of the pivot value

Example

Quicksort(a, 0, 6)

Original	36	71	46	76	41	61	56
							pivot

Example

Quicksort(a, 0, 6)

Original	36	71	46	76	41	61	56
	i →	i			j	← j	pivot

Example

Quicksort(a, 0, 6)

Original	36	71	46	76	41	61	56
	i →	i			j	← j	pivot
1st swap	36	41	46	76	71	61	56

Example

Quicksort(a, 0, 6)

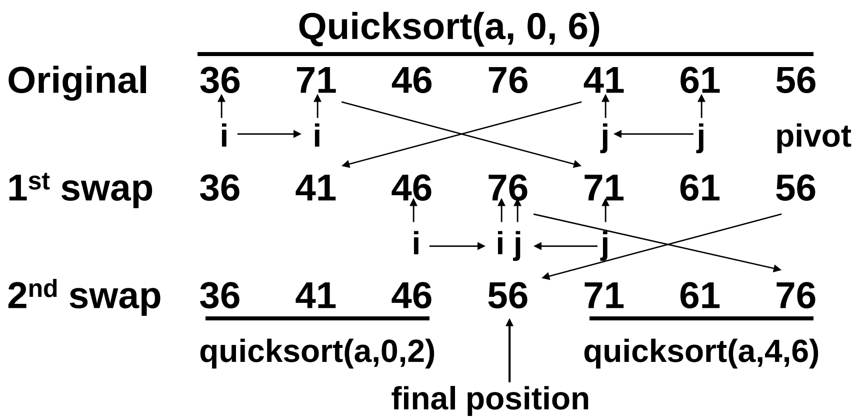
Original	**36**	**71**	**46**	**76**	**41**	**61**	**56**
	i →	**i**			**j**	← **j**	**pivot**
1st swap	**36**	**41**	**46**	**76**	**71**	**61**	**56**
			i →	**i j**	← **j**		

Example

Quicksort(a, 0, 6)

Original	**36**	**71**	**46**	**76**	**41**	**61**	**56**
	i →	**i**			**j**	← **j**	**pivot**
1st swap	**36**	**41**	**46**	**76**	**71**	**61**	**56**
			i →	**i j**	← **j**		
2nd swap	**36**	**41**	**46**	**56**	**71**	**61**	**76**

Example



Partitioning

- **Partitioning is the key step in quicksort.**
- **In our version of quicksort, the `pivot` is chosen to be the last element of the (sub)array to be sorted.**
- **We scan the (sub)array from the left end using index `low` looking for an element >= `pivot`.**
- **When we find one we scan from the right end using index `high` looking for an element <= `pivot`.**
- **If `low` < `high`, we swap them and start scanning for another pair of swappable elements.**
- **If `low >= high`, we are done and we swap `low` with the `pivot`, which now stands between the two partitions.**

Quicksort main(), exchange

```
import javax.swing.*;
public class QuicksortTest {                 // Timing details omitted
    public static void main(String[] args) {
        String input= JOptionPane.showInputDialog("Enter no element");
        int size= Integer.parseInt(input);
        Integer[] sortdata= new Integer[size];
        for (int i=0; i < size; i++)
            sortdata[i]= new Integer( (int)(1000*Math.random()));
        System.out.println(“Start");
        sort(sortdata, 0, size-1);
        System.out.println("Done");
        if (size <= 1000)
           for (int i=0; i < size; i++)
               System.out.println(sortdata[i]);
        System.exit(0);
   }
   public static void exchange(Comparable[] a, int i, int j) {
        Comparable o= a[i];                 //  Swaps a[i] and a[j]
        a[i]= a[j];
        a[j]= o;
    }
```

Quicksort, partition

```
  public static int partition(Comparable[] d, int start, int end) {
       Comparable pivot= d[end];           // Partition element
        int low=  start -1;
        int high= end;
        while (true) {
            while ( d[++low].compareTo(pivot) < 0) ;  // Move indx right
            while ( d[--high].compareTo(pivot) > 0 && high > low); // L
            if (low >= high) break;         // Indexes cross
            exchange(d, low, high);         // Exchange elements
        }
        exchange(d, low, end);              // Exchange pivot, right
        return low;
    }

    public static void sort(Comparable[] d, int start, int end) {
        if (start < end) {                   // If 2 or more elements
            int p= partition(d, start, end);
            sort(d, start, p-1);
            sort(d, p+1, end);
        }
    }
}
```

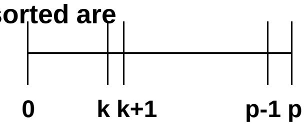
Better Quicksort

- Choice of pivot: Ideal pivot is the median of the subarray but we can't find the median without sorting first.
  - "Median of three" (first, middle and last element of each subarray) is a good substitute for the median.
    - Guarantees that each part of the partition will have at least two elements, provided that the array has at least four, but its performance is usually much better.
    - Median of 9 used on large subfiles
  - Randomize pivot element to avoid worst case behavior of already sorted list.
    - Appears less effective than good medians
- Convert from recursive to iterative
  - Process shortest subarray first (limit stack size, pops, pushes)
  - Makes almost no difference with current Java compiler
- When subarray is small enough (5-10 elements) use insertion sort
  - Makes a small difference

Quicksort performance

1 2 3 4 5 6 7 8 9 10 11

- Worst case:
  - If array is in already sorted order, each partition divides the array into subarrays of length 1 and n-1
  - It thus takes $\sum_{r=2}^{n} r = O(n^2)$ steps to sort the array
- Average case:
  - Partition element data[p] has equal probability of being the k^{th} smallest element, 0 <= k < p in data[0] through data[p-1]
  - The two subarrays remaining to be sorted are
    - data[0] through data[k]
    - data[k+1] through data[p-1]
    - with probability 1/p, 0 <= k < p



Quicksort performance, p.2

$$T(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}[T(k-1) + T(n-k)]$$

$$T(0) = T(1) = 0$$

Multiply both sides by n:

$$nT(n) = n(n+1) + 2(T(0) + T(1) + \ldots + T(n-1))$$

Substitute $(n-1)$ *for* n:

$$(n-1)T(n-1) = n(n-1) + 2(T(0) + T(1) + \ldots + T(n-2))$$

Subtract from previous equation:

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1)$$

$$nT(n) - (n+1)T(n-1) = 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

Repeatedly substitute for T(n - 1), T(n - 2), ... to get

$$\frac{T(n)}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

...

$$= 2\sum_{k=2}^{n+1}\frac{1}{k} \le \int_{1}^{n+1}\frac{dx}{x} = \ln(n+1) - \ln 2$$

$$T(n) \le 2(n+1)(\ln(n+1) - \ln 2) = O(n \ln n)$$

Quicksort: randomized

```
// Class has private static Random generator= new Random();

 public static void rsort(Comparable[] d, int start, int end) {
     int random = Math.abs(generator.nextInt());
     if (start < end) {
       if (end - start > 5)
         // Exchange random element with end and use as pivot
         exchange(d, random % (end - start + 1) + start, end);
       int p= partition(d, start, end);
       rsort(d, start, p-1);
       rsort(d, p+1, end);
     }
 }
```

Quicksort: iterative

```
public static void isort(Comparable[] d, int start, int end) {
    Stack s= new Stack(d.length/10);
    do {
        while (start < end) {
            int p= partition(d, start, end);
            if ((p - start) < (end - p)) {
                s.push(p+1);
                s.push(end);
                end= p-1;
            } else {
                s.push(start);
                s.push(p-1);
                start= p+1;
            }
        }                   // Sort smaller subarray first
        if (s.isEmpty())
            return;
        end= (Integer) s.pop();
        start= (Integer) s.pop();
    } while (true);
}
```

Quicksort: with insertion sort on small subfiles

```
public static void xsort(Comparable[] d, int start, int end) {
    if (start < end) {
        if (end - start < 10)
            InsertionSort.sort(d, start, end);
        else {
            int p= partition(d, start, end);
            xsort(d, start, p-1);
            xsort(d, p+1, end);
        }
    }
}
```

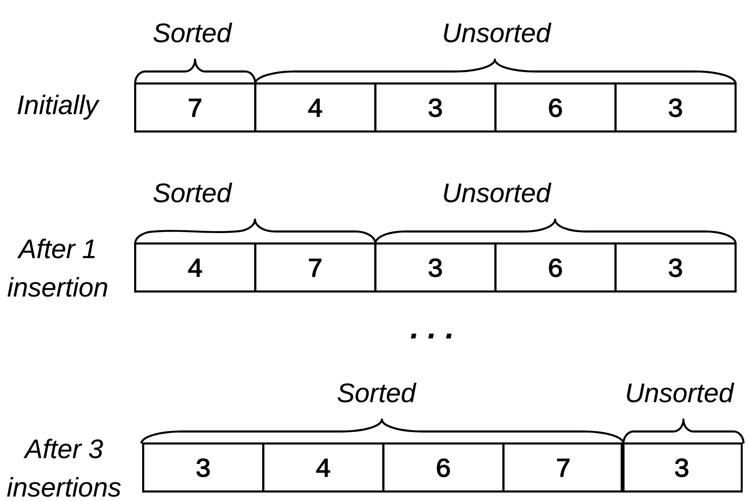
Insertion sort

```
public class InsertionSort {

   public static void sort(Comparable[] d) {
     sort(d, 0, d.length - 1);
   }

   public static void sort(Comparable[] d, int start, int end) {
     Comparable key;
     int i, j;
     for (j = start + 1; j <= end; j++) {
       key = d[j];
       for (i = j - 1; i >= 0 && key.compareTo(d[i]) < 0; i--)
         d[i + 1] = d[i];
       d[i + 1] = key;
     }
   }
}
```

Insertion Sort Diagram



Median of 9 quicksort, with insertionsort

```
public static void msort(Comparable[] d, int start, int end) {
        if (start < end) {
               if (end - start < 10)
                      InsertionSort.sort(d, start, end);
               else {
                      int l= start;
                      int n= end;
                      int m= (end - start)/2;
                      if (end - start > 40) {  // Big enough to matter
                             int s= (end - start)/8;
                             l= med3(d, l, l+s, l+2*s);
                             m= med3(d, m-s, m, m+s);
                             n= med3(d, n-2*s, n-s, n);
                             m= med3(d, l, m, n);
                      }
                      exchange(d, m, end);
                      int p= partition(d, start, end);
                      msort(d, start, p-1);
                      msort(d, p+1, end);
}    } }
// med3() returns median of 3 numbers. Code is obscure
public static int med3(Comparable[] x, int a, int b, int c) {
  return (x[a].compareTo(x[b]) < 0 ?
   (x[b].compareTo(x[c]) <0 ? b : x[a].compareTo(x[c]) <0? c : a) :
   (x[b].compareTo(x[c]) >0 ? b : x[a].compareTo(x[c]) >0? c : a));
}
```

Quicksort sample results

```
Size: 100000
Start regular quicksort, random input
Done, time (ms): 163
Start iterative quicksort, random input
Done, time (ms): 205
Start quicksort with insertionsort, random input
Done, time (ms): 168
Start random quicksort, already sorted input
Done random, time (ms): 142
Start Java Arrays.sort(), random input
Done, time (ms): 180
Start Java Arrays.sort(), already sorted input
Done, time (ms): 16
Start median quicksort, already sorted input
Done, time (ms): 75
```

Java Arrays.sort() code from:

L. Bentley and M. Douglas McIlroy "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) p. 1249-1265 (November 1993). Available as open source.

Selection: find kth smallest item in array

```
public class Select {
   public static void select1(Comparable[] a, int k) {
     int low = 0, up = a.length-1;
     do {
       int j = QuicksortTest.partition(a, low, up);
       if (k == j)     // Found kth item as partition
         return;
       else if (k < j) // kth item earlier in list
         up = j-1;     // Upper limit reset below partition
       else            // kth item later in list
         low = j+1;    // Lower limit reset above partition
     } while (true);
   }
```

Selection: example

```
public static void main(String[] args) {
     // Find kth smallest item (counting from 0, not 1)
     Integer[] a= {65, 70, 75, 80, 85, 60, 55, 50, 45, 99};
     select1(a, 0);        // And output

     Integer[] b= {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0};
     select1(b, 5);        // And output

     Integer[] c= {15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,15};
     select1(c, 6);        // And output

     Integer[] e= {3,7,2,0,-1,8,1,9,6,4,5,55,54};
     select1(e, 6);        // And output

     Integer[] d= {65, 70, 75, 80, 85, 60, 55, 50, 45, -1};
     select1(d, 7);        // And output
}
```

Selection: example output

```
45 70 75 80 85 60 55 50 65 99        // Start counting at 0
0th element is: 45

0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 6
5th element is: 5

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 15
6th element is: 7

3 4 2 0 -1 1 5 9 6 7 8 54 55
6th element is: 5

-1 45 50 55 60 65 70 75 80 85
7th element is: 75
```

Selection: complexity, summary

- **Select has same worst case as quicksort:**
  - **If list is already sorted, select is $O(n^2)$**
- **Same remedies**
  - **Random partition (same as used in quicksort)**
    - **Gives expected O(n) performance, but tends to be slow**
  - **Better pivot element (median selection)**
    - **Gives worst case O(n) performance. Proof long but straightforward**
    - **Horowitz text discusses similar ideas to Bentley-McIlroy algorithm in Arrays.sort() for selection: median, insertionsort, …**

Summary

- Summary: algorithms exist to avoid full sorts:
  - Selection/partition to find percentiles, ranks
  - Heaps to give largest or smallest element
  - If you need or want to sort, improved quicksort is usually best
- Divide and conquer algorithms
  - Binary search (use instead of BST if data static, in array)
  - Quicksort (preferred sort algorithm, partition has many uses)
    - Merge method from mergesort is also broadly useful
  - Selection
- This lecture was a small ‘lab’, typical of industry research practice
  - Find approaches from the literature, implement, analyze and test them
  - Designing and implementing short, clean codes for the algorithms
  - Some proofs
  - Timing a set of variations on an algorithm
  - In many cases, you won’t reproduce published results
    - Call the author, have others review your work, …