Linear Search

Code for a linear search on a sorted list of elements. We assume the elements are sorted in increasing order. Here is the iterative version of linear search:

```
boolean linearSearch( int[] input, int item ) {
    int n = input.length;
    for( int i = 0; i < n; i++ ) {
        if( item == input[ i ] )        return true;
        if( item <  input[ i ] )        return false;
    }
    return false;
}
```

Let us analyse the running time of this algorithm. The `for` loop runs at most n times, with 2 constant-time operations on the inside of the loop. Therefore the whole loop runs in $O(n)$ time.

Now suppose we would like to add the method `linearSearch( int item )` to our `List` class, using recursion. We would first add the abstract definition into our abstract `List` class, and then we would add the implementations to both `Empty` and `Cons`:

```
class Empty extends List {
    ....
    public boolean linearSearch( int item ) {
        return false;
    }
}

class Cons extends List {
    ....
    public boolean linearSearch( int item ) {
        if( item == first )             return true;
        if( item <  first )             return false;
        return rest.linearSearch( item );
    }
}
```

How about the running time for this recursive linear search algorithm? In order to analyse it, we need to first come up with a *recurrence relation* for the cost:

Let $C(n)$ be the cost of performing a linear search on a list/array of length n. Then

$$\begin{aligned} C(n) &= 1 && \text{for } n = 0 \\ C(n) &= 1 + C(n-1) && \text{for } n > 0 \end{aligned}$$

And to actually solve this recurrence relation (which we need to do in order to find the running time of our recursive linear search) we use the *iteration method*, which just iteratively expands the $C(n)$ in the recurrence relation:

$$\begin{aligned} C(n) &= 1 + C(n-1) \\ &= 1 + 1 + C(n-2) \\ &= 1 + 1 + \cdots + 1 + C(1) \\ &= 1 + 1 + \cdots + 1 + 1 \\ &= n \\ &\in O(n) \end{aligned}$$

And so the running time of our recursive linear search is the same as for the iterative linear search, $O(n)$.

Binary Search

The idea of a binary search is that you you compare the item you are looking for to the middle element of your sorted list. If your item is smaller then you recursively search in the left half of your list, otherwise you recursively search in the right half of your list. You are able to do this because your list is sorted. And by dividing your list in half at each step you are making way fewer comparisons than the linear search. The binary search code for an array is as follows:

```
boolean binarySearch( int[] input, int item, int first, int last ) {
    if( first == last )          // size 1 array
        return( item == input[ first ] );
    int pivot = input[ (first + last + 1)/2 ];
    if( item < pivot )
        return binarySearch( input, item, first, (first+last)/2 );
    else
        return binarySearch( input, item, (first+last+1)/2, last );
}
```

What is the running time of the above? Well... the first thing to notice is that this is a recursive algorithm, so you should first try to come up with a recurrence relation. Let $C(n)$ be the cost of the binarySearch algorithm on a list of length n. Then the recurrence relation is:

$$\begin{aligned} C(n) &= 1 && \text{for } n = 1 \\ C(n) &= 1 + C(n/2) && \text{for } n > 1 \end{aligned}$$

Which we solve iteratively, using the assumption that n is a power of 2 (i.e. that $n = 2^k$ for some integer k):

$$\begin{aligned} C(n) &= 1 + C(n/2) \\ &= 1 + 1 + C(n/4) \\ &= 1 + 1 + \cdots + 1 + C(1) \\ &= 1 + 1 + \cdots + 1 \\ &= \log n \\ &\in O(\log n) \end{aligned}$$

Matrix Multiplication

Suppose we are considering the algorithm to multiply two n by n matrices:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Where A, B, C, D, E, F, G, H are $n/2$ by $n/2$ matrices. Then from the above equation, you can see that the cost of multiplying two n by n matrices is the cost of multiplying 8 $n/2$ by $n/2$ matrices (namely AE, BG, AF, ...) and then performing n^2 additions. So we have the following recurrence relation:

$$\begin{aligned} C(n) &= 1 && \text{for } n = 1 \\ C(n) &= n^2 + 8C(n/2) && \text{for } n > 1 \end{aligned}$$

And solving this iteratively, assuming $n = 2^k$ for some integer k, we have:

$$\begin{aligned} C(n) &= n^2 + 8C(n/2) \\ &= n^2 + 8(\frac{n^2}{2^2} + 8C(n/4)) \\ &= n^2 + 2n^2 + 8^2C(n/4) \\ &= n^2 + 2n^2 + 8^2(\frac{n^2}{2^4} + 8C(n/8)) \\ &= n^2 + 2n^2 + 4n^2 + 8^3C(n/8) \\ &= n^2 + 2n^2 + 4n^2 + \cdots + 2^{\log n - 1}n^2 + 8^{\log n}C(1) \\ &= (1 + 2 + 4 + \cdots + 2^{\log n - 1})n^2 + 8^{\log n} \\ &= (2^{\log n} - 1)n^2 + 8^{\log n} \\ &= (n-1)n^2 + 2^{3\log n} \\ &= (n-1)n^2 + 2^{\log n^3} \\ &= (n-1)n^2 + n^3 \\ &\in O(n^3) \end{aligned}$$

Insertion Sort

Insertion sort sorts a list by repeatedly taking the next item from our list and inserting it into a sorted structure in its proper order with respect to the items already inserted. Let's start with the recursive code, since it is a lot cleaner than the iterative version:

```
class Empty extends List {
    ....
    public List insertionSort( ) {
        return this;
    }
}

class Cons extends List {
    ....
    public List insertionSort( ) {
        List sorted = new Empty();
        List l      = this;
        while( !( l instanceof Empty ) ) {
            sorted = insert( l.getFirst(), sorted );
            l = l.getRest();
        }
        return sorted;
    }
```

```
    List insert( int item, List s ) {
        if( s instanceof Empty )       return new Cons( item, s );
        if( item < s.getFirst() )      return new Cons( item, s );
        else return new Cons( s.getFirst(), insert( item, s.getRest() ) );
    }
}
```

What is the running time of the above algorithm? First we have to notice that there is a recursive helper method called `insert` that is called n times from the `insertionSort` method. What is the helper's recursion relation?

$$\begin{aligned} C(n) &= 1 && \text{for } n = 0 \\ C(n) &= 1 + C(n-1) && \text{for } n > 1 \end{aligned}$$

Which we've already solved previously to be $O(n)$. Therefore, if the helper is $O(n)$ and is called n times, then the `insertionSort` method itself is $O(n^2)$.

The iterative code is a *lot* more messy, but is as follows:

```
int[] insertionSort( int[] input ) {
    int   n            = input.length;
    int   sortedLength = 0;
    int[] sorted       = new int[ n ];

    for( int i = 0; i < n; i++ ) {
        int item = input[ i ];
        if( sortedLength == 0 ) {    // insert here
            sorted[ sortedLength++ ] = item;
        }
        else {   // insert item into its proper place
            int j = 0;
            // find item's place
            while( ( j < sortedLength ) && ( item >= sorted[ j ] ) )   j++;
            int k = j;
            int temp = sorted[ k ];
            while( k < sortedLength ) {      // bump all remaining elements
                int temp2 = sorted[ k+1 ];   // back one position to make
                sorted[ k + 1 ] = temp;      // room for the element we want
                temp = temp2;                // to add
                k++;
            }
            sorted[ j ] = item;
            sortedLength++;
        }
    }
}
```

It should also have a running time of $O(n^2)$.

Excercise: Analyse the running time of mergesort!