Lecture 6

Sorting: This lecture covers *sorting*. In this problem, we have a list of items (let's say integers) and would like to sort them from smallest to biggest. A natural method for sorting, which is probably what most of us do in real life, is look for the smallest element, put it at the beginning, then sort what's left. This is known as `selectSort`, and here is an implementation in Python:

```
def selectionSort(L):
    if len(L) == 0:
        return []
    smallest = L[0]
    smallestIndex = 0
    for i in xrange(1, len(L)):
        if L[i] < smallest:
            smallest = L[i]
            smallestIndex = i
    # In Python, a,b = b,a swaps the contents of the variables a,b
    L[0],L[smallestIndex] = L[smallestIndex],L[0]
    return [L[0]] + selectionSort(L[1:])
```

Another approach to sorting is to gradually make prefixes of the list sorted. That is, first we'll make sure L[0:1] is sorted, then L[0:2], etc., all the way up to L[0:len(L)]. The following method is known as `insertionSort`.

```
def insertionSort(L):
    for i in xrange(1, len(L)):
        # We assume L[0:i] is already sorted, and now need to put L[i]
        # in its rightful place.
        j = i - 1
        value = L[i]
        while j>=0 and L[j]>value:
            L[j+1] = L[j]
            j -= 1
        L[j+1] = value
    return L
```

Another problem sorting method is `bubbleSort`. This procedure has several iterations. In each iteration you start at the beginning of the list and move to the end, one item at a time, and for each item you encounter you swap it with its adjacent element on the right if the two elements are inverted. This is done repeatedly until there are no more swaps being performed.

```
def bubbleSort(L):
    swapped = True
    while swapped:
        swapped = False
```

```
        for i in xrange(len(L)-1):
            if L[i]>L[i+1]:
                L[i],L[i+1] = L[i+1],L[i]
                swapped = True
    return L
```

The last sorting method we'll cover today is `mergeSort`. This is a recursive procedure for sorting a list. We break the list in two equal-sized halves (or as equal-sized as possible if the list size is odd), recursively sort each half, then merge the two lists together.

```
def mergeSort(L):
    if len(L)<=1:
        return L
    # recursively sort the first half of L and put the result in A, and
    # recursively sort the second half of L and put the result in B
    A = mergeSort(L[0:len(L)/2])
    B = mergeSort(L[len(L)/2:])

    # now merge A and B, and put the result in C
    C = []
    aindex = 0
    bindex = 0
    for i in xrange(len(L)):
        if aindex==len(A):
            C += B[bindex:]
            break
        elif bindex==len(B):
            C += A[aindex:]
            break
        else:
            if A[aindex] < B[bindex]:
                C += [A[aindex]]
                aindex += 1
            else:
                C += [B[bindex]]
                bindex += 1
    return C
```

As we will see in the next section, `mergeSort` is usually the best choice of these four methods when it comes to algorithm speed.

Analysis of algorithms: This class is mostly concerned with *algorithms*. So, what is an algorithm? An algorithm is a well-defined computational procedure that takes some data as input and computes new data for output. For example, an algorithm for multiplying integers takes two integers as input and outputs their product. An algorithm for sorting a list of numbers takes a list of numbers as input then outputs a list with the same numbers, but in sorted order.

It is often the case that there are many different algorithms which accomplish the same task, and we must choose which one to use. For example, we just saw four different algorithms to accomplish the task of sorting (`selectionSort`, `insertionSort`, `bubbleSort`, and `mergeSort`).

Order of growth: The most popular way of expressing the running time of an algorithm is by expressing how well it scales as the input size gets larger and larger. Also, to have absolute guarantees, we measure the running time of an algorithm in the *worst case* over all inputs of a given size. Running any one of `selectionSort`, `bubbleSort`, and `insertionSort`, on the list [n,n-1,n-2,...,1] takes roughly $\approx n^2$ steps. Meanwhile, it is possible to show that no input can cause `mergeSort` to take more than roughly $\approx n \log_2 n$ steps. Thus, `mergeSort` is the superior choice.

The notation that has become the convention for measuring algorithm running times is the *big-Oh* notation. big-Oh is a way of relating two functions $f(x), g(x)$ (we would either write "$f(x)$ is big-Oh of $g(x)$", or "$f(x) = O(g(x))$"; often we don't write the x and it is understood). The precise definition of big-Oh is that $f(x) = O(g(x))$ if there exists numbers $C, N > 0$ such that once $x > N$ it holds that $f(x) \leq Cg(x)$. In other words, once x gets big enough, f is less than or equal to some constant scaling of g.

In the analysis of the running time of some algorithm A, we often care about $f(n)$ being a function that tells us the worst-case running time of A over all inputs of size n.

Just as big-Oh can be seen as a "less than or equal to", big-Omega can be seen as a "greater than or equal to". We say $f = \Omega(g)$ if f grows at least as fast as g. That is, $f = \Omega(g)$ means the same thing as $g = O(f)$. We also say f is *Theta* of g, or $f = \Theta(g)$, if $f = O(g)$ and $f = \Omega(g)$ simultaneously.

Here are some examples of big-Oh:

- $n^2 = O(n^2)$
- $n^2 = O(n^3)$
- $3n^2 = O(n^2)$
- $n^2 = O(3n^2)$
- $5n^8 + 2n = O(n^8)$
- $3n \log_{10} n = O(n \log_2 n)$, since switching bases of log just changes the value by a constant ($\log_{10} n = (1/\log_2 10) \cdot \log_2 n$).

Now, when measuring running times of the sorting algorithms, the example [n,n-1,n-2,...,1] tells us that the running times of `selectionSort`, `bubbleSort`, and `insertionSort` are all $\Omega(n^2)$. It's also not too hard to see their worst case running times are all $O(n^2)$ (in other words, their running times are $O(n^2)$ on *all* input lists of size n). For example, in `insertionSort`, we have a `for` loop that takes n iterations, and in each iteration we have a `while` loop that goes on for at most n iterations. Thus, these three algorithms all have running times which are $\Theta(n^2)$.

What about `mergeSort`?

Recurrences: Let's look at the code for `mergeSort`:

```
def mergeSort(L):
    if len(L)<=1:
        return L
    A = mergeSort(L[:len(L)/2])
    B = mergeSort(L[len(L)/2:])
    return merge(A, B)
```

We can write a *recurrence* that expresses the running time of `mergeSort`. What is a recurrence? A recurrence is a definition of a function that expresses its value on an input as some combination of its values on smaller inputs. For example, our definition of $\text{fib}(n)$ as the nth Fibonacci number in Lecture 3 was a recurrence:

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

Turning back to `mergeSort`, let $T(n)$ be the function which denotes the worst-case running time of `mergeSort` on any input list of length n. Then, given the code for `mergeSort`, we see that

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2 \cdot T(n/2) + C \cdot n & \text{otherwise} \end{cases}.$$

The $C \cdot n$ term comes from the running time of the `merge` operation, where $C > 1$ is some constant value. The above recurrence isn't *exactly* right since if n is odd we don't break into exactly two pieces of size $n/2$, but to make the presentation simpler let's assume for now that n is a power of 2 so that in all recursive steps we always break the input into exact halves.

Now, let's show that $T(n) \leq Cn \log_2 n + n$. We will show it using a proof technique called *induction*. Specifically, we will first show the inequality holds when $n = 1$ (the **base case**). Then we will show that if the inequality holds for $T(1), T(2), \ldots, T(n-1)$, then it must also hold for $T(n)$ (the **inductive step**).

- **Base case:** When $n = 1$, $T(n) = 1$. We also have $C \cdot 1 \log_2 1 + 1 = 1$, so indeed the claim holds for $n = 1$.
- **Inductive step:** We assume the running time for $n/2$ satisfies $T(n/2) \leq C(n/2)\log_2(n/2) + (n/2)$. Now, we have from the recurrence that

$$\begin{aligned} T(n) &\leq 2 \cdot (C(n/2)\log_2(n/2) + (n/2)) + Cn \\ &= Cn \log_2(n/2) + n + Cn \\ &= Cn \log_2 n + n, \end{aligned}$$

  since $\log(a/b) = \log(a) - \log(b)$. Thus we have shown the claim. Now note that $Cn \log_2 n + n = O(n \log_2 n)$, so the running time of `mergeSort` is $O(n \log_2 n)$. One can similarly also show that in fact its running time is also $\Omega(n \log_2 n)$, so that it is $\Theta(n \log_2 n)$. This means that comparing `mergeSort` with the other three sorting algorithms from yesterday, `mergeSort` is faster by a factor of $\Theta(n/\log_2 n)$, which is quite large (for example, $n/\log_2 n$ is about 50000 when n is one million).