CSCI 104L Lecture 13: Sorting II

Recall MergeSort from last lecture. The recurrence relation for MergeSort is:

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \]
\[ T(1) = \Theta(1). \]

There are many ways you can solve such a recurrence relation, including:

- Draw a recursion tree, calculate the amount of work at each level of the tree, and add it up.
- Do an inductive proof.

QuickSort

```c
void QuickSort (T a[], int l, int r) {
    if (l < r) {
        int m = partition(a, l, r);
        QuickSort(a, l, m-1);
        QuickSort(a, m+1, r);
    }
}

int partition(T a[], int l, int r) {
    int i=1;
    T p = a[r];
    for (int j = l; j < r; j++) {
        if (a[j] <= p) {
            a.swap(i, j);
            i++;
        }
    }
    a.swap(i, r);
    return i;
}
```

**Question 1.** Is QuickSort stable?

**Question 2.** What would be a loop invariant for the partition function?

**Question 3.** What would be a recurrence relation for QuickSort?

**Question 4.** What would be the best and worst choice of pivot, in terms of runtime?

QuickSort is a prime example of an algorithm for which it makes best sense to do average-case runtime analysis.
Keeping a List Sorted

Suppose we want to have a list class where the data is stored in sorted order. Our list previously had the following functions:

```cpp
void set(int pos, const T& data);
const T& get(int pos) const;
void insert(int pos, const T& data);
void remove(int pos);
```

Note the following:

- Nothing need change for get or remove.
- Insert needs to be modified so that it doesn’t accept a position.
- There is no point in having a Set function; accordingly, our interface can be:

```cpp
const T& get(int pos) const;
void insert(const T& data);
void remove(int pos);
```

**Question 5.** What is the runtime for insert/remove/get on a sorted array?

**Question 6.** On a sorted linked list?