CSC148
Lecture 8

Algorithm Analysis
Sorting
Recall definition of Big-Oh: We say a function \( f(n) \) is \( O(g(n)) \) if there exists positive constants \( c, B \) such that
\[
f(n) \leq c \cdot g(n) \text{ for all } n \geq B
\]

Let \( T(n) \) be the worst-case “running time” of an algorithm on input size \( n \). (In this context, “running time” means the number of steps that the algorithm takes.)
Algorithm Analysis

• Loosely speaking, we approximate $T(n)$ by finding a function $g(n)$ such that $T(n)$ is $O(g(n))$.

• Saying that this is an “approximation” for the running time isn't entirely accurate. Consider the algorithm for summing the numbers from 1 to $n$ that we saw last week.
Algorithm Analysis

- The first algorithm, which loops through all the numbers from 1 to n, has time complexity O(n).
- The second algorithm, which uses a formula, has time complexity O(1).
- Is the following statement true: “both algorithms have time complexity O(n^2)”?
- It is! Consider the definition of Big-Oh, and you will see why.
Algorithm Analysis

- Clearly neither algorithm takes anywhere near \( n^2 \) steps.
- We said that Big-Oh notation is used to approximate \( T(n) \), but the last example demonstrates that the notation can lead to inaccurate approximations. What's going on??
- In actuality, Big-Oh notation gives us a convenient way of expressing an upper-bound on the running time of an algorithm.
Algorithm Analysis

- Saying that the summation algorithms take $O(n^2)$ time, although true, doesn't convey as much information as we'd like.

- To make our upper-bound as meaningful as possible, we want to make it “tight”.

- Intuitively, $O(g(n))$ is a tight upper-bound for $T(n)$ if $g(n)$ is the smallest and simplest function that satisfies the big-oh criteria.
Algorithm Analysis

- For example, $O(n)$ is a tight upper-bound for $6n$, but $O(n^2)$ is not.

- More precisely, if for every function $h(n)$ such that $T(n)$ is $O(h(n))$ it is also true that $g(n)$ is $O(h(n))$, then we say $g(n)$ is a tight asymptotic bound on $T(n)$.
  
  - Think carefully about this definition. Why does it capture the intuition described on the previous slide?
Sorting

• Sorting methods that you've seen in 108:
  – Bubble sort
  – Selection Sort
  – Insertion sort

• These sorts all have time complexity $O(n^2)$.

• We'll discuss a new sorting method, called merge sort, that has time complexity $O(n \log n)$. 
Merge Sort

• Merge sort recursively
  – sorts the first half of the list
  – sorts the second half of the list
  – merges the two halves into a newly sorted list

• Lets assume we have a list in which the first and second halves are sorted, but the whole list itself may not be sorted.

• How can we merge the two halves to create a new list that's sorted and contains all the elements of the original list?
Merge Sort

- Examples of merge on board.
Merge Sort

• Before we can actually use the merge procedure we just discussed, we have to somehow get to the point where the two halves of the list are sorted.

• This is done recursively.

• What is our base case?
Merge Sort

- A list containing 1 element is sorted.
Merge Sort

- **Advantages:**
  - $O(n \log n)$ time complexity
    - see discussion on board for why mergesort has this time complexity

- **Disadvantages**
  - requires additional space for the merged list