CSC148
Lecture 8

Algorithm Analysis
Binary Search
Sorting

Algorithm Analysis

- 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) <= c*g(n) for all n >= 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?

Algorithm Analysis

- Big-oh hierarchy on board
- Examples of analyzing algorithms.

Binary Search

- I'm thinking of a number between 1 and 100, each of which is equally likely. After you make a guess, I'll tell you if you guessed the number, or if the number is higher or lower than your guess.
- If you want to determine the number in as few guesses as possible, what strategy should you employ?

Binary Search

- A naïve approach would be to simply start guessing each number from 1 to N, ignoring the high/low information, until you guess the number. But there's a better way...
- You can always eliminate half the possible numbers by guessing the midpoint in the range of remaining possibilities.
- By eliminating half the remaining numbers in each guess, you can determine the number I'm thinking in no more than 7 steps.

Binary Search

- In general, if I'm thinking of a number from 1 to n, you can determine the number in no more than ceil($\log_2 n$) steps.
- We can apply this same idea to searching for an item in a sorted list.
- Given a sorted list of n items, you want to determine whether the item is in the list.

Binary Search

- A naïve approach is to search linearly for the item.
- Since the list is sorted, you can search the list more intelligently.
- As with the guessing numbers game, check to see if the item is at the midpoint of the list.
  - If the item is at the midpoint, you are done.
  - Otherwise, you know whether the item is in the first half or second half of the list. This means you can eliminate half the list from consideration.

Binary Search

- After you've eliminated half the items from consideration, recursively search for the item in the remaining half.
- If the item is NOT in the list, then eventually you'll try searching an empty list, at which point you are done.
- Binary search has time complexity O(log N), where N is the size of the list.

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.
- Lets develop mergesort in Wing.

Merge Sort

- Advantages:
  - O(n log n) time compelxity
    - see discussion on board for why mergesort has this time complexity
- Disadvantages
  - requires additional space for the merged list

Quick Sort

- Recursive, like merge sort, sorting is “in place”. That is, additional space is not required.
- The main idea behind quicksort is contained in the partition procedure. It works by choosing a “pivot” element and
  - finding the correct position of the pivot element in the final sorted list (this is called the “split point”)
  - moving elements less than the pivot before the split point, and other elements after the split point.

Quick Sort

- Quick sort works by partitioning the list (using the partition procedure described above), and then recursively sorting the lists before and after the split point.

Quick Sort

- Examples on board

Quick Sort

- Lets look at the quick sort procedure in Wing.