Notes 9-18

Cameron Chalk

1 Some common runtimes

- $\mathcal{O}(1)$
- $\mathcal{O}(\log n)$
- $\mathcal{O}(\log(n)^c)$ where c is a constant (does not change with input size n)
- $\mathcal{O}(n)$
- $\mathcal{O}(n \log n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^3)$
- $\mathcal{O}(2^n)$

2 Simple Example Algorithms

Compute the max of n numbers: loop through the numbers, and keep track of the largest number. At the end of the list, you have the largest number. $\mathcal{O}(n)$ time.

Merge 2 sorted lists into sorted whole: Compare first two numbers in each list. Add the smaller of the two to the new list and remove it from the old list. Repeat until both original lists are empty. $\mathcal{O}(n)$ time.

Given n points in the plane, find the closest pair. Simple solution: check all pairs, and keep the closest pair. $\mathcal{O}(n^2)$. A better solution exists which works in $\mathcal{O}(n \log n)$ time. The solution is non-trivial and we will see it later, but it's a good example of a simple problem in which the obvious, simple solution is not as fast as a more ingenious solution.

Sorting n numbers. A $\mathcal{O}(n \log n)$ solution exists which we will see later, and again, achieving a solution with this runtime is not obvious.

Given an undirected graph, an *independent set* is a set of nodes such that no two are joined by an edge. Problem: Find the largest indepedent set. Brute force solution: for each subset of vertices (of which there are 2^n), check if that subset is an independent set (this takes n^2 time to check if each pair has no edge), and keep track of the largest one. This takes $\mathcal{O}(2^n \cdot n^2)$. In the worst

case, no one knows if there is a subexponential time algorithm for this problem. If you find one, you literally will win one million dollars from the Clay institute of mathematics. You can google P vs. NP for more information if you want to get ahead of the class.

Given a sorted array, determine if a number p is in it. Simple solution: loop through the list and look for p. This takes $\mathcal{O}(n)$ time. A better solution, since the list is sorted: Start in the middle, and determine if the number in the middle is larger, smaller, or equal to p. If it is larger, cut off the larger half of the list. If it is smaller, cut off the smaller half of the list. Repeat. Since each time we cut off half of the list, the list is of $\mathcal{O}(1)$ size after $\log n$ cuts-in-half, so the runtime is $\mathcal{O}(\log n)$.

3 Lower bound on sorting

We will prove a *lower bound* on the worst case run time of sorting. We will use Ω notation; recall that this is the asymptotic notion of $\geq$, where $\mathcal{O}$ is the asymptotic notion of $\leq$ (check the definitions from the previous notes if you are still confused about these notations).

Since you might be sorting numbers or sorting cars by price or sorting songs from your most favorite to least favorite, we want a general notion (beyond numbers) of sorting, called *comparison sorting*. The rules are that you can only look at the input by pairwise comparisons. Example algorithms which solve comparison sort problems include: mergesort, bubblesort, selection sort, quicksort, sort by heap, etc. This does not include radix sort or counting sort, which sort only numbers (within a certain range).

Claim 1. *Any comparison sorting algorithm runs in time* $\Omega(n \log n)$.

Proof. We can view any comparison sort algorithm as a binary decision tree.

The key point: if the algorithm is correct, it must be able to generate via the decision tree any of the $n!$ possible permutations; that is, any of the $n!$ possible permutations must be a leaf in the binary decision tree. The (worst-case) running time is at least the height of the decision tree, because that is the number of comparisons which must be made in the worst-case input. A binary tree with height h has at most 2^h leaves. Then a binary decision tree with at least $n!$ leaves must have at least $\log n!$ height. In the homework, we prove that $\log n! = \Omega(n \log n)$. □

4 Priority queues (a.k.a. Heaps)

Intuitively, a priority queue combines the benefits of linked lists and sorted arrays.

- (doubly) linked list: insert/delete-with-pointer in $\Theta(1)$ time, find minimum in $\Theta(n)$ time

- sorted array: insert/delete-with-pointer in $\Theta(n)$ time, find minimum in $\Theta(1)$ time

Sorted arrays are good for finding minimum, and linked lists are good for insertion and deletion. We want a data structure with fast insertion/deletion and fast find minimum. Our heap will have the following:

- heap: insert/delete-with-pointer in $\Theta(\log n)$ time, find minimum in $\Theta(1)$ time

Heaps were originally developed for the *heapsort* comparison sorting algorithm. If you believe the insert/delete and find minimum runtimes we've given for a heap, then the heapsort algorithm and runtime is as follows:

Heapsort: insert each number into a heap ($\mathcal{O}(n \log n)$), then repeatedly (until the heap is empty) find minimum ($\mathcal{O}(1)$), add it to your new list, and delete it ($\mathcal{O}(\log n)$).

So the total runtime is $\mathcal{O}(n \log n)$, which meets our lower bound for comparison sort. Next time we will design the heap.