1 Review

Here, we will briefly review the previous lecture.

1.1 Divide and Conquer

Recall from the previous lecture that the idea behind divide and conquer is to:

- Split the problem into parts.
- Recursively solve each part.
- And then somehow recombine these parts to get the final answer.

We will now use this technique to solve several important problems.

1.2 Master Theorem

Theorem 1.1: Master Theorem

Let $T(n)$ be given by the recurrence:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ aT\left(\frac{n}{b} + \mathcal{O}(1)\right) + \mathcal{O}(n^d) & \text{Otherwise} \end{cases}$$

Then we have:

$$T(n) = \begin{cases} \mathcal{O}(n^{\log_b(a)}) & a > b^d \\ \mathcal{O}(n^d \log(n)) & a = b^d \\ \mathcal{O}(n^d) & a < b^d \end{cases}$$

2 Sorting

Given a list L of numbers, return L in sorted order.

2.1 Algorithm Idea

An idea is to:

- Divide L into two parts, L_1 and L_2.
- Sort L_1 and L_2.
- Finally, *merge* both L_1 and L_2 together.

2.2 Merge Operation

One question we have is, how do we define the *merge* operation? Well, suppose we have

$$L_1 = [1, 3, 6, 10]$$

$$L_2 = [2, 4, 5, 7, 8, 9]$$

We can combine the two lists by comparing the two smallest elements an then putting the smaller of the two elements in the new list. So, for example, the first element in L_1 is 1 and the first element in L_2 is 2, so we can put 1 into the final list and somehow point L_1 to the second element so we don't consider 1 again. Then, we can consider 3 in L_1 and 2 in L_2; in this case, we put 2 into the final list and point L_2 to the

second element so we don't consider 2 again. We can keep doing this until we go through every element in both parts of the list.

The algorithm can be described like so:

```
Merge(A, B):
    Let C be the list with the length being len(A) + len(B)
    // Kane uses one-indexing instead of zero-indexing.
    a = 1
    b = 1
    for c = 1 to len(C):
        if b > len(B)
            C[c] = A[a]
            a++
        else if a > len(A)
            C[c] = B[b]
            b++
        else if A[a] < B[b]
            C[c] = A[a]
            a++
        else
            C[c] = B[b]
            b++
    Return C
```

This runs in $\mathcal{O}(|A| + |B|)$ time, particularly due to the `for`-loop.

2.3 Merge Sort Algorithm

```
MergeSort(L):
    // Every divide & conquer algorithm needs a base case
    if len(L) == 1:
        return L

    Split L into approx. equal lists L1, L2
    return Merge(MergeSort(L1), MergeSort(L2))
```

To analyze the runtime, we note the following:

- Base Case: The base case runs in $\mathcal{O}(1)$ time; this is obvious (and is expected for any base case).
- Divide & Conquer: The divide and conquer part comes in several steps.
  - Divide: We divide our list L into approximately equal-sized lists L_1 and L_2. We assume that this takes $\mathcal{O}(1)$ time.
  - Conquer: It takes $2T(n/2 + \mathcal{O}(1))$ time to recursively solve two subproblems, each of size $\frac{n}{2}$. The $\mathcal{O}(1)$ is due to the possibility that one of the two lists may have unequal sizes (e.g. one sublist has size 5 and another sublist has size 6). At the end of the day, we can just simplify this down to $2T(n/2)$.
  - Combine: Merging an n-element sublist takes $\mathcal{O}(n)$ time. This term absorbs the $\mathcal{O}(1)$ time from the divide part of the algorithm.

We then have (by the Master Theorem):

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Note that $a = 2$, $b = 2$, and $d = 1$, so the final runtime is given by the Master Theorem:

$$\mathcal{O}(n \log n)$$

3 Order Statistics

Given a list of numbers L, find the **median** or the **largest element** or the **10th smallest** element of L. Essentially, you want to find something based on the order in some way.

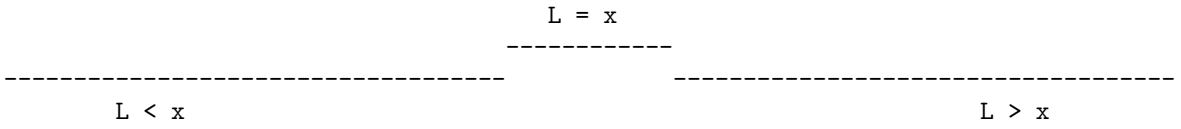
Let's focus on one particular problem: Given L and k, find the kth smallest element of L.

3.1 Easy Algorithm

The easy algorithm is to sort L and then return $L[k]$. The runtime is $\mathcal{O}(n \log n)$, but there are better ways to do this.

3.2 The Idea for the Divide Step

First, we pick a *pivot* $x \in L$ at random here, and we sort all elements in L relative to x. That is, every element to the *left* of x is less than x, every element to the *right* of x is greater than x, and every element in the center is equal to x. In other words:



In this sense, we can sort these elements into categories in $\mathcal{O}(n)$ time. Then:

- We note that the kth smallest element is less than x if and only if $|L_{<x}| \geq k$. If this is the case, then the answer is the kth smallest element of $L_{<x}$.
- The answer is x if and only if $|L_{<x}| < k$ and $|L_{<x}| + |L_{=x}| \geq k$.
- Otherwise, the answer is the $(k - |L_{<x}| - |L_{=x}|)$th smallest in $L_{>x}$.

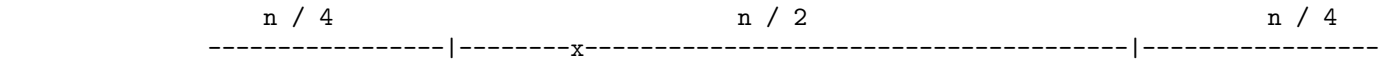
3.3 Divide and Conquer Algorithm

```
OrderStats(L, k)
    Pick x in L randomly
    Sort into L[<x], L[=x], L[>x]
    If len(L[<x]) >= k
        return OrderStats(L[<x], k)
    else if len(L[<x]) + len(L[=x]) >= k
        return x
    else
        return OrderStats(L[>x], k - len(L[<x]) - len(L[=x]))
```

The idea is that we use randomization to improve our runtime. If we didn't pick a random $x \in L$, then it's very possible that the x that we pick is the smallest or biggest x, which means we would essentially have to consider $n-1$ elements, and thus we would have a divide algorithm that takes $T(n-1)$ time (and so $\mathcal{O}(n^2)$ time). By randomly selecting an $x \in L$

- There is a 50% chance that x is selected in the middle half.
- If so, no matter where the answer is, the recursive call is at most $3n/4$.
- On average, we might need two tries to reduce calls.

The idea behind the $3n/4$ is that, when we pick this x, we either recurse on everything bigger than x or smaller than x. Because x is in this middle half, there are at least $n/4$ elements on either sides:



So, we have the runtime

$$T(n) = \mathcal{O}(n) + T(3n/4)$$

By the Master Theorem, we get the *expected*

$$\mathcal{O}(n)$$