This is the last discussion! Some reminders and announcements as we wrap up:

1. As announced this week on Canvas, Duke's course evaluations are open and there is extra credit available! Please see previous Canvas announcements about it.
2. HW10.2 is due Monday evening and will be available on Canvas today (if not already). It is very much in line with today's problems.
3. Since the Exam 3 turned out to be a "Limited Fire Alarm Edition", we will announce some changes to exam grades in light of it by Tuesday's lecture.
4. Our own end-of-semester survey will be available on Qualtrics, which is worth 0.5% of your final grade. Details and links coming soon!
5. Tuesday's lecture is dedicated for final exam review. Please bring questions!

---

1. Consider the MergeSort algorithm below, which takes an array $A[1..n]$ of **distinct** integers as input.

```
MergeSort($A[1..n]$):
  if $n = 2$ and $A[1] > A[2]$:
      swap $A[0] \leftrightarrow A[1]$
  else if $n > 2$:
      $m \leftarrow \lfloor n/2 \rfloor$
      MergeSort($A[1..m]$)
      MergeSort($A[m+1..n]$)
      Merge($A$)
```

Assume that Merge permutes any given array of n distinct integers where the first subarray of $\lfloor n/2 \rfloor$ integers is sorted in increasing order and the remainder of the array is sorted in increasing order so that the overall array is increasing order.

In lecture, we showed proved by induction that the algorithm above is correct: for any array of distinct integers, it terminates with the input array permuted so that its elements are in increasing order.

Now that we have proven the algorithm correct, we wish to prove its asymptotic runtime. Today, we'll focus on bounding the number of *comparisons* performed by the algorithm, meaning the number of times any two elements of the input array are compared to each other. Assume that Merge performs at most n operations given an array of length n.

As a recursive algorithm, we can express the total number of operations as a recurrence relation, as follows. For simplicity, let us assume n is a power of two. Let $T(n)$ be the total number of comparisons for an input array of size n.

$$T(n) \le \begin{cases} 1 & \text{if } n \le 2 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

In CS 201, it is shown that the runtime is $O(n \log_2 n)$. Armed with induction, we are ready to prove this bound for the number of comparisons ourselves. Prove that $T(n) \le n \log_2 n$ for any integer $n \ge 1$.

2. Consider the StoogeSort algorithm below, which takes an array $A[1..n]$ of **distinct** integers as input. For simplicity, we assume that n is a power of 3.

```
StoogeSort($A[1..n]$):
    if $n = 2$ and $A[1] > A[2]$:
        swap $A[1] \leftrightarrow A[2]$
    else if $n > 2$:
        StoogeSort($A[1..2n/3]$)
        StoogeSort($A[n/3 + 1..n]$)
        StoogeSort($A[1..2n/3]$)
```

(a) Using strong induction, prove that StoogeSort($A[1..n]$) an input array A of distinct integers into increasing order. *[Hint: It may be useful to first follow where the largest third of the elements are before/between/after each of the recursive calls.]*

(b) Express the number of comparisons made between the array's elements as a recurrence, then show it is upper-bounded by n^3.

(c) *To think about later*: It turns out that the tighter bound of $cn^{\log_{3/2} 3} = O(n^{2.701...})$ holds for a constant $c > 0$. Use induction to prove this tighter bound, likely relying on laws of logarithms in your proof. (Leave c as a placeholder for the constant value through your analysis, then choose an appropriate constant at the end.)

3. Consider the recursive implementation of the BubbleSort algorithm below, which also is a sorting algorithm like the previous.

```
BubbleSort($A[1..n]$):
    if $n = 1$:
        return
    for $i \leftarrow 1; i < n; i \leftarrow i + 1$:
        if $A[i] > A[i + 1]$:
            swap $A[i] \leftrightarrow A[i + 1]$
    return BubbleSort($A[1..n - 1]$)
```

To prove this algorithm correct, we will rely on induction in a similar way as for the previous examples. However, as this implementation involves a non-trivial for-loop, we first need to reason about the state of the algorithm when the loop exits. To do this, we define a *loop invariant*, a predicate parameterized by the loop counter (variable i here), and prove that the loop invariant holds immediately before each iteration and immediately after the last iteration of the loop.

Looking ahead, the recursive call at the end of the general case suggests that $A[n]$ contains correct element for A to be in sorted order after it. In particular, $A[n]$ must be the largest element in A.

(a) First define an appropriate loop invariant $P(i)$ for the for-loop, then use induction to prove it holds immediately before and immediately after each iteration. *[Hint: Note that $i = n$ when the loop terminates, and we want $A[n]$ to be the maximum element in $A[1..n]$. Generalize this statement for each i-th iteration.]*

(b) With the loop invariant proven, proceed to prove BubbleSort correctly sorts its input array.

(c) Show that the total number of comparisons is at most n^2 for any input array of size n.

1. Consider the MERGESORT algorithm below, which takes an array $A[1..n]$ of **distinct** integers as input.

2 8 5 3 9 4 1 7

2853 9417

28 53 94 17

$\Rightarrow$ 28 35 49 17

2385 1479

1 23456789

```
MERGESORT($A[1..n]$):
  if $n = 2$ and $A[1] > A[2]$:
      swap $A[0] \leftrightarrow A[1]$
  else if $n > 2$:
      $m \leftarrow \lfloor n/2 \rfloor$
      MERGESORT($A[1..m]$)
      MERGESORT($A[m+1..n]$)
      MERGE($A$)
```

Assume that MERGE permutes any given array of n distinct integers where the first subarray of $\lfloor n/2 \rfloor$ integers is sorted in increasing order and the remainder of the array is sorted in increasing order so that the overall array is increasing order.

In lecture, we showed proved by induction that the algorithm above is correct: for any array of distinct integers, it terminates with the input array permuted so that its elements are in increasing order.

Now that we have proven the algorithm correct, we wish to prove its asymptotic runtime. Today, we'll focus on bounding the number of *comparisons* performed by the algorithm, meaning the number of times any two elements of the input array are compared to each other. Assume that MERGE performs at most n operations given an array of length n.

As a recursive algorithm, we can express the total number of operations as a recurrence relation, as follows. For simplicity, let us assume n is a power of two. Let $T(n)$ be the total number of comparisons for an input array of size n.

$n = 2^k$ ↷ # of split

"divide and conquer"

$$T(n) \le \begin{cases} 1 & \text{if } n \le 2 \\ \underbrace{2T(n/2)}_{\text{split recurse}} + \underbrace{n}_{\text{merge}} & \text{otherwise} \end{cases} \iff T(2^k) \le \begin{cases} 1 & , k \le 1 \\ 2T(2^{k-1}) + 2^k & , o/w \end{cases}$$

In CS 201, it is shown that the runtime is $O(n \log_2 n)$. Armed with induction, we are ready to prove this bound for the number of comparisons ourselves. Prove that $T(n) \le n \log_2 n$ for any integer $n \ge 1$.

Pf: Induction on k where $n = 2^k$

$P(k)$: $T(n) \le n \log_2 n$ for $n = 2^k$

$\iff T(2^k) \le 2^k \cdot \log_2 2^k = 2^k \cdot k$

Base: $k = 0$, $T(1) = 0 \le 0 = 1 \log_2 1$

$k = 1$, $T(2) \le 1 < 2 = 2 \log_2 2$ ✓

IH: $P(k)$ holds for some $k \ge 1$

IS: For $k \geq 1$, $2^{k+1} \geq 4 > 2$

$$
\begin{aligned}
T(2^{k+1}) &\leq 2T(2^{k}) + 2^{k+1} \\
&\leq 2 \cdot 2^{k} \cdot k + 2^{k+1} \\
&= 2^{k+1} \cdot k + 2^{k+1} \\
&= 2^{k+1} \cdot (k+1)
\end{aligned}
$$

$\Rightarrow P(k+1)$ holds.

#

First pass Second Third

7 6 4 3   6 4 3 7   4 3 6 7

6 7 4 3   4 6 3 7   3 4 6 7

6 4 7 3   4 3 6 7

6 4 3 7

3. Consider the recursive implementation of the BubbleSort algorithm below, which also is a sorting algorithm like the previous.

```
BUBBLESORT($A[1..n]$):
  if $n = 1$:
      return                    i: 1 → n-1
  for $i \leftarrow 1$; $i < n$; $i \leftarrow i + 1$:
      if $A[i] > A[i+1]$:
          swap $A[i] \leftrightarrow A[i+1]$
  return BUBBLESORT($A[1..n-1]$)
```

To prove this algorithm correct, we will rely on induction in a similar way as for the previous examples. However, as this implementation involves a non-trivial for-loop, we first need to reason about the state of the algorithm when the loop exits. To do this, we define a *loop invariant*, a predicate parameterized by the loop counter (variable i here), and prove that the loop invariant holds immediately before each iteration and immediately after the last iteration of the loop.

Looking ahead, the recursive call at the end of the general case suggests that $A[n]$ contains correct element for A to be in sorted order after it. In particular, $A[n]$ must be the largest element in A.

(a) First define an appropriate loop invariant $P(i)$ for the for-loop, then use induction to prove it holds immediately before and immediately after each iteration. *[Hint: Note that $i = n$ when the loop terminates, and we want $A[n]$ to be the maximum element in $A[1..n]$. Generalize this statement for each i-th iteration.]*

(b) With the loop invariant proven, proceed to prove BUBBLESORT correctly sorts its input array.

(c) Show that the total number of comparisons is at most n^2 for any input array of size n.

(a)

Pf: $P(i)$: Before ith iteration, $A[i]$ is the maximum element in the subarray $A[1 .. i]$. $(1 \le i \le n)$

Base: $i = 1$. $P(1)$ holds immediately.

IH: $P(i-1)$ holds

IS: $A[i-1]$ maximum in $A[1 .. i-1]$

For loop: compare $A[i]$. $A[i-1]$

swap if $A[i-1] > A[i]$

① $A[i-1] > A[i]$

$\Rightarrow$ $A[i-1]$ max in $A[1, ..., i]$

and swap to $A[i]$

$\Rightarrow$ $P(i)$ holds

② $A[i-1] \leq A[i]$

$\Rightarrow$ $A[i]$ max in $A[1 ..., i]$, no swap

$\Rightarrow$ $P(i)$ holds

Therefore, $P(i)$ holds for $1 \leq i \leq n$.

#

(b) Induction on the size of input array, n.

Pf: $P(n)$: BubbleSort correctly sorts for size n.

Base: $n=1$ trivial

IH: $P(n-1)$ holds

IS: By (a), $A[n]$ is max of $A[1, ..., n]$

$\Rightarrow$ BubbleSort$(A[1, ..., n-1])$

By IH, correctly sorts for $n-1$

$\Rightarrow$ $A[1], ...$ $A[n-1]$, $A[n]$ 排列 non-decreasing

$\Rightarrow$ correctly sorts for n (entire array sorted)

$\Rightarrow$ $P(n)$ holds.

#

(c) pf: ① $T(1) = 0$

② $n > 1$

$T(n) = T(n-1) + n - 1$

$$\Rightarrow T(n) = \begin{cases} 0 & , n = 1 \\ T(n-1) + n - 1 & , o/w \end{cases}$$

$P(n)$: $T(n) \le n^2$

Base: $T(1) = 0 \le 1 = 1^2$ ✓

IH: $P(n-1)$ holds ($T(n-1) \le (n-1)^2$)

IS:
$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ &\le (n-1)^2 + n - 1 \\ &= (n-1)(n-1+1) \\ &= (n-1) \cdot n < n^2 \end{aligned}$$

$\Rightarrow$ $P(n)$ holds

#