Review

CSE21 Winter 2017, Day 26 (B00), Day 18 (A00)

March 17, 2017

http://vlsicad.ucsd.edu/courses/cse21-w17
Announcements

Final Exam
A00: Wednesday 3/22 7pm
B00: Friday 3/24 11:30am

Two Final Exam Review Sessions
Saturday 3/18 1-3pm
Sunday 3/19 1-3pm
Locations TBD
Final Exam PPs posted
TTK has been started

CAPEs
Please and thank you!

OHs, 1-1s
Now is the time!
Topics

Correctness of iterative algorithms; Correctness of recursive algorithms

Order notation; time analysis of (iterative and recursive) algorithms

Graphs, trees, and DAGs; graph algorithms

Counting principles; encoding and decoding

Probability and applications
Textbook references

Searching and Sorting algorithms  Rosen 3.1, 5.5

Correctness of iterative algorithms; Correctness of recursive algorithms  Rosen 3.1, 5.5

Order notation; time analysis of (iterative and recursive) algorithms  Rosen 3.2, 3.3, 5.3, 5.4, 8.1, 8.3

Graphs, trees, and DAGs; graph algorithms  Rosen 10.1-10.5, 11.1-11.2

Counting principles; encoding and decoding  Rosen 6.1, 6.3-6.5, 8.5, 4.4

Probability and applications  Rosen 7.1-7.4
Correctness of iterative algorithms

Standard approach: **Loop invariants**

1. **State** the loop invariant.
   - Identify relationship between variables that remains true throughout algorithm.
   - Must imply correctness of algorithm after the algorithm terminates.
   - May need to be stronger statement than correctness.

2. **Prove** the loop invariant by induction on the number of times we have gone through the loop.
   - The induction variable is *not* the size of the input.

3. **Use** the loop invariant to prove correctness of the algorithm.
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

1. Identify relationship between variables that remains true throughout algorithm.
2. Prove the loop invariant
3. Use the loop invariant to prove correctness of the algorithm.
Example: Linear search

LS (a₁, …, aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

1. Identify relationship between variables that remains true throughout algorithm.

After t iterations, ________________________________

Try to fill in this blank.
Example: Linear search

LS (a₁, ..., aᵢ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

1. Identify relationship between variables that remains true throughout algorithm.

After t iterations, Found = true if and only if v is in a₁, ..., aᵢ
Example: Linear search

LS( \(a_1, \ldots, a_n, v\))
1. \(^{}\)Found := false
2. \(^{}\)for \(i := 1\) to \(n\)
3. \(^{}\)if \(a_i = v\) then \(\text{Found := true}\)
4. \(^{}\)return \(\text{Found}\).

2. Prove the loop invariant.

After \(t\) iterations, \(\text{Found = true if and only if } v\) is in \(a_1, \ldots, a_t\)

What's the induction variable?

A. \(n\)
B. \(i\)
C. \(t\)
D. None of the above.
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Base case: \( t = 0 \) then Found = true iff v is in \( a₁, ..., a₀ \)
so Found should be False
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Base case: For t = 0, the loop invariant is claiming that Found = true iff v is in the empty list. Since v is not in the empty list (since nothing is in the empty list), initializing Found to false in line 1 makes the invariant true.
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Induction step:

Let i ≥ 0: Assume Found = true if and only if v is in a₁, ..., aᵢ. We want to show that Found is true if and only if v is in a₁, ..., aᵢ₊₁.
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Induction step: let t be a nonnegative integer and assume that the loop invariant holds after t iterations (this is the IH). We WTS that v is in a₁, ..., aₜ₊₁ if and only if Found = true after the t+1st iteration. Consider two cases:

Case 1: v appears in a₁, ..., aₜ

Case 2: v doesn't appear in a₁, ..., aₜ
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Induction step: ... **Case 1: v appears in a₁, ..., aₜ**

Then by induction hypothesis, after t iterations we'll have set Found = true. Nowhere in the algorithm (after the initialization step) do we ever reset the value of Found to false so after t+1 iterations, the value of Found is true, as required. 😊
Example: Linear search

LS ( a₁, …, aₙ, v)
1. Found := false
2. for i := 1 to n
3.  if aᵢ = v then Found := true
4.  return Found.

2. Prove the loop invariant.

Induction step: … **Case 2: v does not appear in a₁, …, aₜ**

Then by induction hypothesis, after t iterations we'll still have Found = false.
Example: Linear search

LS \( (a_1, \ldots, a_n, v) \)
1. \( \text{Found} := \text{false} \)
2. \( \text{for } i := 1 \text{ to } n \)
3.  \( \text{if } a_i = v \text{ then } \text{Found} := \text{true} \)
4. \( \text{return } \text{Found} \).

2. Prove the loop invariant.

Induction step: \( \ldots \text{Case 2: } v \text{ does not appear in } a_1, \ldots, a_t \)

Then by induction hypothesis, after \( t \) iterations we'll still have \( \text{Found} = \text{false} \).

Case 2a: \( a_{t+1} = v \)
\( \text{Found} = \text{true} \)

Case 2b: \( a_{t+1} \neq v \)
\( \text{Do nothing} \).
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Induction step: … Case 2: v does not appear in a₁, ..., aₜ

Then by induction hypothesis, after t iterations we'll still have Found = false.

Case 2a: aₜ₊₁ = v
Case 2b: aₜ₊₁ ≠ v

In t+1st iteration, we'll set Found:= true, as required. 😊
Example: Linear search

LS (a₁, …, aₙ, v)

1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

2. Prove the loop invariant.

Induction step: … Case 2: v does not appear in a₁, …, aₜ

Then by induction hypothesis, after t iterations we'll still have Found = false.

Case 2a: aₜ₊₁ = v

In t+1st iteration, we'll set Found:= true, as required. 😊

Case 2b: aₜ₊₁ ≠ v

In t+1st iteration, don't change value of Found, so still (IH) false, as required. 😊
Example: Linear search

LS (a₁, ..., aₙ, v)
1. Found := false
2. for i := 1 to n
3. if aᵢ = v then Found := true
4. return Found.

3. Use the loop invariant to prove correctness of the algorithm.

We have shown by induction that for all t≥0,

After t iterations, Found = true if and only if v is in a₁, ..., aₜ.

Since the for loop iterates n times, in particular, when t=n, we have shown that

After n iterations, Found = true if and only if v is in a₁, ..., aₙ.

This is exactly what it means for the Linear Search algorithm to be correct.
Correctness of recursive algorithms

Standard approach: **(Strong) induction on input size**

1. Carefully state what it means for program to be correct.
   - What problem is the algorithm trying to solve?

2. State the statement being proved by induction
   For every input x of size n, Alg(x) "is correct."

3. Proof by induction.
   * Base case(s): state what algorithm outputs. Show this is the correct output.
   * Induction step: For some n, state the (strong) induction hypothesis.
     New goal: for any input x of size n, Alg(x) is correct.
     Express Alg(x) in terms of recursive calls, Alg(y), for y smaller than x.
     Use induction hypothesis.
     Combine to prove that the output for x is correct.
Example: Linear search

RLS (a₁, ..., aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, ..., aₙ₋₁, v)

What kind of induction will we need here?

A. Regular induction
B. Strong induction
Example: Linear search

RLS ( a₁, ..., aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, ..., aₙ₋₁, v)

Standard approach: **(Strong) induction on input size**

1. Carefully state what it means for program to be correct.
   
   \[ RLS(a₁, ..., aₙ, v) \text{ is true iff } v \text{ is in } a₁, ..., aₙ \]

2. State the statement being proved by induction

   **For every input x of size n, Alg(x) "is correct."**

3. Proof by induction.
Example: Linear search

RLS (a₁, ..., aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, ..., aₙ₋₁, v)

Standard approach: (Strong) induction on input size

1. Carefully state what it means for program to be correct.

RLS(a₁, ..., aₙ, v) = True if and only if v is an element in list A.
Example: Linear search

RLS (a_1, ..., a_n, v)
1. If v = a_n then return True
2. If n = 1 then return False
3. return RLS(a_1, ..., a_{n-1}, v)

Standard approach: 
(Strong) induction on input size

2. State statement being proved by induction

Inductive hypothesis:
For every list A of size n and every target v, 
RLS(a_1, ..., a_n, v) = True if and only if v is an element in list A.
Example: Linear search

\( \text{RLS} (\ a_1, \ldots, a_n, \ v) \)
1. If \( v = a_n \) then return True
2. If \( n = 1 \) then return False
3. return \( \text{RLS}(a_1, \ldots, a_{n-1}, \ v) \)

Standard approach: \( \text{(Strong) induction on input size} \)

3. Proof by induction \textbf{on input list size, } n.
Example: Linear search

RLS (a₁, …, aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, …, aₙ₋₁, v)

Standard approach: **(Strong) induction on input size**

3. Proof by induction **on input list size, n.**
   
   **Base case:** (n = 1)
   
   Then a₁ is the only element and RLS returns true iff v = a₁.

What are the base case(s) to consider?

A. n = 1
B. v = aₙ
C. v = a₁
D. More than one of the above.
E. None of the above.
Example: Linear search

RLS (a₁, ..., aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, ..., aₙ₋₁, v)

Standard approach: (Strong) induction on input size

3. Proof by induction on input list size, n.

Base case (n=1). Then A has a single element, a₁.

Goal: RLS(a₁, v) = True if and only if v is an element in list A.

Case 1: a₁ = v
Case 2: a₁ ≠ v
Example: Linear search

RLS (a₁, ..., aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, ..., aₙ₋₁, v)

Standard approach: (Strong) induction on input size

3. Proof by induction on input list size, n.

Base case (n=1). Then A has a single element, a₁.
Goal: RLS(a₁, v) = True if and only if v is an element in list A.
Case 1: a₁ = v  
Case 2: a₁ != v
Since v = a₁ = aₙ, return true in line 1. 😊
Example: Linear search

RLS (a₁, ..., aₙ, v)
1. If v = aₙ then return True
2. If n = 1 then return False
3. return RLS(a₁, ..., aₙ₋₁, v)

Standard approach: (Strong) induction on input size

3. Proof by induction on input list size, n.

Base case (n=1). Then A has a single element, a₁.

Goal: RLS(a₁, v) = True if and only if v is an element in list A.

Case 1: a₁ = v
Since v = a₁ = aₙ, return true in line 1. ☺

Case 2: a₁ != v
Since v != a₁ = aₙ, but n=1, return false in line 2. 😐
Example: Linear search

RLS (a_1, ..., a_n, v)
1. If v = a_n then return True
2. If n = 1 then return False
3. return RLS(a_1, ..., a_{n-1}, v)

Standard approach: (Strong) induction on input size

3. Proof by induction on input list size, n.

Induction step: let n be a nonnegative int, and assume for each list A of size n-1, RLS(a_1, ..., a_{n-1}, v) = True if and only if v is an element in list a_1, ..., a_{n-1}
From pseudocode, we see RLS(a_1, ..., a_n, v) depends on whether v = a_n.
Case 1: v = a_n
   \Rightarrow v is in a_1 ... a_n
   RLS returns True.
Case 2: v != a_n
Example: Linear search

RLS (a_1, ..., a_n, v)
1. If v = a_n then return True
2. If n = 1 then return False
3. return RLS(a_1, ..., a_{n-1}, v)

Standard approach: (Strong) induction on input size

3. Proof by induction on input list size, n.

Induction step: let n be a nonnegative int, and assume for each list A of size n-1, RLS(a_1, ..., a_{n-1}, v) = True if and only if v is an element in list a_1, ..., a_{n-1}

From pseudocode, we see RLS(a_1, ..., a_n, v) depends on whether v = a_n.

**Case 1: v = a_n**
Return true in line 1. 😊

**Case 2: v != a_n**
Don't return in lines 1,2. In line 3 return (by IH) true iff v is in a_1, ..., a_{n-1} 😊
Asymptotic analysis

Big O

For functions \( f(n) \), \( g(n) \) from the non-negative integers to the real numbers,

\[
f(n) \in O(g(n))
\]

means there are constants, \( C \) and \( k \) such that  

\[|f(n)| \leq C|g(n)| \quad \text{for all } n > k.
\]

What about big \( \Omega \)? big \( \Theta \)?
True or false, with justification:

\[ 2^{2\log n} \in O(2^{\log n}) \]

\[ n \in \Omega(n/(\log n)) \]

\[ ((n + 1)^2 + 1)^3 \in \Theta(n^6) \]

\[ \checkmark \text{true.} \]
Example: Multiplication

Multiply ( \( x = x_{m-1}…x_0 \) an m-bit integer, \( y = y_{n-1}…y_0 \) an n-bit integer)

1. If \( n = 1 \) and \( y_0 = 0 \) then return 0.
2. If \( n = 1 \) and \( y_0 = 1 \) then return \( x \).
3. \( \text{product} := \text{Multiply}(x, y_{n-1} \cdots y_1) \).
4. \( \text{product} := \text{Add}(\text{product}, \text{product}) \).
5. If \( y_0 = 1 \) then \( \text{product} := \text{Add}(\text{product}, x) \).

What's the input size?
A. \( m \)
B. \( n \)
C. \( m+n \)
D. \( mn \)
E. None of the above.
Example: Multiplication

Multiply ( \( x = x_{m-1} \ldots x_0 \) an m-bit integer, \( y = y_{n-1} \ldots y_0 \) an n-bit integer)
1. If \( n = 1 \) and \( y_0 = 0 \) then return 0.
2. If \( n = 1 \) and \( y_0 = 1 \) then return \( x \).
3. product := Multiply(\( x, y_{n-1} \ldots y_1 \)).
5. If \( y_0 = 1 \) then product := Add(product, \( x \)).

**How fast is this algorithm?**

** Assume we have access to algorithm for adding integers, and assume it takes time linear in N. **
Example: Multiplication

Multiply \((x = x_{m-1} \ldots x_0 \text{ an } m\text{-bit integer}, y = y_{n-1} \ldots y_0 \text{ an } n\text{-bit integer})\)

1. If \(n = 1\) and \(y_0 = 0\) then return 0.
2. If \(n = 1\) and \(y_0 = 1\) then return \(x\).
3. \(\text{product} := \text{Multiply}(x, y_{n-1} \ldots y_1)\).
4. \(\text{product} := \text{Add}(\text{product}, \text{product})\).
5. If \(y_0 = 1\) then \(\text{product} := \text{Add}(\text{product}, x)\).

How fast is this algorithm? Need recurrence.

Base case of recurrence is for smallest value of \(N\).

What's the smallest possible value of \(N\)?

A. 0
B. 1
C. 2
D. 3
E. None of the above.
Example: Multiplication

Multiply (\(x = x_{m-1} \ldots x_0\) an \(m\)-bit integer, \(y = y_{n-1} \ldots y_0\) an \(n\)-bit integer)
1. If \(n = 1\) and \(y_0 = 0\) then return 0.
2. If \(n = 1\) and \(y_0 = 1\) then return \(x\).
3. \(\text{product} := \text{Multiply}(x, y_{n-1} \ldots y_1)\). \(N = m+n\)
4. \(\text{product} := \text{Add}(\text{product}, \text{product})\).
5. If \(y_0 = 1\) then \(\text{product} := \text{Add}(\text{product}, x)\).

Base case of recurrence is for smallest value of \(N = 2\).
In this case, \(m=n=1\) so algorithm returns in either line 1 or line 2.

If \(T(N)\) is running time of algorithm for input of size \(n\), then
\[T(2) = c\]
where \(c\) is a constant.
Example: Multiplication

Multiply ( $x = x_{m-1}...x_0$ an m-bit integer, $y = y_{n-1}...y_0$ an n-bit integer)

1. If $n = 1$ and $y_0 = 0$ then return 0.
2. If $n = 1$ and $y_0 = 1$ then return $x$.
3. product := Multiply($x, y_{n-1} ... y_1$).
5. If $y_0 = 1$ then product := Add(product, $x$).

General case of the recurrence:

Lines 1, 2: constant time
Line 3: takes time $T(m+n-1) = T(N-1)$
Line 4, 5: linear time in $N$ via Add subroutine

$T(N) = T(N-1) + c'N$

for $N \geq 3$, where $c'$ is a constant
Example: Multiplication

Now solving recurrence:

Method 1: Unravel

Method 2: *Guess* (formula) and *Check* (with induction)
Example: Multiplication

Now solving recurrence:

Method 1: Unravel

\[ T(N) = T(N-1) + c'N \]
\[ = T(N-2) + c'(N-1) + c'N \]
\[ = T(N-3) + c'(N-2) + c'(N-1) + c'N \]
\[ = \ldots \]
\[ = T(N-k) + c'(N-k+1) + \ldots + c'(N-1) + c'N \]

What should we plug in for \( k \)?

A. \( N-2 \)
B. \( N+2 \)
C. \( N \)
D. \( 2 \)
E. None of the above.

\[ T(2) = c \]
Example: Multiplication

Now solving recurrence:

Method 1: Unravel

\[ T(N) = T(N-1) + c'N \]
\[ = T(N-2) + c'(N-1) + c'N \]
\[ = T(N-3) + c'(N-2) + c'(N-1) + c'N \]
\[ = \ldots \]
\[ = T(N-k) + c'(N-k+1) + \ldots + c'(N-1) + c'N \]
\[ = T(2) + c'(3) + \ldots + c'(N-1) + c'N \]
\[ = c + c'(3) + \ldots + c'(N-1) + c'N \]

\[ \Theta(N^2) \]
Graphs

To define a graph, must answer

**What are vertices?**

**What are edges?**

"connect vertex i to vertex j iff…"

Special classes of graphs:

**DAGs**  
directed acyclic graphs  
(impossible to find path from any vertex back to itself)
To define a graph, must answer

**What are vertices?**

**What are edges?**

"connect vertex \( i \) to vertex \( j \) iff…"

Special classes of graphs:

**Rooted trees**

directed acyclic graph, every vertex \( v \) assigned some height \( h(v) \), special vertex called the root [height 0, no incoming edges], all other vertices have exactly one incoming edge.
To define a graph, must answer

**What are vertices?**

**What are edges?**
"connect vertex i to vertex j iff…"

Special classes of graphs:

**Unrooted trees**
undirected graph, connected, acyclic

\[ \text{how many edges?} \]
\[ |V| - 1 \]
Fleury's algorithm: To find an Eulerian tour, don't burn your bridges.

Topological ordering algorithm: To find a "good" ordering, start with sources.

Rooting a tree: Convert unrooted tree into a rooted tree by directing its edges.

Graph search: Which other vertices can be reached from a given vertex in a graph?
Counting techniques

**Product rule:** When number of choices have doesn't depend on previous decisions, multiply number of choices together.

**Sum rule:** If cases have no overlap, count each case separately and add them up.

**Inclusion-Exclusion:** If cases do have overlap, adjust count:

\[ |A \cup B| = |A| + |B| - |A \cap B| \]

\[ |A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| \]

**Categories:** If two objects are being counted as "the same,"

\[ \# \text{ categories} = \frac{\# \text{ objects}}{\text{size of each category}} \]
(a) How many rearrangements are there of the letters in MISSISSIPPI?

11 total

4 I's 4 S's 2 P's

\[ \frac{11!}{(4!)(4!)(2!)} \]

(b) How many of the rearrangements in (a) are palindromes?

5!

\[ \frac{5!}{(2!)(2!)} \]

(c) How many 3 letter words can be made from the letters of MISSISSIPPI if all the letters must be distinct?

\[ \frac{(4)(3)(2)}{1P, 2S', 2I's} \}

(d) How many 3 letter words can be made from the alphabet \{M,I,S,P\}, with no restrictions?

\[ (4)(4)(4) \]
A random walk starts at the origin and can go either right or left along the x axis. At each step it can go 1, 2, 3, or 4 units in either the right or left direction.

How many walks of n steps are possible?

A. 4!
B. $8^n$
C. 2n
D. $n^4$
E. None of the above.
Example: Encoding / decoding

A random walk starts at the origin and can go either right or left along the x axis. At each step it can go 1, 2, 3, or 4 units in either the right or left direction.

How many walks of n steps are possible?
A. $4!$
B. $8^n$
C. $2n$
D. $n^4$
E. None of the above.

How many bits to represent each such walk?

$$\log (8^n) = 3n$$
A random walk starts at the origin and can go either right or left along the x axis. At each step it can go 1, 2, 3, or 4 units in either the right or left direction.

How many walks of n steps are possible?
A. $4!$  
B. $8^n$  
C. $2n$  
D. $n^4$  
E. None of the above.

How many bits to represent each such walk?

Encoding scheme?
**Probability**

A **Sample Space** is a set of elements called **Outcomes**.

A **probability distribution** is an assignment of probabilities (between 0 and 1) to each outcome of a sample space, so that the total probability is 1.

An **event** is a subset of the sample space, i.e. a collection of possible outcomes.

A **Random Variable** is a function from the sample space to the set of real numbers.

The **Expected Value** of a random variable $X$ is the “average value” of $X$.

$$E(X) = \sum_{s \in S} P(s)X(s) = \sum_r P(X = r)r$$
Probability

Conditional probability and Bayes' theorem

Random variables

Independence … of events … of random variables

Expected value or average value (and linearity of expectation)

Variance, a measure of concentration or spread
Example: Probability

Suppose 5-card hands are dealt at random from a standard deck of 52. What is the probability that your hand contains exactly two Aces?

\[
\binom{4}{2} \binom{48}{3} \quad \frac{\binom{52}{5}}{} 
\]
Example: Probability

A bitstring of length 4 is generated randomly one bit at a time.

So far, you can see that the first bit is a 1. What is the probability that the string will have at least two consecutive 0s?

\[
\frac{3}{8}
\]
A new employee at the coat check forgets to put numbers on people’s coats, so when people come back to claim their coats, he gives them back a coat chosen at random. What is the expected number of coats that are returned correctly?

\[ X = \sum_{i=1}^{n} X_i \]

\[ X_i = \begin{cases} 1 & \text{if person } i \text{ gets their coat back} \\ 0 & \text{otherwise} \end{cases} \]

\[ E(X_i) = P(X_i = 1) = \frac{1}{n} \]

\[ E(X) = \sum_{i=1}^{n} E(X_i) = n \left( \frac{1}{n} \right) = 1 \]
Example: Probability

In a board game, you attack another character by giving them damage equal to the difference of the numbers that appear when you roll two 4-sided dice. If damage can never be negative, what is the expected value and variance of the damage?

Recall: \[ V(X) = E \left( (X - E(X))^2 \right) \]
\[ = E(X^2) - E(X)^2 \]

\[
E(X) = 0.4 \cdot \frac{1}{16} + 1.6 \cdot \frac{1}{16} + 2.4 \cdot \frac{1}{16} + 3.2 \cdot \frac{1}{16} = \frac{20}{16} = \frac{5}{4}
\]

\[
E(X^2) = 0^2 \cdot \frac{1}{16} + 1^2 \cdot \frac{1}{16} + 2^2 \cdot \frac{1}{16} + 3^2 \cdot \frac{1}{16} = \frac{40}{16} = \frac{5}{2}
\]

\[
V(X) = \frac{5}{2} - \left( \frac{5}{4} \right)^2 = \frac{5}{2} - \frac{25}{16} = \frac{40}{16} - \frac{25}{16} = \frac{15}{16}
\]
Have a Successful Finals Week!

Final Exam
A00: Wednesday 3/22 7pm
B00: Friday 3/24 11:30am

Final Exam PPs and TTKs!
+ prepare your notes pages!

Two Final Exam Review Sessions
Saturday 3/18 1-3pm
Sunday 3/19 1-3pm
Locations TBD
Final Exam PPs posted
TTK has been started

CAPEs
Please and thank you!

OHs, 1-1s
Now is the time!