<table>
<thead>
<tr>
<th>Lecture</th>
<th>Instructor</th>
<th>Days</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture A</td>
<td>Tiefenbruck</td>
<td>Tu, Th</td>
<td>11am-12:20pm</td>
<td>Center 119</td>
</tr>
<tr>
<td>Lecture B</td>
<td>Tiefenbruck</td>
<td>Tu, Th</td>
<td>9:30am-10:50am</td>
<td>Center 119</td>
</tr>
<tr>
<td>Lecture C</td>
<td>Minnes</td>
<td>Tu, Th</td>
<td>3:30pm-4:50pm</td>
<td>WLH 2005</td>
</tr>
</tbody>
</table>

http://cseweb.ucsd.edu/classes/fa15/cse21-abc/

Dec 3, 2015
Topics

Searching and Sorting algorithms

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
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching and Sorting algorithms</td>
<td>Rosen 3.1, 5.5</td>
</tr>
<tr>
<td>Correctness of iterative algorithms; Correctness of recursive algorithms</td>
<td>Rosen 3.1, 5.5</td>
</tr>
<tr>
<td>Order notation; time analysis of (iterative and recursive) algorithms</td>
<td>Rosen 3.2, 3.3, 5.3, 5.4, 8.1, 8.3</td>
</tr>
<tr>
<td>Graphs, trees, and DAGs; graph algorithms</td>
<td>Rosen 10.1-10.5, 11.1-11.2</td>
</tr>
<tr>
<td>Counting principles; encoding and decoding</td>
<td>Rosen 6.1, 6.3-6.5, 8.5, 4.4</td>
</tr>
<tr>
<td>Probability and applications</td>
<td>Rosen 7.1-7.4</td>
</tr>
</tbody>
</table>
Sorting algorithms
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

\( \text{LS} ( a_1, \ldots, a_n, v) \)

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. \( \text{return } \text{Found.} \)
Example: Linear search

\[ \text{LS} (a_1, ..., a_n, v) \]
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. \( \text{return } \text{Found}. \)

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 (a_1, ..., a_n, v)

1. Found := false
2. for i := 1 to n
3. if a_i = 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 a_1, ..., a_t
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.

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

What's the induction variable?
A. n
B. i
C. t
D. None of the above.

\[ \text{goal: show this is true } \forall t \geq 0 \]
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:
Example: Linear search

\[ LS \left( a_1, \ldots, a_n, v \right) \]

1. \( \text{Found} := \text{false} \) \hspace{1cm} \text{after 0 iterations}
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.

Base case: For \( t = 0 \), the loop invariant is claiming that \( \text{Found} = \text{true} \) iff \( v \) is in the empty list. Since there are no elements in the empty list, what we are trying to show reduces to \( \text{Found} \neq \text{true} \). This is, in fact, the case since we initialize \( \text{Found} \) to \( \text{false} \) in line 1.
Example: Linear search

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

2. Prove the loop invariant.

*Induction step:*
Example: Linear search

\[ \text{LS} \left( a_1, \ldots, a_n, v \right) \]

1. \text{Found} := \text{false}
2. \text{for } i := 1 \text{ to } n
3. \quad \text{if } a_i = v \text{ then } \text{Found} := \text{true}
4. \text{return } \text{Found}.

2. Prove the loop invariant.

\text{Induction step: let } t \text{ be a nonnegative integer and assume that the loop invariant holds after } t \text{ iterations (this is the IH). We WTS that } v \text{ is in } a_1, \ldots, a_{t+1} \text{ iff and only if } \text{Found} = \text{true} \text{ after the next iteration. Consider two cases:}

\text{Case 1: } v \text{ appears in } a_1, \ldots, a_t \quad \text{Case 2: } v \text{ doesn't appear in } a_1, \ldots, a_t
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.

What do we want to prove next?

A. In this iteration, Found is set to true.
B. In this iteration, Found remains false.
C. In this iteration, Found gets the value aₜ₊₁
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.

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**
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+1\textsuperscript{st} iteration, we'll set Found := true,*

**Case 2b: aₜ₊₁ \neq v**

*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)
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: \textbf{(Strong) induction on input size}

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

2. State the statement being proved by induction
   \textbf{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, \ldots, a_n, v)
1. If v = a_n then return True
2. If n = 1 then return False
3. return RLS(a_1, \ldots, a_{n-1}, v)

Standard approach: (Strong) induction on input size

2. State statement being proved by induction

For every list A of size n and every target v,

\[
\text{RLS}(a_1, \ldots, a_n, v) = \text{True if and only if } v \text{ is an element in list } A.
\]

\text{Goal: show this } A \geq \text{ base case}
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.
Example: Linear search

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

Standard approach: \((Strong)\) induction on input size \(n\).

3. Proof by induction on input list size, \(n\).

What are the base case(s) to consider?

A. \(n = 1\)
B. \(v = a_n\)
C. \(v = a_1\)
D. More than one of the above.
E. None of the above.

\(regular\) induction \(\text{OK}\) for this example

\((Strong)\) induction on input 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 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( 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.

Induction step: let n be a nonnegative int, and assume for each list A of size n-1, RLS(a₁, …, aₙ₋₁, v) = True if and only if v is an element in list a₁, …, aₙ₋₁
From pseudocode, we see RLS(a₁, …, aₙ, v) depends on whether v = aₙ.
Case 1: v = aₙ
Case 2: v != aₙ
Example: Linear search

\[ \text{RLS} \left( a_1, \ldots, a_n, v \right) \]
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 on input list size, \( n \).

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

Case 1: \( v = a_n \)
Return true in line 1. 😊

Case 2: \( v \neq a_n \)
Don't return in lines 1,2. In line 3 return (by IH)
true iff \( v \) is in \( a_1, \ldots, a_{n-1} \) 😊
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 \( \Theta \)?

- tools
- use def'n
- limits

as \( n \) gets large
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. \(product := Multiply(x, y_{n-1} \ldots y_1)\). (recursive call)
4. \(product := Add(product, product)\).
5. If \(y_n = 1\) then \(product := Add(product, x)\).

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

Multiply ( \( x = x_{m-1}...x_0 \) 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. \( \text{product} := \text{Multiply}(x, y_{n-1} \ldots y_1) \).
4. \( \text{product} := \text{Add}(	ext{product}, \text{product}) \).
5. If \( y_n = 1 \) then \( \text{product} := \text{Add}(	ext{product}, x) \).

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 \( x = x_{m-1} \ldots x_0 \) 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 \( y_n = 1 \) then product := Add(product, \( x \)).

How fast is this algorithm? Need recurrence.

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

\[ N = m + n \]

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. \( \text{product} := \text{Multiply}(x, y_{n-1} \ldots y_1) \).
4. \( \text{product} := \text{Add}((\text{product}, \text{product})) \).
5. If \( y_n = 1 \) 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

\[
T(2) = c \quad \leq \quad \text{generic constant}
\]
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. \text{product} := \text{Multiply}(x, y_{n-1} \ldots y_1).
4. \text{product} := \text{Add}(\text{product}, \text{product}).
5. If y_n = 1 then \text{product} := \text{Add}(\text{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 \text{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

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

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

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

\[ T(2) = c \]

Set \( N-K = 2 \)
\[ K = N-2 \]
Now solving recurrence:

Method 1: Unravel

\[
T(N) = T(N-1) + c'N
\]

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

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

\[
= \ldots
\]

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

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

\[
= c + c'(3) + \ldots + c'(N-1) + c'N
\]

\[
\Theta(N^2)
\]
Example: Multiplication

Now solving recurrence:

Method 1: Unravel

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

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 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
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 vertex back to itself)
Some Graph Algorithms

**Fleury's algorithm:** To find an Eulerian tour, don't burn your bridges.

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

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

**Graph search:** Can any vertex in the graph be reached by any others?
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,"

\[
\# \text{ categories} = \frac{\# \text{ objects}}{\text{size of each category}}
\]
Example: Counting

(a) How many rearrangements are there of the letters in MISSISSIPPI?

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

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

\[ \frac{S!}{2!2!} \]

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

\[ \binom{4}{3} \cdot 3! \]

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

\[ 4 \cdot 4 \cdot 4 = 4 \]
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?

$$\log_2 (8^n) = 3n$$
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?

For each step, use 3 bits.

$0 = L$
$1 = R$
$00 = 4$
$01 = 1$
$10 = 2$
$11 = 3$
A probability distribution is an assignment of probabilities (between 0 and 1) to each element of a sample space $S$, so that the total probability is 1.

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

Conditional probability and Bayes' rule

Random variables

Independence … of events … of random variables

Expectation & Variance
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?

\[ p = \frac{\# \text{ hands with 2 A's}}{\# \text{ hands}} = \frac{\binom{4}{2} \cdot \binom{48}{3}}{\binom{52}{5}} \]
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?

\[
P(\text{have 00 | first bit is 1}) = \frac{P(\text{have 00 AND first bit is 1})}{P(\text{first bit is 1})}
\]

\[
= \frac{3/16}{1/2} = \frac{3}{8}
\]
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?

\[
X = \# \text{ coats returned correctly}
\]

\[
\text{goal: } E[X] \quad \text{for } i = 1 \text{ to } n
\]

\[
\begin{align*}
X &= X_1 + X_2 + \ldots + X_n = \\
&= \sum_{i=1}^{n} X_i.
\end{align*}
\]

by linearity:

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

What is the variance of the number of coins that turn up heads in 3 coin flips?

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

\[
X = \text{\# Heads in 3 flips}
\]

\[
V(X) = E(X^2) - E(X)^2
\]

\[
E(X) = 1.5
\]

\[
E(X^2) = 3 - (1.5)^2 = 3/4
\]

<table>
<thead>
<tr>
<th>X</th>
<th>P(X)</th>
<th>( (X - 3/2)^2 )</th>
<th>X^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/8</td>
<td>9/4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3/8</td>
<td>1/4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3/8</td>
<td>1/4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1/8</td>
<td>9/4</td>
<td>9</td>
</tr>
</tbody>
</table>

\[
E\left((X-3/2)^2\right) = \frac{9}{4} \cdot \frac{1}{8} + \frac{1}{4} \cdot \frac{3}{8} + \frac{1}{4} \cdot \frac{3}{8} + \frac{9}{4} \cdot \frac{1}{8} = 3/4
\]
Reminders

Final exam: Saturday, December 5, 11:30-2:30. That’s 2 days from today.

- Lecture B00 (9:30am, Tiefenbruck) - Final exam in Peterson 110.
- Lecture A00 (11am, Tiefenbruck) - Final exam in Peterson 108.
- Lecture C00 (3:30pm, Minnes) - Final exam in Warren Lecture Hall 2001.
- Seating charts on course website.