“Notes” slides from before lecture

CSE 21, Winter 2017, Section A00

Lecture 18 Notes

Class URL: http://vlsicad.ucsd.edu/courses/cse21-w17/
Notes March 15 (1)

• Today: Day 26 of posted slides = Review

• HW8 is now due on FRIDAY

• Final exam review sessions:
  – Saturday 1-3pm York 2722
  – Sunday 1-3pm Center 101

• FINAL EXAM WILL BE IN CENTER 101 **AND** CENTER 212 !!!
  – ONE piece of paper allowed
  – Look for seating chart that gives everyone plenty of room

• Please fill out CAPEs – thank you !!!

• Any logistic, other issues ?
Announcements

Final Exam
A00: Wednesday 3/22 7pm
CENTER 101 *AND* 212
B00: Friday 3/24 11:30am

HW 8
Due FRIDAY
11:59pm

CAPEs
Please and thank you!

Ohs, 1-1s
Now is the time!

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

Two Final Exam Review Sessions
Saturday 3/18 1-3pm York 2722
Sunday 3/19 1-3pm Center 101
Final Exam PPs posted
TTK has been started
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

* See TTK for Final Exam “anatomy”

No Bayes Theorem
No Week 10 (randomized —)
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page(s)</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

- Selection
- Insertion
- Bucket
- Bubble
- Merge sort
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

\( L_S (a_1, \ldots, a_n, v) \)

1. Found := false
2. for \( i := 1 \) to \( n \)
3. if \( a_i = v \) then Found := true
4. return Found.

Look for \( v \) in list \( a_1, \ldots, a_n \)

Step through the list

If ever see \( v \), then Found := true

\( v = 7 \)

\( A = 4, 6, 9, 7, 3, 13 \)

\( \text{Found} = F \)

\( i = 1 \) \( F \)
\( i = 2 \) \( F \)
\( i = 3 \) \( F \)
\( i = 4 \) \( T \)
\( i = 5 \) \( T \)
\( i = 6 \) \( T \)
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.

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, $\implies$ Found is true if $v$ is one of the first $t$ elements $a₁, \ldots, aₜ$

and Found is false if none of the first $t$ elements is $=$ $v$

Try to fill in this blank.
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. \quad \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.

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

2. Prove the loop invariant.

For any `t geq 0`,

After `t` iterations, `Found = true if and only if v is in a_1, ..., a_t`

What's the induction variable?

- A. `n`
- B. `i`
- C. `t`
- D. None of the above.

B.C.: `t = 0`
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.

2. Prove the loop invariant.

Base case:

\[ t = 0 \quad \checkmark \]

- After 0 iterations of loop \((\equiv \text{before entering loop})\),

\[ \text{Found} = \text{false} \]

- Indeed, \(v\) is not one of the first 0 elements of list \(A\)
Example: Linear search

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

2. Prove the loop invariant.

Base case: For \( t = 0 \), the loop invariant is claiming that \( \text{Found} = \text{true} \iff v \text{ is in the empty list} \). Since \( v \text{ is not in the empty list (since nothing is in the empty list)} \), initializing \( \text{Found to false in line 1} \) makes the invariant true.
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. \( \text{if} \ a_i = v \text{ then } \text{Found} := \text{true} \)
4. \( \text{return } \text{Found} \).

2. Prove the loop invariant.

\[ \checkmark \text{ ordinary induction } \]
\[ t \rightarrow t + 1 \]

Induction step:

\[ \begin{align*}
\bullet & \ \text{Found} = \text{true} \text{?} \\
\text{1. } & \ \text{Found} = \text{true} \text{?} \\
\text{2. } & \ \text{Found} = \text{false} \text{?} \\
\text{2A} & \ a_{t+1} = v \text{?} \\
\text{2B} & \ a_{t+1} \neq v \text{?}
\end{align*} \]
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

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

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.

\textit{Induction step: } \ldots \textbf{Case 1: } v \textit{ appears in } a_1, \ldots, a_t \ldots \]

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

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

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

2. Prove the loop invariant.

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

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

\[
\text{Found} := \text{true} \quad \text{if} \quad a_{t+1} = v
\]

\[
\Rightarrow \quad \text{Found remains false} \quad \text{if} \quad a_{t+1} \neq v
\]

What do we want to prove next?

A. \text{In this iteration,} \text{Found} \text{ is set to true.}
B. \text{In this iteration,} \text{Found remains false.}
C. \text{In this iteration,} \text{Found gets the value} \ a_{t+1}
D. \text{None of the above.}
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. \( \text{if } a_i = v \text{ then } \text{Found} := \text{true} \)
4. \( \text{return } \text{Found} \).

2. Prove the loop invariant.

\text{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 \( \text{Found} = \text{false} \).

\text{Case 2a: } a_{t+1} = v \quad \Rightarrow \quad \text{Found} = \text{true} \quad \text{Case 2b: } a_{t+1} \neq v \quad \text{fails test} \quad \Rightarrow \quad \text{Found stays false}
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.

2. Prove the loop invariant.

Induction step: … Case 2: \( v \) does not appear in \( a_1, ..., a_t \)

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

Case 2a: \( a_{t+1} = v \)
In \( t+1^{\text{st}} \) 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, ..., aᵣ

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? ("spec")

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

\(v = 7\) \hspace{1cm} A = 4, 6, 9, 7, 3, 13

\((n = 6) \quad (7 \neq 13) \Rightarrow \text{return \ RLS}(4, 6, 9, 7, 3; 7)\)
\((n = 5) \quad (7 \neq 3) \Rightarrow \text{return \ RLS}(4, 6, 9, 7; 7)\)

\(7 = 7 \Rightarrow \text{return True (Line 1)}\)

What kind of induction will we need here?

A. Regular induction
B. Strong induction

(for this example, suffices)
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:

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

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

3. Proof by induction.

("We use induction which may sometimes be strong induction.
This example: regular")
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.

~ “spec”

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

“For every list A of size n and every target 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

2. State statement being proved by induction

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

3. Proof by induction **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.

WTS: works for all \( n \geq 1 \)

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

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

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, \( n \).

\textbf{Base case} \( (n=1) \). Then A has a single element, \( a_1 \).

\textbf{Goal:} \( \text{RLS}(a_1, v) = \text{True} \) if and only if \( v \) is an element in list A.

\textbf{Case 1:} \( a_1 = v \) \hspace{1cm} \text{(line 1)}

\textbf{Case 2:} \( a_1 \neq v \) \hspace{1cm} \text{((line 1 and) 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.

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
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 (2 cases)
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
Return true in line 1. 😊 ✔
Case 2: v ≠ aₙ
Don't return in lines 1,2. In line 3 return (by IH) true iff v is in a₁, ..., aₙ₋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 \( \Theta \)?

\[
f \in \Omega(g) \iff g \in O(f)
\]
\[
f \in \Theta(g) \iff f \in O(g) \quad \text{and} \quad g \in O(f)
\]
Asymptotic analysis

True or false, with justification:

- $2^{2\log n} \in O(2^{\log n})$ (False)
- $n \in \Omega(n/(\log n))$ (True)
- $((n + 1)^2 + 1)^3 \in \Theta(n^6)$ (True)

\[ \lim_{n \to \infty} \frac{n}{n/\log n} = \infty \]
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 \( y_0 = 1 \) 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}...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).

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}…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).

How fast is this algorithm? Need recurrence.

Base case of recurrence is for smallest value of N.

<table>
<thead>
<tr>
<th>N = m+n</th>
<th>2^1</th>
<th>2^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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}...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).

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
where c is a constant.
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 := \text{Multiply}(x, y_{n-1} \ldots y_1).
4. product := \text{Add}(\text{product}, \text{product}).
5. If \( y_0 = 1 \) then 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

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.

\[
k = N - 2
N - (N - 2) = 2 = B.C.
\]
Example: Multiplication

Now solving recurrence:

Method 1: Unravel

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

\( T(2) = c \)

\( \Theta(N^2) \)

\( c' \cdot \left(3 + 4 + 5 + \ldots + \frac{(N-1)}{N}\right) \)
To define a graph, must answer

**What are vertices?**

set \( V \)

**What are edges?**

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

Special classes of graphs:

**DAGs**

directed acyclic graphs
( impossible to find directed path from any vertex back to itself )
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 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?**

"connect vertex i to vertex j iff…"

**What are edges?**

Special classes of graphs:

**Unrooted trees**

undirected graph, connected, acyclic
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.

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?

BFS, DFS
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?

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

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

(d) How many cycles of 3 letters can be made from the letters of MISSISSIPPI if all the letters must be distinct?

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

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?

\[
\frac{\binom{4}{2} \cdot \binom{48}{3}}{\binom{52}{5}}
\]
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?

\[ \frac{3}{8} \]
Example: Probability

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?

\[ X = \sum_{i=1}^{100} X_i \]

\[ E(X) = 100 \cdot \frac{1}{100} = 1 \]
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: \[ V(X) = E \left( (X - E(X))^2 \right) \]
\[ = E(X^2) - E(X)^2 \]
Have A Successful Finals Week!

Final Exam
A00: Wednesday 3/22 7pm CENTER 101 *AND* 212
B00: Friday 3/24 11:30am

HW 8
Due FRIDAY 11:59pm

OHs, 1-1s
Now is the time!

CAPEs
Please and thank you!

Two Final Exam Review Sessions
Saturday 3/18 1-3pm York 2722
Sunday 3/19 1-3pm Center 101
Final Exam PPs posted
TTK has been started

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