Before We Start...
Midterm Overview

- 6 hours, multiple sittings
- Open book, open notes, open lecture slides
- No collaboration

Possible Topics:
- Basically, everything you’ve seen on homework assignments to this point
- Relational model
  - relations, keys, relational algebra operations (queries, modifications)
- SQL DDL commands
  - `CREATE TABLE`, `CREATE VIEW`, integrity constraints, etc.
  - Altering existing database schemas
  - Indexes
Midterm Overview (2)

- Possible Topics (cont):
  - SQL DML commands
    - SELECT, INSERT, UPDATE, DELETE
    - Grouping and aggregation, subqueries, etc.
    - Aggregates of aggregates 😊
    - Translation to relational algebra, performance considerations, etc.
  - Procedural SQL
    - User-defined functions (UDFs)
    - Stored procedures
    - Triggers
    - Cursors
Midterm Overview (2)

- You can use a MySQL database, if you want to
  - e.g. make sure your DDL syntax is correct, check schema-alteration steps, verify that UDFs work

- **WARNING**: Don’t let it become a time-sink!
  - I won’t necessarily give you actual data for problems
  - Don’t waste time making up data just to test your SQL
Assignments and Solution Sets

- Some assignments may not be graded in time for the midterm (e.g. HW3, HW4)
- HW1-HW4 solution sets will be on Moodle by the time of the midterm
Be familiar with the relational model:
- What’s a relation? What’s a relation schema? What’s a tuple? etc.

Remember, relations are different from SQL tables in a very important way:
- Relations are sets of tuples. SQL tables are multisets of tuples.
Keys in the Relational Model

- **Be familiar with the different kinds of keys**
  - Keys uniquely identify tuples within a relation

- **Superkey**
  - Any set of attributes that uniquely identifies a tuple
  - If a set of attributes $K$ is a superkey, then so is any superset of $K$

- **Candidate key**
  - A *minimal* superkey
  - If any attribute is removed, no longer a superkey

- **Primary key**
  - A particular candidate key, chosen as the primary means of referring to tuples
Keys and Constraints

- Keys constrain the set of tuples that can appear in a relation
  - In a relation $r$ with a candidate key $K$, no two tuples can have the same values for $K$

- Can also have foreign keys
  - One relation contains the key attributes of another relation
  - Referencing relation has a foreign key
  - Referenced relation has a primary (or candidate) key
  - Referencing relation can only contain values of foreign key that also appear in referenced relation
  - Called referential integrity
Foreighn Key Example

- Bank example:
  
  `account(account_number, branch_name, balance)`
  
  `depositor(customer_name, account_number)`

- `depositor` is the referencing relation
  - `account_number` is a foreign-key to `account`

- `account` is the referenced relation
A Note on Notation

- Depositor relation:
  - `depositor(customer_name, account_number)`

- In the relational model:
  - Every `(customer_name, account_number)` pair in `depositor` is unique

- When translating to SQL:
  - `depositor` table could be a multiset...
  - Need to ensure that SQL table is actually a set, not a multiset
  - `PRIMARY KEY (customer_name, account_number)` after all columns are declared
In the relational model, you must pay attention to referential integrity constraints.

Make sure to perform modifications in an order that maintains referential integrity.

Example: Remove customer “Jones” from bank.

- Customer name appears in `customer`, `depositor`, and `borrower` relations.
- Which relations reference which?
  - `depositor` references `customer`
  - `borrower` references `customer`

Remove Jones records from `depositor` and `borrower` first.
Then remove Jones records from `customer`.
Relational Algebra Operations

- Six fundamental operations:
  - $\sigma$: select operation
  - $\Pi$: project operation
  - $\cup$: set-union operation
  - $-$: set-difference operation
  - $\times$: Cartesian product operation
  - $\rho$: rename operation

- Operations take one or two relations as input
- Each produces another relation as output
Additional Relational Operations

- Several additional operations, defined in terms of fundamental operations:
  - $\cap$: set-intersection
  - $\bowtie$: natural join (also theta-join $\bowtie_\theta$)
  - $\div$: division
  - $\gets$: assignment

- Extended relational operations:
  - $\Pi$: generalized project operation
  - $G$: grouping and aggregation
  - $\bowtie\bowtie\bowtie\bowtie$: left outer join, right outer join, full outer join
Join Operations

- Be familiar with different join operations in relational algebra
- Cartesian product $r \times s$ generates every possible pair of rows from $r$ and $s$
- Summary of other join operations:

\[
\begin{array}{c|c|c}
\text{attr1} & \text{attr2} & \text{attr3} \\
\hline
a & r1 & \text{null} \\
b & r2 & s2 \\
c & r3 & s3 \\
d & \text{null} & s4
\end{array}
\]
Rename Operation

- Mainly used when joining a relation to itself
  - Need to rename one instance of the relation to avoid ambiguities
- Remember you can specify names with both $\Pi$ and $G$
  - Can rename attributes
  - Can assign a name to computed results
  - Naming computed results in $\Pi$ or $G$ is shorter than including an extra $\rho$ operation
- Use $\rho$ when you are only renaming things
  - Don’t use $\Pi$ or $G$ just to rename something
  - Also, $\rho$ doesn’t create a new relation-variable! Assignment $\leftarrow$ does this.
Examples

- Schema for an auto insurance database:
  \[ \text{car(license, vin, make, model, year)} \]
  - \text{vin} is also a candidate key, but not the primary key
  \[ \text{customer(driver_id, name, street, city)} \]
  \[ \text{owner(license, driver_id)} \]
  \[ \text{claim(driver_id, license, date, description, amount)} \]

- Find names of all customers living in Los Angeles or New York.
  \[ \Pi_{name}(\sigma_{\text{city="Los Angeles" } \lor \text{city="New York"}}(\text{customer})) \]
  - Select predicate can refer to attributes, constants, or arithmetic expressions using attributes
  - Conditions combined with \( \land \) and \( \lor \)
Examples (2)

- **Schema:**
  
  car(license, vin, make, model, year)
  customer(driver_id, name, street, city)
  owner(license, driver_id)
  claim(driver_id, license, date, description, amount)

- **Find customer name, street, and city of all Toyota owners**
  
  - Need to join customer, owner, car relations
  - Could use Cartesian product, select, etc.
  - Or, use natural join operation:
    \[ \Pi_{\text{name, street, city}}(\sigma_{\text{make}=\text{“Toyota”}}(\text{customer} \bowtie \text{owner} \bowtie \text{car})) \]
Examples (3)

» Schema:

- `car(license, vin, make, model, year)`
- `customer(driver_id, name, street, city)`
- `owner(license, driver_id)`
- `claim(driver_id, license, date, description, amount)`

» Find how many claims each customer has
  - Don’t include customers with no claims…
  - Simple grouping and aggregation operation
    `driver_id G count(license) as num_claims(claim)`
    - The specific attribute that is counted is irrelevant here…
  - Aggregate operations work on multisets by default
  - Schema of result?
    `(driver_id, num_claims)`
Examples (4)

- Now, include customers with no claims
  - They should have 0 in their values
  - Requires outer join between customer, claim
  - “Outer” part of join symbol is towards relation whose rows should be null-padded
  - Want all customers, and claim records if they are there, so “outer” part is towards customer

\[
driver_id \ G\text{count}(\text{license}) \text{ as } num\_claims(customer \bowtie \text{claim})
\]

- Aggregate functions ignore null values
Selecting on Aggregate Values

- Grouping/aggregation op produces a relation, not an individual scalar value
  
  **You cannot use aggregate functions in select predicates!!!**

- To select rows based on an aggregate value:
  - Create a grouping/aggregation query to generate the aggregate results
    - This is a relation, so…
  - Use Cartesian product (or another appropriate join operation) to combine rows with the relation containing aggregated results
  - Select out the rows that satisfy the desired constraints
Selecting on Aggregate Values (2)

- General form of grouping/aggregation:
  \[ G_{1}, G_{2}, \ldots G_{F(A_1), F(A_2), \ldots}(\ldots) \]

- Results of aggregate functions are unnamed!

- This query is **wrong**:
  \[ \sigma_{F(A_1)} = \ldots(G_{1}, G_{2}, \ldots G_{F(A_1), F(A_2), \ldots}(\ldots)) \]
  
  Attribute in result does **not** have name \( F(A_1) \)!

- **Must assign** a name to the aggregate result
  \[ G_{1}, G_{2}, \ldots G_{F(A_1)} \text{ as } V_{1}, F(A_2) \text{ as } V_{2}, \ldots(\ldots) \]

- Then, can properly select against the result:
  \[ \sigma_{V_1} = \ldots(G_{1}, G_{2}, \ldots G_{F(A_1)} \text{ as } V_{1}, F(A_2) \text{ as } V_{2}, \ldots(\ldots)) \]
An Aggregate Example

- **Schema:**
  - car(license, vin, make, model, year)
  - customer(driver_id, name, street, city)
  - owner(license, driver_id)
  - claim(driver_id, license, date, description, amount)

- **Find the claim(s) with the largest amount**
  - Claims are identified by (driver_id, license, date), so just return all attributes of the claim
  - Use aggregation to find the maximum claim amount:
    \[ G_{\text{max(amount)}} \text{ as max_amt(claim)} \]
  - This generates a relation! Use Cartesian product to select the row(s) with this value.
    \[ \Pi_{\text{driver_id,license,date,description,amount}} (\sigma_{\text{amount=\text{max_amt}}} (\text{claim} \times G_{\text{max(amount)}} \text{ as max_amt(claim)})) \]
Another Aggregate Example

- **Schema:**
  - `car(license, vin, make, model, year)`
  - `customer(driver_id, name, street, city)`
  - `owner(license, driver_id)`
  - `claim(driver_id, license, date, description, amount)`

- Find the customer with the most insurance claims, along with the number of claims

- This involves two levels of aggregation
  - Step 1: generate a count of each customer’s claims
  - Step 2: compute the maximum count from this set of results

- Once you have result of step 2, can reuse the result of step 1 to find the final result

- Common subquery: computation of how many claims each customer has
Another Aggregate Example (2)

- Use assignment operation to store temporary result
  \[\text{claim\_counts} \leftarrow \text{driver\_id} \quad \text{\#count(license) as num\_claims(claim)}\]
  \[\text{max\_count} \leftarrow \text{\#max(num\_claims) as max\_claims(claim\_counts)}\]

- Schemas of \text{claim\_counts} and \text{max\_count}?
  \[\text{claim\_counts(driver\_id, num\_claims)}\]
  \[\text{max\_count(max\_claims)}\]

- Finally, select row from \text{claim\_counts} with the maximum count value
  - Obvious here that a Cartesian product is necessary
  \[\Pi_{\text{driver\_id, num\_claims}}(\sigma_{\text{num\_claims}=\text{max\_claims}}(\text{claim\_counts} \times \text{max\_count}))\]
Can add rows to a relation

\[ r \leftarrow r \cup \{ (\ldots), (\ldots) \} \]

- \{ (\ldots), (\ldots) \} is called a constant relation
- Individual tuple literals enclosed by parentheses ( )
- Set of tuples enclosed with curly braces { }

Can delete rows from a relation

\[ r \leftarrow r - \sigma_p(r) \]

Can modify rows in a relation

\[ r \leftarrow \Pi(r) \]

- Uses generalized project operation
Modifying Relations (2)

- **Remember to include unmodified rows!**
  
  \[ r \leftarrow \Pi(\sigma_p(r)) \cup \sigma_{\neg p}(r) \]

- Relational algebra is **not** like SQL for updates!
  - Must *explicitly* include unaffected rows

- **Example:**

  Transfer $10,000 in assets to all Horseneck branches.

  \[
  \begin{align*}
  &\text{branch} \leftarrow \Pi_{\text{branch\_name,branch\_city,assets} + 10000}(\sigma_{\text{branch\_city}=\text{“Horseneck”}}(\text{branch})) \\
  &\text{Wrong: This version throws out all branches not in Horseneck!}
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{branch} \leftarrow \Pi_{\text{branch\_name,branch\_city,assets} + 10000}(\sigma_{\text{branch\_city}=\text{“Horseneck”}}(\text{branch})) \cup \\
  &\quad \sigma_{\text{branch\_city} \neq \text{“Horseneck”}}(\text{branch}) \\
  &\text{Correct. Non-Horseneck branches are included, unmodified.}
  \end{align*}
  \]
Some major differences between SQL and relational algebra!

Tables are like relations, but are multisets

Most queries generate multisets

- SELECT queries produce multisets, unless they specify SELECT DISTINCT ...

Some operations do eliminate duplicates!

- Set operations: UNION, INTERSECT, EXCEPT
  - Duplicates are eliminated automatically, unless you specify UNION ALL, INTERSECT ALL, EXCEPT ALL
**SELECT** is most ubiquitous

\[
\text{SELECT } A_1, A_2, \ldots \text{ FROM } r_1, r_2, \ldots \text{ WHERE } P;
\]
- Equivalent to: \( \prod_{A_1, A_2, \ldots} (\sigma_P(r_1 \times r_2 \times \ldots)) \)

**INSERT, UPDATE, DELETE** all have common aspects of **SELECT**
- All support **WHERE** clause, subqueries, etc.
- Also **INSERT** ... **SELECT** statement
Join Alternatives

- **FROM r1, r2**
  - Cartesian product
  - Can specify join conditions in **WHERE** clause

- **FROM r1 JOIN r2 ON (r1.a = r2.a)**
  - Most like theta-join operator: \( r \bowtie_\theta s = \sigma_\theta(r \times s) \)
  - Doesn’t eliminate any columns!

- **FROM r1 JOIN r2 USING (a)**
  - Eliminates duplicate column \( a \)

- **FROM r1 NATURAL JOIN r2**
  - Uses all common attributes to join \( r1 \) and \( r2 \)
  - Also eliminates all duplicate columns in result
Join Alternatives (2)

- Can specify inner/outer joins with `JOIN` syntax
  - `r INNER JOIN s ...`
  - `r LEFT OUTER JOIN s ...`
  - `r RIGHT OUTER JOIN s ...`
  - `r FULL OUTER JOIN s ...`
- Can also specify `r CROSS JOIN s`
  - Cartesian product of `r` with `s`
  - Can’t specify `ON` condition, `USING`, or `NATURAL`
- Can actually leave out `INNER` or `OUTER`
  - `OUTER` is implied by `LEFT/RIGHT/FULL`
  - If you just say `JOIN`, this is an `INNER` join
Self-Joins

- Sometimes helpful to do a self-join
  - A join of a table with itself
- Example: employees
  - employee(emp_id, emp_name, salary, manager_id)
- Tables can contain foreign-key references to themselves
  - manager_id is a foreign-key reference to employee table’s emp_id attribute
- Example:
  - Write a query to retrieve the name of each employee, and the name of each employee’s boss.
    SELECT e.emp_name, b.emp_name AS boss_name
    FROM employee AS e JOIN employee AS b
    ON (e.manager_id = b.emp_id);
Subqueries

- Can include subqueries in **FROM** clause
  - Called a derived relation
  - Nested **SELECT** statement in **FROM** clause, given a name and a set of attribute names

- Can also use subqueries in **WHERE** clause
  - Can compare an attribute to a scalar subquery
    - This is different from the relational algebra!
  - Can also use set-comparison operations to test against a subquery
    - **IN, NOT IN** – set membership tests
    - **EXISTS, NOT EXISTS** – empty-set tests
    - **ANY, SOME, ALL** – comparison against a set of values
Scalar Subqueries

- Find name and city of branch with the least assets
  - Need to generate the “least assets” value, then use this to select the specific branch records

- Query:
  ```sql
  SELECT branch_name, branch_city FROM branch
  WHERE assets = (SELECT MIN(assets) FROM branch);
  ```
  - This is a scalar subquery: one row, one column
  - Don’t need to name \( \text{MIN}(\text{assets}) \) since it doesn’t appear in final result, and we don’t refer to it

- Don’t do this:
  ```sql
  WHERE assets=ALL (SELECT MIN(assets) FROM branch)
  ```
  - \( \text{ANY}, \text{SOME}, \text{ALL} \) are for comparing a value to a set of values
  - Don’t need these when comparing to a scalar subquery
Subqueries vs. Views

- Don’t create views unnecessarily
  - Views are part of a database’s schema
  - Every database user sees the views that are defined

- Views should generally expose “final results,” not intermediate results in a larger computation
  - Don’t use views to compute intermediate results

- If you really want functionality like this, read about the `WITH` clause (Book, 6th ed: §3.8.6, pg. 97)
  - MySQL doesn’t support `WITH`, so unfortunately you can’t use it in CS121 😞
**WHERE Clause**

- **WHERE** clause specifies selection predicate
  - Can use **AND**, **OR**, **NOT** to combine conditions
  - **NULL** values affect comparisons!
    - Can’t use **=** NULL or **<>** NULL
      - *Never* evaluates to true, regardless of other value
    - Must use **IS NULL** or **IS NOT NULL**
  - Can use **BETWEEN** to simplify range checks
    - \( a \geq v_1 \text{ AND } a \leq v_2 \)
    - \( a \text{ BETWEEN } v_1 \text{ AND } v_2 \)
Grouping and Aggregation

- SQL supports grouping and aggregation
- **GROUP BY** specifies attributes to group on
  - Apply aggregate functions to non-grouping columns in **SELECT** clause
  - Can filter results of grouping operation using **HAVING** clause
    - **HAVING** clause can refer to aggregate values too
- Difference between **WHERE** and **HAVING**?
  - **WHERE** is applied before grouping; **HAVING** is applied after grouping
  - **HAVING** can refer to aggregate results, too
    - Unlike relational algebra, can use aggregate functions in **HAVING** clause
Another difference between relational algebra notation and SQL syntax

Relational algebra syntax:

\[ G_1, G_2, \ldots, G_n \left( F_1(A_1), F_2(A_2), \ldots, F_m(A_m) \right)(E) \]

- Grouping attributes appear only on \textbf{left} of \( G \)
- Schema of result: \( (G_1, G_2, \ldots, F_1, F_2, \ldots) \)
  - (Remember, \( F_i \) generate \textit{unnamed} results.)

SQL syntax:

\[
\text{SELECT } G_1, G_2, \ldots, F_1(A_1), F_2(A_2), \ldots
\text{ FROM } r_1, r_2, \ldots \text{ WHERE } P
\text{ GROUP BY } G_1, G_2, \ldots
\]

- To include group-by values in result, specify grouping attributes in \textbf{SELECT} clause and in \textbf{GROUP BY} clause
Grouping and Distinct Results

- SQL grouping syntax:
  ```sql
  SELECT  G_1, G_2, ..., F_1(A_1), F_2(A_2), ...
  FROM  r_1, r_2, ..., WHERE P
  GROUP BY  G_1, G_2, ...
  ```

- If all grouping attributes are in SELECT clause:
  - Are all rows in the results distinct?
  - Yes! \((G_1, G_2, \ldots)\) is a superkey on the results.
  - Each group in result has a unique set of values for the set of grouping attributes.
  - Don’t need to specify `SELECT DISTINCT G_1, G_2, \ldots` if all grouping attributes are listed.
Another example:

```sql
SELECT G_3, F_1(A_1), F_2(A_2)
FROM r_1, r_2, ...
GROUP BY G_1, G_2, G_3
```

- You can specify only a subset of the grouping attributes in the `SELECT` clause (or even none of the grouping attributes)
  - Results no longer guaranteed to be distinct, of course

- **Main constraint (approximately):**
  - Can’t specify non-grouping attributes in `SELECT` clause unless they are arguments to an aggregate function
  - Default MySQL configuration allows you to violate this rule, but the results are not well-defined!
    - This has been turned off all term, hopefully this has helped... 😊
SQL Query Example

- Schema:
  - `car(license, vin, make, model, year)`
  - `customer(driver_id, name, street, city)`
  - `owner(license, driver_id)`
  - `claim(driver_id, license, date, description, amount)`

- Find customers with more claims than the average number of claims per customer

- This is an aggregate of another aggregate

- Each `SELECT` can only compute one level of aggregation
  - `AVG(COUNT(*))` is not allowed in SQL
    (or in relational algebra, so no big surprise)
Aggregates of Aggregates

- **Two steps** to find average number of claims

- **Step 1:**
  - Must compute a count of claims for each customer
    
    ```sql
    SELECT COUNT(*) AS num_claims
    FROM claim GROUP BY driver_id
    ```
  - Then, compute the average in a second **SELECT**:
    
    ```sql
    SELECT AVG(num_claims)
    FROM (SELECT COUNT(*) AS num_claims
          FROM claim GROUP BY driver_id) AS c
    ```

- This generates a single result
  - Can use it as a scalar subquery if we want.
Finally, can compute the full result:

```sql
SELECT driver_id, COUNT(*) AS num_claims
FROM claim GROUP BY driver_id
HAVING num_claims >=
  (SELECT AVG(num_claims)
    FROM (SELECT COUNT(*) AS num_claims
          FROM claim GROUP BY driver_id) AS c);
```

- Comparison **must** be in **HAVING** clause

This won’t work:

```sql
SELECT driver_id, COUNT(*) AS num_claims
FROM claim GROUP BY driver_id
HAVING num_claims = AVG(num_claims);
```

- Tries to do two levels of aggregation in one **SELECT**
Alternative 1: Make a View

- Knowing each customer’s total number of claims could be generally useful...

- Define a view for it:

```sql
CREATE VIEW claim_counts AS
  SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id;
```

- Then the query becomes:

```
SELECT * FROM claim_counts
WHERE num_claims >
  (SELECT AVG(num_claims) FROM claim_counts)
```

- View hides one level of aggregation
Alternative 2: Use **WITH** Clause

- **WITH** is like defining a view for a single statement
- Using **WITH**:  
  ```sql
  WITH claim_counts (driver_id, num_claims) AS (  
    SELECT driver_id, COUNT(*)  
    FROM claim GROUP BY name)  
  SELECT * FROM claim_counts  
  WHERE num_claims > (SELECT AVG(num_claims)  
    FROM claim_counts);
  ```
- **WITH** doesn’t pollute the database schema with a bunch of random views
- Can specify multiple **WITH** clauses, too (see book)
- (Unfortunately, MySQL doesn’t support **WITH**...
SQL Data Definition

- Specify table schemas using **CREATE TABLE**
  - Specify each column’s name and domain
  - Can specify domain constraint: **NOT NULL**
  - Can specify key constraints
    - **PRIMARY KEY**
    - **UNIQUE** (candidate keys)
    - **REFERENCES table (column)** (foreign keys)
  - Key constraints can go in column declaration
  - Can also specify keys after all column decls.

- Be familiar with common SQL data types
  - **INTEGER, CHAR, VARCHAR, date/time types**, etc.
DDL Example

- Relation schema:
  
  ```
  car(license, vin, make, model, year)
  ```
  
  - `vin` is also a candidate key

- CREATE TABLE statement:
  
  ```
  CREATE TABLE car (
    license CHAR(10) PRIMARY KEY,
    vin CHAR(30) NOT NULL UNIQUE,
    make VARCHAR(20) NOT NULL,
    model VARCHAR(20) NOT NULL,
    year INTEGER NOT NULL
  );
  ```
DDL Example (2)

- **Relation schema:**
  \[ \text{claim}(\text{driver_id}, \text{license}, \text{date}, \text{description}, \text{amount}) \]

- **CREATE TABLE statement:**
  ```sql
  CREATE TABLE claim ( 
    driver_id   CHAR(12),
    license     CHAR(10),
    date        TIMESTAMP,
    description VARCHAR(4000) NOT NULL,
    amount      NUMERIC(8,2),
    PRIMARY KEY (driver_id, license, date),
    FOREIGN KEY driver_id REFERENCES customer,
    FOREIGN KEY license REFERENCES car
  );
  ```
Key Constraints and **NULL**

- Some key constraints automatically include **NOT NULL** constraints, but **not all do**.

  - **PRIMARY KEY** constraints
    - Disallows **NULL** values

  - **UNIQUE** constraints
    - Allows **NULL** values, unless you specify **NOT NULL**

  - **FOREIGN KEY** constraints
    - Allows **NULL** values, unless you specify **NOT NULL**

- Understand how **NULL** values affect **UNIQUE** and **FOREIGN KEY** constraints that allow **NULLs**
Referential Integrity Constraints

- Unlike relational algebra, SQL DBs automatically enforce referential integrity constraints for you.
  - You still need to perform operations in the correct order, though.
- Same example as before:
  - Remove customer “Jones” from the bank database.
  - DBMS will ensure that referential integrity is enforced, but you still have to delete rows from `depositor` and `borrower` tables first!

```sql
DELETE FROM depositor WHERE customer_name = 'Jones'
DELETE FROM borrower WHERE customer_name = 'Jones'
DELETE FROM customer WHERE customer_name = 'Jones'
```
Midterm Details

- Midterm posted online around Thursday, October 27
- Due Thursday, November 3 at 2:00AM (the usual time)

- No homework to do next week

- Good luck! 😊