UPE Tutoring:

CS 32 Midterm 2 Review + Project 3 Q&A

Sign-in https://bit.ly/39W9B3q

Slides link available upon sign-in

Quick Announcement - CS Town Hall

Date & Time: Wed, Feb 26th 6-8 pm (Week 8)
Location: Mong Auditorium, Eng VI

The CS Town Hall is an opportunity for all students in CS and related disciplines to interact with the UCLA CS Department and provide feedback, suggestions or discuss problems.

Please make your voice heard! Fill out this survey: https://forms.gle/JrE9HYXmwftrYB6NA

This year, we will also have a discussion about cheating in CS classes. Voice your opinion in this **completely anonymous** survey: https://forms.gle/KuRKjmfWqJZp8xix5

Food will be provided! RSVP Link: https://www.facebook.com/events/528498164437871/

STL Data Structures and Iterators

Templates

- A class template describes a class in terms of data-type parameter
- C++ includes the STL, or standard template library
- Consider a simple class template, where T is the data-type parameter

```
template <typename T>
class NewClass
{
public:
  NewClass();
  NewClass(T initialData);
  void setData(T newData);
  T getData();
private:
  T theData;
}; // end NewClass
```

```
int main()
{
  NewClass<int> first;
  NewClass<double> second(4.8);

  first.setData(5);
  cout << second.getData() << endl;
}
```

STL: Stacks and Queues

- Two abstract data types (ADTs) used to store items
- Stacks follow LIFO (Last in, first out)
- Queues follow FIFO (First in, first out)
- The C++ STL has implementations of both of these
    - #include <stack>
    - #include <queue>
    - stack<Type> stack_name;
    - queue<Type> queue_name;

STL: Stacks

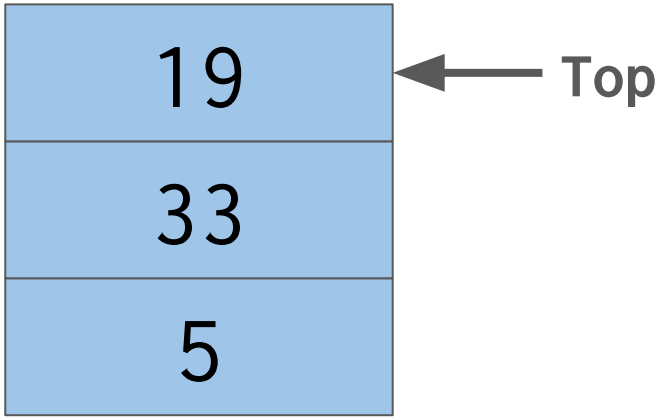
```
stack<int> s;
s.push(33);             // Inserts an element.
int top = s.top();      // Access top element.
s.pop();                // Removes top element.
int size = s.size();
bool empty = s.empty();
```

STL: Stack Example

```
#include <stack>
...
stack<int> stk;
stk.push(5);
stk.push(33);
stk.push(87);
cout << stk.top() << endl;
stk.pop();
stk.push(19);
cout << stk.top() << endl;
```

STL: Stack Example

```
#include <stack>
...
stack<int> stk;
stk.push(5);
stk.push(33);
stk.push(87);
cout << stk.top() << endl;
stk.pop();
stk.push(19);
cout << stk.top() << endl;
```




```
> 87
> 19
```

STL: Stacks

- When to use?
  - Whenever you need **LIFO** data storage
- Example: Backtracking
  - Store your path through a maze in a stack
    - **Push** each new position onto the stack
  - Backtrack to a fork in the road when you reach a dead-end
    - **Pop** positions from the stack until you're back where you want to be
- How to implement on your own?
  - Use an array with an index referring to the "top" of the stack
  - Use a linked list, **pushing** a new head and **popping** off the head

STL: Queues

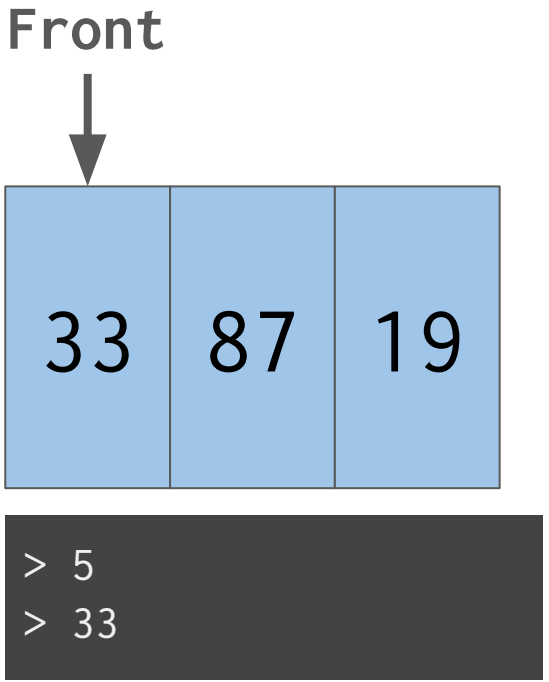
```
queue<int> q;
q.push(32);              // Queues an element.
int front = q.front(); // Access front element.
int back = q.back();   // Access back element.
q.pop();                 // Remove front element.
int size = q.size();
bool empty = q.empty();
```

STL: Queue Example

```
#include <queue>
...
queue<int> q;
q.push(5);
q.push(33);
q.push(87);
cout << q.front() << endl;
q.pop();
q.push(19);
cout << q.front() << endl;
```

STL: Queue Example

```
#include <queue>
...
queue<int> q;
q.push(5);
q.push(33);
q.push(87);
cout << q.front() << endl;
q.pop();
q.push(19);
cout << q.front() << endl;
```



STL: Queues

- When to use?
  - Whenever you need **FIFO** data storage
- Example: CPU Scheduling
  - Store process IDs on a queue
    - **Push** a process' ID that wants to run onto the queue
    - **Pop** the front process ID from the queue, give the process some CPU time, then **push** it back on the queue if it isn't finished yet
- How to implement on your own?
  - Use a linked list with a front and back pointer, **pushing** to the back and **popping** from the front

Practice Question: Parenthesis Validation

Write a function that takes an input string of parentheses and returns true or false on whether it is a valid grouping.

For example,

```
true:  validGrouping(“()(()())”)
true:  validGrouping(“()()”)
false: validGrouping(“())()(”)
```

(Problem contributed by Jerry Li)

Solution: Parenthesis Validation

```
bool validGrouping(string str) {
  stack<char> parenStack;

  for(int i = 0; i < str.size(); i++) {
    if(str[i] == '(')
      parenStack.push(str[i]);
    else {
      if(parenStack.empty())
        return false;
      parenStack.pop();
    }
  }

  // (continued on next column)
```

```
  if(!parenStack.empty())
    return false;

  return true;
}
```

As we traverse the string from left to right, we ensure we have never seen fewer left parentheses, (, than right parentheses,).

Practice Question: String Reversal using a Stack

Write a function called `reverseString` that takes in as its parameter one string and returns a string with the characters in reverse order. Use the STL stack to reverse the string. This isn't a very efficient way to perform string reversal, but it's good stack practice.

Solution: String Reversal using a Stack

```
string reverseString(string text) {
     string reversedString = "";
     stack<char> myStack;

     for(int i =0; i < text.size(); i++)
          myStack.push(text[i]);

     while(!myStack.empty()) {
          reversedString += myStack.top();
          myStack.pop();
     }
     return reversedString;
}
```

Practice Question: Stack from a Queue

Implement the following class that functions as a stack of ints using only a STL queue<int>.

```
class Stack {
public:
   bool empty() const;
   int size() const;
   int top();
   void push(int value);
   void pop();
private:
   queue<int> q;
};
```

Solution: Stack from a Queue

```
bool Stack::empty() const {
   return q.empty();
}

int Stack::size() const {
   return q.size();
}

int Stack::top() {
   return q.back();
}
```

```
void Stack::push(int value) {
   q.push(value);
}

void Stack::pop() {
   int n = q.size() - 1;
   for (int i = 0; i < n; i++) {
      q.push(q.front());
      q.pop();
   }
   q.pop();
}
```

Also try implementing a queue by using one or more stacks!

STL: Vectors

```
vector<int> v;
v.push_back(31);  // Adds element to end.
int front = v[0]; // Square bracket to access.
v.pop_back();     // Deletes element at end.
int size = v.size();
bool empty = v.empty();

// Takes an iterator, number to copy, and
// value to copy. v = [33, 33, 33, 33, 33].
v.insert(v.begin(), 5, 33);

// Takes two iterators, deleting the values
// between [begin, end).
v.erase(v.begin(), v.begin() + 3);
```

STL: Lists

```
list<int> l;
l.push_front(32); // Adds element to front.
l.pop_front();    // Deletes element at front.
l.push_back(31);  // Adds element to end.
l.pop_back();     // Deletes element at end.
int size = l.size();
bool empty = l.empty();

// Insert and erase, as before.
l.insert(l.begin(), 5, 33);
l.erase(l.begin(), l.begin() + 3);
```

STL: Vector or List?

Vector

- Contiguous memory (like an Array)
- Insert/erase at the end average to constant time, but insertions elsewhere are a costly O(n)
- You can randomly access elements
- Iterators are invalidated if you add or remove elements to or from the vector

Need fast random access? Use a vector.

List

- Non-contiguous memory (like a Linked List)
- Insert/erase are cheap no matter where in the list they occur.
- It's cheap to combine lists with splicing.
- You *cannot* randomly access elements
- Iterators remain valid even when you add or remove elements from the list

Doing lots of insert/erase anywhere except the end? Use a list.

STL: Sets

```
set<int> s;
s.insert(31);      // Adds element to set (unique).
s.erase(31);       // Remove element from set.
int size = s.size();
bool empty = s.empty();

// Returns s.end() if value is not found.
set<int>::iterator it = s.find(30);
if (it != s.end()) {
  s.erase(it);
}

// Takes two iterators, deleting the values
// between [begin, end).
v.erase(v.begin(), v.begin + 3);
```

STL: Maps

```
map<string, int> m;
m["das"] = 33;    // "das" -> 33
m.erase("das");   // Removes element by key.
int size = s.size();
bool empty = s.empty();

// Returns m.end() if value is not found.
map<string, int>::iterator it = m.find("das");
if (it != m.end()) {
  m.erase(it);
}

// Takes two iterators, deleting the values
// between [begin, end).
m.erase(m.begin(), m.begin + 3);
```

Iterators

- Iterators abstract the process of iterating through a number of STL containers
- The order of iteration reflects the underlying data structure
  - `list` and `vector` iterators traverse in the order of the collection (not necessarily sorted)
  - `unordered_set` and `unordered_map` iterators *seem to* traverse in reverse order of insertion
    - They are implemented as hash maps and hash sets
    - Don't rely on this order of traversal, it isn't guaranteed
  - `map` and `set` iterators traverse in sorted order
    - They are implemented as BSTs

Iterators

- For some given container c, we have the following iterators:
  - **`c.begin()`** returns an `iterator` to the first element in c;
  - **`c.cbegin()`** returns a `const_iterator` to the first element in c;
  - **`c.end()`** returns an `iterator` to the element following the last element in c;
  - **`c.cend()`** returns a `const_iterator` to the element following the last element in c.
- An iterator object has several overloaded operators that allow you to use it like a pointer, typically including:

  `=` (assignment) `++` (increment) `==` (equals)

  `!=` (not equals) `*` (dereference) `->` (member)
- Calling `erase` on an STL structure using an iterator invalidates the iterator, but `erase` returns an iterator to the next item, which can be assigned back

Iterator Syntax

```
vector<int>::iterator it = v.begin();

// Here, * "dereferences" the iterator.
// Use `rbegin` and `rend` to iterate
// in reverse order.
for (; it != v.end(); it++) {
  cout << *it << endl;
}
```

Iterator Syntax (Maps)

```
map<string, int>::iterator it = m.begin();

// Here, `first` refers to the key;
// `second` refers to the value.
// Use `rbegin` and `rend` to iterate
// in reverse order.
for (; it != m.end(); it++) {
  cout << it->first << " " << it->second << endl;
}
```

Iterators Exercise

1. Declare a list of strings `li` and insert these words: "hybrid", "appear", "freeze", "friend", "appear".
2. Remove all occurrences of the word "appear" (use an `iterator`).
3. Print out the processed list (use a `const_iterator`).

Iterators Exercise Solution

```
list<string> li;
li.push_back("hybrid");
li.push_back("appear");
li.push_back("freeze");
li.push_back("friend");
li.push_back("appear");

for (list<string>::iterator it = li.begin(); it != li.end();)
  if (*it == "appear")
    it = li.erase(it);
  else
    ++it;

for (list<string>::const_iterator it = li.cbegin(); it != li.cend(); ++it)
  cout << *it << endl;
```

Inheritance and Polymorphism

Inheritance

- Define new classes based on old classes
- Based on an “is-a” relationship
    - Subclass “is-a” type of Superclass

Animal

Cat *is-an Animal*

Jaguar *is-a Cat*

Inheritance Benefits

- Reuse code
  - Code in base class is inherited in derived classes
- Specialization
  - New code can be added to the derived class to distinguish it from its base parent
- Overriding
  - Certain inherited code (defined as virtual functions) can be overridden in the derived class

Inheritance Example

```
class Animal
{
public:
    Animal(): fatigueLevel(3)
    {}

    void sleep()
    {
     fatigueLevel--;
    }

private:
    int fatigueLevel;
};
```

```
class Cat : public Animal
{
public:
    Cat()
    {}
};

int main()
{
   Cat c;
   c.sleep();
}
```

Inheritance Example - Specialization

```
class Animal
{
public:
    Animal(): fatigueLevel(3)
    {}

    void sleep()
    {
     fatigueLevel--;
    }

private:
    int fatigueLevel;
};
```

```
class Cat : public Animal
{
public:
    Cat() : thirsty(true) {}
    void drinkMilk() { thirsty = false; }
private:
    bool thirsty;
};


int main()
{
   Cat c;
   c.sleep();
}
```

Inheritance Example - Overriding

```
class Animal
{
public:
    Animal(): fatigueLevel(3)
    {}

    virtual void sleep()
    {
     fatigueLevel--;
    }

private:
    int fatigueLevel;
};
```

```
class Cat : public Animal
{
public:
    Cat() : thirsty(true) {}
    void drinkMilk() { thirsty = false; }
    virtual void sleep() { thirsty = true; }
private:
    bool thirsty;
};
```

Notice, fatigueLevel never changes!

```
int main()
{
   Cat c;
   c.sleep();
}
```

Inheritance - Things to Know

What is a derived class allowed to access from its parent class(es)?

Only the member variables/functions in the parent that are defined as **public** or **protected.**

(In the previous example, a method in the Cat class could therefore not alter the `fatigueLevel` private variable owned by the Animal class.)

Inheritance - Things to Know

What order are classes constructed in inheritance situations?

Classes are constructed in order from most **basic** to most **derived**.

In other words: from furthest ancestor to child.

Note: member variables of an “ancestor” class are also constructed before the derived child class’ constructor is called.

Polymorphism

- Occurs when we use a base pointer or reference to access an object derived from the base's class.
- This behavior will be allowed by our compiler.
- For example, we can create a function that causes an Animal to sleep:

```
Void sleepForNumHours(Animal* a, int n)
{
     while(n > 0)
     {
          a->sleep();
          n--;
     }
}
```

Polymorphism

```
void sleepForNumHours(Animal* a, int n)
{
      while(n > 0)
      {
            a->sleep();
            n--;
      }
}

int main()
{
    Animal* d = new Cat;      // This is fine because a Cat is a derived class of an Animal.
    sleepForNumHours(d, 2);   // Polymorphism occurs in this call because we are using a
                              //  base Animal pointer to access a derived Cat.
    delete d;                 // In order for this to work (i.e. call the destructor for Cat and then
                              //  for Animal), we need some destructor in Cat and a virtual
                              //  destructor in Animal
}
```

General Rules of Polymorphism

- Always have a virtual destructor in the base class, to prevent memory leaks!
- Base class pointers can point to derived class objects
  - `Animal* a = new Cat;`
  - Every `Cat` *is-an* `Animal`
- Derived class pointers CANNOT point to an object of its base class!
  - `Cat* c = new Animal;` //ERROR!
  - An `Animal` is not necessarily a `Cat`. We cannot guarantee that an `Animal` can do everything a `Cat` can do.
  - For example, `Cats` can drinkMilk(). However, `Animal` does not have this functionality! `Cat` has a superset of `Animal's` functionality.
- Polymorphism goes hand in hand with the principles of inheritance. Make sure to understand both for maximum success!

Adv. Practice Q.: Default Parameter Inheritance 1

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x = 5) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   d.foo();
   d.foo(3);
}
```

Solution: Default Parameter Inheritance 1

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x = 5) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   d.foo();
   d.foo(3);
}
```

Derived 5
Derived 3

Adv. Practice Q.: Default Parameter Inheritance 2

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x = 5) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   Base* bp = &d;
   bp->foo();
}
```

Solution: Default Parameter Inheritance 2

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x = 5) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   Base* bp = &d;
   bp->foo();
}
```

// Doesn't compile!!
// "Too few arguments to function call"

Adv. Practice Q.: Default Parameter Inheritance 3

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x = 3) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   Base* bp = &d;
   bp->foo();
}
```

Solution: Default Parameter Inheritance 3

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x = 3) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   Base* bp = &d;
   bp->foo();
}
```

Derived 3

Adv. Practice Q.: Default Parameter Inheritance 4

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x = 3) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x = 5) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   Base* bp = &d;
   bp->foo();
}
```

Adv. Practice Q.: Default Parameter Inheritance 4

What is the output of the code?

```
class Base {
public:
   virtual void foo(int x = 3) {
      cout << "Base " << x << endl;
   }
};

class Derived: public Base {
public:
   virtual void foo(int x = 5) {
      cout << "Derived " << x << endl;
   }
};
```

```
int main() {
   Derived d;
   Base* bp = &d;
   bp->foo();
}
```

Derived 3

// This is not a typo!! It's so weird!!!

Abstract Base Classes

```
class Food
{
public:
    Food()
    {}

    virtual void taste()
    {
      cout << “Who knows what food this is,” <<
      “so why am I giving it a taste??” << endl;
    }

    virtual ~Food() {...} // Don’t Forget!
};
```

```
class Pizza : public Food
{
public:
    Pizza()
    {}

    virtual void taste()
    {
      cout << "MMM...pizza!" << endl;
      // Note: virtual is not needed here if
      //       no other class derives from Pizza
    }
};
```

Abstract Base Classes - Pure Virtual Functions

- It makes sense to have an implementation tasting a `Pizza`
- It really doesn't for generic `Food`...
- We can avoid useless code in the base class through **pure** virtual functions!

Note: Pure virtual destructors are allowed, but must have an implementation later, outside the class definition like `BaseClass::~BaseClass() { }`

```
class Food
{
public:
    Food()
    {}

    virtual void taste() = 0;
    // Syntax for a pure virtual function

    virtual ~Food() {...} // Don't Forget!
};
```

Abstract Base Classes - No Instantiation

- If a class has any pure virtual functions, it is called an Abstract Base Class (ABC)
- Facts about ABC's
  - Cannot be instantiated
    - Food f; **// ERROR!**
  - Any class derived from an ABC that does not include code for each of the pure virtual functions is also an ABC
    - AmericanFood af; **// ERROR!**

```
class AmericanFood : public Food
{
public:
    AmericanFood() { … }

    // No definition for the taste func
};
```

Abstract Base Classes - Why are they useful?

- Why are ABC's useful?
  - Avoid writing useless functions in the base class
  - Require anyone deriving from the ABC to define certain functions
    - Helps prevent errors

```
// We want to avoid this
virtual void taste()
{
    cout << "Who knows what food this is, so why am I giving it a taste??" << endl;
}
```

Abstract Base Classes - Polymorphism

- You can’t instantiate an object of an ABC.
- However, You can use an ABC pointer or reference!

```
void sample(Food& f) // Food is ABC
{
  // take a bite
     f.taste();
}
```

Recursion

Recursion

What is recursion?

- Recursion is a technique to solve problems by first solving smaller problems and using their solutions to construct the final answer.

How does it work?

- Base Case or Terminating Condition
  - Since a recursive function calls itself, there must be a way to stop the calling
- Simplifying Step
  - Each time the recursive function is called, it must be called with a simpler case of the problem to ensure that we reach the base case

Recursion Example - Binary Conversion

Goal: Print binary representation of a positive decimal integer

How to do this recursively? (Let’s use N=43 as an example)

Recursion Example - Binary Conversion

Goal: Print binary representation of a positive decimal integer

How to do this recursively? (Let's use N=43 as an example)

1. Base Case: stop when N = 0
2. Write '1' if N is odd; '0' if n is even
3. Move pencil one position to left
4. Recursive call to print the binary representation of N / 2 (using integer division not binary division)

Recursion Example - Binary Conversion

Goal: Print binary representation of a positive decimal integer

How to do this recursively? (Let's use N=43 as an example)

1. Base Case: stop when N = 0
2. Write '1' if N is odd; '0' if n is even
3. Move pencil one position to left
4. Recursive call to print the binary representation of N / 2 (using integer division not binary division)

```
43          1
21         11
10        011
5        1011
2       01011
1      101011
0
```

Recursion Example - Binary Conversion

Goal: Print binary representation of a positive decimal integer

How to do this recursively? (Let's use N=43 as an example)

1. Base Case: stop when N = 0
2. Write '1' if N is odd; '0' if n is even
3. Move pencil one position to left
4. Print binary representation of N / 2

```
43                1
21               11
10              011
5              1011
2             01011
1            101011
0
```

Notice how this corresponds to doing it by hand!

Recursion Example - Binary Conversion

- Be careful: computer naturally prints from left to right
  - So we need to first convert N / 2
  - Then write ’0’ or ’1’

```
void bin_convert(int N) {    // 1. Declare function



}
```

Recursion Example - Binary Conversion

- Be careful: computer naturally prints from left to right
  - So we need to first convert N / 2
  - Then write ’0’ or ’1’

```
void bin_convert(int N) {     // 1. Declare function
    if (N == 0) return;       // 2. Base case N=0 -> Stop


}
```

Recursion Example - Binary Conversion

- Be careful: computer naturally prints from left to right
  - So we need to first do the recursive call for N / 2
  - Then write ’0’ or ’1’

```
void bin_convert(int N) {     // 1. Declare function
    if (N == 0) return;       // 2. Base case N=0 -> Stop
    bin_convert(N / 2);       // 3. Convert N/2

}
```

Recursion Example - Binary Conversion

- Be careful: computer naturally prints from left to right
  - So we need to first do the recursive call for N / 2
  - Then write ’0’ or ’1’

```
void bin_convert(int N) {     // 1. Declare function
    if (N == 0) return;       // 2. Base case N=0 -> Stop
    bin_convert(N / 2);       // 3. Convert N/2
    cout << N % 2;            // 4. Print ‘0’ or ‘1’ for
}                                    even or odd
```

Recursion Example - Binary Conversion

```
convert(43)
    convert(21)
        convert(10)
            convert(5)
                convert(2)
                    convert(1)
                        convert(0)
                    printf("1")
                printf("0")
            printf("1")
        printf("0")
    printf("1")
printf("1")
```

Recursive Calls (check base case and recurse further)

Base Case

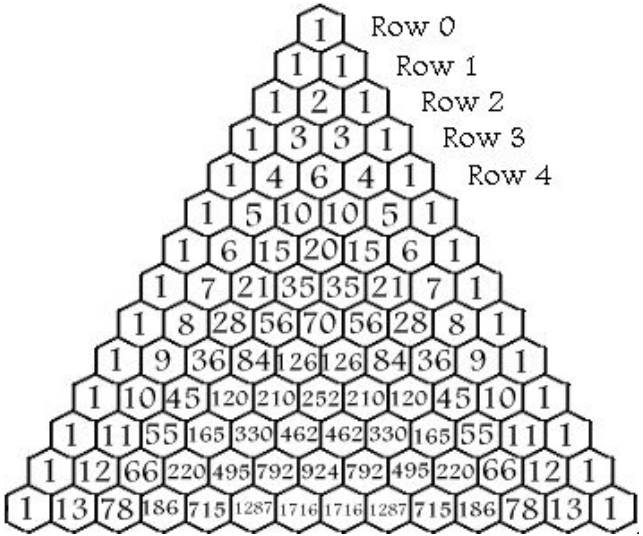
Output Execution

Recursion Tips

- Make sure to check your base case before recursing!
- A recursive subprogram must have at least one base case and one recursive case (it's OK to have more than one base case, and more than one recursive case).
- The problem must be broken down in such a way that the recursive call is closer to the base case than the top-level call.
- The base case must be reachable in a finite number of steps.
- The recursive call must not skip over the base case.
- The non-recursive portions of the subprogram must operate correctly.

Practice Question: Pascal's Triangle

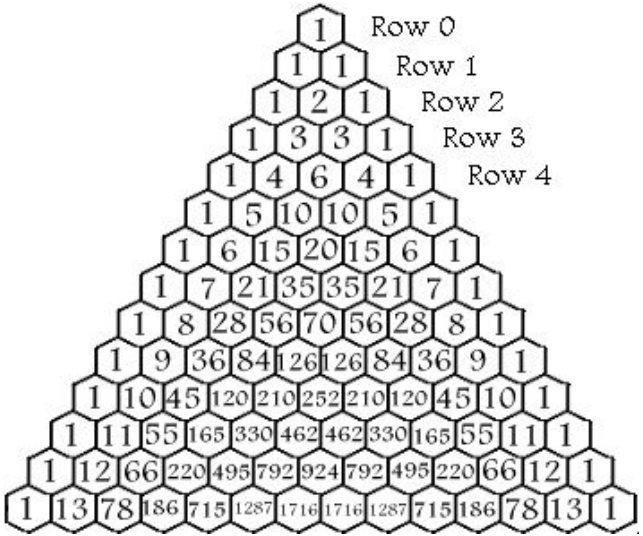
Write a function called **`pascals`** that uses recursion to return the value at a specified (row, column) pair in Pascal's Triangle.



Example: `pascals(2, 1) = 1 + 1 = 2`

Solution: Pascal's Triangle

Write a function called **pascals** that uses recursion to return the value at a specified (row, column) pair in Pascal's Triangle.



Example: `pascals(2, 1) = 1 + 1 = 2`

```
int pascals(int row, int col) {
  // Base case: Far left and far right are always one
  if (col == 0 || col == row) {
    return 1;
  }

  // Simplifying Step: Inner elements are the sum of the
  // (prev row, prev column) element and (prev row, same col)
  else {
    return pascals(row - 1, col - 1) + pascals(row - 1, col);
  }

}
```

Practice Question: Binary Search

Write a function called findPos that finds the position of a given number in a sorted array. Do this using a recursive binary search. Return -1 if the number is not found in the array. You may assume that the numbers in the array are distinct. You may use a helper function to do the recursion if you'd like.

int findPos(int a[], int size, int target);
Example: a is `1,4,5,6,8,9`. `findPos(a, 6, 5)` returns 2. `findPos(a, 6, 7)` returns −1.

Solution: Binary Search

```
int search(int a[], int first, int last, int target) {
      if (first > last)
            return -1;

      int mid = (first + last) / 2;
      if(a[mid] == target)
            return mid;
      else if(a[mid] < target)
            return search(a, mid+1, last, target);
      else
            return search(a, first, mid-1, target);
}

int findPos(int a[], int size, int target) {
      return search(a, 0, size - 1, target);
}
```

Practice Question: `isMeasurable`

Write a function named **`isMeasurable`** that determines whether it is possible to measure out the desired target amount with a given set of weights. For example, if the sample weights are {1, 3}, you can measure a target weight of 2 by putting the 1 on the left and 3 on the right.

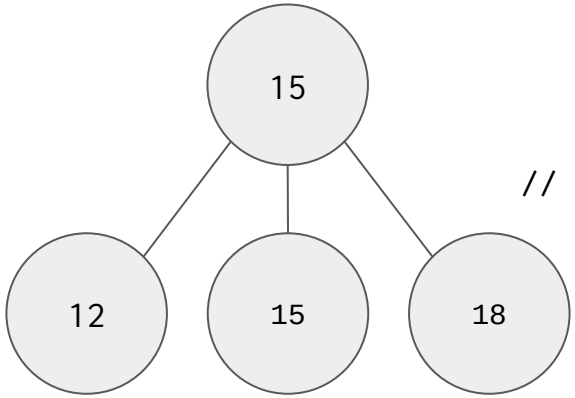
```
bool isMeasurable(int target, vector<int> weights);
isMeasurable(2, {1, 3})    // Returns true
isMeasurable(5, {1, 3})    // Returns false
isMeasurable(6, {2, 3, 7}) // Returns true
```

isMeasurable

It's practically a (recursive) search problem! Let's look at the subproblems:

1. **There aren't any more weights.**
   In this case, we need to know if we have any unbalanced weight. If we do, return `false`. (Another way to say this is that if the two sides are not balanced right now, we return false.) Otherwise, return `true`.

isMeasurable




```
// Assume the next weight is 3.
```

It's practically a (recursive) search problem! Let's look at the subproblems:

2. **There are weights!**
   In this case, we have three search options. Let's start out by taking out the last weight in the list. We can either pass on the current weight, put it to the left, or put it on the right.

Solution: isMeasurable

```
bool isMeasurable(int target, vector<int> weights) {
  if (weights.empty()) {
    return target == 0;
  }
  int back = weights.back();
  weights.pop_back();
  return isMeasurable(target, weights) ||     // Don't use this weight.
      isMeasurable(target - back, weights) || // Add it to the left?
      isMeasurable(target + back, weights);   // Or add it to the right.
}
```

Practice Question: `longestCommonSubsequence`

Write a function named **`longestCommonSubsequence`** that returns the longest common subsequence of both strings.

```
string longestcommonSubsequence(string s1, string s2);
longestCommonSubsequence("mars","megan")  // Returns "ma"
longestCommonSubsequence("chris", "cs32") // Returns "cs"
```

`longestCommonSubsequence`

Search strategy.

1. If either string is empty: return `""`.
2. Compare the first characters.
    a. If they're the same, then we know that we have this character in our longest subsequence. So, we return `s1[0] + longestCommonSubsequence(s1.substr(1), s2.substr(1))`.
    b. If they're different, then we have two choices. We can either progress `s1`, comparing `longestCommonSubsequence(s1.substr(1), s2)`, or vice versa.

Solution: longestCommonSubsequence

```
string longestCommonSubsequence(string s1, string s2) {
  if (s1.length() == 0 || s2.length() == 0) {
    return "";
  } else if (s1[0] == s2[0]) {
    return s1[0] + longestCommonSubsequence(s1.substr(1), s2.substr(1));
  } else {
    string choice1 = longestCommonSubsequence(s1, s2.substr(1));
    string choice2 = longestCommonSubsequence(s1.substr(1), s2);
    return choice1.length() > choice2.length() ? choice1 : choice2;
  }
}
```

Practice Question: Recursive Merge LL

Implement a recursive function that merges two sorted linked lists into a single sorted linked list. The lists are singly linked; the last node in a list has a null next pointer. The function should return the head of the merged linked list. No new Nodes should be created while merging. Example:

List 1 = 1 -> 4 -> 6 -> 8
List 2 = 3 -> 9 -> 10
After merge: 1 -> 3 -> 4 -> 6 -> 8 -> 9 -> 10

Practice Question: Recursive Merge LL

List 1 = 1 -> 4 -> 6 -> 8 List 2 = 3 -> 9 -> 10 After merge: 1 -> 3 -> 4 -> 6 -> 8 -> 9 -> 10

Use the following definition of a Node of a linked list:

```
struct Node {
    int val;
    Node* next;
};
```

Use the following function header to get started:

```
Node* merge(Node* l1, Node* l2);
```

Solution: Recursive Merge LL

```
Node* merge(Node* l1, Node* l2) {

   if (l1 == nullptr) return l2;
   if (l2 == nullptr) return l1;

   Node* head;
   if (l1->val < l2->val) {
      head = l1;
      head->next = merge(l1->next, l2);
   } else {
      head = l2;
      head->next = merge(l1, l2->next);
   }

   return head;
}
```

// base cases: if a list is empty, return the other list

// determine which head should be the head of the current merged list
// then set the next pointer to the head of the recursive calls

// return the head of the merged list

Midterm 2 Tips

- Open book open notes, but try to consolidate info onto a cheat sheet
- Expect these kinds of problems on the midterm:
  - Understanding and writing some stack and queue code
    - E.g. tracing the states of a stack
  - Inheritance and polymorphism:
    - How inheritance affects constructors/deconstructors
    - Tracing polymorphism code
    - Understanding virtual and pure virtual
  - Writing recursive functions
    - That may deal specifically with some given data structure

Good luck!

Sign-in https://bit.ly/39W9B3q
Slides https://bit.ly/39ShhU8
Practice https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.