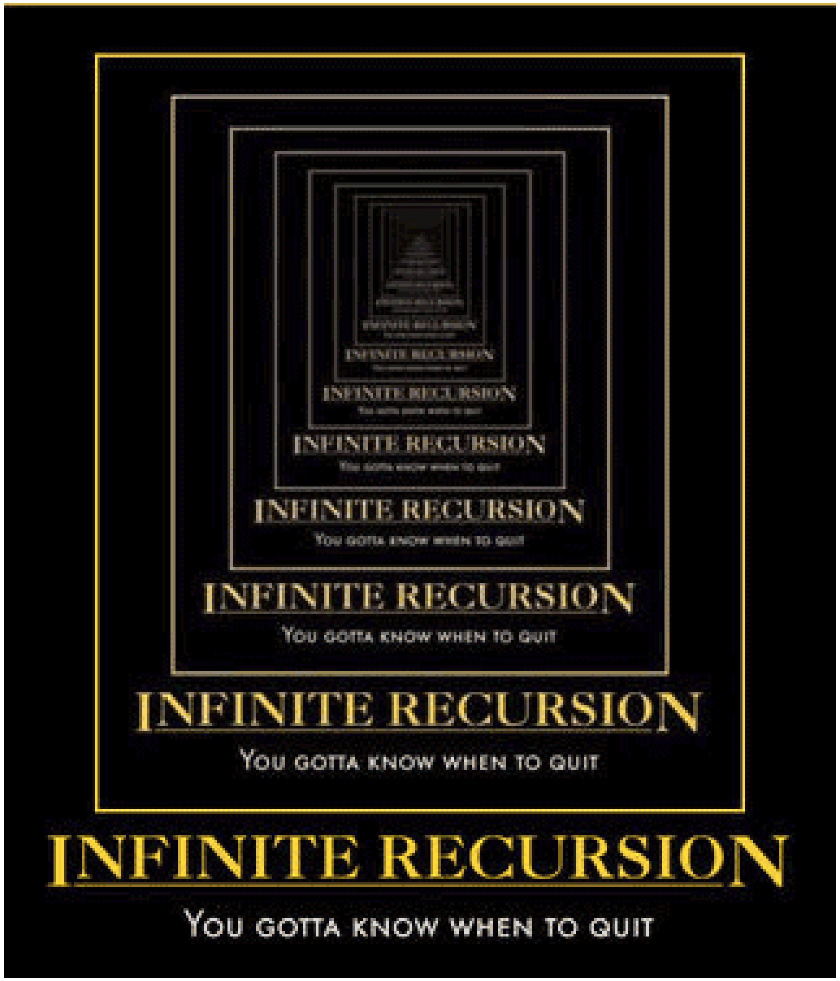
Lecture #8

- Recursion
- How to *design* an Object Oriented program (on your own study)
- Project 3 Design Tips

This is a ship-shipping ship, shipping shipping ships.

Recursion


INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION

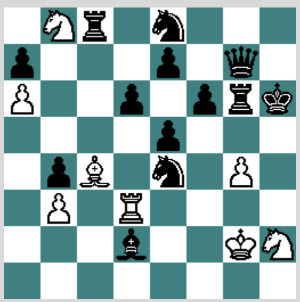
Recursion

Why should you care?

Recursion is one of the most difficult… but powerful computer science topics.

Use it for things like…

AI for Games



Solving SuDoKu

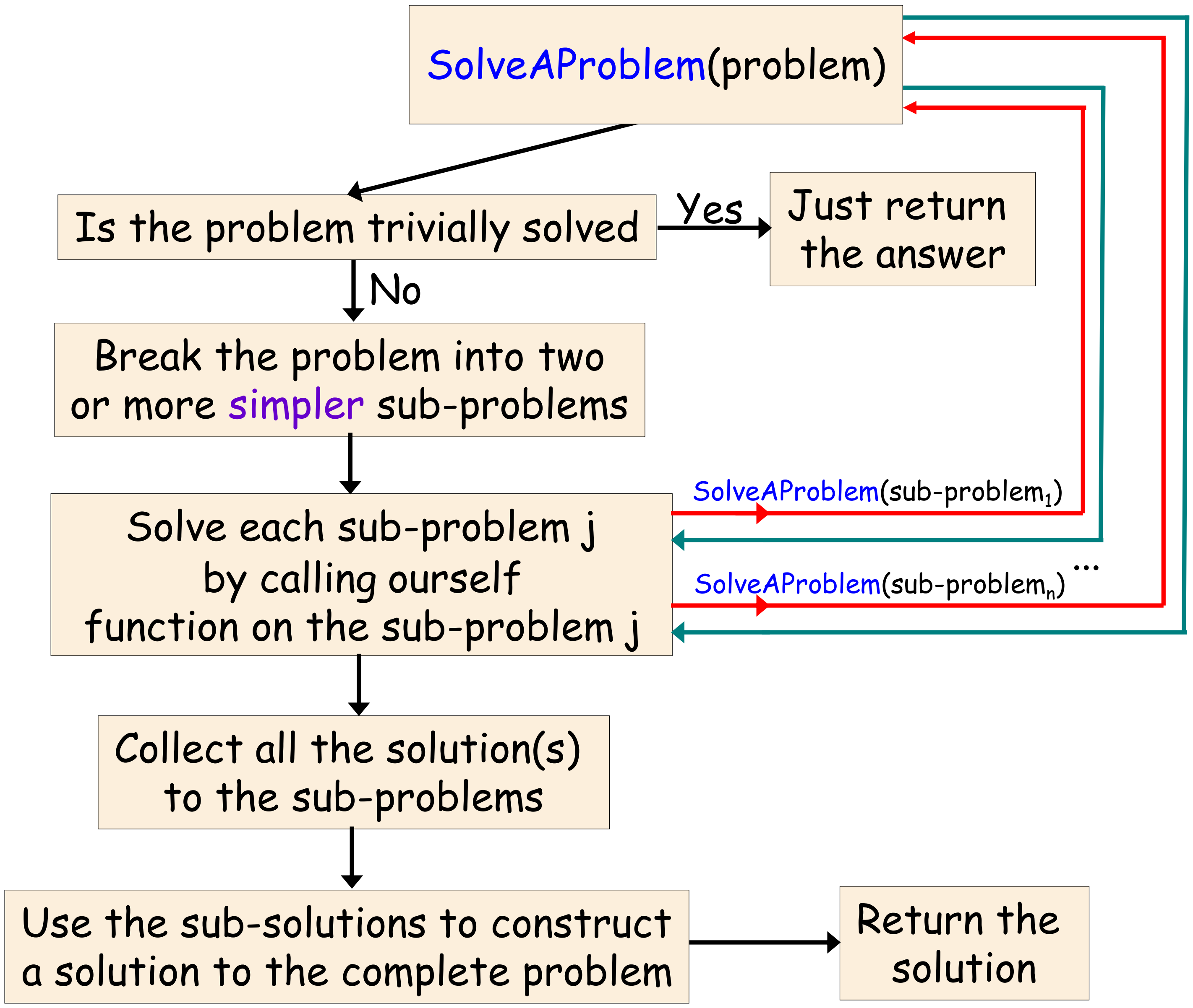
9				6				3
1		5		9	3	2		6
	4			5				9
8						4	7	1
		4	8	7				
7		2	6		1			8
2								
5				3	2		9	4
	8	7		1	6	3	5	

Cracking Codes

And they *love* to ask you to write recursive functions during job interviews.

So pay attention!

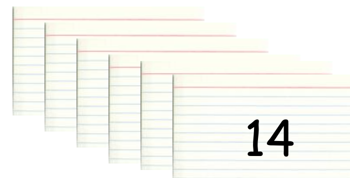
Idea Behind Recursion



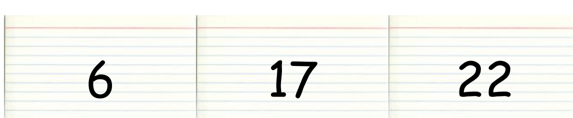
"The Lazy Person's Sort"

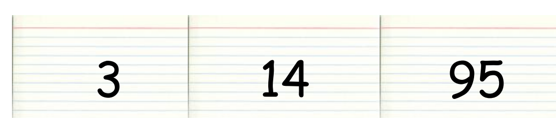
Let's design a new sorting algorithm, called the "lazy person's sort"…

The input to this sort are a bunch of index cards with #s.

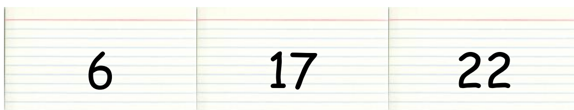


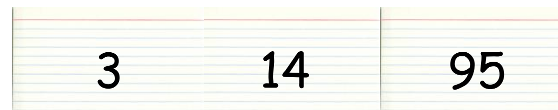
Lazy Person's Sort:
- Split the cards into two roughly-equal piles
- Hand one pile to nerdy student A and ask them to sort it
- Hand the other pile to nerdy student B and ask them to sort it
- Take the two sorted piles and merge them into a single sorted pile





"The Lazy Person's Sort"

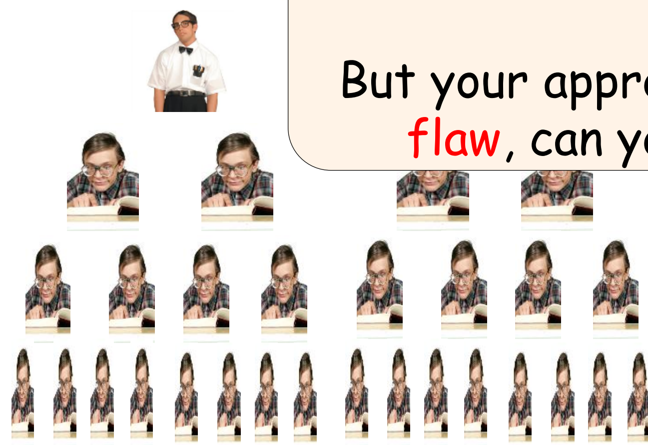




Lazy Person's Sort:

- Split the cards into two roughly-equal piles
- Hand one pile to nerdy student A and ask them to sort it
- Hand the other pile to nerdy student B and ask them to sort it
- Take the two sorted piles and merge them into a single sorted pile

"The Lazy Person's Sort"



Very clever, students.

But your approach has one flaw, can you see it?

Lazy Person's Sort:
- Split the cards into two roughly-equal piles
- Hand one pile to nerdy student A and say "do the Lazy Person's Sort"
- Hand the other pile to hot student B and say "do the Lazy Person's Sort"
- Take the two sorted piles and merge them into a single sorted pile

"The Lazy Person's Sort"

Correct!

Amazing, huh? By having an algorithm use itself over and over, you can solve big problems!

Lazy Person's Sort:

If you're handed just one card, then just give it right back.
Split the cards into two roughly-equal piles
Hand one pile to studly student A and say "do the Lazy Person's Sort"
Hand the other pile to hot student B and say "do the Lazy Person's Sort"
Take the two sorted piles and merge them into a single sorted pile

```
void MergeSort(an array)
{
  if (array's size == 1)
     return;                              // array has just 1 item, all done!

  MergeSort( first half of array );      // process the 1st half of the array
  MergeSort( second half of array);      // process the 2nd half of the array

  Merge(the two array halves);           // merge the two sorted halves
  // now the complete array is sorted
}
```

The Lazy Person's Sort (also known as Merge Sort) is a perfect example of a recursive algorithm!

Every time our MergeSort function is called, it breaks up its input into two smaller parts and calls itself to solve each sub-part.

When you write a recursive function…

Your job is to figure out how the function can use itself (on a subset of the problem) to get the complete problem solved.

When you add the code to make a function call itself, you need to have faith that that call will work properly (on the subset of data).

It takes some time to learn to think in this way, but once you "get it," you'll be a programming Ninja!

The Two Rules of Recursion

RULE ONE:

Every recursive function must have a "stopping condition!"

The Stopping Condition (aka Base Case):

Your recursive function must be able to solve the simplest, most basic problem *without using recursion*.

Remember: A recursive function calls itself.

Therefore, every recursive function must have some mechanism to allow it to stop calling itself.

The Stopping Condition

```
void eatCandy(int layer)
{
  if (layer == 0)
  {
    cout << "Eat center!";
    return;
  }

  cout<<"Lick layer "<<layer;

  eatCandy(layer-1);
}
```

```
main()
{
  eatCandy(3);
}
```

Here's a simple recursive function that shows how to eat a tootsie-roll pop.

Can you identify the stopping condition in this function?

What if we didn't have this stopping condition/base case?

Right! Our function would never stop running.
(We'd just keep licking forever)

The Two Rules of Recursion

RULE TWO:

Every recursive function must have a "simplifying step".

Simplifying Step:

Every time a recursive function calls itself, it must pass in a smaller sub-problem that ensures the algorithm will eventually reach its stopping condition.

Remember: A recursive function must eventually reach its stopping condition or it'll run forever.

Simplifying Code

```
void eatCandy(int layer)
{
  if (layer == 0)
  {
    cout << "Eat center!";
    return;
  }

  cout<<"Lick layer "<<layer;

  eatCandy(layer-1);
}
```

```
main()
{
  eatCandy(3);
}
```

Can you identify the simplifying code in our eatCandy function?

What if we didn't have simplifying code?

Our function would never get closer to our stopping condition and never stop running.

Most recursive functions simplify their inputs in one of two ways:

1. Each recursive call divides its input problem in half (like MergeSort)
2. Each recursive call operates on an input that's one smaller than the last

(Rule 2.5 of Recursion)

Recursive functions should never use global, static or member variables.

They should only use local variables and parameters!

(So be forewarned... If your recursive functions use globals/statics/members on a test/HW, you'll get a ZERO!)

Tracing Through our Function

```
void eatCandy(int layer)
{
  if (layer == 0)
  {
    cout << “Eat center!”;
    return;
  }
  cout<<“Lick layer “<<layer;

  eatCandy(layer-1);
}
```

```
void eatCandy(int layer)
{
  if (layer == 0)
  {
    cout << “Eat center!”;
    return;
  }
  cout<<“Lick layer “<<layer;

  eatCandy(layer-1);
}
```

```
void eatCandy(int layer)
{
  if (layer == 0)
  {
    cout << “Eat center!”;
    return;
  }
  cout<<“Lick layer “<<layer;

  eatCandy(layer-1);
}
```

It's very difficult to trace through a function that calls itself…

So, let's use a little trick and pretend like this call is actually calling a different function

(one that just happens to have the same name ☺).

```
main()
{
  int layers = 2;

  eatCandy(layers);
}
```

Writing (Your Own) Recursive Functions: 6 Steps

What if we want to write our own recursive function?

Here's a proven six-step method to help you!

Step #1:
Write the function header

Step #2:
Define your magic function

Step #3:
Add your base case code

Step #4:
Solve the problem w/the magic function

Step #5:
Remove the magic

Step #6:
Validate your function

Let's use these steps to write a recursive function to calculate factorials.

Recall, the definition of fact(N) is:

1	for N = 0
N * fact (N-1)	for N > 0

Example #1: Factorial

Step #1: Write the function header

Figure out what argument(s) your function will take and what it needs to return (if anything).

First, a factorial function takes in an integer as a parameter, e.g., factorial(6).

Second, the factorial computes (and should return) an integer result. Let's add a return type of int.

And here's how we'd call our factorial function to solve a problem of size n…

So far, so good. Let's go on to step #2.

```
int fact(int n)
{

}
```

```
int main()
{
    int n = 6, result;

    result = fact( n );

}
```

Step #2: Define your magic function

Pretend that you are given a magic function that can compute a factorial. It's already been written for you and is guaranteed to work!

It takes the same parameters as your factorial function and returns the same type of result/value.

There's only one catch! You are forbidden from passing in a value of n to this magic function.

So you can't use it to compute n!

But you can use it to solve smaller problems, like (n-1)! or (n/2)!, etc.

Show how you could use this magic function to compute (n-1)!.

```
// provided for your use!
int magicfact(int x) { … }

int fact(int n)
{

}
```

```
int main()
{
     int n = 6, result;
     // use magicfact to solve subproblems
     result = magicfact( n-1 );

}
```

Step #3: Add your base case Code

Determine your base case(s) and write the code to handle them *without recursion!*

Our goal in this step is to identify the simplest possible input(s) to our function…

And then have our function process those input(s) without calling itself (i.e., just like a normal function would).

Ok, so what is the simplest factorial we might be asked to compute?

Well, the user could pass 0 into our function.

0!, by definition, is equal to 1.

Let's add a check for this and handle it *without using any recursion*.

In this example, this is the only base condition, but some problems may require 2 or 3 different checks.

```
// provided for your use!
int magicfact(int x) { … }

int fact(int n)
{
   if (n == 0)
      return 1;  // base case

   // Always consider all possible
   // base cases and add checks
   // for them before proceeding!
}
```

```
int main()
{
    int n = 6, result;
    // use magicfact to solve subproblems
    result = magicfact( n-1 );

}
```

Step #4: Solve the problem using the magic function

Now try to figure out how to use the magic function in your new function to help you solve the problem.

Unfortunately, you can't use the magic function to do all the work for you... (it can't solve problems of size n)

So let's try to break our problem into two (or more) simpler sub-problems and use our magic function to solve those.

Well, by definition, N! = N * (N-1)!
So it's already split into two parts for us, & each part is simpler than the original problem.

Let's figure out a way to solve each of these sub-problems.

Cool! Now we can combine the results of our sub-problems to get the overall result!

```
// provided for your use!
int magicfact(int x) { … }

int fact(int n)
{
    if (n == 0)
        return 1;  // base case

    int part1 = n;
    int part2 = magicfact( n-1 );

    return part1 * part2;
}
```

```
int main()
{
    int n = 6, result;

    // use magicfact to solve subproblems
    result = magicfact( n-1 );

}
```

Step #5: Remove the m

```
int magicfact(int x)
{
    return fact(x);
}
```

OK, so let's see what this magic function really looks like!

Wait a second! Our magicfact function basically just calls fact!

That means that fact is really just calling itself!

The magicfact function hid this from us, but that's what's really happening!

OK, well in that case, let's replace our call(s) to the magic function with calls directly to our own function.

Will that work? Yup!

Woohoo! We've just created our first recursive function from scratch!


```
// provided for your use!
int magicfact(int x) { … }

int fact(int n)
{
    if (n == 0)
        return 1;  // base case

    int part1 = n;
    int part2 = magicfact( n-1 );

    return part1 * part2;
}
```

```
int main()
{
    int n = 6, result;

    // use magicfact to solve subproblems
    result = magicfact( n-1 );

}
```

Step #6: Validating our Function

You SHOULD do this step EVERY time your write a recursive function!

Start by testing your function with the simplest possible input.

Next test your function with incrementally more complex inputs.
(You can usually stop once you've validated at least one recursive call)

```
int fact(int n)
{
  if (n == 0)
     return 1;

  return  n * fact(n-1);
}
```

Excellent! We've tested all of the base case(s) as well as validated a single level of recursion…

We can be pretty certain our function works now…

```
int fact(int n)
{
  if (n == 0)
     return 1;

  return  n * fact(n-1);
}
```

```
int main()
{
  cout << fact( 0 );
  cout << fact( 1 );
}
```

Factorial Trace-through

```
int fact(int n)          n  0
{
  if (n == 0)
     return (1);

  return(n * fact(n-1));
}
```

```
int fact(int n)          n  1
{
  if (n == 0)
     return (1);

  return(n * fact(n-1));
}
```

```
int fact(int n)          n  2
{
  if (n == 0)
     return (1);

  return(n * fact(n-1));
}
```

```
int main()
{
  int result;
  result = fact(2);
  cout << result;
}
```

Example #2: Recursion on an Array

For our next example, let's learn how to use recursion to get the sum of all the items in an array.

arr	10	20	70	14	39
	0				n-1

Step #1: Write the

Figure out what argument(s) your function will take and what it needs to return (if anything).

To sum up all of the items in an array, we need a pointer to the array and its size.

Our function will return the total sum of items in the array, so we can make the return type an int.

And here's how we'd call our array-summer function to solve a problem of size n…

So far, so good.

Let's go on to step #2.

You could also have written:
int *arr
It's the same thing!

```
int sumArr(int arr[ ], int n)
{

}
```

```
int main()
{
    const int n = 5;
    int arr[n] = { 10, 100, 42, 72, 16}, s;

    s = sumArr( arr , n);           // whole array

}
```

Step #2: Define your magic function

Pretend that you are given a magic function that sums up the values in an array and returns the result…

There's only one catch! You are forbidden from passing in an array with n elements to this function.

So you can't use it to sum up an entire array (one with all n items)…

But you can use it to sum up smaller arrays (e.g., with n-1 elements)!

Show how to use the magic function to sum the first n-1 items of the array.

Now show how to use the magic function to sum the last n-1 items of the array.

Now show how to use the magic function to sum the first half of the array.

Finally show how to use the magic function to sum the last half of the array.

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{

}
```

```
int main()
{
    const int n = 5;
    int arr[n] = { 10, 100, 42, 72, 16}, s;

    s = magicsumArr( arr, n-1 );  // first n-1
    s = magicsumArr( arr+1, n-1 );  // last n-1
    s = magicsumArr( arr, n/2 );  // sums 1st half
    s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

Step #3: Add your base case Code

Determine your base case(s) and write the code to handle them *without recursion!*

Ok, so what is the smallest array that might be passed into our function?

Well, someone could pass in a totally empty array of size n = 0. What should we do in that case?

Well, what's the sum of an empty array? Obviously it's zero. Let's add the code to deal with this case.

Do we have any other base cases? For example, what if the user passes in an array with just one element?

Let's see what that would look like…

Good. Let's keep both of those.

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{
   if (n == 0)  return 0;
   if (n == 1)  return arr[0];


}
```

```
int main()
{
    const int n = 5;
    int arr[n] = { 10, 100, 42, 72, 16}, s;

    s = magicsumArr( arr, n-1 );  // first n-1
    s = magicsumArr( arr+1, n-1 );  // last n-1
    s = magicsumArr( arr, n/2 );  // sums 1st half
    s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

Step #4: Solve the problem using the magic function

Now try to figure out how to use the magic function in your new function to help you solve the problem.

Unfortunately, you can't use the magic function to do all the work for you... (it can't solve problems of size n)

So let's try to break our problem into two (or more) simpler sub-problems and use our magic function to solve those.

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{
  if (n == 0)  return 0;
  if (n == 1)  return arr[0];

  int s = magicsumArr( arr, n );

  return s;

}
```

```
int main()
{
   const int n = 5;
   int arr[n] = { 10, 100, 42, 72, 16}, s;

   s = magicsumArr( arr, n-1 );  // first n-1
   s = magicsumArr( arr+1, n-1 );  // last n-1
   s = magicsumArr( arr, n/2 );  // sums 1st half
   s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

Step #4: Solve the problem using the magic function

Now try to figure out how to use the magic function in your new function to help you solve the problem.

Unfortunately, you can't use the magic function to do all the work for you... (it can't solve problems of size n)

So let's try to break our problem into two (or more) simpler sub-problems and use our magic function to solve those.

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{
  if (n == 0)  return 0;
  if (n == 1)  return arr[0];

  int front = magicsumArr( arr, n-1 );

  int total = front + a[n-1];

  return total;
}
```

Strategy #1: Front to back

Your function uses the magic function to process the first n-1 elements of the array, ignoring the last element.

Once it gets the result from the magic function, it combines it with the last element in the array.

It then returns the full result.

```
int main()
{
   const int n = 5;
   int arr[n] = { 10, 100, 42, 72, 16}, s;

   s = magicsumArr( arr, n-1 );  // first n-1
   s = magicsumArr( arr+1, n-1 );  // last n-1
   s = magicsumArr( arr, n/2 );  // sums 1st half
   s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

Step #4: Solve the problem using the magic function

Now try to figure out how to use the magic function in your new function to help you solve the problem.

Unfortunately, you can't use the magic function to do all the work for you… (it can't solve problems of size n)

So let's try to break our problem into two (or more) simpler sub-problems and use our magic function to solve those.

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{
  if (n == 0)  return 0;
  if (n == 1)  return arr[0];

  int rear = magicsumArr( arr+1, n-1 );

  int total = a[0] + rear;

  return total;
}
```

Strategy #2: Back to front

Your function uses the magic function to process the last n-1 elements of the array, ignoring the first element.

Once it gets the result from the magic function, it combines it with the first element in the array.

It then returns the full result.

```
int main()
{
    const int n = 5;
    int arr[n] = { 10, 100, 42, 72, 16}, s;

    s = magicsumArr( arr, n-1 );  // first n-1
    s = magicsumArr( arr+1, n-1 );  // last n-1
    s = magicsumArr( arr, n/2 );  // sums 1st half
    s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

Step #4: Solve the problem using the magic function

Now try to figure out how to use the magic function in your new function to help you solve the problem.

Unfortunately, you can't use the magic function to do all the work for you… (it can't solve problems of size n)

So let's try to break our problem into two (or more) simpler sub-problems and use our magic function to solve those.

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{
  if (n == 0)  return 0;
  if (n == 1)  return arr[0];

  int first = magicsumArr( arr, n/2 );

  int last = magicsumArr( arr+n/2,
                          n - n/2 );
  return first + last;
}
```

Strategy #3: Divide and conquer

Your function uses the magic function to process the first half of the array.

Your function uses the magic function to process the last half of the array.

Once it gets both results, it combines them and returns the full result.

```
int main()
{
   const int n = 5;
   int arr[n] = { 10, 100, 42, 72, 16}, s;

   s = magicsumArr( arr, n-1 );  // first n-1
   s = magicsumArr( arr+1, n-1 );  // last n-1
   s = magicsumArr( arr, n/2 );  // sums 1st half
   s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

```
int magicsumArr(int arr[], int x)
{
    return sumArr(arr,x);
}
```

...ve the magic

```
// provided for your use!
int magicsumArr(int arr[], int x) { … }

int sumArr(int arr[ ], int n)
{
   if (n == 0)  return 0;
   if (n == 1)  return arr[0];

   int first = magicsumArr( arr, n/2 );
   int scnd = magicsumArr( arr+n/2,
                           n - n/2 );
   return first + scnd;
}
```

OK, so let's see what this magic function really looks like!

Wait a second! Our magicsumArr function just calls sumArr!

This means that sumArr is really just calling itself!

The magic function hid this from us, but that's what's really happening!

OK, well in that case, let's replace our calls to the magic function with calls directly to our own function.

Will that work? Yup!

Woohoo! We've just created our second recursive function!

```
int main()
{
    const int n = 5;
    int arr[n] = { 10, 100, 42, 72, 16}, s;

    s = magicsumArr( arr, n-1 );  // first n-1
    s = magicsumArr( arr+1, n-1 );  // last n-1
    s = magicsumArr( arr, n/2 );  // sums 1st half
    s = magicsumArr( arr+n/2, n - n/2 ); // 2nd
}
```

Step #6: Validating our Function

As before, make sure to test your function with at least one input that exercises the base case...

and one input that causes a recursive call.

```
int sumArr(int arr[], int n)
{
    if (n == 0) return 0;
    if (n == 1)  return arr[0];
    int first =   sumArr( arr, n/2 );
    int scnd = sumArr( arr+n/2, n-n/2);
    return first + scnd;
}
```

```
int main()
{
  int arr[2] = { 10, 20 };

  cout << sumArr( arr, 0 );

  cout << sumArr( arr, 2 );
}
```

Array-summer Trace-through

```
int sumArr(int arr[], int n)
{
    if (n == 0) return 0;
    if (n == 1)  return arr[0];
    int first =   sumArr( arr, n/2 );
    int scnd = sumArr( arr+n/2, n-n/2);
    return first + scnd;
}
```

```
int sumArr(int arr[], int n)
{
    if (n == 0) return 0;
    if (n == 1)  return arr[0];
    int first =   sumArr( arr, n/2 );
    int scnd = sumArr( arr+n/2, n-n/2);
    return first + scnd;
}
```

```
int sumArr(int arr[], int n)
{
    if (n == 0) return 0;
    if (n == 1)  return arr[0];
    int first =   sumArr( arr, n/2 );
    int scnd = sumArr( arr+n/2, n-n/2);
    return first + scnd;
}
```

```
int main()
{
    const int n = 3;
    int nums[n] = { 10, 20, 42 };


    cout << sumArr( nums , n );
}
```

Your Turn: Recursion Challenge

Write a recursive function called printArr that prints out an array of integers in reverse from bottom to top.

Step #1:
Write the function header

Step #2:
Define your magic function

Step #3:
Add your base case code

Step #4:
Solve the problem w/the magic function

Step #5:
Remove the magic

Step #6:
Validate your function

Recursion Challenge

Step #1: Write the function header

Step #2: Define your magic function

Step #3: Add your base case code

Step #4: Solve the problem using your magic function

Step #5: Remove the magic

Step #6: Validate your function

Write a recursive function called printArr that prints out an array from bottom to top.

```
int main()
{
   const int size = 5;
   int arr[size] = {7, 9, 6, 2, 4};



}
```

Recursion

```
void reversePrint(string arr[ ], int size)
{
    if (size == 0) // an empty array
        return;
    else
    {
        reversePrint(arr + 1, size - 1 );
        cout << arr[0] << "\n";
    }
}
```

```
{
    if (size == 0) // an empty array
        return;
    else
    {
        reversePrint(arr + 1, size - 1 );
        cout << arr[0] << "\n";
    }
}
```

```
{
    if (size == 0) // an empty array
        return;
    else
    {
        reversePrint(arr + 1,  size - 1 );
        cout << arr[0] << "\n";
    }
}
```

names

[0]	Leslie	2000
[1]	Phyllis	2020
[2]	Nan	2040

```
main()
{
    string names[3];
    ...
    reversePrint(names,3);
}
```

Recursion

```
void reversePrint(string arr[ ], int size)
{
    if (size == 0) // an empty array
        return;
    else
    {
        reversePrint(arr + 1, size - 1 );
        cout << arr[0] << "\n";
    }
}
```

arr	2040
size	1

```
v
{
    if (size == 0) // an empty array
        return;
    else
    {
        reversePrint(arr + 1, size - 1 );
        cout << arr[0] << "\n";
    }
}
```

arr	2020
size	2

```
{
    if (size == 0) // an empty array
        return;
    else
    {
        reversePrint(arr + 1,  size - 1 );
        cout << arr[0] << "\n";
    }
}
```

arr	2000
size	3

names

[0]	Leslie	2000
[1]	Phyllis	2020
[2]	Nan	2040

```
main()
{
    string names[3];
    ...
    reversePrint(names,3);
}
```

Recursion

```
void reversePrint(string arr[ ], int size)
{
   if (size == 0) // an empty array

      reversePrint(a      + 1, size - 1 );
      cout << arr[0] << "\n";
   }
}
```

arr [0] is
" Nan "

arr 2040
size 1

```
{
   if (size == 0) // an empty array

   {
      reversePrint(a      + 1, size - 1 );
      cout << arr[0] << "\n";
   }
}
```

arr [0] is
" Phyllis "

arr 2020
size 2

```
{
   if (size == 0) // an empty array

   el
   {
      reversePrint(ar      1,  size - 1 );
      cout << arr[0] << "\n";
   }
}
```

arr [0] is
" Leslie "

arr 2000
size 3

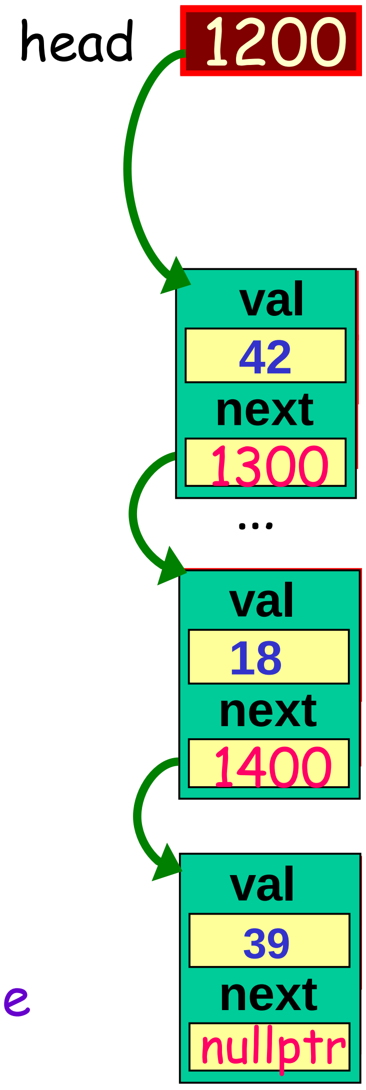
names		
[0]	Leslie	2000
[1]	Phyllis	2020
[2]	Nan	2040

```
main()
{
   string names[3];
   ...
                   2000  3
   reversePrint(names,3);
}
```

Example #3: Recursion on a Linked List

When we process a linked list using recursion, it's very much like processing an array using strategy #2!

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



There are two differences:

1. Instead of passing in a pointer to an array element, you pass in a pointer to a node
2. You don't need to pass in a size value for your list (this is determined via the next pointers)

Let's see an example. We'll write a function that finds the biggest number in a NON-EMPTY linked list.

Step #1: Write the function header

Figure out what argument(s) your function will take and what it needs to return (if anything).

To find the biggest item in a linked list, what kind of parameter should we pass to our function?

Right! All we need to pass in is a pointer to a node of the linked list.

Our function will return the biggest value in the list, so we can make the return type an int.

So far, so good. Let's go on to step #2.

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

int biggest( Node *cur  )
{



}
```

Step #2: Define your magic function

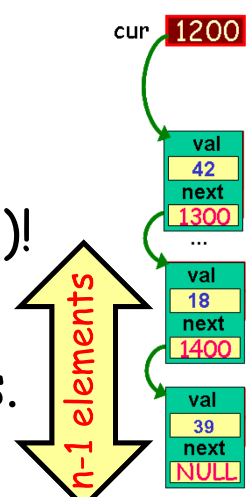
Pretend that you are given a magic function that finds the biggest value in a linked list and returns it…

There's only one catch! You are forbidden from passing in a full linked list with all n elements to this function.

So you can't use it to find the biggest item in the entire list (one with all n items)…

But you can use it to find the biggest item in a partial list (e.g., with n-1 elements)!

Let's see how to do this.




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

// provided for your use!
int magicbiggest(Node *n) { … }

int biggest( Node *cur  )
{

}
```

```
int main()
{
  Node *cur = createLinkedList();
  int biggest = magicbiggest(cur->next);
}
```

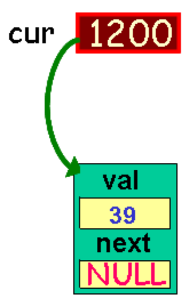
Step #3: Add your base case Code

Determine your base case(s) and write the code to handle them *without recursion!*

For this problem, we're assuming that the user must pass in a linked list with at least one element.

So, what's the simplest case that our function must handle?

Well, if a linked list has only one node...



Then *by definition* that node *must* hold the biggest (only!) value in the list, right?

Are there any other base cases?

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

// provided for your use!
int magicbiggest(Node *n) { … }

int biggest( Node *cur )
{
   if (cur->next == nullptr)  // the only node
      return cur->val;    // so return its value
}
```

```
int main()
{
  Node *cur = createLinkedList();
  int biggest = magicbiggest(cur->next);
}
```

Step #4: Solve the problem using the magic function

Now try to figure out how to use the magic function in your new function to help you solve the problem.

Unfortunately, you can't use the magic function to process all n nodes of the list.

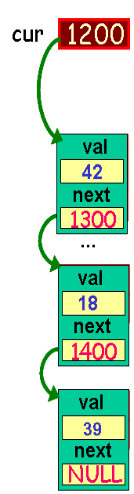
So let's break our problem into two (or more) simpler sub-problems and use our magic function to solve those.

Strategy for Linked Lists:

Use the magic function to process the last n-1 elements of the list, ignoring the first element.

Once you get a result from the magic function for the last n-1 nodes, combine it with the first element in the list.

Then return the full result.




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

// provided for your use!
int magicbiggest(Node *n) { … }

int biggest( Node *cur )
{
   if (cur->next == nullptr)  // the only node
       return cur->val;    // so return its value

   int rest = magicbiggest(cur->next);

   // pick biggest of 1st node and last n-1 nodes
   return max( rest , cur->val );
}
```

```
int main()
{
   Node *cur = createLinkedList();

   int biggest = magicbiggest(cur->next);
}
```

Step #5: Remove th

```
int magicbiggest(Node *n)
{
    return biggest(n);
}
```

OK, so let's see what this magic function really looks like!

Wait a second! Our magicbiggest function just calls biggest!

That means our biggest function is really just calling itself!

The magic function hid this from us, but that's what's really happening!

OK, well in that case, let's replace our calls to the magic function with calls directly to our own function.

Will that work? Yup!

Woohoo! We've just created our third recursive function!

```
    No      xt;
};

// provide  for your use!
int magicbiggest(Node *n) { … }

int biggest( Node *cur )
{
  if (cur->next == nullptr)  // the only node
      return cur->val;    // so return its value

  int rest = magicbiggest(cur->next);
  // pick biggest of 1st node and last n-1 nodes
  return max( rest , cur->val );
}
```

```
int main()
{
  Node *cur = createLinkedList();
  int biggest = magicbiggest(cur->next);
}
```

Step #6: Validating our Function

Don't forget to test your function!

Since the problem states that the list is never empty…

The two simplest test cases would be a
one-node list and a two-node list!

```
int biggest( Node *cur )
{
   if (cur->next == nullptr)  // the only node
      return cur->val;    // so return its value

   int rest = biggest( cur->next );

   return max ( rest, cur->val );
}
```

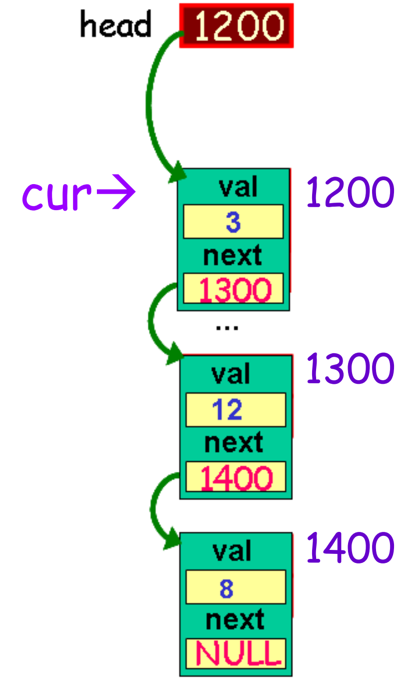
```
int main()
{
   Node *head1 = mkLstWith1Item();

   cout << biggest( head1 );

   Node *head2 = mkLstWith2Items();

   cout << biggest( head2 );
}
```

Biggest-in-List Trace-through

```
int biggest(Node *cur)
{
    if (cur->next == nullptr)
        return(cur->val);

    int rest = biggest( cur->next );

    return max( rest, cur-val );
}
```

```
int biggest(Node *cur)
{
    if (cur->next == nullptr)
        return(cur->val);

    int rest = biggest( cur->next );

    return max( rest, cur-val );
}
```

```
int biggest(Node *cur)
{
    if (cur->next == nullptr)
        return(cur->val);

    int rest = biggest( cur->next );

    return max( rest, cur-val );
}
```




```
main()
{
    Node *head;
    ...   // create linked list
    cout << biggest(head);
}
```

Writing Recursive Functions: A Critical Tip!

Your recursive function should generally only access the current node/array cell passed into it!

Your recursive function should rarely/never access the value(s) in the node(s)/cell(s) below it!

```
// good examples!
int recursiveGood(Node *p)
{
   ...
   if (p->value == someValue)
      do something;

  if (p == nullptr || p->next == nullptr)
      do something;

   int v = p->value +
         recursiveGood(p->next);

   if (p->value > recursiveGood(p->next))
      do something;
}
```

```
// bad examples!!!
int recursiveBad(Node *p)
{
   ...
   if (p->next->value == someValue)
      do something;

  if (p->next->next == nullptr)
      do something;

  int v = p->value + p->next->value +
         recursiveBad(p->next->next);

  if (p->value > p->next->value)
      do something;
}
```

Writing Recursive Functions: A Critical Tip!

Your recursive function should generally only access the current node/array cell passed into it!

Your recursive function should rarely/never access the value(s) in the node(s)/cell(s) below it!

```
// good examples!
int recursiveGood(int a[], int count)
{
   ...
   if (count == 0 || count == 1)
     do something;

   if (a[0] == someValue)
     do something;

   int v = a[0] +
         recursiveGood(a+1, count-1);

   if (a[0] > recursiveGood(a+1, count-1))
     do something;
}
```

```
// bad examples!!!
int recursiveBad(int a[], int count)
{
   ...
   if (count == 2)
     do something;

   if (a[1] == someValue)
     do something;

   int v = a[0] + a[1] +
         recursiveBad(a+2,count-2);

   if (a[0] > a[1])
     recursiveBad(a+2,count-2);
}
```

Recursion Challenge #2

Write a recursive function called count that counts the number of times a number appears in an array.

```
main()
{
    const int size = 5;
    int arr[size] = {7, 9, 6, 7, 7};
    cout << countNums(arr,size,7);
    // should print 3
}
```

Recursion Challenge #2

Step #1: Write the function header

Step #2: Define your magic function

Step #3: Add your base case code

Step #4: Solve the problem using your magic function

Step #5: Remove the magic!

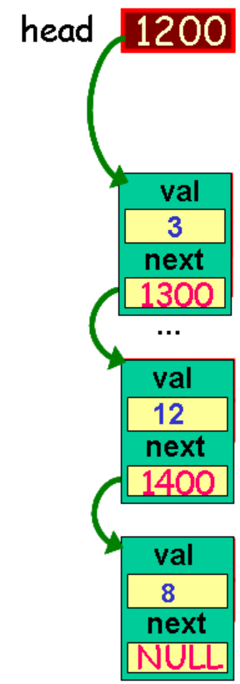
Step #6: Validate your function

Write a recursive function called count that counts the number of times a number appears in an array.

Recursion Challenge #3

Write a function that finds and returns the earliest position of a number in a linked list. If the number is not in the list or the list is empty, your function should return -1 to indicate this.

```
main()
{
   Node *cur = <make a linked list>;

   cout << findPos(cur,3);  // prints 0
   cout << findPos(cur,8);  // prints 2
   cout << findPos(cur,19); // prints -1
}
```



Step #1: Write the function header
Step #2: Define your magic function
Step #3: Add your base case code
Step #4: Solve the problem using your magic function
Step #5: Remove the magic!
Step #6: Validate your function

Recursion Challenge #3

Write a function that finds and returns the earliest position of a number in a linked list. If the number is not in the list or the list is empty, your function should return -1 to indicate this.

Let's see some REAL examples!

Ok, enough with the toy recursion examples!

Let's see some situations where recursion really shines!

Recursion: Binary Search

Goal: Search a *sorted* array of data for a particular item.

Idea: Use recursion and a divide-and-conquer approach!

Pseudocode:

```
bool Search(sortedArray, findMe)
{
  if (array.size == 0) // empty array
    return false;

  middle_word = sortedArray.size / 2;
  if (findMe == sortedArray[middle_word])
    return true;

  if (findMe < sortedArray[middle_word])
    return Search( first ½ of sortedArray );
  else // findWord > middle_word
    return Search( second ½ of sortedArray );
}
```

Notice how Binary Search code recurses on either the first half *or* the second half of the array… But never both. This is for efficiency.

Binary Search: C++ Code

Here's a real binary search implementation in C++. Let's see how it works!

```
int BS(string A[], int top, int bot, string f)
{
   if (numItemsBetween(top, bot) == 0)
      return (-1);     // Value not found
   else
   {
      int Mid = (top + bot) / 2;
      if (f == A[Mid])
          return Mid; // found - return where!
      else if (f < A[Mid])
          return BS(A,top,Mid - 1,f);
      else if (f > A[Mid])
          return BS(A, Mid + 1,bot,f);
    }
}
```

Recurs ... ch

As our binary search progresses, top will move down and bot will move up.

If the # of items between them is zero, it means our value was not in the array.

top →	0	Albert
	1	Brandy
	2	Carol
	3	David
	4	Eugene
Mid →	5	Frank
	6	Gordon
	7	Grendel
	8	Hank
	9	Wayne
bot →	10	Yentle

```
int BS(string A[], int top, int bot, string f)
{
  if (numItemsBetween(top, bot) == 0)
    return (-1);     // Value not found
  else
  {
    int Mid = (top + bot) / 2;
    if (f == A[Mid])
       return Mid; // found - return where!
    else if (f < A[Mid])
       return BS(A,top,Mid - 1,f);
    else if (f > A[Mid])
       return BS(A, Mid + 1,bot,f);
   }
}
```

```
= {"Albert",…};
,"David") != -1)
 it!";
}
```

Recursion: Binary Search

```
int BS(string A[], int top, int bot, string f)
{
  if (numItemsBetween(top, bot) == 0)
    return (-1);     // Value not found
  else
  {
    int Mid = (top + bot) / 2;
    if (f == A[Mid])
       return Mid; // found - return where!
    else if (f < A[Mid])
       return BS(A, top, Mid - 1,f);
    else if (f > A[Mid])
       return BS(A, Mid + 1,bot,f);
   }
}
```

```
         return BS(A,top,Mid - 1,f);
     else if (f > A[Mid])
         return BS(A, Mid + 1,bot,f);
   }
}
```

```
= {"Albert",…};
,"David") != -1)
 it!";
```

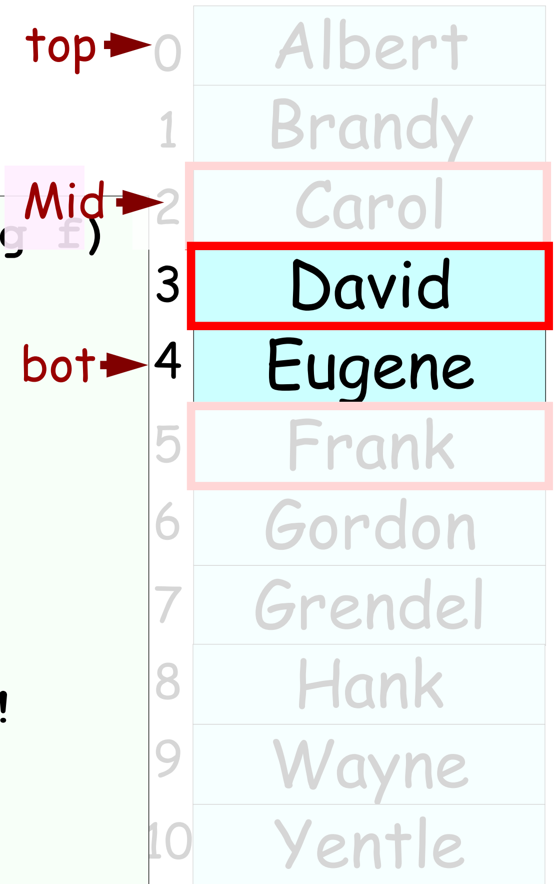
top	0	Albert
	1	Brandy
Mid	2	Carol
	3	David
	4	Eugene
Mid	5	Frank
	6	Gordon
	7	Grendel
	8	Hank
	9	Wayne
bot	10	Yentle

Recursion: Binary Search

```
int BS(string A[], int top, int bot, string f)
{
  if (numItemsBetween(top, bot) == 0)
    return (-1);    // Value not found
  else
  {
    int Mid = (top + bot) / 2;
    if (f == A[Mid])
       return Mid; // found - return where!
    else if (f < A[Mid])
       return BS(A, top, Mid - 1,f);
    else if (f > A[Mid])
       return BS(A, Mid + 1,bot,f);
   }
}
```

```
    return                        ;
  }
}
}
```

```
{"Albert",…};
"David") != -1)
it!";
```



Recursion: Binary Search

top ►	0	Albert
	1	Brandy
Mid ►	2	Carol
	3	David
bot ►	4	Eugene
	5	Frank
	6	Gordon
	7	Grendel
	8	Hank
	9	Wayne
	10	Yentle

```
int BS(string A[], int top, int bot, string f)
{
  if (numItemsBetween(top, bot) == 0)
    return (-1);     // Value not found
  else
  {
    int Mid = (top + bot) / 2;
    if (f == A[Mid])
       return Mid; // found - return where!
    else if (f < A[Mid])
       return BS(A, top, Mid - 1,f);
    else if (f > A[Mid])
       return BS(A, Mid + 1,bot,f);
   }
}
```

```
}
```

```
{"Albert",…};
"David") != -1)
it!";
```

Recursion: Binary Search

top ►	0	Albert
	1	Brandy
	2	Carol
	3	David
	4	Eugene
Mid ►	5	Frank
	6	Gordon
	7	Grendel
	8	Hank
	9	Wayne
bot ►	10	Yentle

```
int BS(string A[], int top, int bot, string f)
{
  if (numItemsBetween(top, bot) == 0)
    return (-1);    // Value not found
  else
  {
    int Mid = (top + bot) / 2;
    if (f == A[Mid])
       return Mid; // found - return where!
    else if (f < A[Mid])
       return BS(A,top,Mid - 1,f);
    else if (f > A[Mid])
       return BS(A, Mid + 1,bot,f);
   }
}
```

```
= {"Albert",…};
,"David") != -1)
 it!";
}
```

Recursion Helper Functions

So we just saw a recursive version of Binary Search:

```
int BS(string A[], int top, int bot, string f)
{
...
}
```

Notice how many crazy parameters it takes?
What is top? What's bot? That's going to be really confusing for the user!

Wouldn't it be nicer if we just provided our user with a simple function (with a few, obvious params) and then hid the complexity?

```
int SimpleBinarySearch(string A[], int size, string findMe)
{
   return BS(A , 0 , size-1 , findMe);
}
```

This simple function can then call the complex recursive function to do the dirty work, without confusing the user.

Solving a Maze

We can also use recursion to find a solution to a maze.

In fact, the recursive solution works in the same basic way as the stack-based solution we saw earlier.

The algorithm uses recursion to keep moving down paths until it hits a dead end.

Once it hits a dead end, the function returns until it finds another path to try.

This approach is called "backtracking" or "depth first search."

Solving a Maze

```
void solve(int sx, int sy)
{
  m[sy][sx] = '#'; // drop crumb
  if (sx == dx && sy == dy)
    solveable = true; // done!
  if (m[sy-1][sx] == ' ')
    solve(sx,sy-1);
  if (m[sy+1][sx] == ' ')
    solve(sx,sy+1);
  if (m[sy][sx-1] == ' ')
    solve(sx-1,sy);
  if (m[sy][sx+1] == ' ')
    solve(sx+1,sy);
}
```

```
vo
{
    solve(sx,sy+1);
  if (m[sy][sx-1] == ' ')
    solve(sx-1,sy);
  if (m[sy][sx+1] == ' ')
    solve(sx+1,sy);
}
```

```
bool solvable; // globals
int  dx, dy;
char m[11][11] = {
Start "***********",
      "*         *",
      "* * * **  *",
      "*** *  *  *",
      "* * ** *  *",
      "*    ***  *",
      "*  *   *  *",
      "*  *****  *",
      "*     *   *",
      "********** *"
  };              Finish

main()
{
   solvable = false;
   dx = dy = 10;

   solve(1,1);
   if (solvable == true)
     cout << "possible!";
};
```

```
void solve(int sx, int sy)
{
  m[sy][sx] = '#'; // drop crumb
  if (sx == dx && sy == dy)
    solveable = true; // done!
  if (m[sy-1][sx] == ' ')
    so
  if (
    solve(sx,sy+1);
  if (m[sy][sx-1] == ' ')
    solve(sx-1,sy);
  if (m[sy][sx+1] == ' ')
    solve(sx+1,sy);
}
```

And on it goes...

```
v
{
    solve(sx,sy+1);
  if (m[sy][sx-1] == ' ')
    solve(sx-1,sy);
  if (m[sy][sx+1] == ' ')
    solve(sx+1,sy);
}
```

```
vo
{
    solve(sx+1,sy);
}
```

```
vo
{
    solve(sx+1,sy);
}
```

```
bool solvable; // globals
int  dx, dy;
char m[11][11] = {
art  "**********",
     "*        *",
     "* * * ** *",
     "*** *  * *",
     "* * ** * *",
     "*    *** *",
     "*  *   * *",
     "*  ***** *",
     "*     *  *",
     "**********"
 };                Finish

main()
{
   solvable = false;
   dx = dy = 10;

   solve(1,1);
   if (solvable == true)
     cout << "possible!";
};
```

```
void solve(int sx, int sy)
{
  m[sy][sx] = '#'; // drop crumb
  if (sx == dx && sy == dy)
    solveable = true; // done!
  if (m[sy-1][sx] == ' ')
    so
  if (
    solve(sx,sy+1);
  if (m[sy][sx-1] == ' ')
    solve(sx-1,sy);
  if (m[sy][sx+1] == ' ')
    solve(sx+1,sy);
}
```

And on it goes...

```
v
{
    solve(sx,sy+1);
  if (m[sy][sx-1] == ' ')
    solve(sx-1,sy);
  if (m[sy][sx+1] == ' ')
    solve(sx+1,sy);
}
```

```
vo
{
    solve(sx+1,sy);
}
```

```
vo
{
    solve(sx+1,sy);
}
```

```
bool solvable; // globals
int  dx, dy;
char m[11][11] = {
    "**********",
    "*########*",
    "*#*#*#**#*",
    "***#*##*#*",
    "*#*#**#*#*",
    "*####***#*",
    "*##*###*#*",
    "*##*****#*",
    "*#####* #*",
    "**********"
  };

main()
{
    solvable = false;
    dx = dy = 10;

    solve(1,1);
    if (solvable == true)
      cout << "possible!";
};
```

art

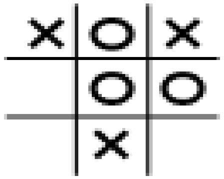
Finish

Writing a TicTacToe Player

```
bool gameIsOver()
{
   if (X has three in a row)   // X wins
      return true;
   if (O has three in a row)  // O wins
      return true;
   if (all squares are filled)  // tie game
      return true;

  return false;
}
```

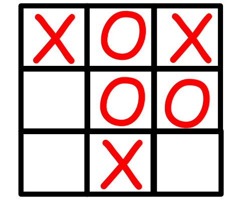
Have you ever wondered how to build an intelligent chess player?

Let's learn how – but for simplicity, we'll look at TicTacToe!



```
GameBoard b;

while (!gameIsOver())
{
  move = getBestMoveForX();  // Get X move from AI
  applyMove('X', move);

  move = GetHumanMove(); // Get O move from human
  applyMove('O', move);
}
```

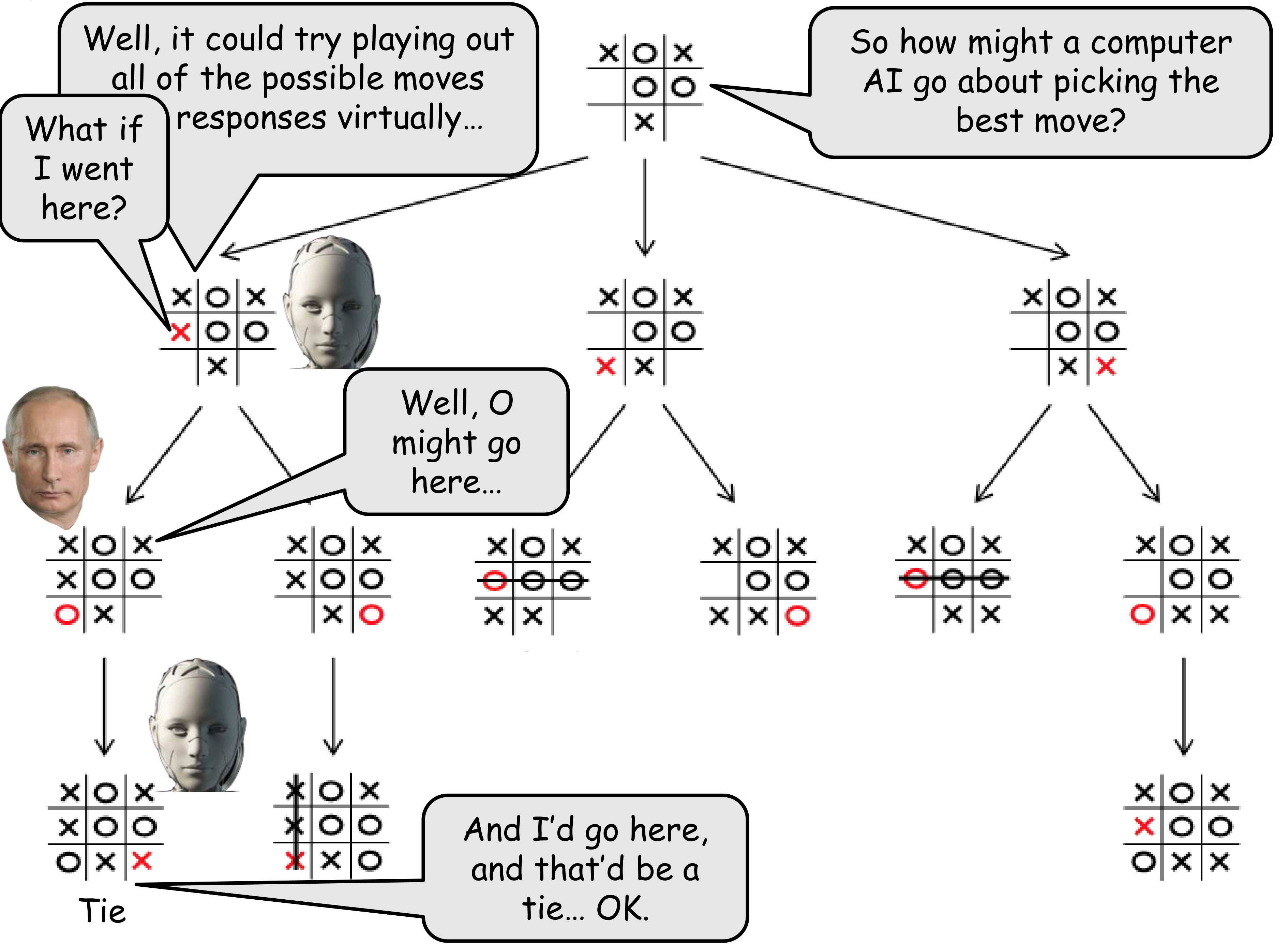
Writing a TicTacToe Player



First, let's see what our game looks like at a high level.

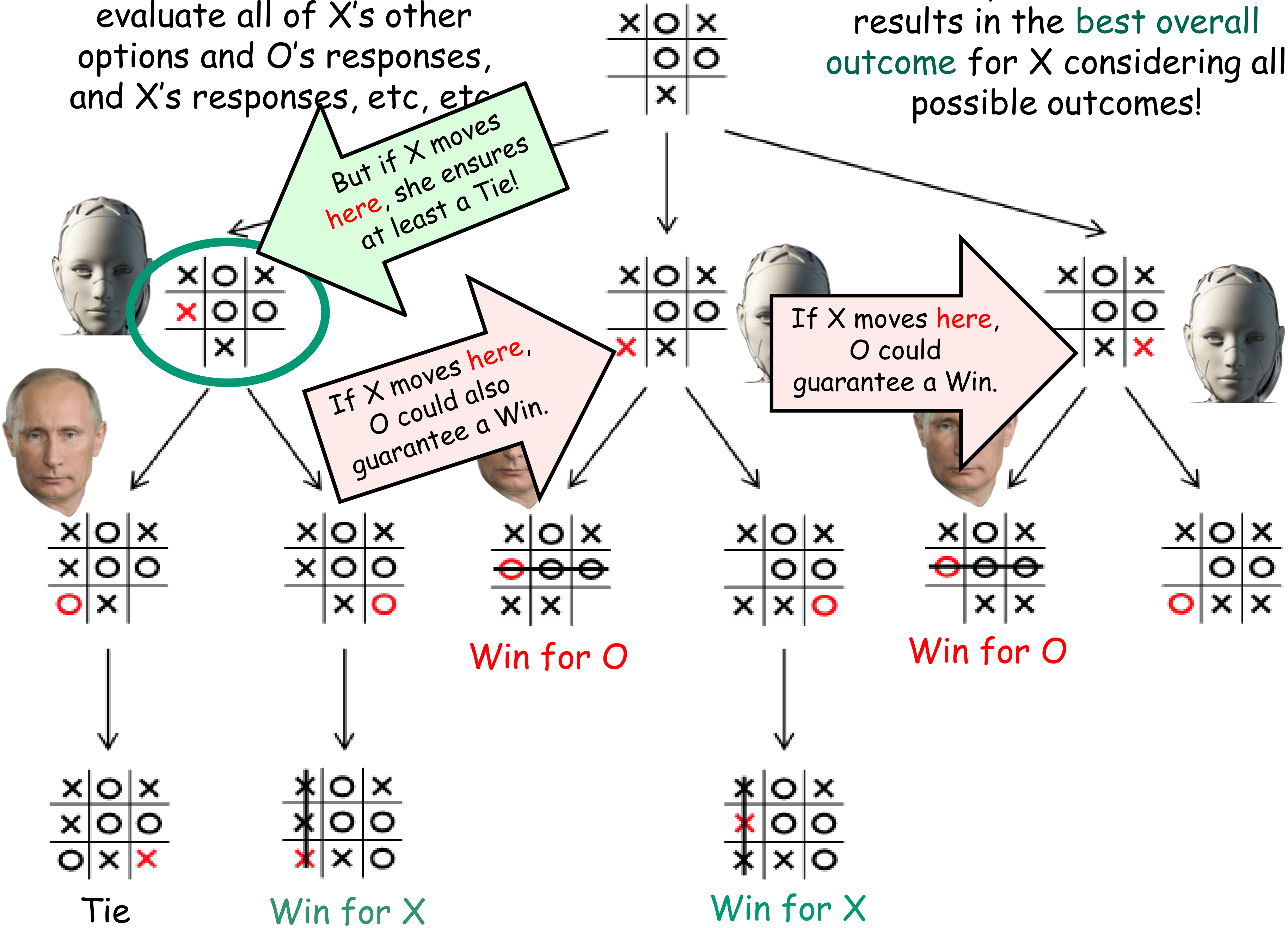
OK, now let's consider how the getBestMoveForX() function might work!

```
GameBoard b;

while (!gameIsOver())
{
  move = getBestMoveForX();  // Get X move from AI
  applyMove('X', move);

  move = GetHumanMove(); // Get O move from human
  applyMove('O', move);
}
```


So how might a computer AI go about picking the best move?
Well, it could try playing out all of the possible moves and responses virtually…
What if I went here?
Well, O might go here…
And I'd go here, and that'd be a tie… OK.
Tie

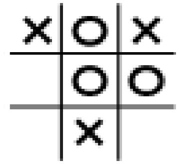
And we could similarly evaluate all of X's other options and O's responses, and X's responses, etc, etc

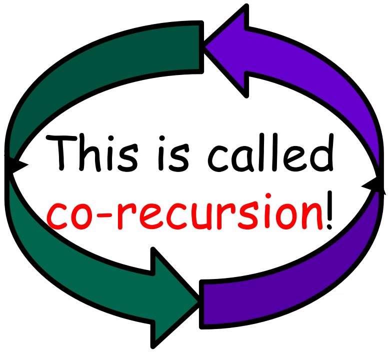
And then pick the move that results in the best overall outcome for X considering all possible outcomes!



So how can we teach a computer to perform this deep evaluation?

Using recursion, of course!





getBestMoveForX():
1. Try each X move
    a. If that ends the game, return result
    b. Otherwise, see how O would respond
2. Return the best move we found for X

getBestMoveForO():
1. Try each O move
    a. If that ends the game, return result
    b. Otherwise, see how X would respond
2. Return the best move we found for O

The function for X calls the function for O…

getBestMoveForX():
1. Try each X move
    a. If that ends the game, return result
    b. Otherwise, see how O would respond
2. Return the best move we found for X

And the function for O calls the function for X…

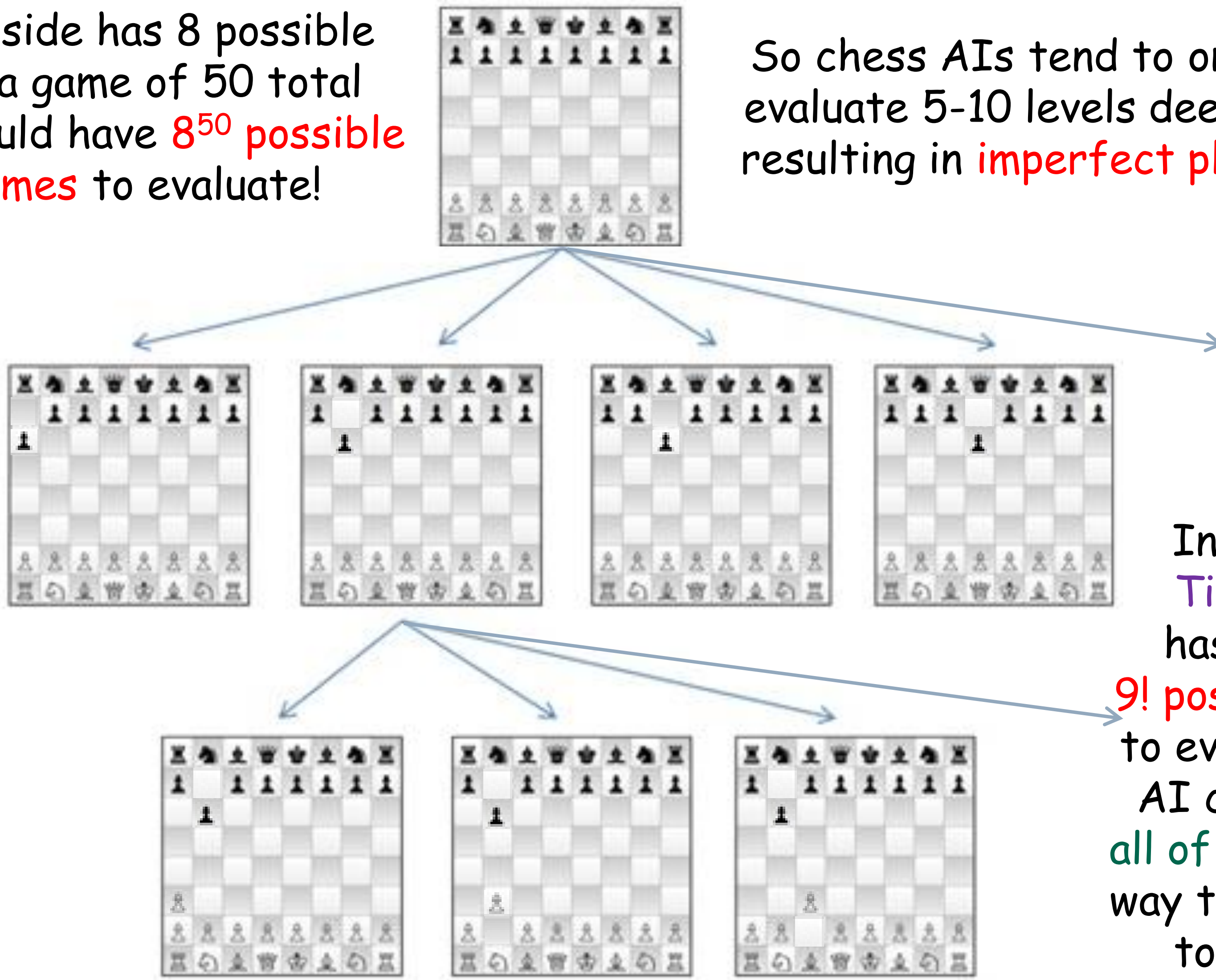
getBestMoveForO():
1. Try each O move
    a. If that ends the game, return result
    b. Otherwise, see how X would respond
2. Return the best move we found for O

And the function for X calls the function for O…

Until we hit the bottom…

Chess Is The Same… But Much Deeper!

If each side has 8 possible moves, a game of 50 total moves would have 8^{50} possible outcomes to evaluate!



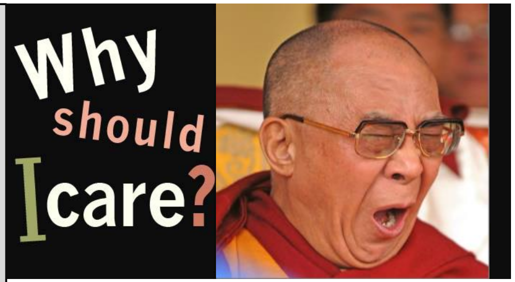
So chess AIs tend to only evaluate 5-10 levels deep… resulting in imperfect play.

In contrast, Tic Tac Toe has less than 9! possible boards to evaluate, so an AI can evaluate all of them, all the way to the bottom to make the perfect move!

Object Oriented Design

(for on-your-own study)

Object Oriented Programming Design
Why should you care?



Good software design can dramatically reduce bugs, reduce development time, and simplify team programming.

So far, you've learned the basics of OOP.

But you haven't learned how properly design OOP programs.

It's like the difference between writing simple sentences and writing a great novel.

So go learn this stuff!

Object-Oriented Design

So, how does a computer scientist go about designing a program?

How do you figure out all of the classes, methods, algorithms, etc. that you need for a program?

At a high level, its best to tackle a design in two phases:

(Well, it's not easy! Many senior engineers are horrible at it!)

First, determine the classes you need, what data they hold, and how they interact with one another.

Second, determine each class's data structures and algorithms.

Class Design Steps

1. Determine the classes and objects required to solve your problem.

2. Determine the outward-facing functionality of each class. How do you interact with a class?

3. Determine the data each of your classes holds and…

4. How they interact with each other.

An Example

Often, we start with a textual specification of the problem.

For instance, let's consider a spec for an electronic calendar.

Each user's calendar should contain appointments for that user. There are two different types of appointments, one-time appts and recurring appts. Users of the calendar can get a list of appointments for the day, add new appointments, remove existing appointments, and check other users' calendars to see if a time-slot is empty. The user of the calendar must supply a password before accessing the calendar. Each appointment has a start-time and an end-time, a list of participants, and a location.

Step #1: Identify Objects

Start by identifying potential classes.

The easiest way to do this is identify all of the nouns in the specification!

Each **user's calendar** should contain **appointments** for that user. There are two different types of appointments, **one-time appts** and **recurring appts**. Users of the calendar can get a list of appointments for the day, add new appointments, remove existing appointments, and check other users' calendars to see if a **time-slot** is empty. The user of the calendar must supply a **password** before accessing the calendar. Each appointment has a **start-time** and an **end-time**, a list of **participants**, and a **location**.

Step #1b: Identify Objects

Now that we know our nouns, let's identify potential classes.

We don't need classes for every noun, just for those key components of our system…

Which nouns should we turn into classes?

Calendar

Appointment

Recurring Appointment

One-time Appointment

- user
- calendar
- password
- time-slot
- start-time
- appointments
- recurring appts
- one-time appts
- participants
- location
- end-time

Step #2a: Identify Operations

Next we have to determine what actions need to be performed by the system.

To do this, we identify all of the verb phrases in the specification!

Each user's calendar should contain appointments for that user. There are two different types of appointments, one-time appts and recurring appts. Users of the calendar can **get a list of appointments** for the day, **add new appointments**, **remove existing appointments**, and **check other users' calendars** to see if a time-slot is empty. The user of the calendar must **supply a password** before accessing the calendar. Each appointment **has a start-time and an end-time, a list of participants**, and a location.

Step #2b: Associate Operations w/Classes

Next we have to determine what actions go with which classes. (let's just look at the first two)

Calendar

```
list getListOfAppts(void)
bool addAppt(Appointment *addme)
bool removeAppt(string &apptName)
bool checkCalendars(Time &slot,
                    Calendar others[])
bool login(string &pass)
bool logout(void)
```

Appointment

```
bool setStartTime(Time &st)
bool setEndTime(Time &st)
bool addParticipant(string &user)
bool setLocation(string &location)
```

Verbs

- get a list of appointments
- add new appointments
- remove existing
- check other users' calendars
- supply a password
- has a start-time
- has an end-time
- has a list of participants
- has a location

Step #2b: Associate Operations w/Classes

So, do we need all of our classes?

Calendar

- Calendar() and ~Calendar()
- list getListOfAppts(void)
- bool addAppt(Appointment *addme)
- bool removeAppt(string &apptName)
- bool checkCalendars(Time &slot, Calendar others[])
- bool login(string &pass)
- bool logout(void)

~~OneTimeAppointment~~

- ~~OneTimeAppointment()~~
- ~~~OneTimeAppointment()~~
- ~~bool setStartTime(Time &st)~~
- ~~bool setEndTime(Time &st)~~
- ~~bool addParticipant(string &user)~~
- ~~bool setLocation(string &location)~~

Appointment

- Appointment() and ~Appointment()
- bool setStartTime(Time &st)
- bool setEndTime(Time &st)
- bool addParticipant(string &user)
- bool setLocation(string &location)

RecurringAppointment

- RecurringAppointment()
- ~RecurringAppointment()
- bool setStartTime(Time &st)
- bool setEndTime(Time &st)
- bool addParticipant(string &user)
- bool setLocation(string &location)
- bool setRecurRate(int numDays)

Step 3: Determine Relationships & Data

Now you need to figure out how the classes relate to each other and what data they hold.

There are three relationships to consider:

1. Uses: Class X uses objects of class Y, but may not actually hold objects of class Y.
2. Has-A: Class X contains one or more instances of class Y (composition).
3. Is-A: Class X is a specialized version of class Y.

This will help you figure out what private data each class needs, and will also help determine inheritance.

Step 3: Determine Relationships & Data

```
Calendar
   Calendar()   and ~Calendar()
   list getListOfAppts(void)
   bool addAppt(Appointment *addme)
   bool removeAppt(string &apptName)
   bool checkCalendars(Time &slot,
                        Calendar others[])
   bool login(string &pass)
   bool logout(void)
private:
    Appointment  m_app[100];
    String       m_password;
```

A Calendar contains appointments

A Calendar must have a password

A Calendar uses other calendars, *but* it doesn't need to hold them.

In general, if a class naturally holds a piece of data, your design should place the data in that class.

Of course, you might not get it right the first time.

In this case, it helps to "re-factor" your classes.
(i.e. iterate till you get it right)

Step 3: Determine Relationships & Data

Appointment
    Appointment()
    virtual ~Appointment()
    bool setStartTime(Time &st)
    bool setEndTime(Time &st)
    bool addParticipant(string &user)
    bool setLocation(string &location)
 private:
   Time m_startTime;
   Time m_endTime;
   string m_participants[10];
   string m_location;

An Appointment has a start time

An Appointment has an end time

An Appointment is associated with a set of particpants.

An Appointment is held at a location.

Now, how about our RecurringAppointment?

It's shares all of the attributes of an Appointment. So should a Recurring Appointment contain an Appointment or use inheritance?

RecurringAppointment
    : public Appointment
    RecurringAppointment()
    ~RecurringAppointment()

 private:
    int m_numDays;
    bool setRecurRate(int numDays)

Step 4: Determine Interactions

Here, we want to determine how each class interacts with the others.

The best way to determine the interactions is by coming up with use cases…

Use Case Examples

1. The user wants to add an appointment to their calendar.
2. The user wants to determine if they have an appointment at 5pm with Joe.
3. The user wants to locate the appointment at 5pm and update it to 6pm.

Use Case #1

1. The user wants to add an appointment to their calendar.

A. The user creates a new Appointment object and sets its values:

```
Appointment *app = new Appointment ("10am","11am","Dodd",...);
```

B. The user adds the Appointment object to the Calendar:

```
Calendar c;
c.addAppointment(app);
```

It looks like we're OK here. Although it might be nicer if we could set the Appointment's values during construction

Use Case #2

2. The user wants to determine if they have an appointment at 5pm with Joe.

Hmm... Can we do this with our classes?

Nope. We'll need to add this to our Appointment class!

```
Calendar c;
...
Appointment *appt;

appt = c.checkTime("5pm");
if (appt == NULL)
   cout << "No appt at 5pm";
else if (appt->isAttendee("Joe"))
   cout << "Joe is attending!";
```

```
Calendar
   Calendar()   and ~Calendar()
   list getListOfAppts(void)
```

```
Appointment
    Appointment() & virtual ~Appointment()
    bool setStartTime(Time &st)
    bool setEndTime(Time &st)
    bool addParticipant(string &user)
    bool setLocation(string &location)
    bool isAttendee(string &person)
 private:
   Time m_startTime;
   Time m_endTime;
   string m_participants[10];
   string m_location;
```

Class Design Conclusions

First and foremost, class design is an iterative process.

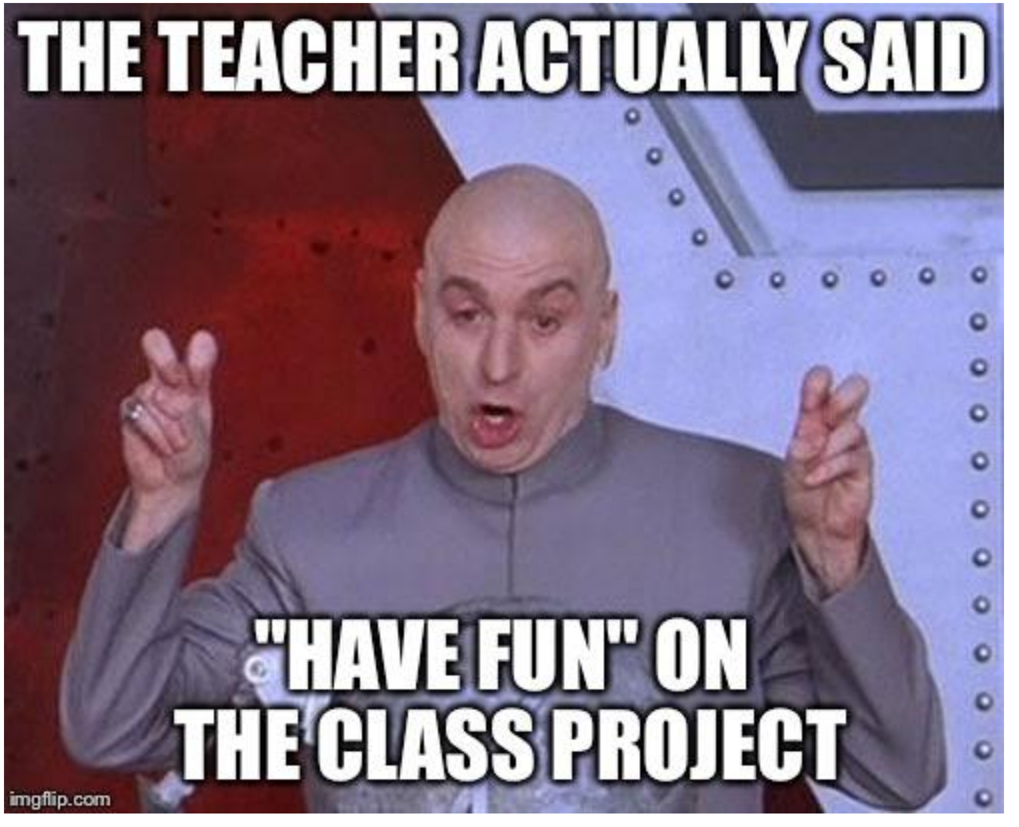
Before you ever start to program your class implementations, it helps to determine your classes, their interfaces, their data, and their interactions.

It's important to go through all of the use cases in order to make sure you haven't forgotten anything.

This is something that you only get better at with experience, so don't feel bad if its difficult at first!

Class Design Tips

Helpful tips for Project #3!



Tip #1

Avoid using dynamic cast to identify common types of objects. Instead add methods to check for various classes of behaviors:

Don’t do this:

```
void decideWhetherToAddOil(Actor *p)
{
    if (dynamic_cast<BadRobot *>(p) != nullptr ||
        dynamic_cast<GoodRobot *>(p) != nullptr ||
        dynamic_cast<ReallyBadRobot *>(p) != nullptr ||
        dynamic_cast<StinkyRobot *>(p) != nullptr)
        p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor *p)
{
     // define a common method, have all Robots return true, all biological
          // organisms return false
    if (p->requiresOilToOperate())
        p->addOil();
}
```

Tip #2

Always avoid defining specific isParticularClass() methods for each type of object. Instead add methods to check for various common behaviors that span multiple classes:

Don’t do this:

```
void decideWhetherToAddOil (Actor *p)
{
    if (p->isGoodRobot() || p->isBadRobot() || p->isStinkyRobot())
        p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor *p)
{
    // define a common method, have all Robots return true, all biological
    // organisms return false
   if (p-> requiresOilToOperate())
       p->addOil();
}
```

Tip #3

If two related subclasses (e.g., BadRobot and GoodRobot) each directly define a member variable that serves the same purpose in both classes (e.g., m_amountOfOil), then move that member variable to the common base class and add accessor and mutator methods for it to the base class. So the Robot base class should have the m_amountOfOil member variable defined once, with getOil() and addOil()functions, rather than defining this variable directly in both BadRobot and GoodRobot.

Don't do this:

```
class SmellyRobot: public Robot
{
		...
private:
		int m_oilLeft;
};

class GoofyRobot: public Robot
{
		...
private:
		int m_oilLeft;
};
```

Do this instead:

```
class Robot
{
public:
		void addOil(int oil)
			{ m_oilLeft += oil; }
		int getOil() const
			{ return m_oilLeft; }
private:
		int m_oilLeft;
};
```

Tip #4

Never make any class's data members public or protected. You may make class constants public, protected or private.

Tip #5

Never make a method public if it is only used directly by other methods within the same class that holds it. Make it private or protected instead.

Tip #6

Your StudentWorld methods should never return a vector, list or iterator to StudentWorld's private game objects or pointers to those objects. Only StudentWorld should know about all of its game objects and where they are. Instead StudentWorld should do all of the processing itself if an action needs to be taken on one or more game objects that it tracks.

Don't do this:

```
class StudentWorld
{
   public:
      vector<Actor *> getZappableActors (int x, int y)
      {
         ... // creates a vector with
             // actor pointers and return it
      }
};
```

```
class NastyRobot
{
public:
   virtual void doSomething()
   {
      ...
      vector<Actor *> v;
      vector<Actor *>::iterator p;

      v = studentWorldPtr-> getZappableActors(getX(), getY());
      for (p = actors.begin(); p != actors.end(); p++)
         p->zap();
   }
};
```

Do this instead:

```
class StudentWorld
{
   public:
      void zapAllZappableActors(int x, int y)
      {
        for (p = actors.begin(); p != actors.end(); p++)
           if (p->isAt(x,y) == true && p->isZappable())
              p->zap();
      }
};
```

```
class NastyRobot
{
public:
      virtual void doSomething()
      {
            ...
      studentWorldPtr-> zapAllZappableActors (getX(), getY());

      }
};
```

Tip #7

If two subclasses have a method that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:

Don't do this:

```
class StinkyRobot: public Robot
{
...
protected:
	virtual void doDifferentiatedStuff()
	{
		doCommonThingA();
		doCommonThingB();

		passStinkyGas();
		pickNose();
	}
};

class ShinyRobot: public Robot
{
...
protected:
	virtual void doDifferentiatedStuff()
	{
		doCommonThingA();
		doCommonThingB();

		polishMyChrome();
		wipeMyDisplayPanel();
	}
};
```

Do this instead:

```
class Robot
{
public:
	virtual void doSomething()
	{
		// first do all the common things that all robots do:
		doCommonThingA();
		doCommonThingB();

		// then call out to a virtual function to do the differentiated stuff
		doDifferentiatedStuff();
	}

protected:
	virtual void doDifferentiatedStuff() = 0;
};

class StinkyRobot: public Robot
{
...
protected:
	// define StinkyRobot's version of the differentiated function
	virtual void doDifferentiatedStuff()
	{
		// only Stinky robots do these things
		passStinkyGas();
		pickNose();
	}
};

class ShinyRobot: public Robot
{
...
protected:
	// define ShinyRobot's version of the differentiated function
	virtual void doDifferentiatedStuff()
	{
		// only Shiny robots do these things
		polishMyChrome();
		wipeMyDisplayPanel();
	}
};
```

Appendix

Tracing Through Recursion (on Paper)

You're taking a CS exam and see this:

How do you solve it quickly?

```
5. What does this print?

int mystery(int a)
{
    if (a == 0)
        return a+1;
    cout << a;

    if (a % 2 == 0)
        a = mystery(a/3);
    else
        a = mystery(a-1);

    return a+5;
}

int main()
{
    cout << mystery(3);
}
```

Output: 3

a = 3

a = mystery(2); *

Steps:

1. Put a blank sheet of paper next to the func.
2. Trace the func with your finger.
   A. When you hit/update a var, write its value down.
   B. Write all output on your original sheet.
   C. When you call a func:
      i. Write its params down
      ii. Write a * to mark where to continue tracing later
      iii. Fold the sheet in half and continue tracing

Tracing Through Recursion (on Paper)

You're taking a CS exam and see this:

How do you solve it quickly?

```
5. What does this print?

int mystery(int a)
{
    if (a == 0)
        return a+1;
    cout << a;

    if (a % 2 == 0)
        a = mystery(a/3);
    else
        a = mystery(a-1);

    return a+5;
}

int main()
{
    cout << mystery(3);
}
```

Output: 3 2

a = 2

a = mystery(0); *

Steps:

1. Put a blank sheet of paper next to the func.
2. Trace the func with your finger.
   - A. When you hit/update a var, write its value down.
   - B. Write all output on your original sheet.
   - C. When you call a func:
     - i. Write its params down
     - ii. Write a * to mark where to continue tracing later
     - iii. Fold the sheet in half and continue tracing

Tracing Thr… ursion (on Paper)

Returning a value of 1

You're taking a CS exam and see …

How do you solve it quickly?

```
5. What does this print?

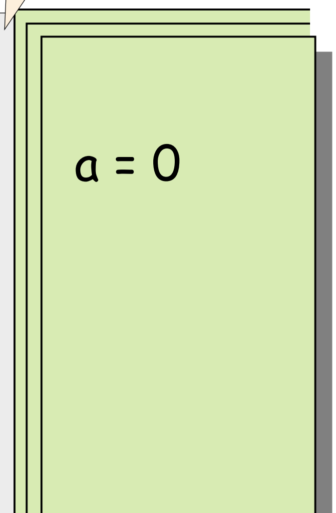
int mystery(int a)
{
    if (a == 0)
        return a+1;
    cout << a;

    if (a % 2 == 0)
        a = mystery(a/3);
    else
        a = mystery(a-1);

    return a+5;
}

int main()
{
    cout << mystery(3);
}
```

Output: 3 2



Steps:

D. To return from a function:

i. Determine what value is being returned (if any)

ii. Unfold your paper once.

iii. Find the * that points to the line where you were (you'll continue from here)

iv. Erase the *

Tracing Thro… ursion (on Paper)

Returning a value of 6

You're taking a CS exam and see…

How do you solve it quickly?

```
5. What does this print?

int mystery(int a)
{
    if (a == 0)
        return a+1;
    cout << a;

    if (a % 2 == 0)
        a = mystery(a/3);
    else
        a = mystery(a-1);

    return a+5;
}

int main()
{
    cout << mystery(3);
}
```

Output: 3 2

a = 2

a = ~~mystery(0)~~; * 1

Steps:

D. To return from a function:

i. Determine what value is being returned (if any)

ii. Unfold your paper once.

iii. Find the * that points to the line where you were (you'll continue from here)

iv. Erase the *

v. Write the returned value above your function

vi. Continue tracing normally.

Tracing Thro ursion (on Paper)

Returning a value of 11

You're taking a CS exam and see t

How do you solve it quickly?

```
5. What does this print?

int mystery(int a)
{
    if (a == 0)
        return a+1;
    cout << a;

    if (a % 2 == 0)
        a = mystery(a/3);
    else
        a = mystery(a-1);

    return a+5;
}

int main()
{
    cout << mystery(3);
}
```

Output: 3 2 11

a = 3

6

a = ~~mystery(2);~~ *

Steps:

D. To return from a function:

i. Determine what value is being returned (if any)

ii. Unfold your paper once.

iii. Find the * that points to the line where you were (you'll continue from here)

iv. Erase the *

v. Write the returned value above your function

vi. Continue tracing normally.

Recursion Tracing Exercise

Use the paper tracing technique to determine what the following program prints:

```
int mystery(int a)
{
    cout << a;
    if (a == 0)
       return 1;
    int b = mystery(a/2);
    cout << b;
    return b + 1;
}

int main()
{
    cout << mystery(3);
}
```