Recursion

Using materials from Stanford CS 106B Summer 2022
(Instructors: Jenny Han and Kylie Jue)
and learncplusplus.com

Definition

recursion

A problem-solving technique in which tasks are completed by reducing them into repeated, smaller versions of themselves.

Recursion

- Recursion is a problem-solving technique in which tasks are completed by reducing them into **repeated, smaller tasks of the same form**.
- Recursion has two main parts: the **base case** and the **recursive case**.
- The solution will get built up **as you come back up the call stack**.
- When solving problems recursively, look for **self-similarity** and think about **what information is getting stored in each stack frame**.

- A *stack frame* is a region of memory (allocated on the stack)
- A function call creates a new stack frame which contains the local variables for that function

3 Musts of Recursion

1. Your code must have a case for all valid inputs.
2. You must have a base case.
3. When you make a recursive call it should be to a simpler instance (forward progress towards base case).

Example:

`isPalindrome()`

Write a function that returns if a string is a palindrome

A string is a palindrome if it reads the same both forwards and backwards:

- isPalindrome(“level”) →true
- isPalindrome(“racecar”) →true
- isPalindrome(“step on no pets”) →true
- isPalindrome(“high”) →false
- isPalindrome(“hi”) →false
- isPalindrome(“palindrome”) → false
- isPalindrome(“X”) →true
- isPalindrome(“”) →true

Approaching recursive problems

- Look for self-similarity.

- Try out an example and look for patterns.
  - Work through a simple example and then increase the complexity.
  - Think about what information needs to be “stored” at each step in the recursive case (like the current value of **`n`** in each **`factorial`** stack frame).

- Ask yourself:
  - What is the base case? (What is the simplest case?)
  - What is the recursive case? (What pattern of self-similarity do you see?)

Discuss:

What are the base and recursive cases?

isPalindrome()

- Look for self-similarity: **racecar**

isPalindrome()

- Look for self-similarity: **racecar**
    - Look at the first and last letters of “racecar” →both are ‘r’

isPalindrome()

- Look for self-similarity: **racecar**
  - Look at the first and last letters of “racecar” →both are ‘r’
  - Check if “aceca” is a palindrome:

isPalindrome()

- Look for self-similarity: **racecar**
  - Look at the first and last letters of “racecar” →both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” →both are ‘a’
    - Check if “cec” is a palindrome:

isPalindrome()

- Look for self-similarity: **racecar**
  - Look at the first and last letters of “racecar” →both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” →both are ‘a’
    - Check if “cec” is a palindrome:
      - Look at the first and last letters of “cec” →both are ‘c’
      - Check if “e” is a palindrome:

isPalindrome()

- Look for self-similarity: **racecar**
  - Look at the first and last letters of “racecar” →both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” →both are ‘a’
    - Check if “cec” is a palindrome:
      - Look at the first and last letters of “cec” →both are ‘c’
      - Check if “e” is a palindrome:
        - **Base case**: “e” is a palindrome

isPalindrome()

- Look for self-similarity: **racecar**
  - Look at the first and last letters of “racecar” →both are ‘r’
  - Check if “aceca” is a palindrome:
    - Look at the first and last letters of “aceca” →both are ‘a’
    - Check if “cec” is a palindrome:
      - Look at the first and last letters of “cec” →both are ‘c’
      - Check if “e” is a palindrome:
        - **Base case**: “e” is a palindrome

What about the `false` *case?*

isPalindrome()

- Look for self-similarity: **hunch**

isPalindrome()

- Look for self-similarity: **hunch**
  - Look at the first and last letters of “hunch” →both are ‘h’

isPalindrome()

- Look for self-similarity: **hunch**
  - Look at the first and last letters of “hunch” →both are ‘h’
  - Check if “unc” is a palindrome:

isPalindrome()

- Look for self-similarity: **hunch**
  - Look at the first and last letters of “hunch” →both are ‘h’
  - Check if “unc” is a palindrome:
    - Look at the first and last letters of “unc” →not equal
    - **Base case**: Return `false`

isPalindrome()

- **Base cases**:
  - isPalindrome(“”) →**true**
  - isPalindrome(string of length 1) →**true**
  - If the first and last letters are not equal →**false**

- **Recursive case:** If the first and last letters are equal,
  isPalindrome(string) = isPalindrome(string minus first and last letters)

isPalindrome()

There can be multiple base (or recursive) cases!

- **Base cases**:
  - isPalindrome(“”) →**true**
  - isPalindrome(string of length 1) →**true**
  - If the first and last letters are not equal →**false**

- **Recursive case:** If the first and last letters are equal, isPalindrome(string) = isPalindrome(string minus first and last letters)

isPalindrome()

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
      if (s[0] != s[s.length() - 1]) {
         return false;
      }
      return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

isPalindrome() in action

```
int main() {
    cout << boolalpha <<
         isPalindrome("racecar")
         << noboolalpha << endl;
    return 0;
}
```

isPalindrome() in action

```
int main() {
    cout << boolalpha <<
        isPalindrome("racecar")
        << noboolalpha << endl;
    return 0;
}
```

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

s

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

s

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

s

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

s

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

isPalindrome() in action

```
int main() {

}
```

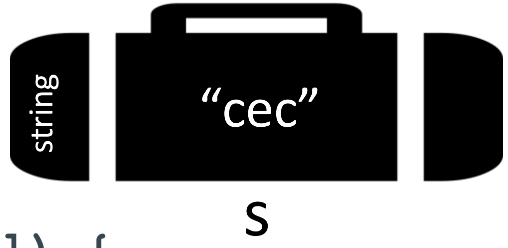
```
bool isPalindrome (string s) {

}
```

```
bool isPalindrome (string s) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```



isPalindrome() in action

```
int main() {

}
```

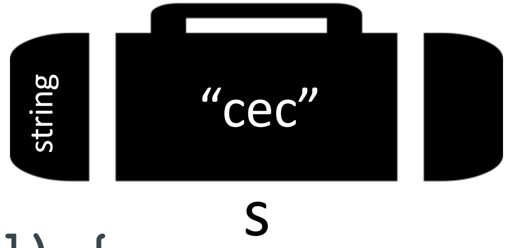
```
bool isPalindrome (string s) {
    if (s.length() < 2) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```



isPalindrome() in action

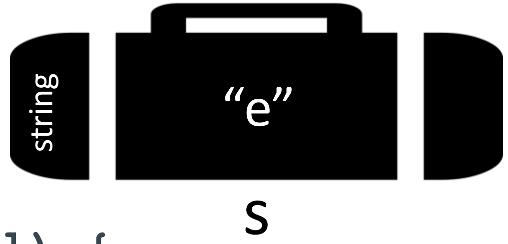
```
int main() {
}
```

```
bool isPalindrome (string s) {
}
```

```
bool isPalindrome (string s) {
}
```

```
bool isPalindrome (string s) {
}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```



isPalindrome() in action

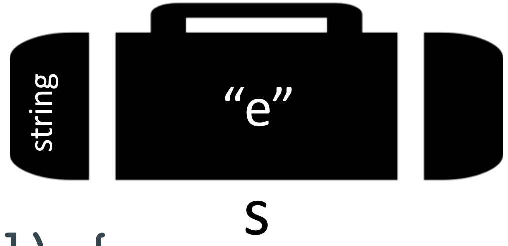
```
int main() {
}
```

```
bool isPalindrome (string s) {
}
```

```
bool isPalindrome (string s) {
}
```

```
bool isPalindrome (string s) {
}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```



isPalindrome() in action

```
int main() {

}
```

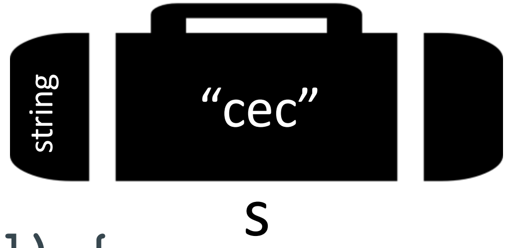
```
bool isPalindrome (string s) {

}
```

```
bool isPalindrome (string s) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```



true

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

true

isPalindrome() in action

```
int main() {

}
```

```
bool isPalindrome (string s) {
    if (s.length() < 2) {
        return true;
    } else {
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        return isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

true

s

isPalindrome() in action

```
int main() {
    cout <<
        isPalindrome("racecar")
        << endl;
    return 0;
}
```

Prints **1***!*

Why do we use recursion?

Why do we use recursion?

- Elegance
  - Allows us to solve problems with very clean and concise code
- Efficiency
  - Allows us to accomplish better runtimes when solving problems
- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

An **efficient** example: Binary Search

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Idea #1: We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

We could just go through each element in order and do a linear search.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Linear search is **`O(n)`**

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Can we do better? Can we take advantage of the structure of the data?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Idea #2: Binary search

- Eliminate half of the data at each step.

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you've found your element!

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Start by looking at index:

`(startIndex + endIndex) / 2`

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Start by looking at index:

`(0 + 9) / 2`

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Start by looking at index:

4

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Too small

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Eliminate left half

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

`(startIndex + endIndex) / 2 =`

`(5 + 9) / 2 =`

`7`

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

```
(startIndex + endIndex) / 2 =
(5 + 9) / 2 =
7
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Too small

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Eliminate left half

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

`(startIndex + endIndex) / 2 =`
`(8 + 9) / 2 =`
`8`

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

```
(startIndex + endIndex) / 2 =
(8 + 9) / 2 =
8
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Success!

Defining binary search recursively

- **Algorithm**: Check the middle element at `(startIndex + endIndex) / 2`
    - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
    - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
    - Otherwise, you've found your element!

Defining binary search recursively

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you've found your element!

- Recursive cases
  - Element at middle is too small →binarySearch(right half of data)
  - Element at middle is too large →binarySearch(left half of data)

Defining binary search recursively

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you’ve found your element!

- Recursive cases
  - Element at middle is too small →binarySearch(right half of data)
  - Element at middle is too large →binarySearch(left half of data)
- Base cases
  - Element at middle == desired element
  - Desired element is not in your data

Let’s read the code for **`binarySearch()`** and identify the base/recursive cases.

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Base cases

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Recursive cases

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

What's the runtime?

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains:

```
N, N/2, N/4, N/8, ..., 4, 2, 1
```

  - How many divisions does it take?

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains.

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach **N**?

```
1, 2, 4, 8, ..., N/4, N/2, N
```

  - Call this number of multiplications **x**:

$$2^x = N$$
$$\mathbf{x = \log_2 N}$$

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains.

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach **N**?

`1, 2, 4, 8, ..., N/4, N/2, N`

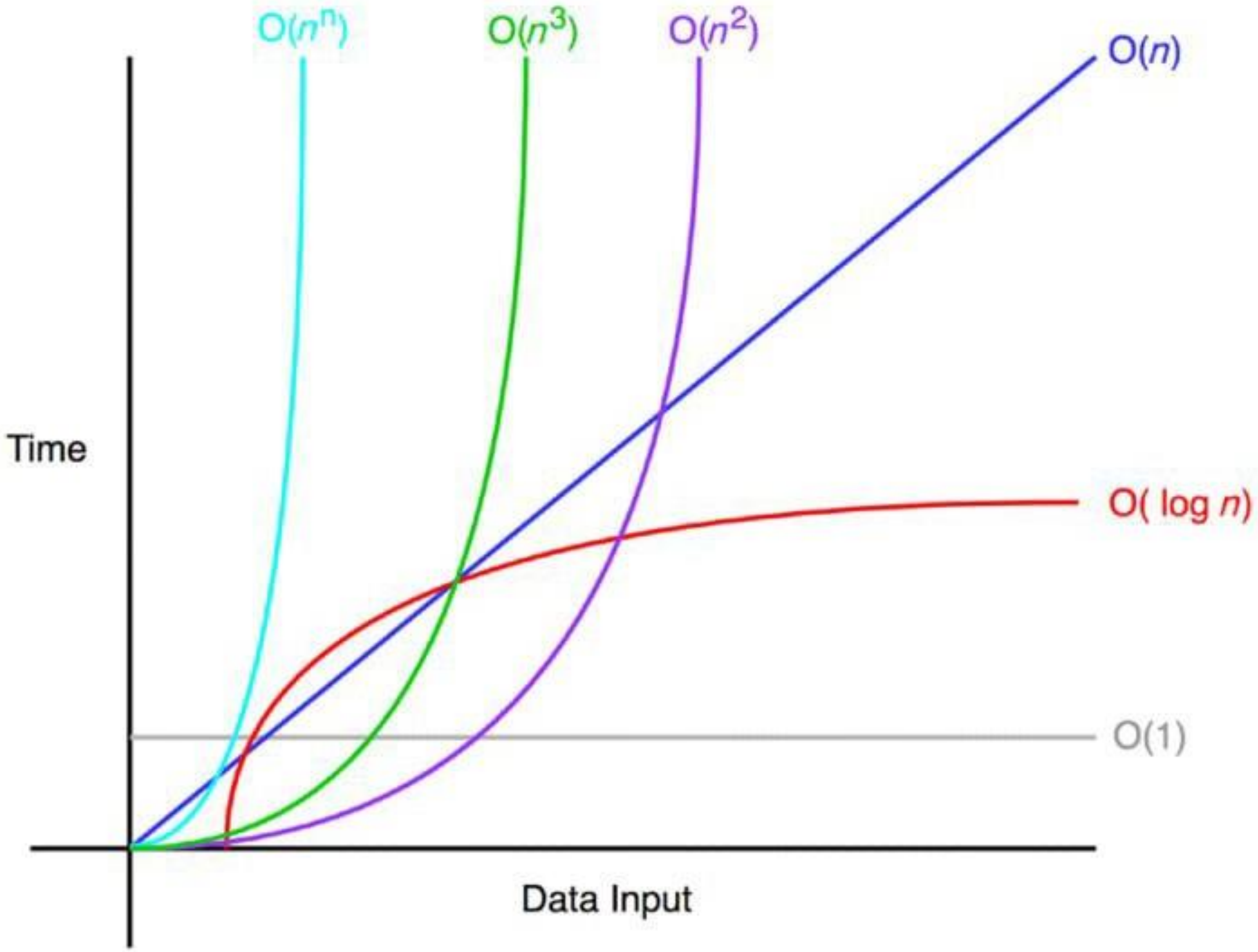
  - Call this number of multiplications **x**:

$$2^x = N$$
$$\mathbf{x = \log_2 N}$$

- Binary search has logarithmic complexity: **`O(log N)`**

Logarithmic runtime

- Better than linear
- A common runtime when you're able to "divide and conquer" in your algorithm, like with binary search



Why do we use recursion?

- Elegance
  - Allows us to solve problems with very clean and concise code
- Efficiency
  - Allows us to accomplish better runtimes when solving problems
- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

Recursive vs. Iterative

- Why use recursion if you can do the same thing iteratively (using a for loop or while loop)?
  - You can always solve a recursive problem iteratively
  - Sometimes the recursive version is much simpler to write (and read)

- Iterative functions (using a for-loop or while-loop) are usually more efficient than their recursive counterparts
  - This is due to the overhead of function calls
  - Each time you call a function, you push/pop things from the stack
  - Iterative functions don’t need to do this

- But sometimes the recursive implementation is much cleaner and easier to follow
  - Incurring a little extra overhead is may be worth it for the benefit in maintainability
  - Particularly if the algorithm doesn’t need to recurse too many times to find a solution

Recursive vs. Iterative

- In general, recursion is a good choice when most of the following are true:
  - The recursive code is much simpler to implement
  - The recursion depth can be limited (e.g. there's no way to provide an input that will cause it to recurse down 100,000 levels)
  - The iterative version of the algorithm requires managing a stack of data
  - This isn't a performance-critical section of code

- If you're not sure
  - Do whichever is easier for you to understand and implement
  - Then, if it's not efficient, re-think the implementation to see if the other way is better

Homework

- Read Convex Optimization Book: Chapters 9.1-9.1.1; 9.2; 9.3-9.3.1; 9.5-9.5.2; 9.5.4
- Read Numerical Differentiation (up to Complex-variable Methods)
- Homework 4 due Weds.