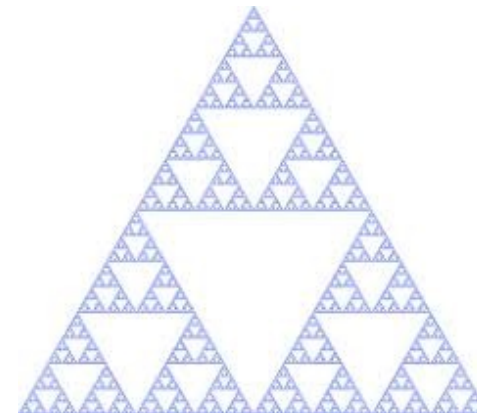# Recursion

Recursion

- *Recursion* is a powerful technique for breaking up complex computational problems into simpler ones,

  where the "simpler one" *is* the **solution** to the whole problem!

- Recursion is often the most natural way of thinking about a problem, and there are some computations that are difficult to perform without recursion.

Recursive Functions

A recursive function is a function that calls itself, reducing the problem a bit on each call:

```
void solveIt(the-Problem)
{
    . . .
    solveIt(the-Problem-a-bit-reduced);
}
```

Of course, there's a lot/little more to it than this.

How to think recursively?

Can we break down the problem into smaller, similar pieces?

Example: walk to a target point.

How many steps?

Iterative

```
howManySteps(start,end)
{



}
```

Recursive

```
howManySteps(start,end)
{



}
```

How to think recursively? The Two Key Requirements

The two keys requirements for a successful recursive function:

- Every recursive call must simplify the task in some way.

- There must be special cases to handle the simplest tasks directly so that the function will stop calling itself.
    - base case(s) – simplest task(s)
    - recursive case – break problem into smaller version of itself

How to think recursively?

Triangle of Boxes:

```
[]
[][]
[][][]
[][][][]
```

```
void print_triangle(int side_length);
```
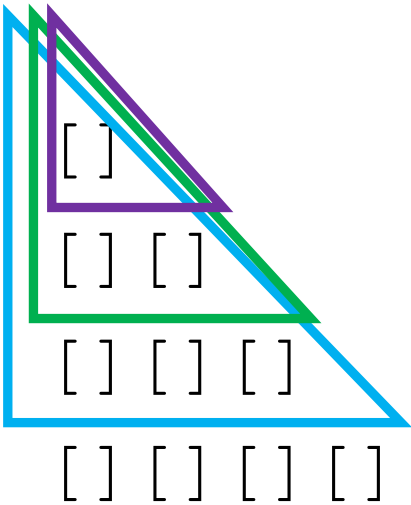
Triangle of Boxes Code

```cpp
void print_triangle(int side_length)
{
    if (side_length < 1)
    {
        return;
    }

    print_triangle(side_length - 1);

    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

Function calls - `void print_triangle(int side_length);`

[ ]

[ ] [ ]

[ ] [ ] [ ]

[ ] [ ] [ ] [ ]

no triangle + print 1st row

smaller triangle + print 2nd row

smaller triangle + print 3rd row

smaller triangle + print 4th row

How to think recursively?

Triangle of Boxes:

```
[] [] [] []
[] [] []
[] []
[]
```

```
void print_triangle(int side_length);
```

Triangle of Boxes Code

```cpp
void print_triangle(int side_length)
{
    if (side_length < 1)
    {
        return;
    }

    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;

    print_triangle(side_length - 1);
}
```

Thinking Recursively – Palindromes

Palindrome: a string that is equal to itself when you reverse all characters

Example: Madam, I'm Adam

IT'S PALINDROME WEEK!

5.10.15	5.14.15
5.11.15	5.15.15
5.12.15	5.16.15
5.13.15	

● USA TODAY

Thinking Recursively – Palindromes (1)

The problem: Write a function to test if a string is a palindrome.

```
bool is_palindrome(string s)
```

Thinking Recursively – Palindromes – an aside: iteratively

The problem: Write a function to test if a string is a palindrome.

```
bool is_palindrome(string s)
```

Thinking Recursively – Palindromes: recursively

The problem: Write a function to test if a string is a palindrome.

```
bool is_palindrome(string s)
```

Step 1: Break the input into parts that can themselves be inputs to the problem.

Focus on a particular input or set of inputs for the problem.

Think how you can simplify the inputs in such a way that the same function can be applied to the simpler input.

Thinking Recursively – Palindromes (2)

To get simpler inputs, how about :

- Remove the first character?
- Remove the last character?
- Remove a character from the middle?
- Cut the string into two halves?
- Remove both the first and the last character?

Thinking Recursively – Palindromes (3)

- Every palindrome's first half is the same as its other half.
- In this problem, chopping in half seems to be a good way to reduce the problem. But:

"rotor"

(chop)

"rot"   "or"

- Not sure how chopping in half gets us closer to a way to determine a palindromic situation.

Thinking Recursively – Palindromes (4)

- One character at a time seems not so good.
- How about chopping off BOTH ends at the same time?

"rotor"

(chop)   (chop)

"r"   "oto"   "r"

- We can reduce the problem to the "middle" of the string for the recursive call.

Thinking Recursively – Palindromes (5)

- **Step 2:** Combine solutions with simpler inputs to a solution of the original problem.

<div align="center">

"rotor"

(chop)   (chop)

"r"   "oto"   "r"

</div>

- If the end letters are the same AND **is_palindrome(** *the middle word* **)** then the string is a palindrome!

Thinking Recursively – Palindromes (6)

Step 3: Find solutions to the simplest inputs.

- A recursive computation keeps simplifying its inputs.

- Eventually it arrives at very simple inputs. To make sure that the recursion comes to a stop, deal with the simplest inputs separately.

- That leaves us with two possible end situations, both of which are palindromes themselves:

string of length 0, and string of length 1

```
bool is_palindrome(string s)
{
    // Separate case for shortest strings
    if (s.length() <= 1 ){ return true; }
...
```

Thinking Recursively – Palindromes (7)

Step 4: Implement the solution by combining the simple cases and the reduction step.

```cpp
...
// Get first and last character, converted to lowercase
    char first = tolower(s[0]);
    char last = tolower(s[s.length() - 1]);

    if (first == last)
    {
        string shorter = s.substr(1, s.length() - 2);
        return is_palindrome(shorter);
    }
    else
    {
        return false;
    }
}
```

Iteration vs. Recursion

- So is the iterative solution always faster than the recursive?

- Look at the iterative palindrome solution

```
bool is_palindrome(string s)
{
    int start = 0;
    int end = s.length() - 1;
    while (start < end)
    {
        if (s[start] != s[end]) { return false; }
        start++;
        end--;
    }
    return true;
}
```

Iteration vs. Recursion (2)

- Both the iteration and the recursion run at about the same speed.

- If a palindrome has n characters,

    the iteration *executes* the loop **n/2** times.

    the recursive solution *calls itself* **n/2** times, because two characters are removed in each step.

The Fibonacci Sequence

- Recursion can lead to simpler solutions to problems, but recursive algorithms many perform poorly.

- The Fibonacci sequence is a sequence of numbers defined by the equations:

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Each value in the sequence is the sum of the 2 preceding.

The first ten terms of the sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...

Fibonacci Call Tree

Fibon

This can be shown more clearly as a *call tree.*

Notice that the same values, for example, **fib(2)**, are computed over and over, **and each recursive call generates 2 more calls**

```
                                fib(6)
                  fib(5)                      fib(4)
           fib(4)        fib(3)        fib(3)        fib(2)
       fib(3) fib(2)  fib(2) fib(1)  fib(2) fib(1)
    fib(2) fib(1)
```