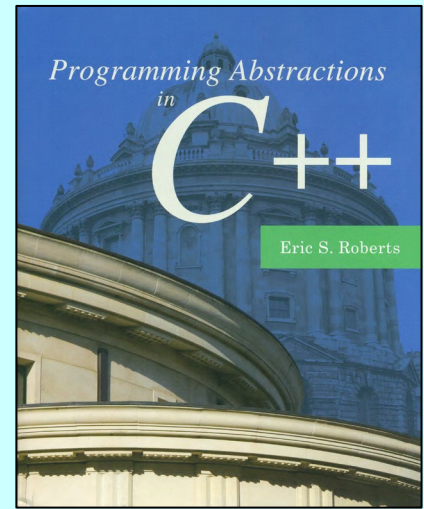


## CHAPTER 8

# Recursive Strategies

*Tactics without strategy is the noise before defeat.*

—Sun Tzu, ~5<sup>th</sup> century BCE



8.1 The Towers of Hanoi

8.2 The subset-sum problem

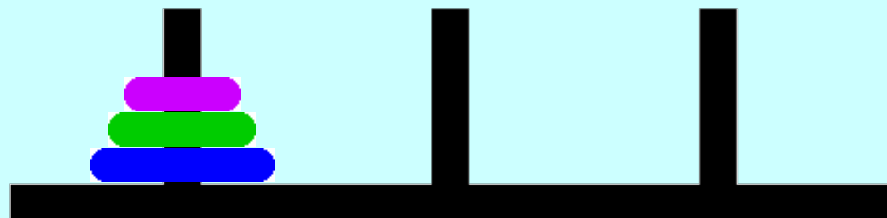
8.3 Generating permutations

8.4 Graphical recursion

# The Towers of Hanoi

In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty **must not move more than one disk at a time** and that he **must place this disk on a needle so that there is no smaller disk below it**. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

—Henri de Parville, *La Nature*, 1883



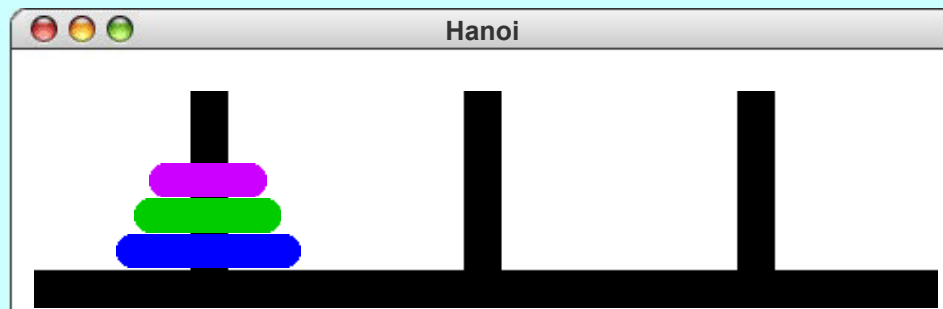
# The Tower of Hanoi Solution

- Framing the problem:
  1. The number of disks to move
  2. The name of the spire where the disks start out
  3. The name of the spire where the disks should finish
  4. The name of the spire used for **temporary storage**
- Finding a recursive strategy:
  1. There must be a **simple case**.
  2. There must be a **recursive decomposition**.
- If you play the game for a few moments, it becomes clear that you can solve the problem by dividing it into these three steps:
  1. Move the entire stack consisting of the top  $n-1$  disks from spire A to spire C (with the help of B).
  2. Move the bottom disk from spire A to spire B.
  3. Move the stack of  $n-1$  disks from spire C to spire B (with the help of A).

# The Towers of Hanoi Solution

```
int main() {  
    void moveTower(int n, char start, char finish, char temp) {  
        if (n == 1) {  
            moveSingleDisk(start, finish);  
        } else {  
            moveTower(n - 1, start, temp, finish);  
            moveSingleDisk(start, finish);  
            moveTower(n - 1, temp, finish, start);  
        }  
    }  
}
```

n	start	finish	temp
3	'A'	'B'	'C'



# The Recursive “Leap of Faith”



*recursive leap of faith.*

# Example: Generating Subsets

- Write a function

```
Vector<string> generateSubsets(string set) ;
```

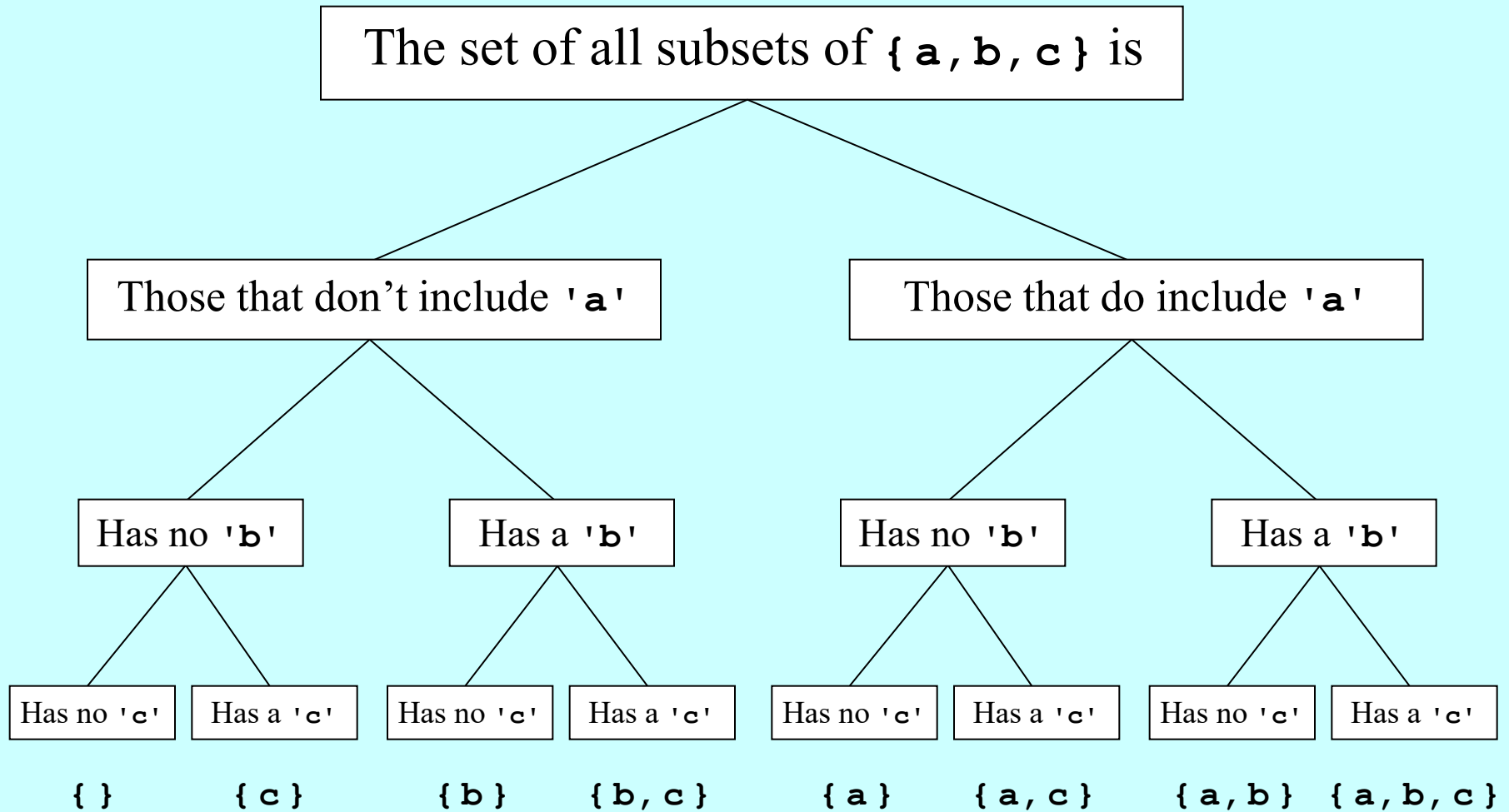
that generates a vector showing all subsets of the set formed from the letters in **set**.

- For example, calling **generateSubsets("abc")** should return a vector containing the following eight strings, in some order:

" "	"a"	"ab"	"abc"
	"b"	"ac"	
	"c"	"bc"	

- The solution process requires a branching structure. At each level of the recursion, you can either *exclude* or *include* the current letter from the list of subsets, as illustrated on the following slide.

# The Subset Tree





# The Subset Problem

- This problem is difficult to solve without using recursion, mostly because any solution strategy requires exploring a solution space that doubles in size each time you choose whether to include a particular value from the set or not.
- The recursive solution to this problem tries to find a solution that *includes* a particular element of the set and then tries to find a solution that *excludes* that element. This strategy is often called the *inclusion-exclusion pattern*.
- When you are looking for a **recursive decomposition**, you need to find some value in the inputs—arguments in the C++ formulation of the problem—that you can make **smaller**.



# Generating Subsets

```
/*
 * Function: generateSubsets
 * Usage: Vector<string> subsets = generateSubsets(str);
 * -----
 * Generates a vector containing all subsets of the specified
 * set. Each set is represented as a string in which the letters
 * correspond to the individual elements. Thus, the string
 * "abd" represents the set { a, b, d }.
 */

Vector<string> generateSubsets(string set) {
    Vector<string> result;
    if (set == "") {
        result.add("");
    } else {
        result = generateSubsets(set.substr(1));
        int n = result.size();
        for (int i = 0; i < n; i++) {
            result.add(set[0] + result[i]);
        }
    }
    return result;
}
```

# The Subset-Sum Problem

- The *subset-sum problem* can be expressed as follows:

*Given a set of integers and a target value, determine whether it is possible to find a subset of those integers whose sum is equal to the specified target.*

- For example, given the set  $\{-2, 1, 3, 8\}$  and the target value 7, the answer to the subset-sum question is?
- Yes, because the subset  $\{-2, 1, 8\}$  adds up to 7.
- What if the target value is 5?
- The answer would be no.
- This problem is a little more difficult than the previous subset problem because there is an extra target value in play.

# The Subset-Sum Problem

- The recursive solution to this problem can again follow the *inclusion-exclusion pattern*.
- The key insight you need to solve this problem is that there are two ways that you might be able to produce the desired target sum after you have identified a particular element.
  - One possibility is that the subset you're looking for *excludes* that element, in which case it must be possible to generate the value `target` using only the leftover set of elements.
  - The other possibility is that the subset you're looking for *includes* that element. For that to happen, it must be possible to take the rest of the set and produce the value `target-element`.

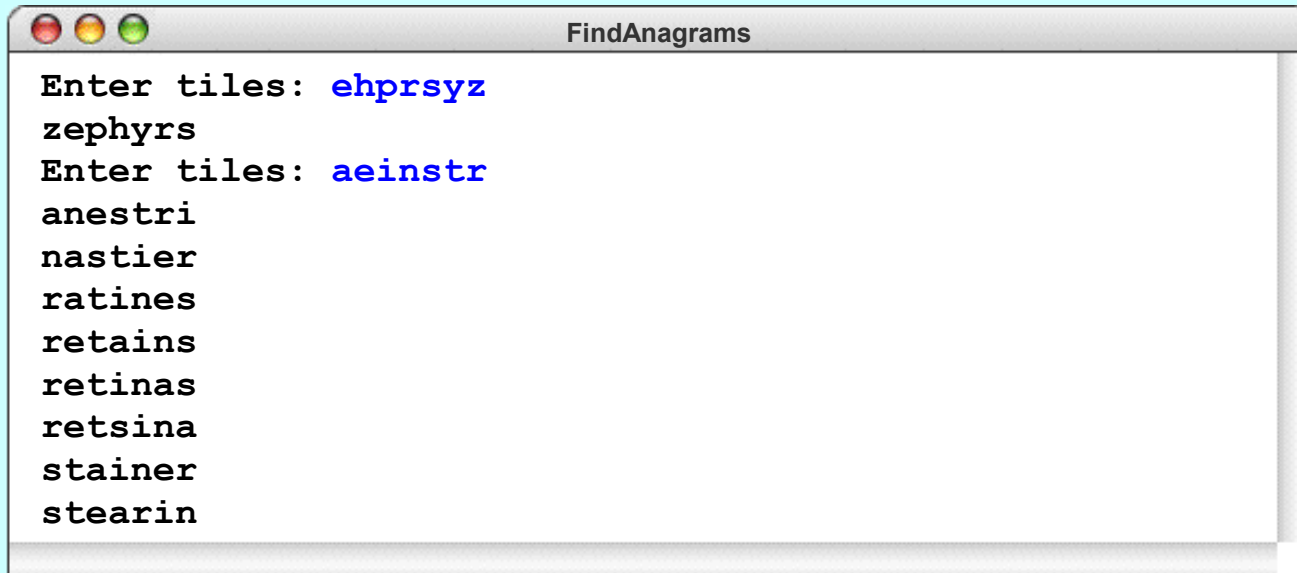
# Code for the Subset-Sum Program

```
/*
 * Function: subsetSumExists
 * Usage: if (subsetSumExists(set, target) . . .
 * -----
 * Determines whether it is possible to choose elements from set so that
 * the sum is equal to target. The recursion uses the inclusion-exclusion
 * pattern. This pattern begins by choosing a particular element (this
 * code chooses the first element) and then continues by checking whether
 * it is possible to create the target without using that element or
 * whether it is possible to include that element and then find a subset
 * of the rest that adds up to the target after subtracting the element.
 */

bool subsetSumExists(Set<int> & set, int target) {
    if (set.isEmpty()) {
        return target == 0;
    } else {
        int element = set.first();
        Set<int> rest = set - element;
        return subsetSumExists(rest, target)
            || subsetSumExists(rest, target - element);
    }
}
```

# Exercise: Finding Anagrams

- Write a program that reads in a set of letters and sees whether any anagrams of that set of letters are themselves words:



```
FindAnagrams
Enter tiles: ehprsy
zephyrs
Enter tiles: aeinstr
anestri
nastier
ratines
retains
retinas
retsina
stainer
stearin
```

- Generating all anagrams of a word is not a simple task. Most solutions require some tricky *recursion*, but can you think of another way to solve this problem?

Hint: What if you had a function that sorts the letters in a word? Would that help?

# Generating Permutations

- A *permutation* of a collection of elements is simply an ordering of those elements (anagrams in word games).
- To generate all permutations of a collection of  $N$  items, you can apply the following recursive decomposition:
  - Choose every element of the  $N$  items to be the first element of the permutation.
  - Add all possible permutations of the remaining  $N - 1$  items to the set of permutations.
- For example, the permutations of the string "**ABCDE**" are:
  - The character '**A**' followed by all permutations of "**BCDE**".
  - The character '**B**' followed by all permutations of "**ACDE**".
  - The character '**C**' followed by all permutations of "**ABDE**".
  - The character '**D**' followed by all permutations of "**ABCE**".
  - The character '**E**' followed by all permutations of "**ABCD**".

# Code for the Permutations Function

```
/*
 * Function: generatePermutations
 * Usage: Set<string> permutations = generatePermutations(str);
 * -----
 * Returns a set consisting of all permutations of the specified string.
 */

Set<string> generatePermutations(string str) {
    Set<string> result;
    if (str == "") {
        result += "";
    } else {
        for (int i = 0; i < str.length(); i++) {
            char ch = str[i];
            string rest = str.substr(0, i) + str.substr(i + 1);
            for (string s : generatePermutations(rest)) {
                result += ch + s;
            }
        }
    }
    return result;
}
```

# Exercise: Generating Permutations

- The previous solution finds all possible permutations by exploring in a **branching structure** in a systematic way. The inclusion-exclusion pattern is essentially a two-branch structure.
- Any alternative strategy for generating permutations?
- To generate all permutations of a collection of  $N$  items, you can apply the following different recursive decomposition:
  - Find all possible permutations of any one specific subset of  $N - 1$  items.
  - Insert the remaining element into every permutation generated above, at all possible  $N$  positions.
- Complexity compared to the previous branch-searching strategy?





# Graphical Recursion

the most exciting applications of recursion use to create intricate pictures in which a particular repeated at many different scales.

- Graphics libraries, e.g., `gwindow.h` in the Stanford C++ libraries, are needed to implement the following graphical recursion programs.
- Be aware that the graphics libraries in the Stanford C++ libraries include a set of complicated class hierarchies, with limited documentations.
- There are many other professional graphics libraries, e.g., OpenGL, you can use to develop graphics applications.

# Methods in the Graphics Library

**GWindow gw**(*width*, *height*)

Creates a graphics window with the specified dimensions.

**gw.drawLine**( $x_0$ ,  $y_0$ ,  $x_1$ ,  $y_1$ )

Draws a line connecting the points  $(x_0, y_0)$  and  $(x_1, y_1)$ .

**gw.drawPolarLine**( $x_0$ ,  $y_0$ ,  $r$ ,  $\theta$ )

Draws a line  $r$  pixels long in direction  $\theta$  from  $(x_0, y_0)$ . To make chaining line segments easier, this function returns the ending coordinates as a **GPoint**.

**gw.getWidth()**

Returns the width of the graphics window.

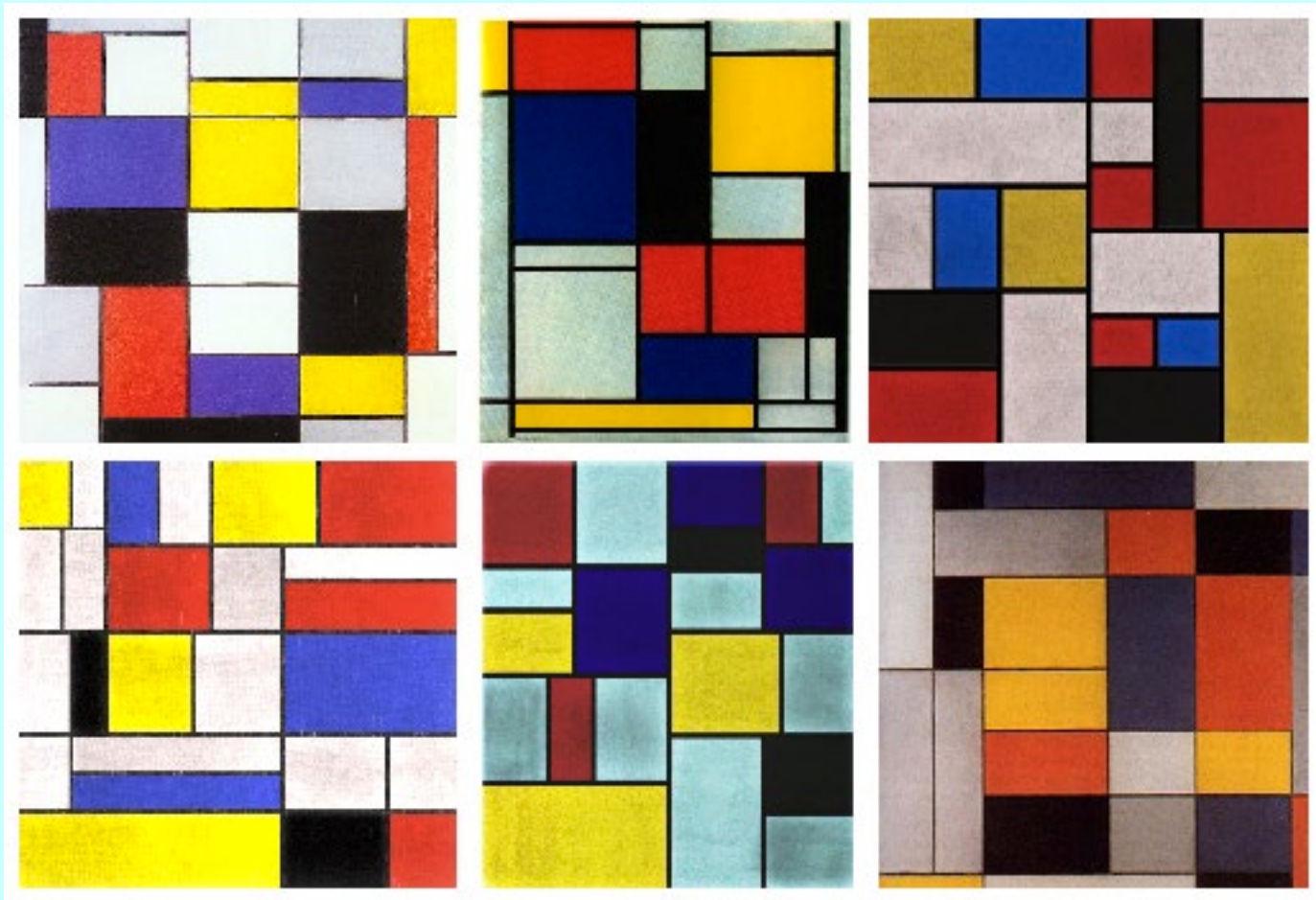
**gw.getHeight()**

Returns the height of the graphics window.

Many more functions exist in the interfaces under the **graphics** directory in the Stanford C++ libraries. A brief (and possibly outdated) documentation is available here:

<https://cs.stanford.edu/people/eroberts/StanfordCPPLib/doc/GWindow-class.html>

# Generating Mondrian-Style Paintings



**Fig. 11:** Three real Mondrian paintings, and three samples from our targeting function. Can you tell which is which? (Source: Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun, “Metropolis Procedural Modeling,” *ACM Transactions on Graphics*, April 2011.)

S

44)

```
if (dst == NULL || src == NULL)
    return; // Handle null pointers
int i;
for (i = 0; i < dst_size - 1 && src[i] != '\0'; i++)
    dst[i] = src[i];
dst[i] = '\0'; // Ensure null-termination
```





# Code for the Mondrian Program

```
#include <iostream>
#include "gwindow.h"
#include "random.h"
using namespace std;

/* Constants */

const double MIN_AREA = 10000;    /* Smallest square that will be split */
const double MIN_EDGE = 20;      /* Smallest edge length allowed */

/* Function prototypes */

void subdivideCanvas(GWindow & gw, double x, double y,
                    double width, double height);

/* Main program */

int main() {
    GWindow gw;
    subdivideCanvas(gw, 0, 0, gw.getWidth(), gw.getHeight());
    return 0;
}
```

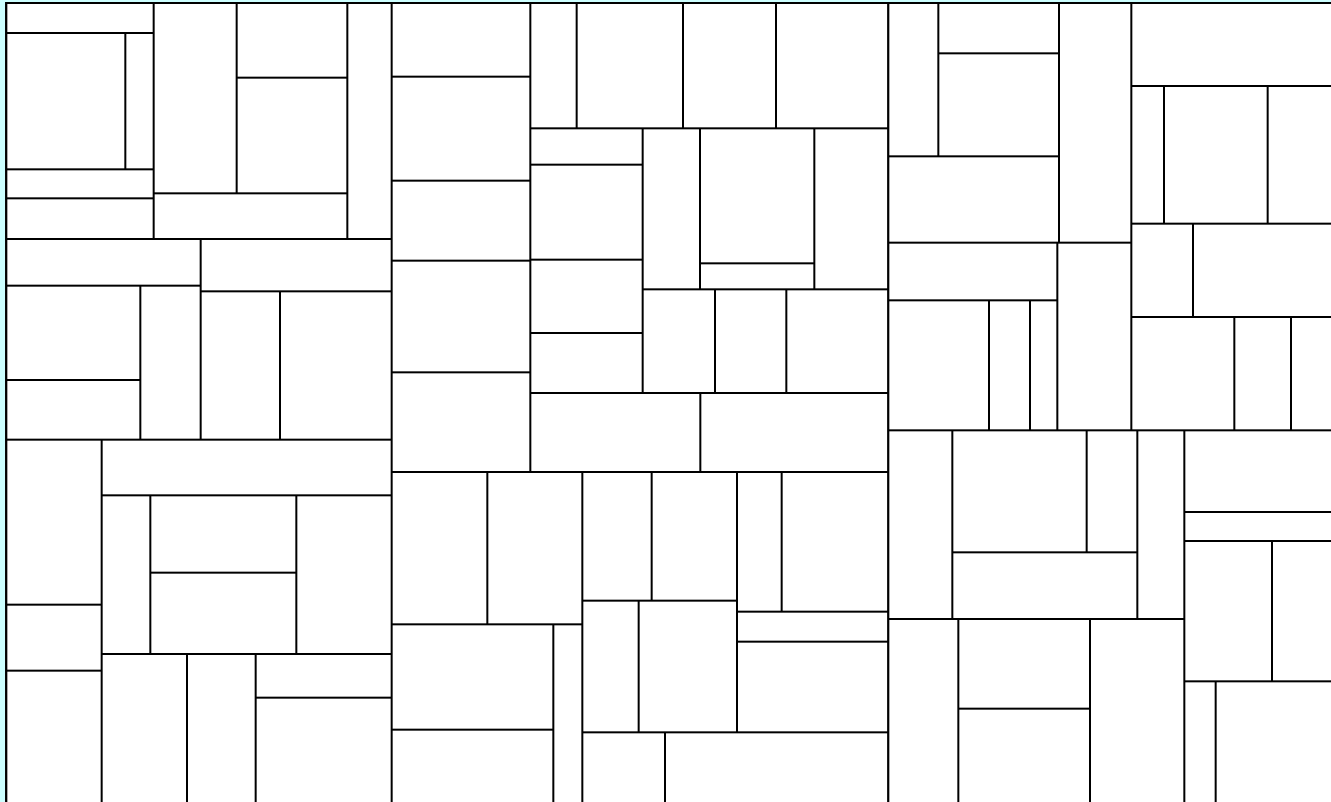


# Code for the Mondrian Program

```
void subdivideCanvas(GWindow & gw, double x, double y,  
                    double width, double height) {  
    if (width * height >= MIN_AREA) {  
        if (width > height) {  
            double mid = randomReal(MIN_EDGE, width - MIN_EDGE);  
            subdivideCanvas(gw, x, y, mid, height);  
            subdivideCanvas(gw, x + mid, y, width - mid, height);  
            gw.drawLine(x + mid, y, x + mid, y + height);  
        } else {  
            double mid = randomReal(MIN_EDGE, height - MIN_EDGE);  
            subdivideCanvas(gw, x, y, width, mid);  
            subdivideCanvas(gw, x, y + mid, width, height - mid);  
            gw.drawLine(x, y + mid, x + width, y + mid);  
        }  
    }  
}
```

- In graphical recursion, there seems to be no simple cases to be solved.
- There must be, however, some stopping criteria to stop the recursion.

# Mondrian Decomposition



# Exercise: A Better Mondrian Program

- Can you do a better job of emulating Mondrian's style?
- Suppose that you have the following additional functions:

**gw.drawRect** (*x*, *y*, *width*, *height*)

Draws the outline of a rectangle with the specified bounds.

**gw.fillRect** (*x*, *y*, *width*, *height*)

Fills the outline of the specified rectangle using the current color.

**gw.setColor** (*color*)

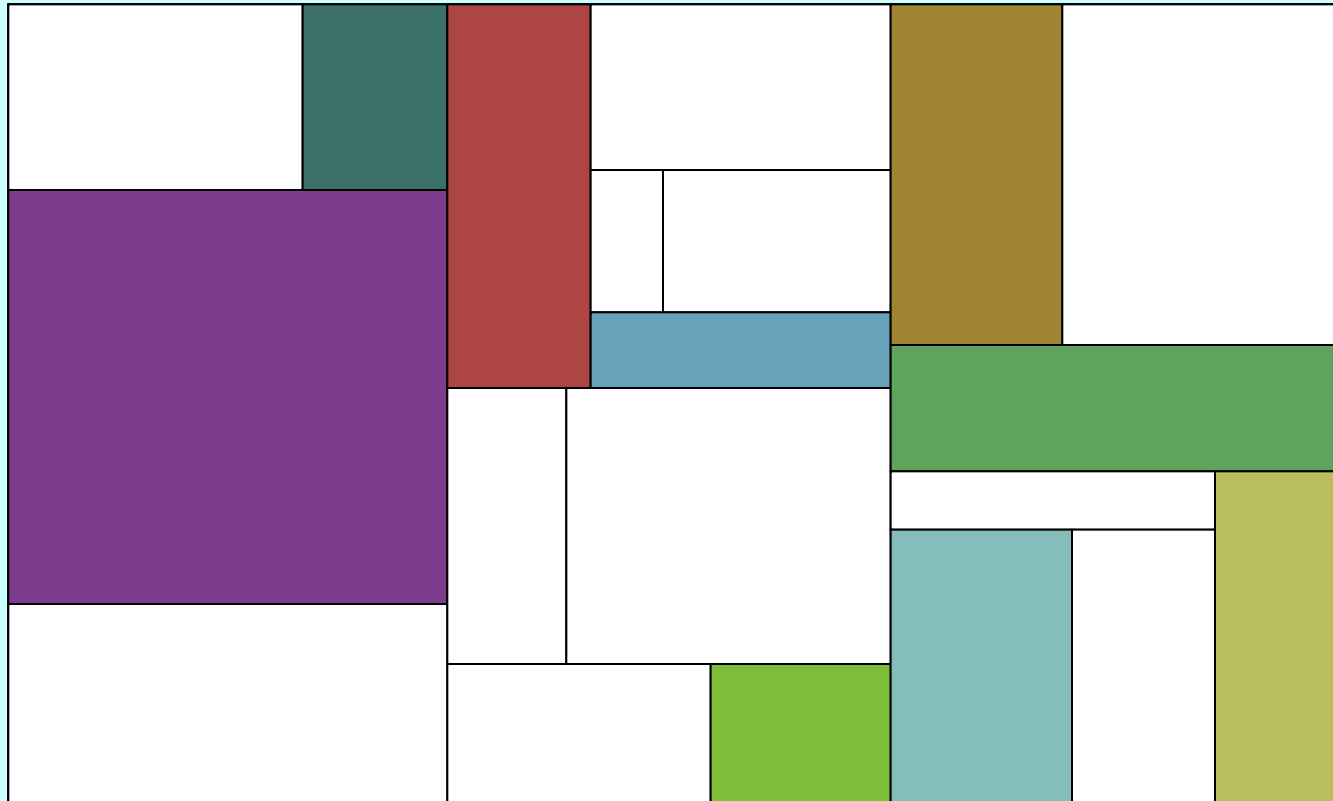
Sets the pen color to the specified color string (such as "**BLACK**" or "**RED**")

**gw.setColor** ("**#rrggbb**")

Sets the red/green/blue components to the specified hexadecimal values.

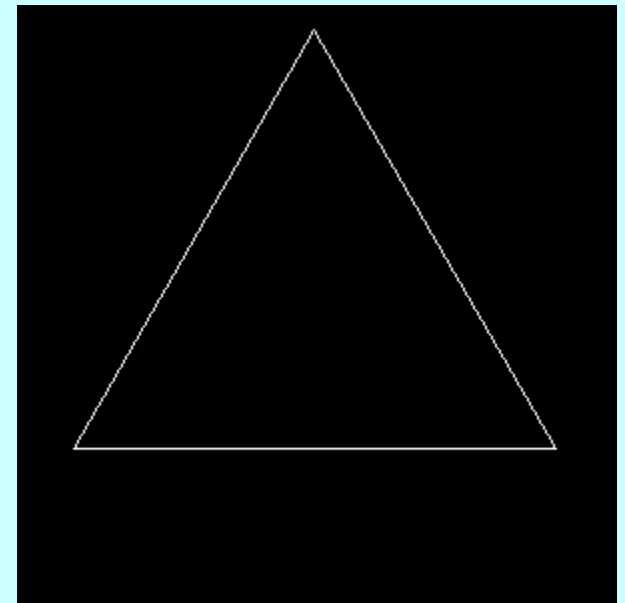


# Revised Mondrian Decomposition



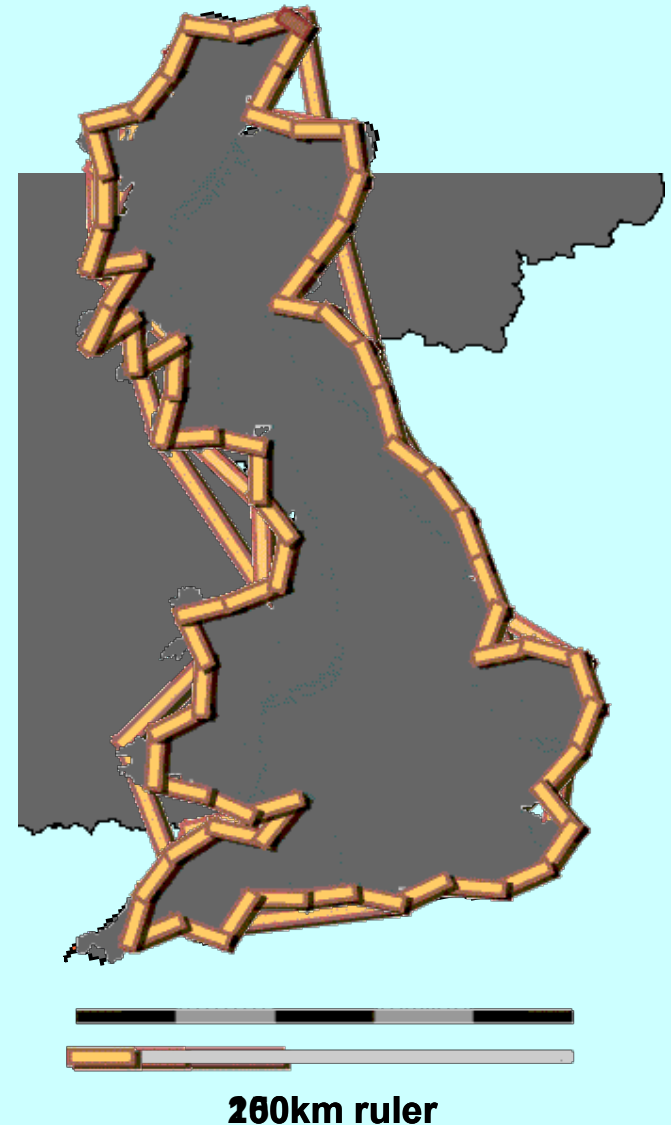
# Graphical Recursion

- Recursion comes up in many graphical applications, most notably in the creation of *fractals*, which are mathematical structures consisting of similar figures at various different scales. Fractals were popularized in a 1982 book by the late **Benoit Mandelbrot** (1924-2010) entitled *The Fractal Geometry of Nature*.
- One of the simplest fractal patterns to draw is the *Koch fractal*, named after its inventor, the Swedish mathematician **Helge von Koch** (1870-1924). The Koch fractal is sometimes called a *snowflake fractal* because of the beautiful, six-sided symmetries it displays as the figure becomes more detailed. as illustrated in the following diagram:



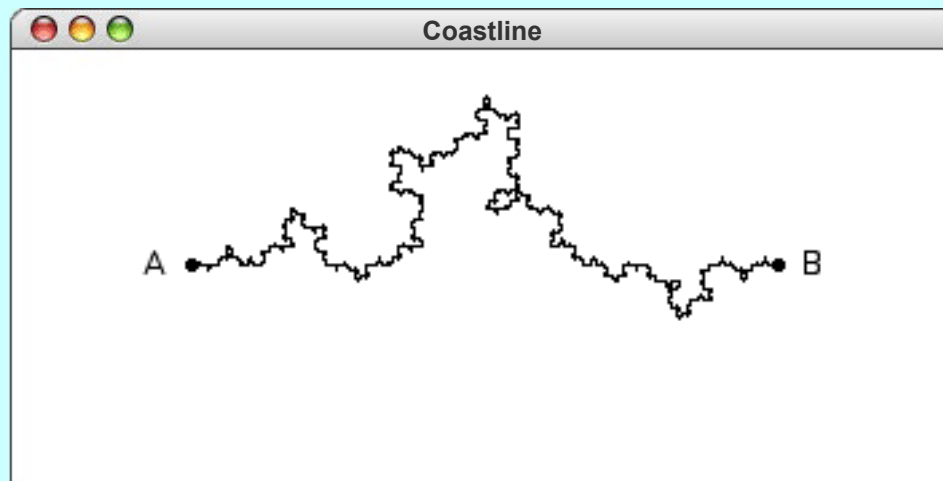
# How Long is the Coast of England?

- The first widely circulated paper about fractals was a 1967 article in *Science* by Mandelbrot that asked the seemingly innocuous question, “How long is the coast of England?”
- The point that Mandelbrot made in the article is that the answer depends on the measurement scale, as these images from Wikipedia show.
- This thought-experiment serves to illustrate the fact that coastlines are *fractal* in that they exhibit the same structure at every level of detail.



# Exercise: Fractal Coastline

- Exercise 15 on page 384 asks you to draw a fractal coastline between two points, A and B, on the graphics window.
  - The order-0 coastline is just a straight line.
  - The order-1 coastline replaces that line with one containing a triangular wedge pointing randomly up or down.
  - The order-2 coastline does the same for each line in the order-1.
  - Repeating this process eventually yields an order-5 coastline.



The End