Advanced Programming

Fall 97

University of Tehran

Recursion

By: Hadi Safari

hadisafari.ir

What is Recursion?

What is Recursion?

- Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller **subproblems** until you get to a small enough problem that it can be solved trivially.

What is Recursion?

- Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller **subproblems** until you get to a small enough problem that it can be solved trivially.

- Usually recursion involves a function **calling itself**.

The Three Laws of Recursion

The Three Laws of Recursion

- A recursive algorithm must have a **base case**.

The Three Laws of Recursion

- A recursive algorithm must have a **base case**.
- A recursive algorithm must **change its state** and move **toward the base case**. (Avoiding loops)

The Three Laws of Recursion

- A recursive algorithm must have a **base case**.
- A recursive algorithm must **change its state** and move **toward the base case**. (Avoiding loops)
- A recursive algorithm must **call itself**, recursively.

1. Integer Factorial

1. Integer Factorial

```
unsigned int factorial(unsigned int n) {
    if(n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

1. Integer Factorial

Base Case

```
unsigned int factorial(unsigned int n) {
    if(n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

1. Integer Factorial

Base Case

Change State toward Base Case

```
unsigned int factorial(unsigned
    if(n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

1. Integer Factorial

Base Case

```
unsigned int factorial(unsigned
    if(n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Change State toward Base Case

Recursive Call

1. Integer Factorial

```
unsigned int factorial(unsigned
    if(n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Base Case

Change State toward Base Case

Do Sth with Result of Subproblem

Recursive Call

2. Fibonacci Seq.

2. Fibonacci Seq.

```
unsigned int fibonacci(unsigned int n) {
    if(n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

2. Fibonacci Seq.

Multiple Base Cases

```
unsigned int fibonacci(unsigned int n) {
    if(n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

2. Fibonacci Seq.

Multiple Base Cases

```
unsigned int fibonacci(unsigned int n) {
    if(n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Multiple Recursive Call

2. Fibonacci Seq.

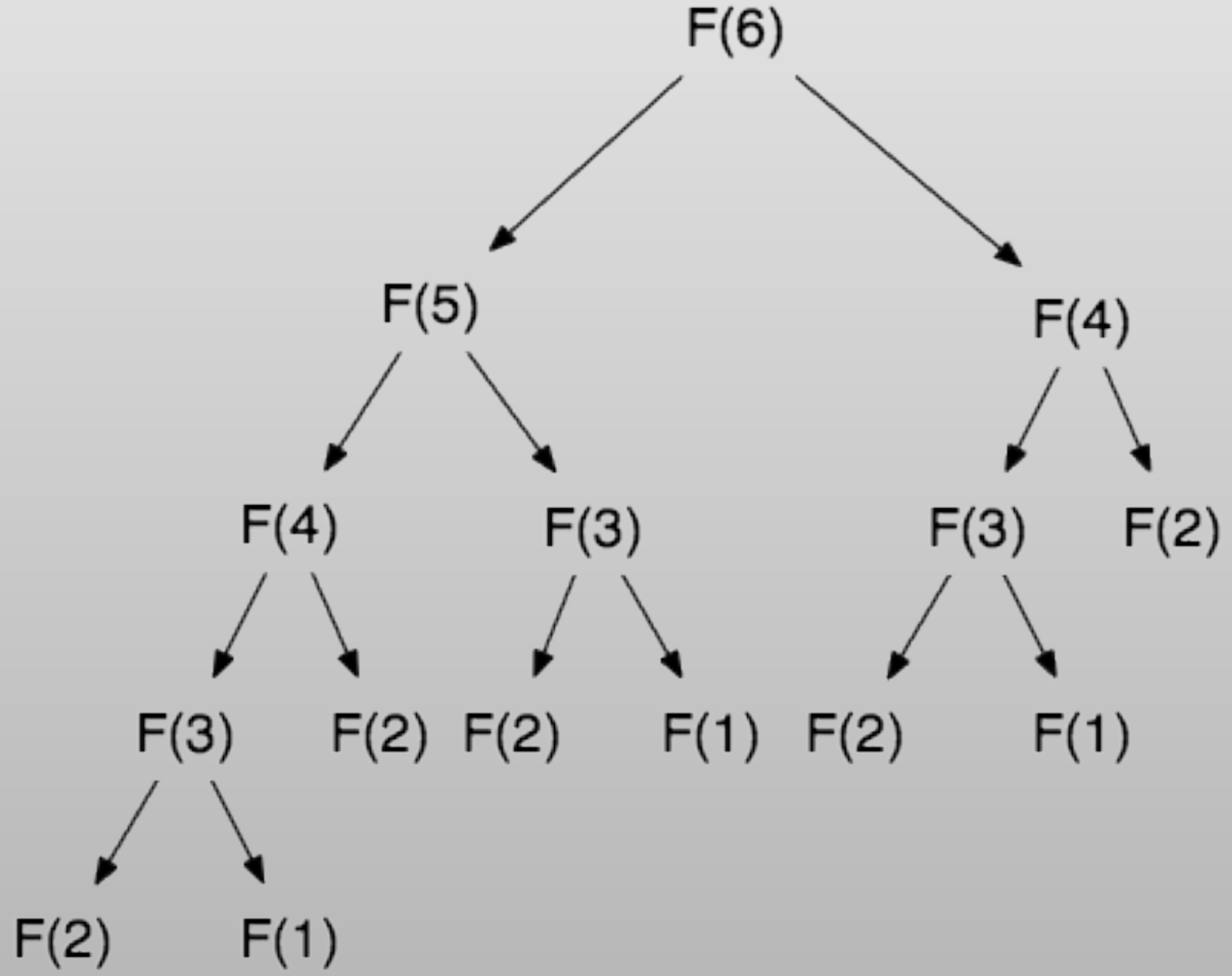
Multiple Base Cases

```
unsigned int fibonacci(unsigned int n) {
    if(n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

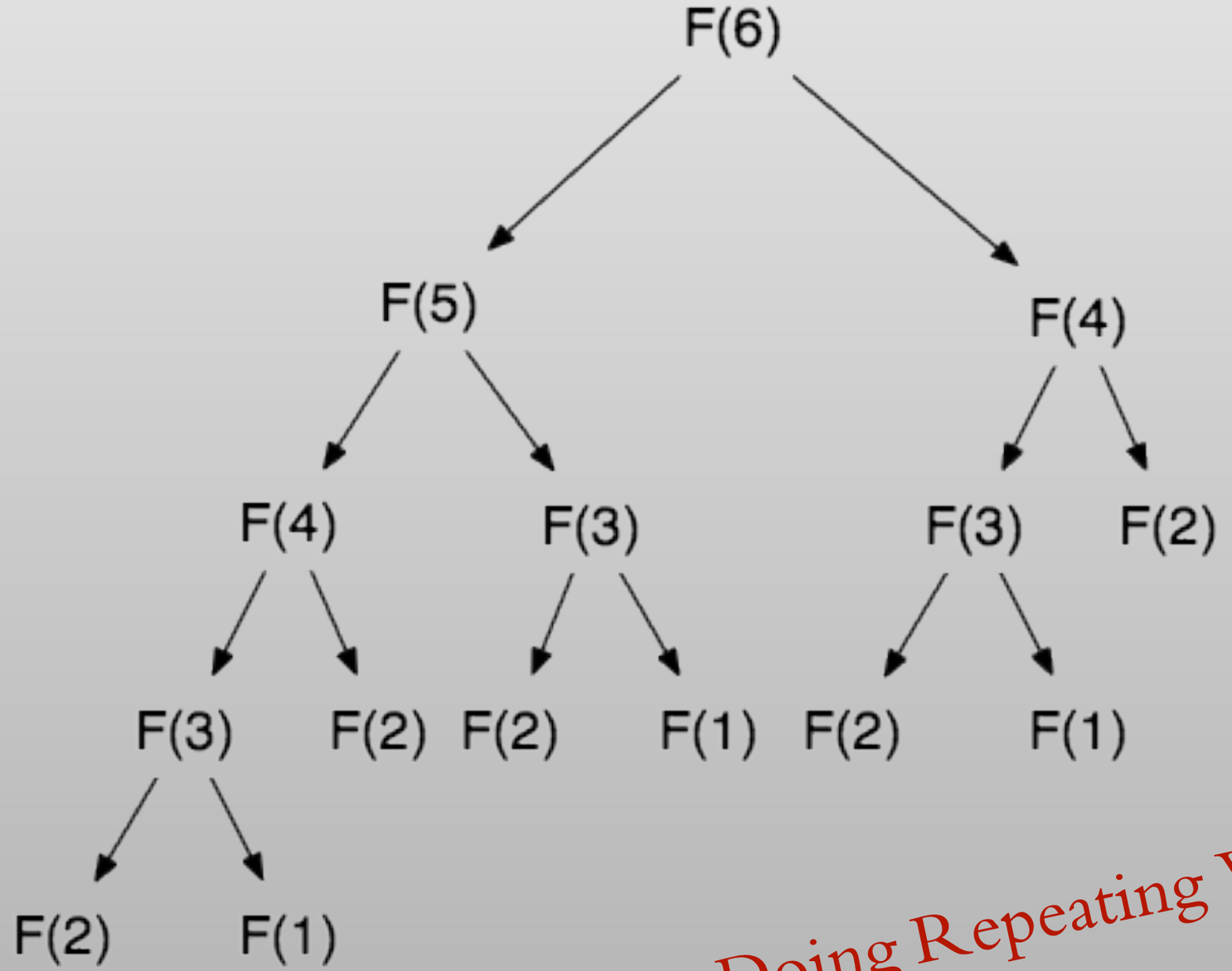
Do Sth with Result of Subproblems

Multiple Recursive Call

2. Fibonacci Seq.



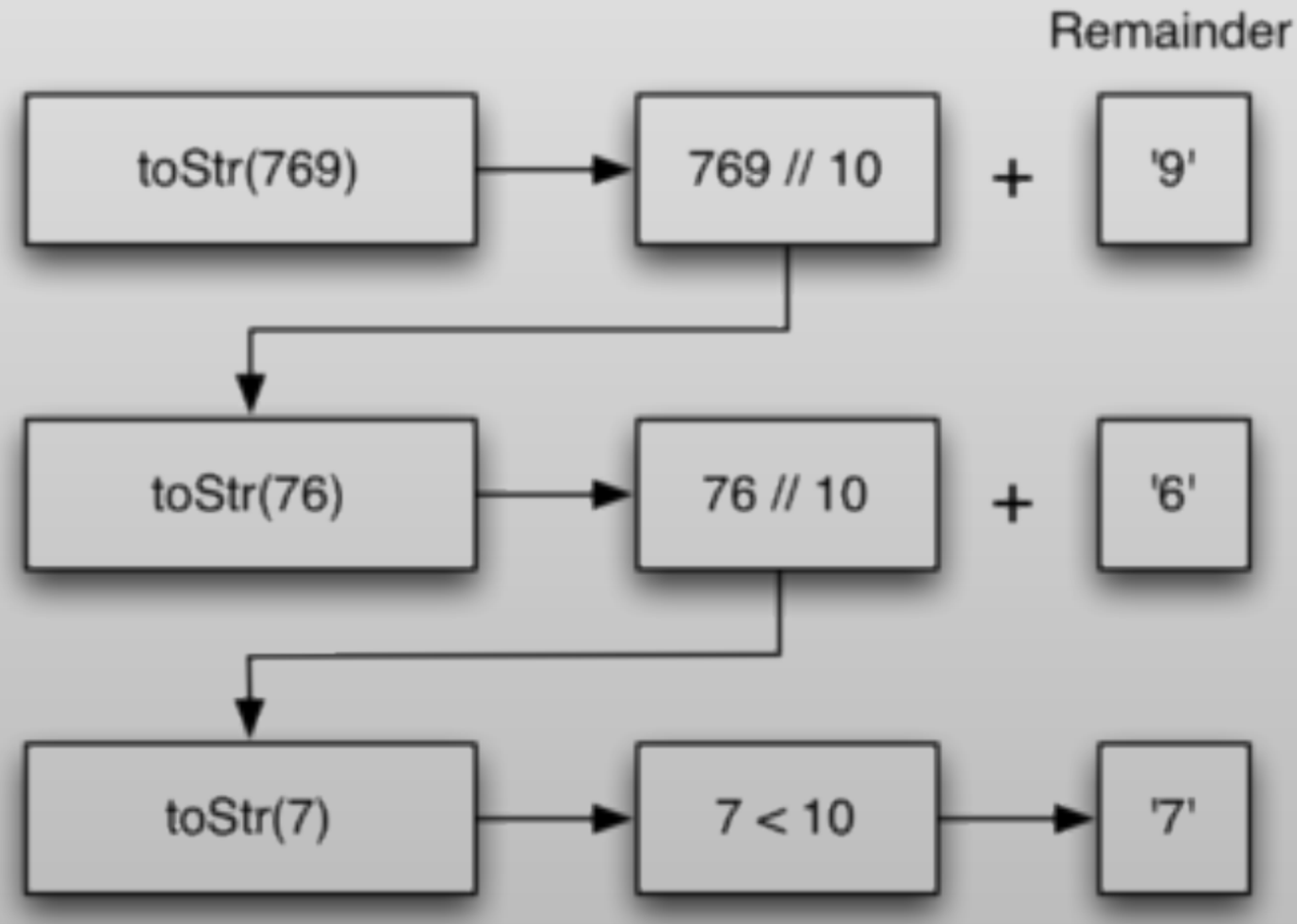
2. Fibonacci Seq.



Doing Repeating Work ?

3. Converting an Integer to a String in Any Base

3. Converting an Integer to a String in Any Base



3. Converting an Integer to a String in Any Base

```
string to_string(unsigned int n, unsigned int base) {
    static const string digits = "0123456789ABCDEF";
    if(base > digits.size())
        throw invalid_argument("invalid base");
    if(n < base)
        return string(digits[n], 1);
    else
        return to_string(n / base, base) + string(digits[n % base], 1);
}
```

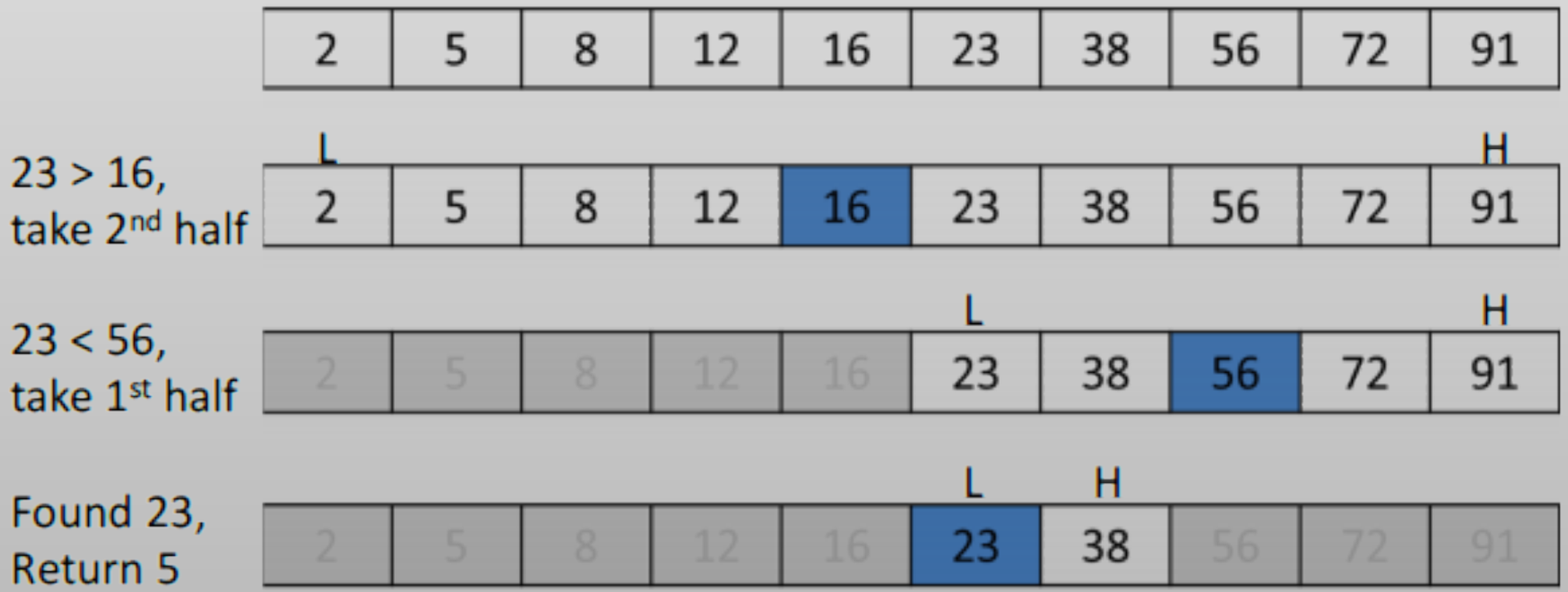
4. Binary Search

```
int binary_search(vector<int> list, int key);
```

4. Binary Search

```
int binary_search(vector<int> list, int key);
```

If searching for 23 in the 10-element array:



4. Binary Search

```
int binary_search(vector<int> list, int from, int to, int key)
{
    if (list.size() == 0 || from > to)
        return -1;

    int mid = (from + to) / 2;
    if (list[mid] == key)
        return mid;
    else if (list[mid] < key)
        return binary_search(list, mid + 1, to, key);
    else
        return binary_search(list, from, mid - 1, key);
}
```

4. Binary Search

```
int binary_search(vector<int> list, int from, int to, int key)
{
    if (list.size() == 0 || from > to)
        return -1;

    int mid = (from + to) / 2;
    if (list[mid] == key)
        return mid;
    else if (list[mid] < key)
        return binary_search(list, mid + 1, to, key);
    else
        return binary_search(list, from, mid - 1, key);
}
```

Solving More General Problem

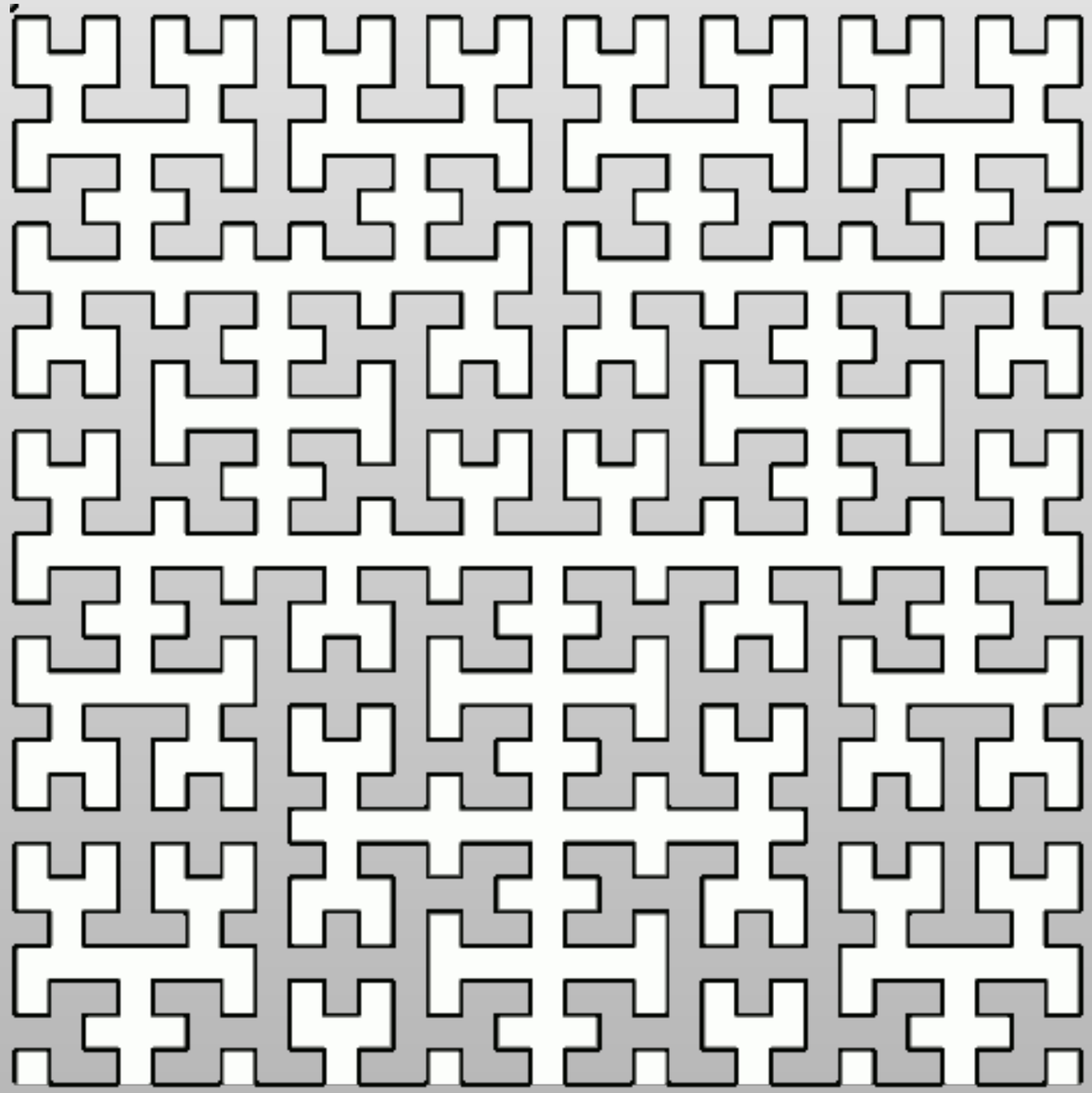
4. Binary Search

```
int binary_search(vector<int> list, int from, int to, int key)
{
    if (list.size() == 0 || from > to)
        return -1;

    int mid = (from + to) / 2;
    if (list[mid] == key)
        return mid;
    else if (list[mid] < key)
        return binary_search(list, mid + 1, to, key);
    else
        return binary_search(list, from, mid - 1, key);
}
int binary_search(vector<int> list, int key) {
    return binary_search(list, 0, list.size() - 1, key);
}
```

Solving More General Problem

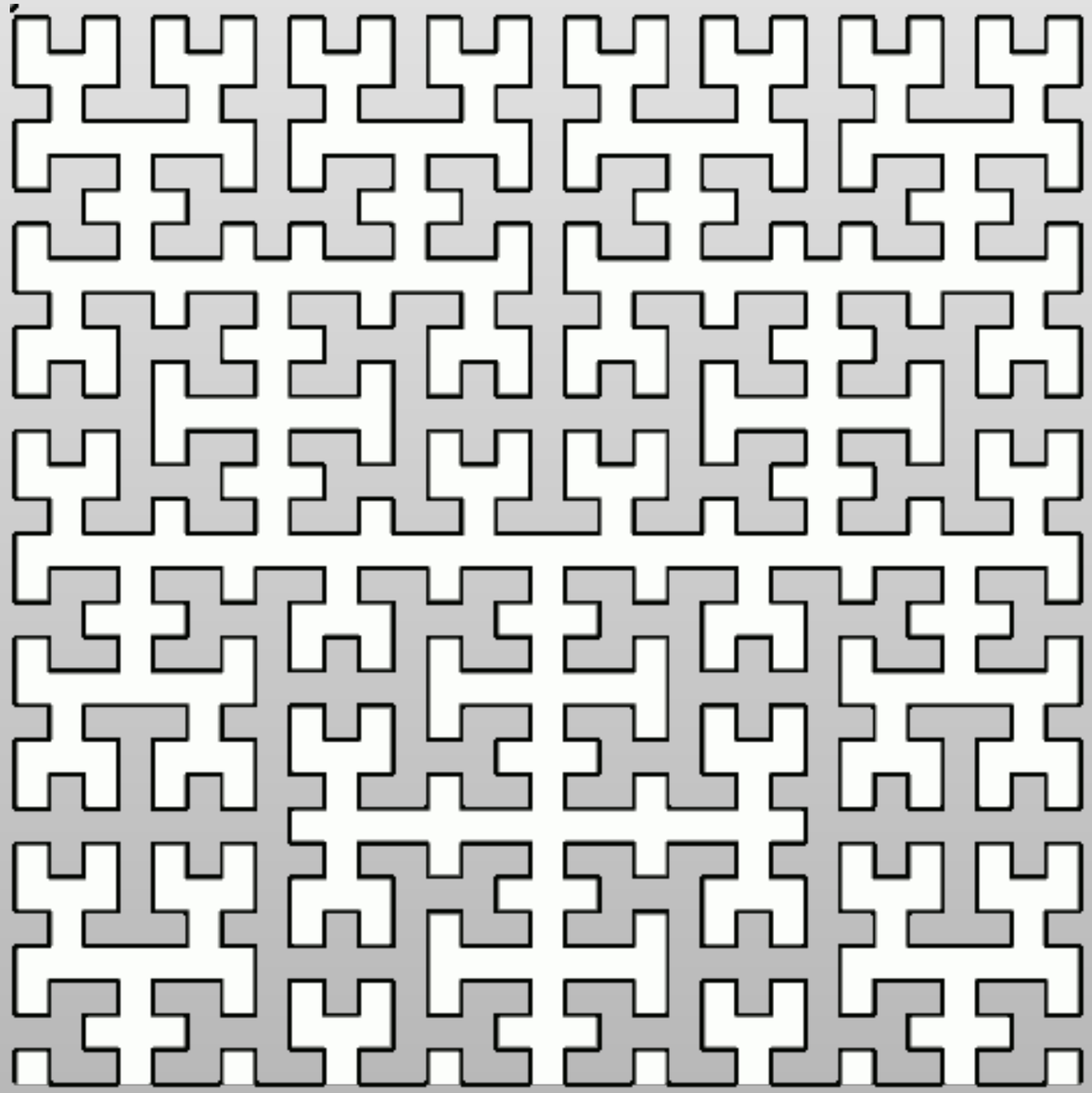
5. Hilbert Curve



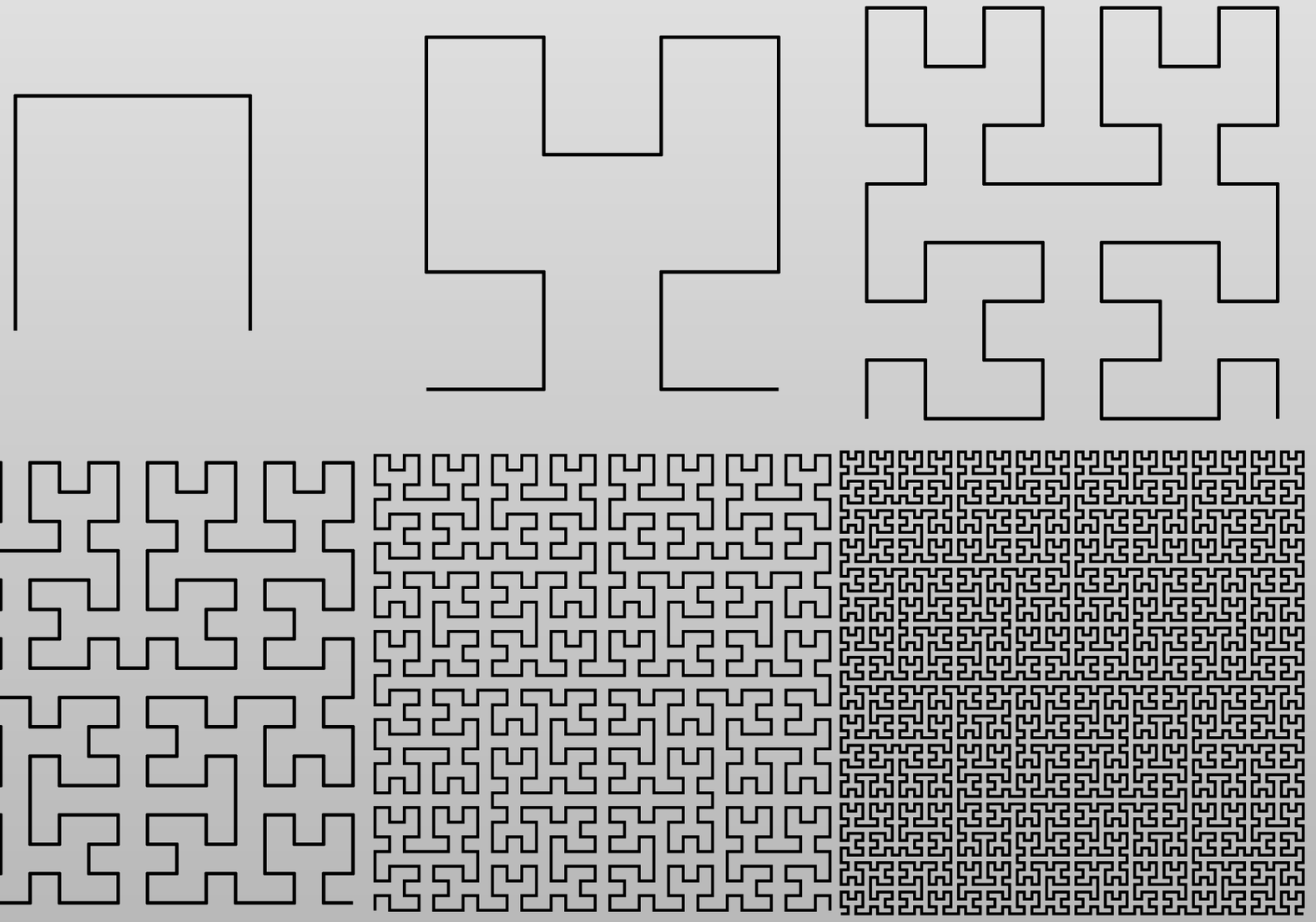
5. Hilbert Curve

David Hilbert

5. Hilbert Curve

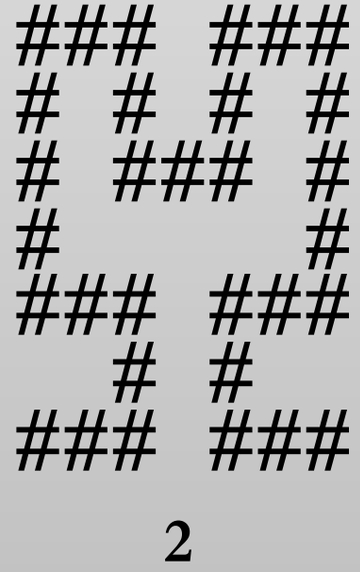


5. Hilbert Curve



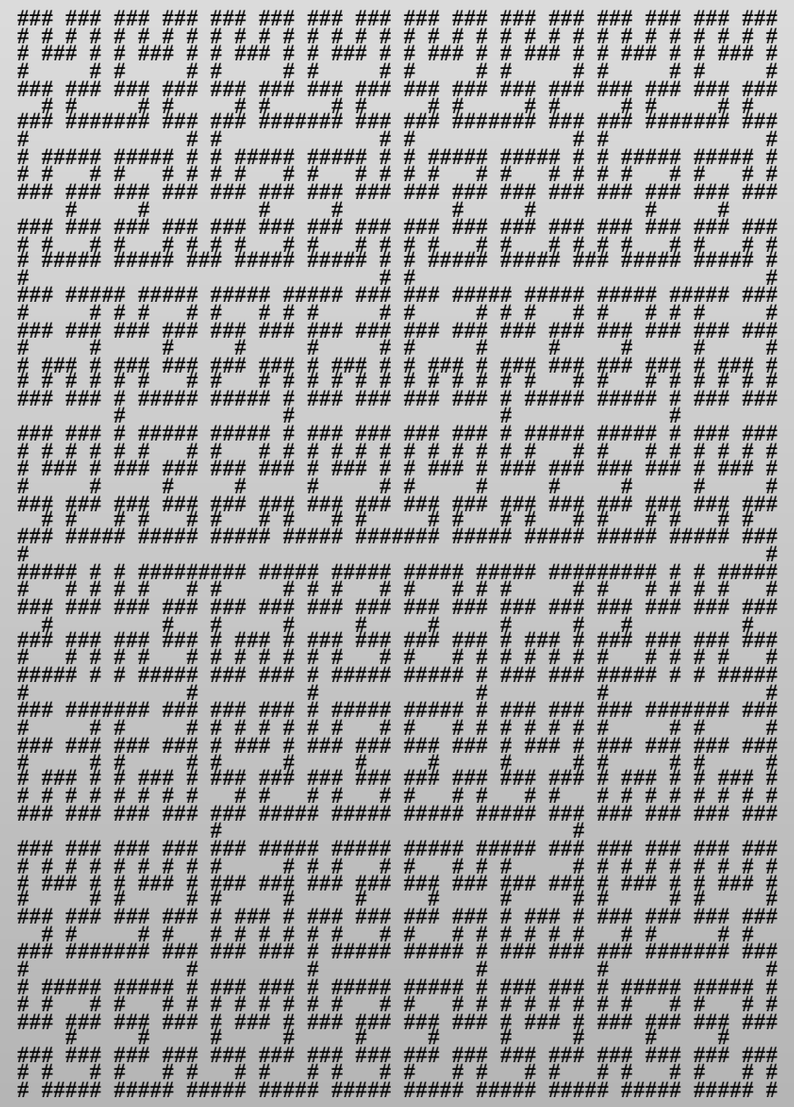
5. Hilbert Curve

5. Hilbert Curve



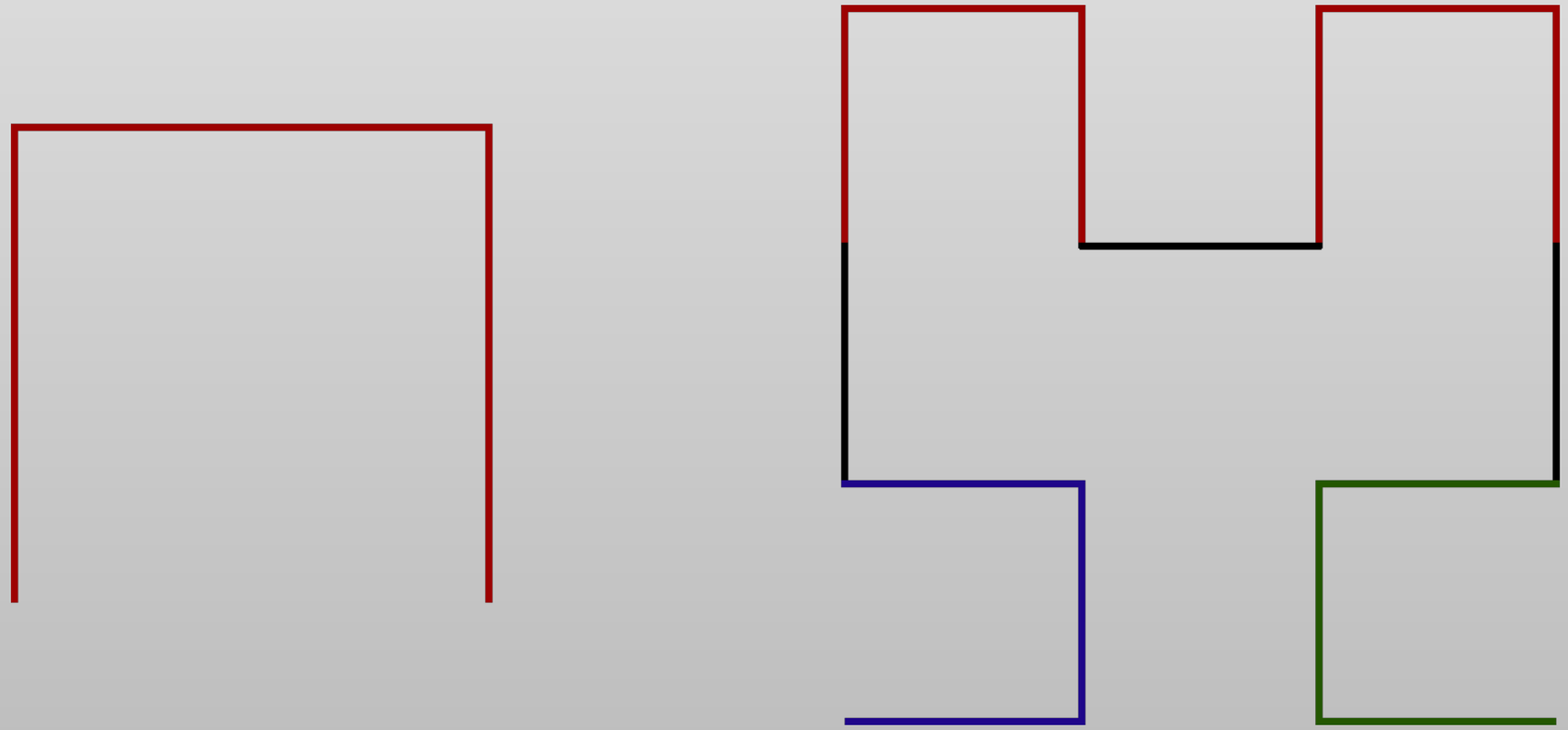
2

5. Hilbert Curve

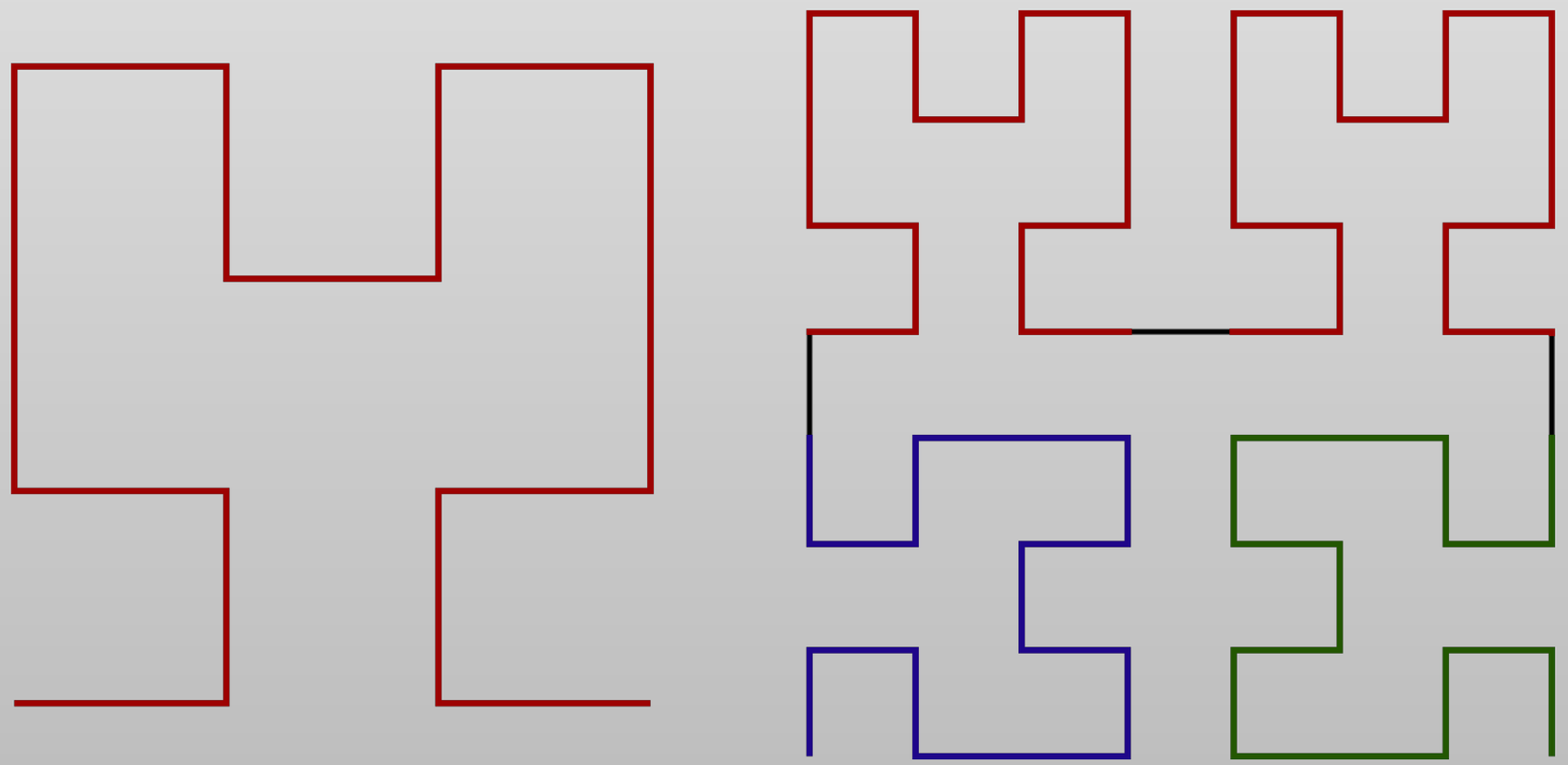


5

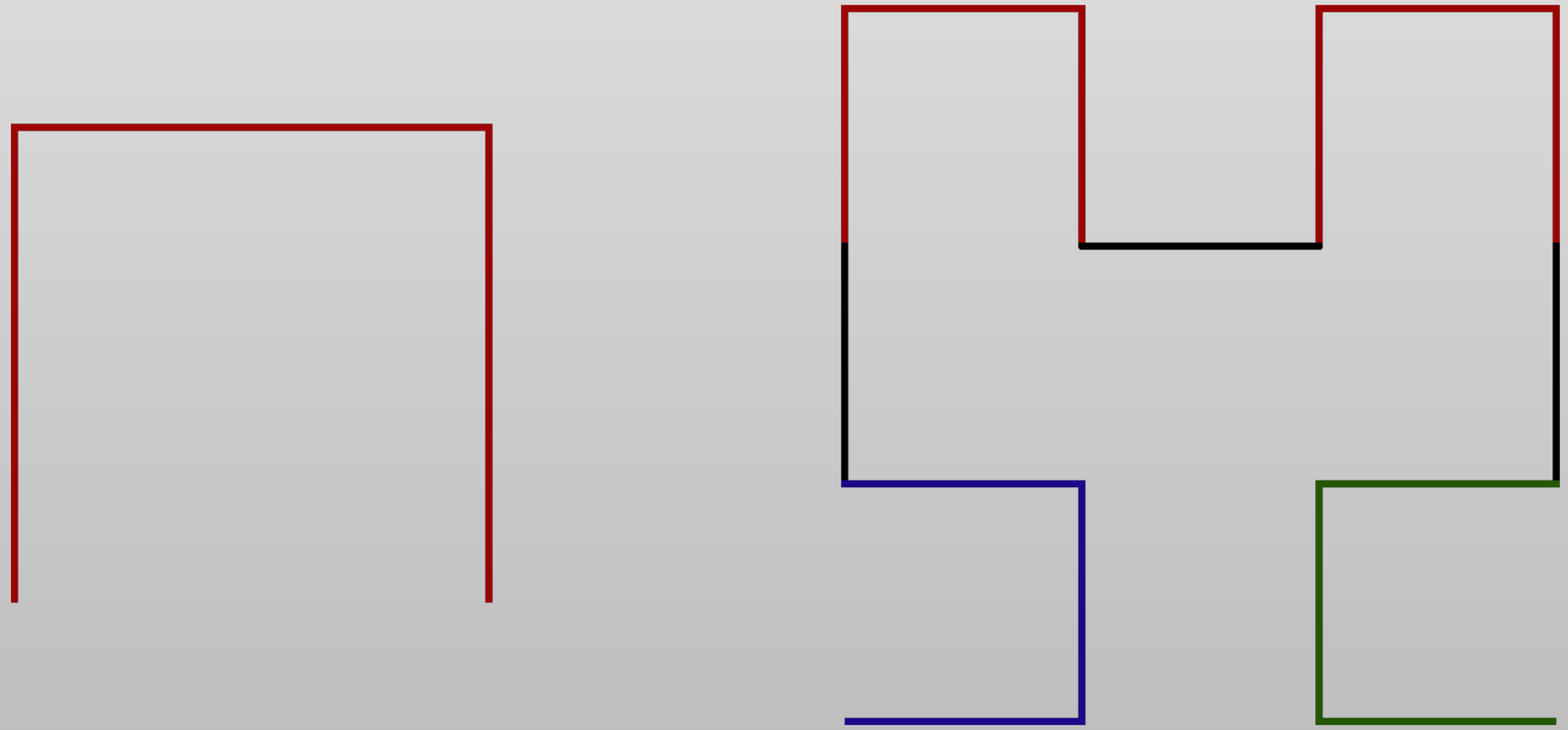
5. Hilbert Curve



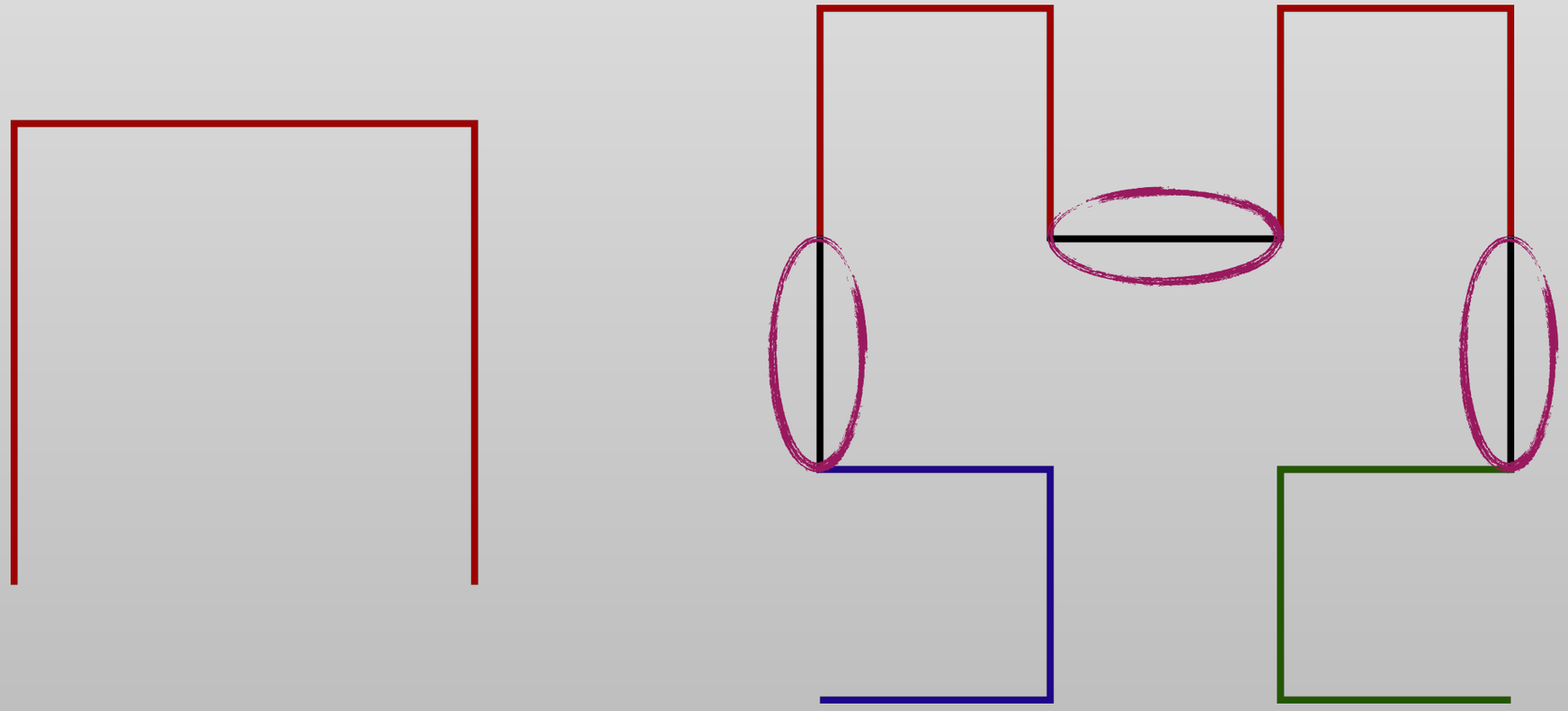
5. Hilbert Curve



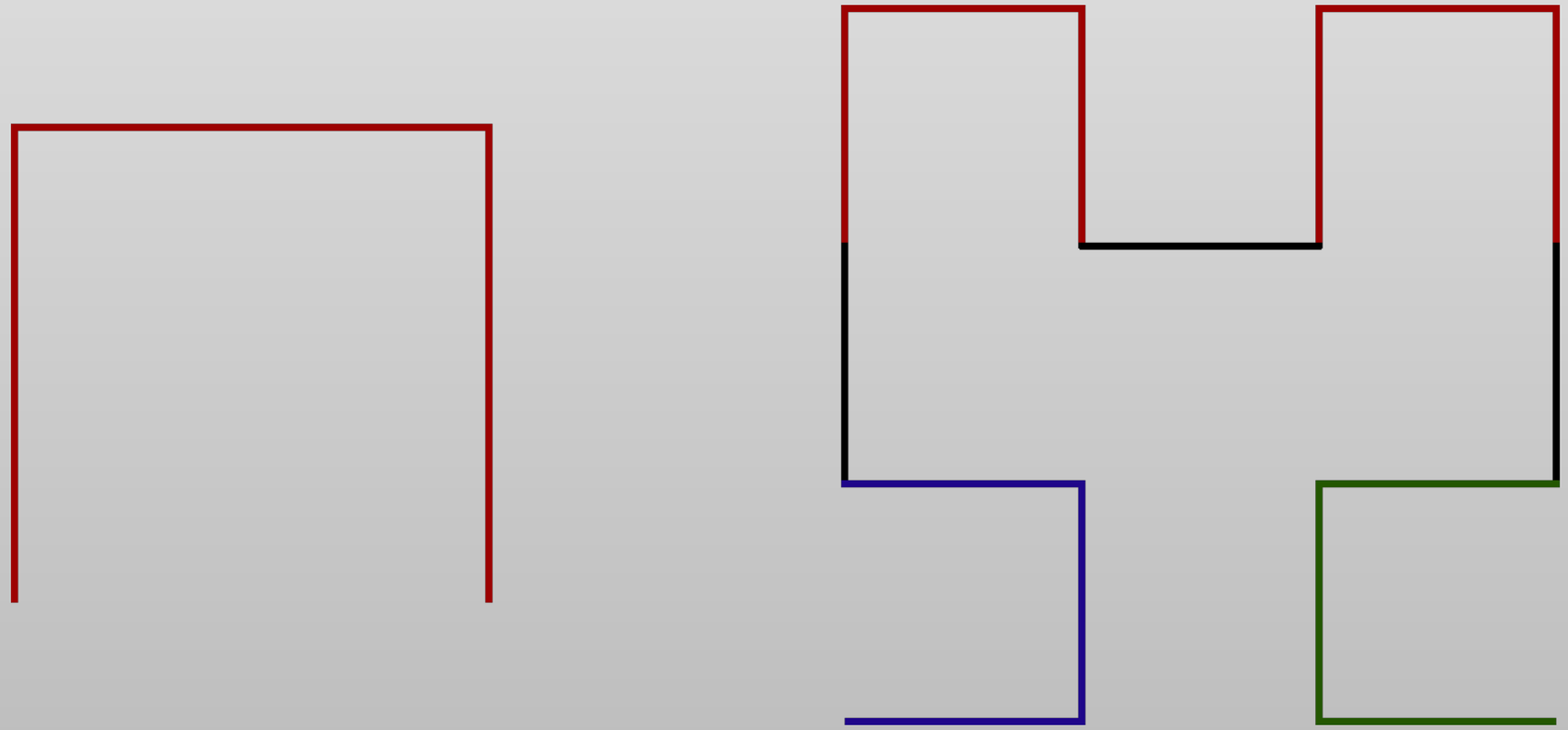
5. Hilbert Curve



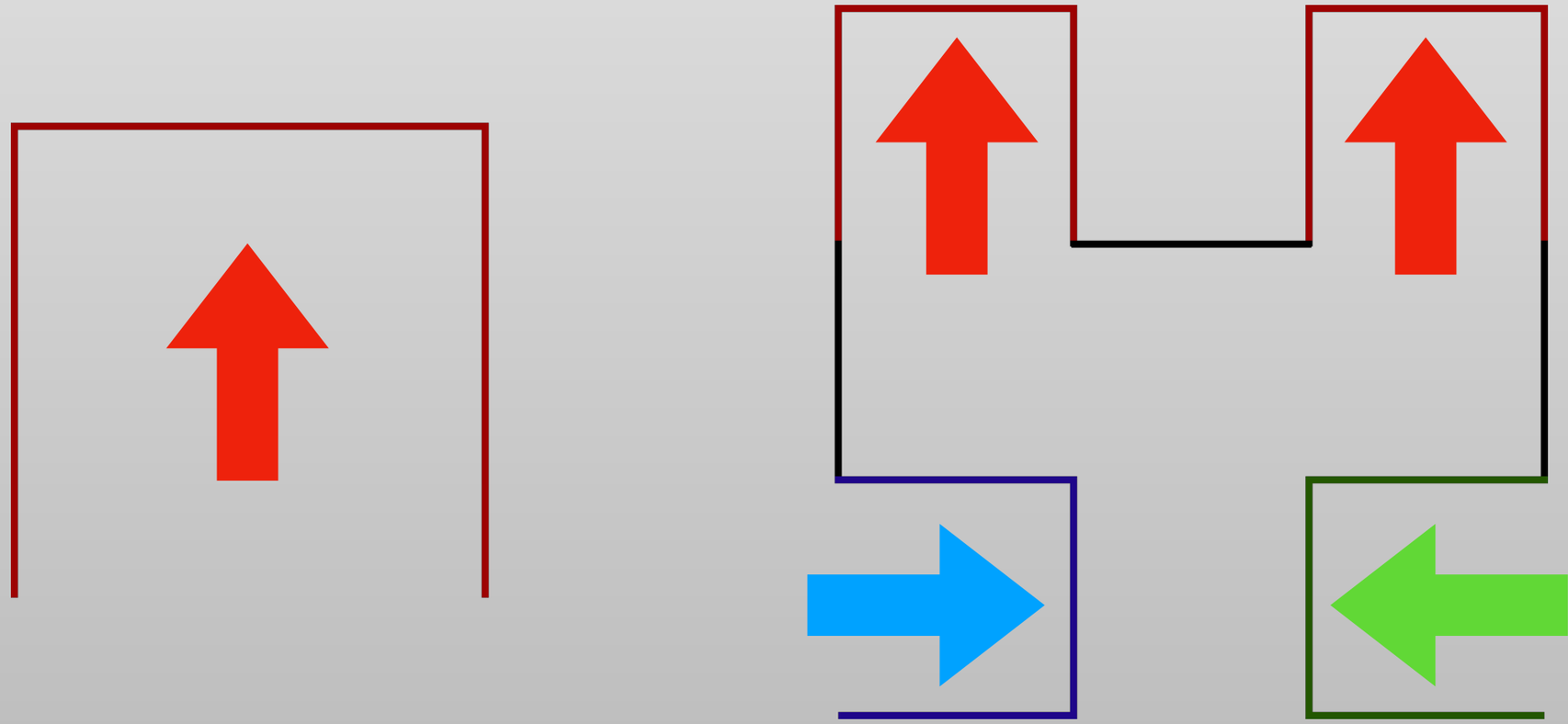
5. Hilbert Curve



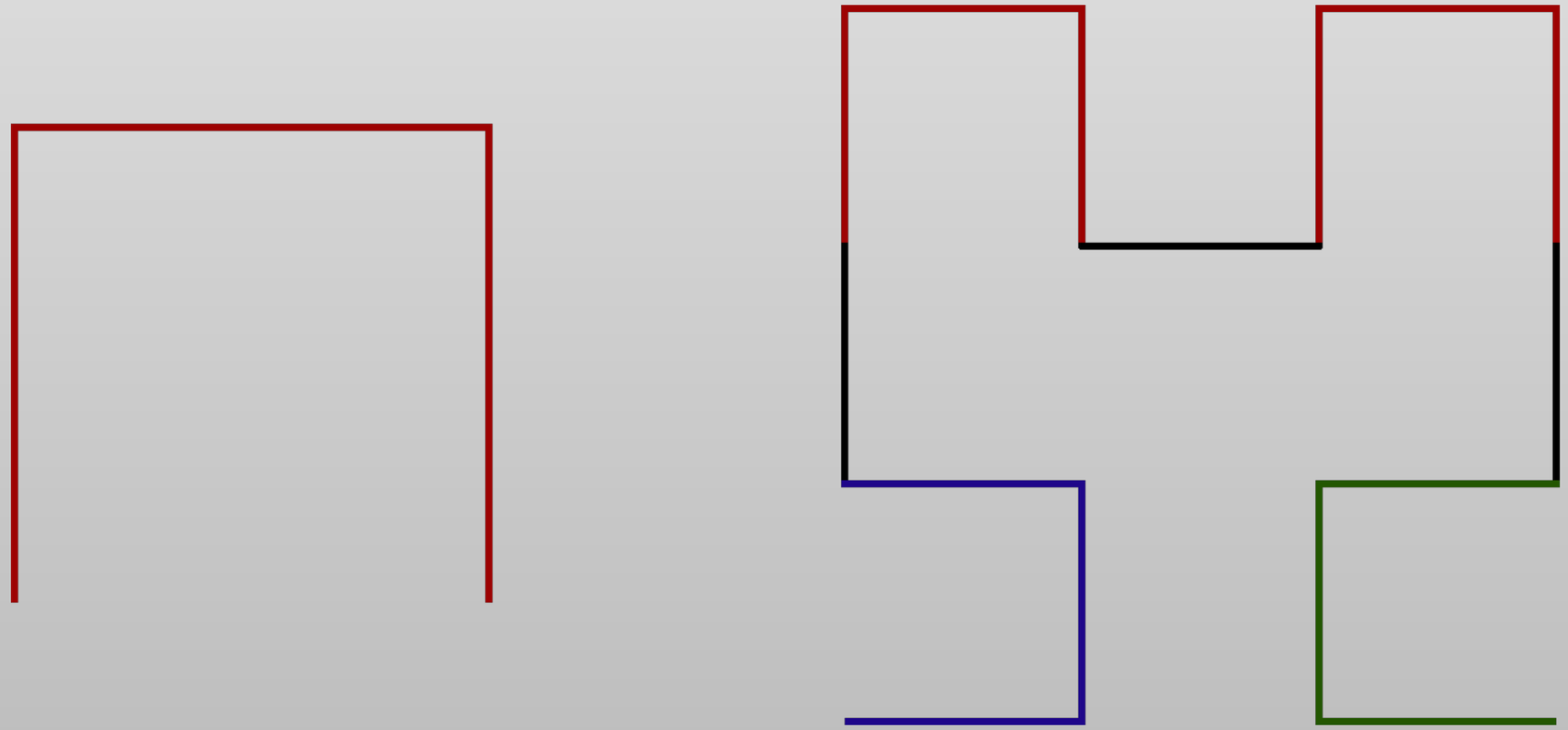
5. Hilbert Curve



5. Hilbert Curve



5. Hilbert Curve



5. Hilbert Curve

```
typedef pair<int, int> Coord;
typedef vector<bool> Row;
typedef vector<Row> Matrix;
enum Direction { Up, Down, Right, Left };
```

5. Hilbert Curve

```
const Direction recursive_directions[4][2][2] = {
    {{Up, Up}, {Right, Left}},      // Up
    {{Left, Right}, {Down, Down}}, // Down
    {{Down, Right}, {Up, Right}},  // Right
    {{Left, Down}, {Left, Up}}     // Left
};
```

5. Hilbert Curve

```
int get_grid_length(int n) {
    return pow(2, n) - 1;
}
```

5. Hilbert Curve

```
void make_grid(unsigned int n, Matrix &grid, Coord start, Direction direction) {
  if (n == 1) {
    grid[start.first][start.second] = true;
    return;
  }
  int length = get_grid_length(n);
  if (direction == Up || direction == Down) {
    grid[start.first + length / 2][start.second] = true;
    grid[start.first + length / 2][start.second + length - 1] = true;
    grid[start.first + length / 2 + (direction == Up ? -1 : 1)]
        [start.second + length / 2] = true;
  } else {
    grid[start.first][start.second + length / 2] = true;
    grid[start.first + length - 1][start.second + length / 2] = true;
    grid[start.first + length / 2]
        [start.second + length / 2 + (direction == Left ? -1 : 1)] = true;
  }
  make_grid(n - 1, grid, start, recursive_directions[direction][0][0]);
  make_grid(n - 1, grid, Coord(start.first, start.second + length / 2 + 1),
            recursive_directions[direction][0][1]);
  make_grid(n - 1, grid, Coord(start.first + length / 2 + 1, start.second),
            recursive_directions[direction][1][0]);
  make_grid(n - 1, grid,
            Coord(start.first + length / 2 + 1, start.second + length / 2 + 1),
            recursive_directions[direction][1][1]);
}
```

5. Hilbert Curve

```
Matrix make_grid(unsigned int n) {
  Matrix grid(get_grid_length(n), Row(get_grid_length(n), false));
  make_grid(n, grid, Coord(0, 0), Up);
  return grid;
}
```

5. Hilbert Curve

```
void make_grid(unsigned int n, Matrix &grid, Coord start, Direction direction) {
  if (n == 1) {
    grid[start.first][start.second] = true;
    return;
  }
  int length = get_grid_length(n);
  if (direction == Up || direction == Down) {
    grid[start.first + length / 2][start.second] = true;
    grid[start.first + length / 2][start.second + length - 1] = true;
    grid[start.first + length / 2 + (direction == Up ? -1 : 1)]
        [start.second + length / 2] = true;
  } else {
    grid[start.first][start.second + length / 2] = true;
    grid[start.first + length - 1][start.second + length / 2] = true;
    grid[start.first + length / 2]
        [start.second + length / 2 + (direction == Left ? -1 : 1)] = true;
  }
  make_grid(n - 1, grid, start, recursive_directions[direction][0][0]);
  make_grid(n - 1, grid, Coord(start.first, start.second + length / 2 + 1),
            recursive_directions[direction][0][1]);
  make_grid(n - 1, grid, Coord(start.first + length / 2 + 1, start.second),
            recursive_directions[direction][1][0]);
  make_grid(n - 1, grid,
            Coord(start.first + length / 2 + 1, start.second + length / 2 + 1),
            recursive_directions[direction][1][1]);
}
```

References

- https://www.topcoder.com/community/data-science/data-science-tutorials/an-introduction-to-recursion-part-1/
- http://interactivepython.org/runestone/static/pythonds/Recursion/WhatIsRecursion.html
- http://ramtung.ir/apnotes/html/05_Recursion.html
- https://medium.com/@nkhaja/memoization-and-decorators-with-python-32f607439f84
- https://www.geeksforgeeks.org/binary-search/
- http://people.cs.aau.dk/~normark/prog3-03/html/notes/fu-intr-2_themes-hilbert-sec.html
- http://datagenetics.com/blog/march22013/index.html