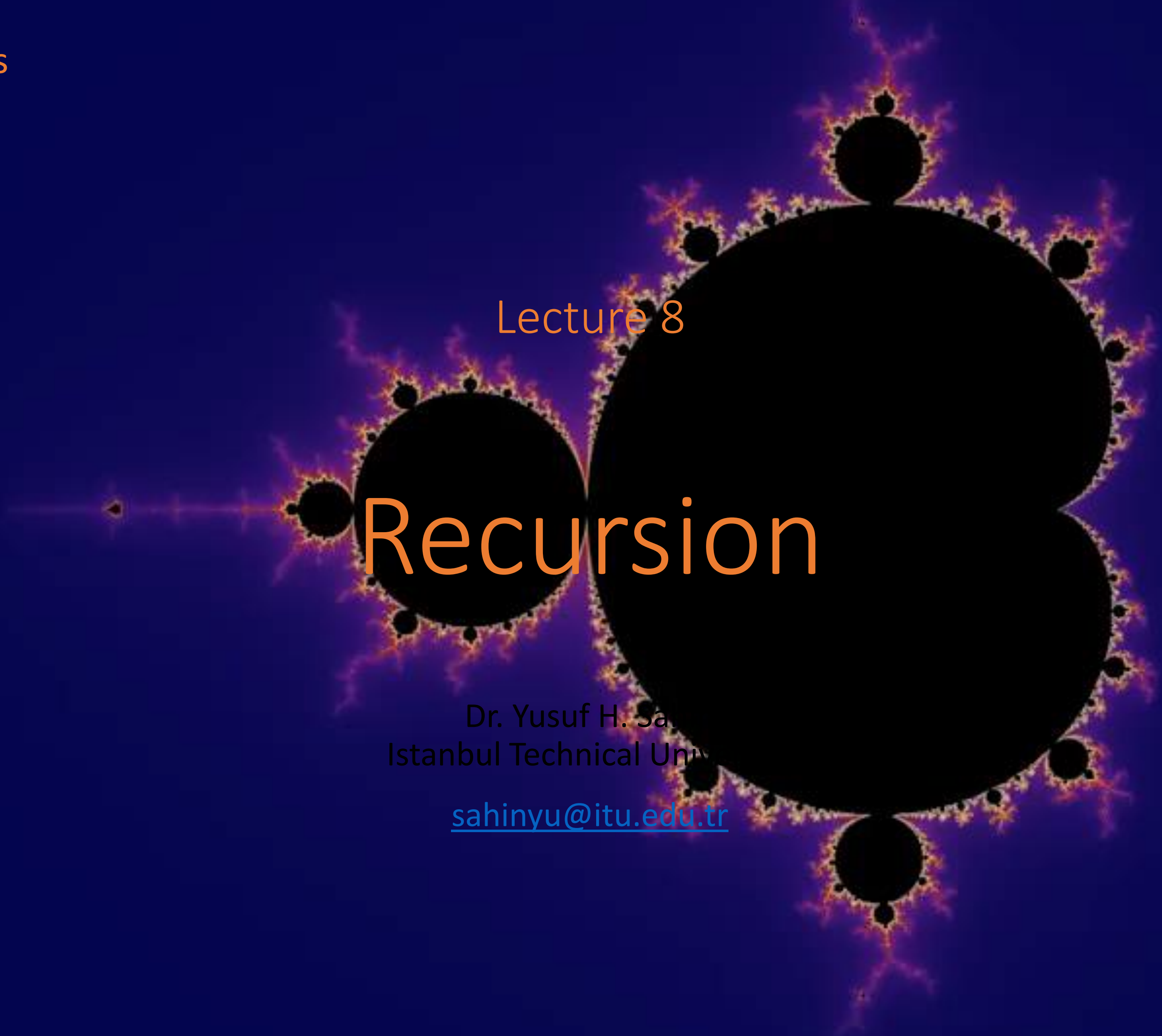


Lecture 8

Recursion

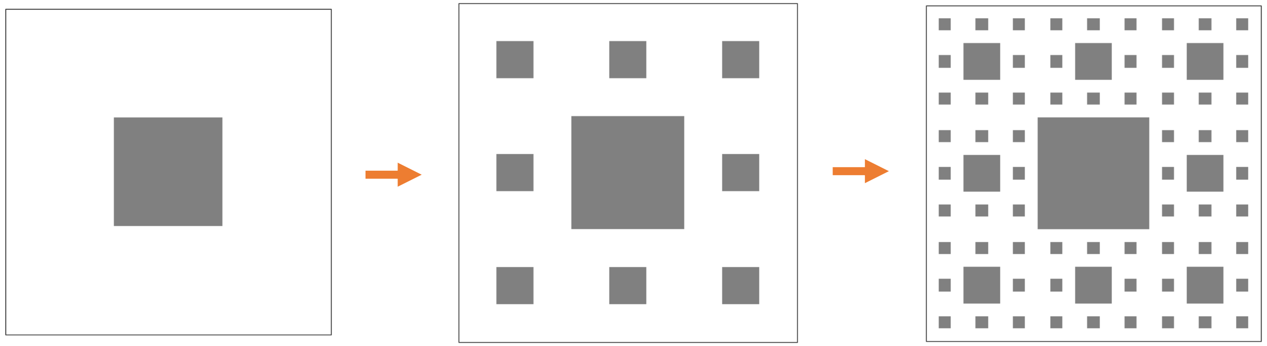
Dr. Yusuf H. Sa
Istanbul Technical Univ

sahinyu@itu.edu.tr

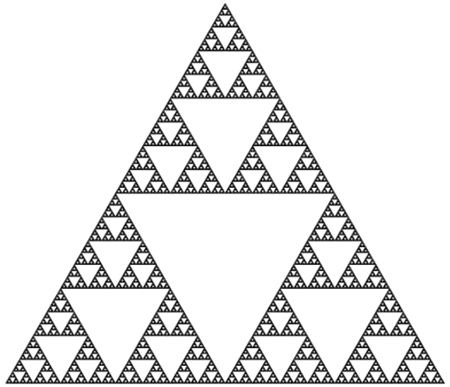
What is recursion?

- Recursion is the process of repeating a task in a self-referencing manner.

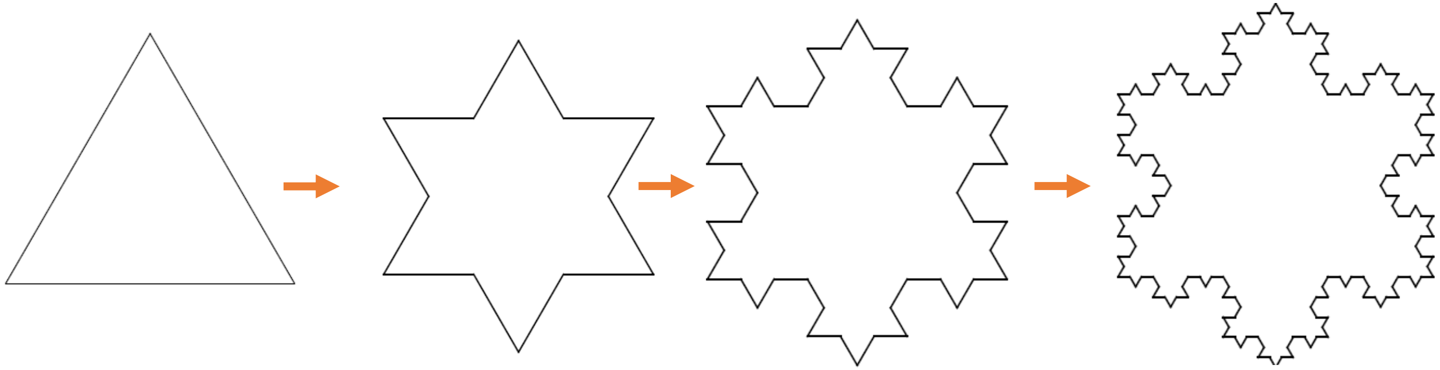
Fractal: A geometric figure that replicates its pattern at progressively smaller scales in a recursive manner.



Sierpiński's carpet



Sierpiński's triangle



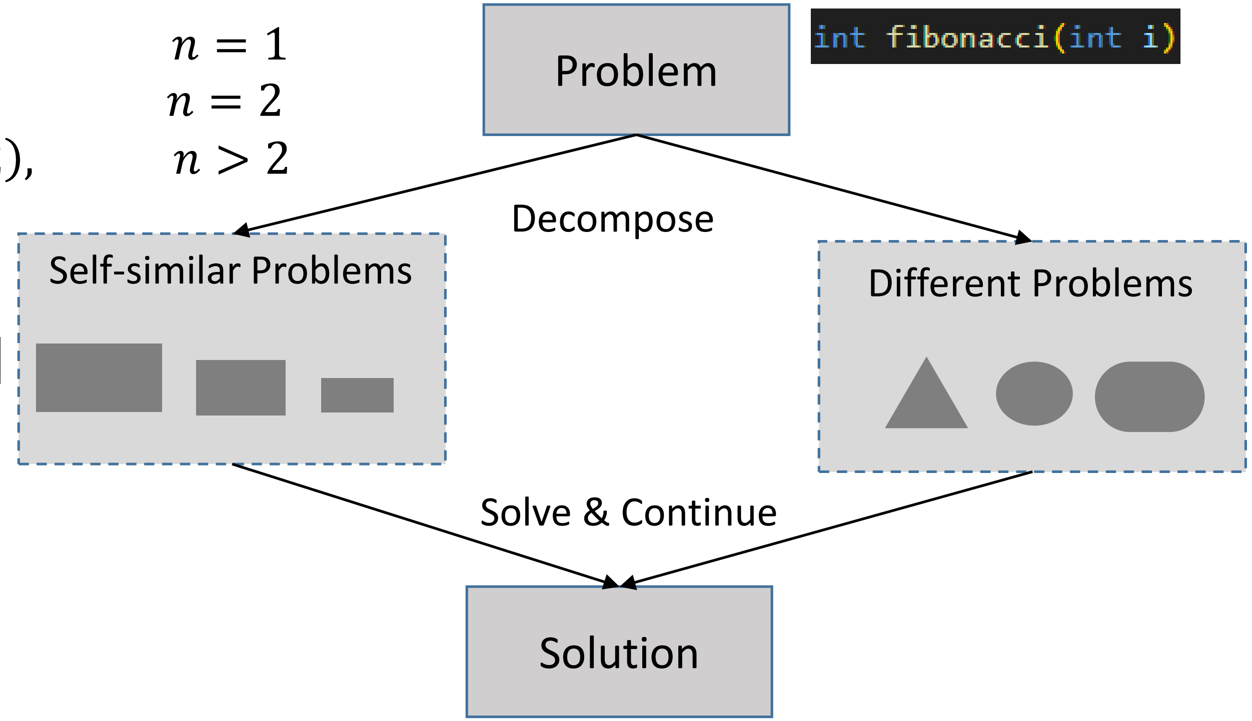
Koch's snowflake

Problem decomposition

- To effectively use recursion in programming, the problem should be decomposed into **self-similar problems** and **different problems**.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n > 2 \end{cases}$$

```
int fibonacci(int i)
```

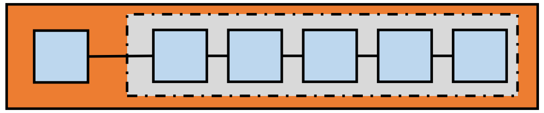



```
return fibonacci(i-1) + fibonacci(i-2);
```

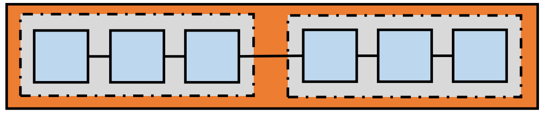
```
if (i == 1)
    return 1;
if (i == 2)
    return 1;
```

Recursion in Data & Code Structures

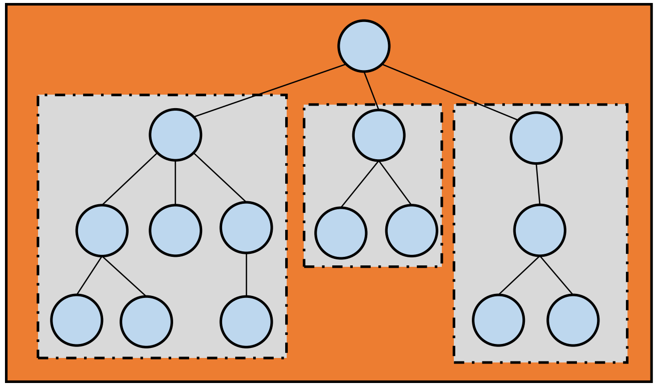
- Recursion could be used to design data structures and functions using these data structures.
- Lists and trees could be decomposed recursively.



First element + Remaining part of the list



A collection of two lists (left part and right part)



A tree could be considered as a collection of other trees pointed by a head node.

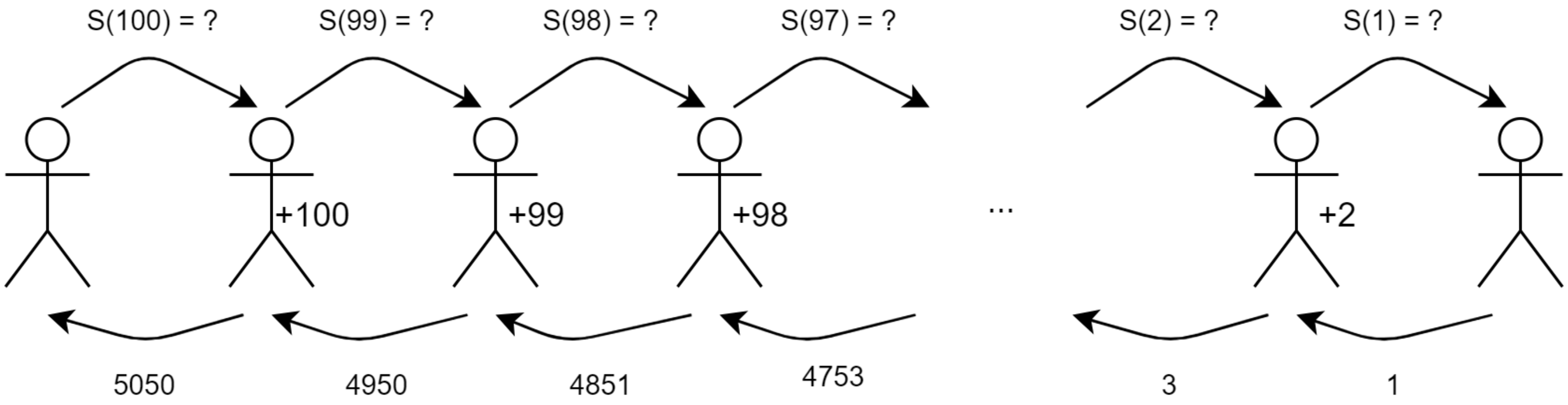
Depending on the data structure's self-referencing patterns, many problems could be solved recursively.

Proof

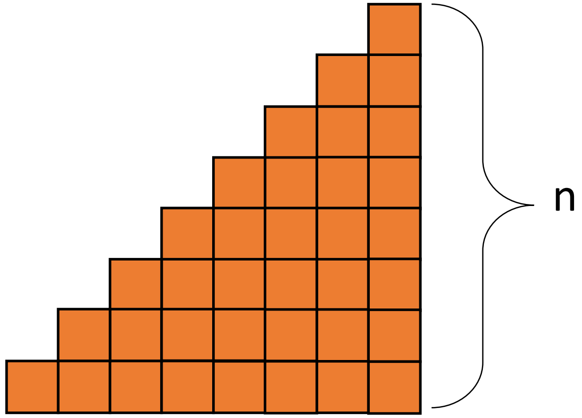
- Induction could be used to prove a recursive algorithm.

- **Base Case:** Confirm that the algorithm works successfuly for the smallest value n_0.
- **Inductive Step:** Assume that the algorithm is true for n, check whether it is true for $n + 1$.

Problem: Calculate the sum of first n positive integers.



Problem decomposition

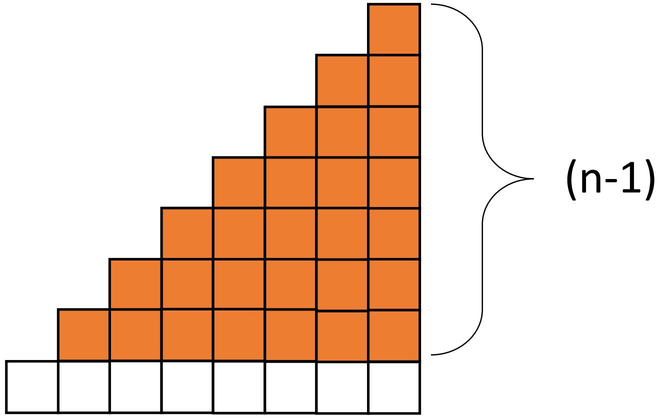


- Problem: Calculate the sum of first n positive integers.

Recursive Solution 1:

S(n) = S(n-1) + n

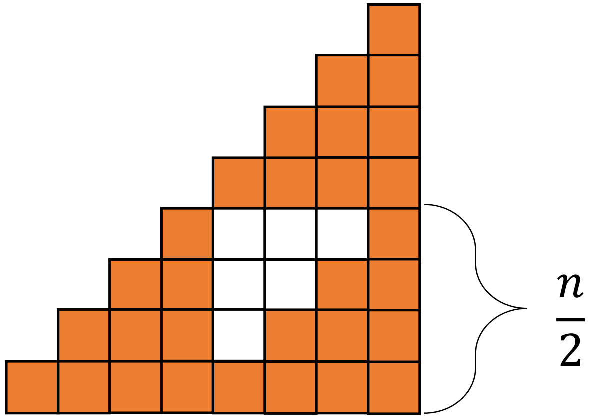
```c
int sum_first_naturals(int i)
{
    if (i == 1)
        return 1;
    else
        return sum_first_naturals(i-1) + i;
}
```

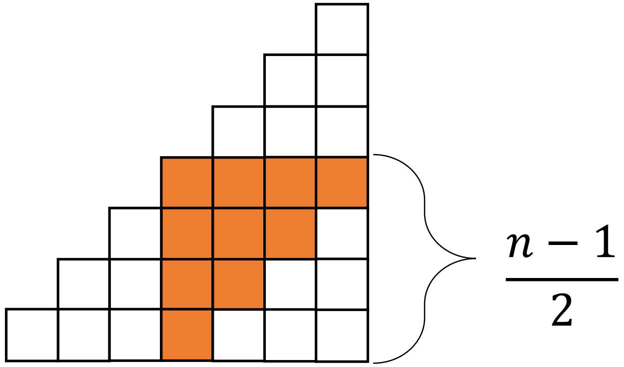


Recursive Solution 2:

$$S(n) = \begin{cases} 1, & n = 1 \\ 3, & n = 2 \\ 3S\left(\frac{n}{2}\right) + S\left(\frac{n}{2} - 1\right), & n > 2 \text{ and even} \\ 3S\left(\frac{n-1}{2}\right) + S\left(\frac{n+1}{2}\right), & n > 2 \text{ and odd} \end{cases}$$

```c
int sum_first_naturals(int i)
{
    if (i == 1)
        return 1;
    else if (i == 2)
        return 3;
    else if (i % 2 == 0)
        return 3* sum_first_naturals(i/2) + sum_first_naturals(i/2 - 1);
    else
        return 3* sum_first_naturals((i-1)/2) + sum_first_naturals((i+1)/2);
}
```





Problem decomposition

Greatest Common Divisor

```
int gcd(int x, int y)
{
    if (y == 0)
        return x;
    else
        return gcd(y, (x % y));
}
```

Factorial

```
int factorial(int x)
{
    if (x >1)
        return x* factorial(x-1);
    else
        return 1;
}
```

Power of a Natural Number

```
int power(int x, int y)
{
    if(y == 0)
        return 1;
    else
        return x*power(x,y-1);
}
```

Power, computationally simpler way

```
int power(int x, int y)
{
    if(y == 0)
        return 1;
    else if (y % 2 == 0)
    {
        int res = power(x, y/2);
        return res*res;
    }
    else
        return x*power(x,y-1);
}
```

Recursion vs. Iteration

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

- Iterations and recursion are equivalent that, any problem solved by recursion could also be solved in an iterative manner.
- **Church-Turing Thesis: «**A computation on the natural numbers is computable if and only if it can be effectively performed by a Turing Machine.»
- General recursive functions and Turing machines (iterative solution) serve as interchangeable frameworks for defining computability.

```
int power(int x, int y)
{
    if(y == 0)
        return 1;
    else
        return x*power(x,y-1);
}
```

```
int power(int x, int y)
{
    int result = 1;

    for(int i=0; i<y; i++)
        result = result*x;

    return result;
}
```

Types of Recursion

- **Linear Recursion:** The method makes a single recursive call to itself, but this call is not the last step in the recursion sequence.

$$f(n) = \begin{cases} 1, & n = 1 \; or \; n = 2 \\ \left\lfloor \phi \, f(n-1) + \frac{1}{2} \right\rfloor, & otherwise \end{cases}$$

- **Tail Recursion:** Also makes a single recursive call. Different from the linear recursion, the result from the recursive call is not altered.

$$f(n, a, b) = \begin{cases} b, & n = 1 \\ f(n-1, a+b, a), & n > 1 \end{cases}$$

Types of Recursion

- **Multiple Recursion:** General pattern of divide&conquer algorithms. The method calls itself multiple times.

$$S(n) = \begin{cases} 1, & n = 1 \\ 3, & n = 2 \\ 3S\left(\frac{n}{2}\right) + S\left(\frac{n}{2} - 1\right), & n > 2 \text{ and even} \\ 3S\left(\frac{n-1}{2}\right) + S\left(\frac{n+1}{2}\right), & n > 2 \text{ and odd} \end{cases}$$

- **Mutual Recursion:** Multiple functions calling each other in a circular scheme.

$$A(n) = \begin{cases} 0, & n = 1 \\ A(n-1) + B(n-1), & n > 1 \end{cases} \quad B(n) = \begin{cases} 1, & n = 1 \\ A(n-1), & n > 1 \end{cases}$$

- **Nested Recursion:** The parameter of the recursive function is a result of another recursive call.

$$f(n, s) = \begin{cases} 1 + s, & n = 1 \text{ or } n = 2 \\ f\big(n - 1, s + f(n - 2, 0)\big), & n > 2 \end{cases}$$

Ex: Recursive Quine

- A quine program, or simply a quine, is a program designed to output its own source code when executed.

- The traditional method for creating a self-replicating program involves two steps:

1. Define a string variable that includes a placeholder for self-referencing.
2. Print this string, inserting the string itself into the placeholder.

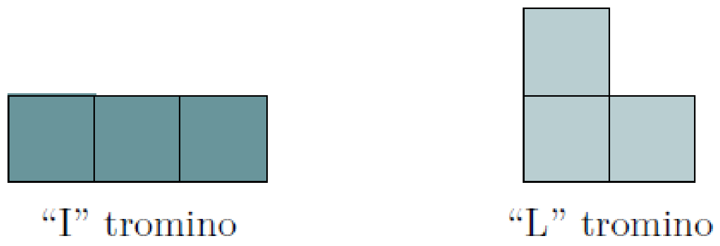
```
#include <stdio.h>
int main(){
char*s="#include <stdio.h>%cint
main(){%cchar*s=%c%s%c;%cprintf(s,10,10,34,s,34,10);return 0;}";
printf(s,10,10,34,s,34,10);return 0;}
```

```
#include <stdio.h>
int main(){
char*s="#include <stdio.h>%cint main(){%cchar*s=%c%s%c;%cprintf(s,10,10,34,s,34,10);return 0;}";
printf(s,10,10,34,s,34,10);return 0;}
```

A Recursive Quine: Creates a new code file, which creates another code file…

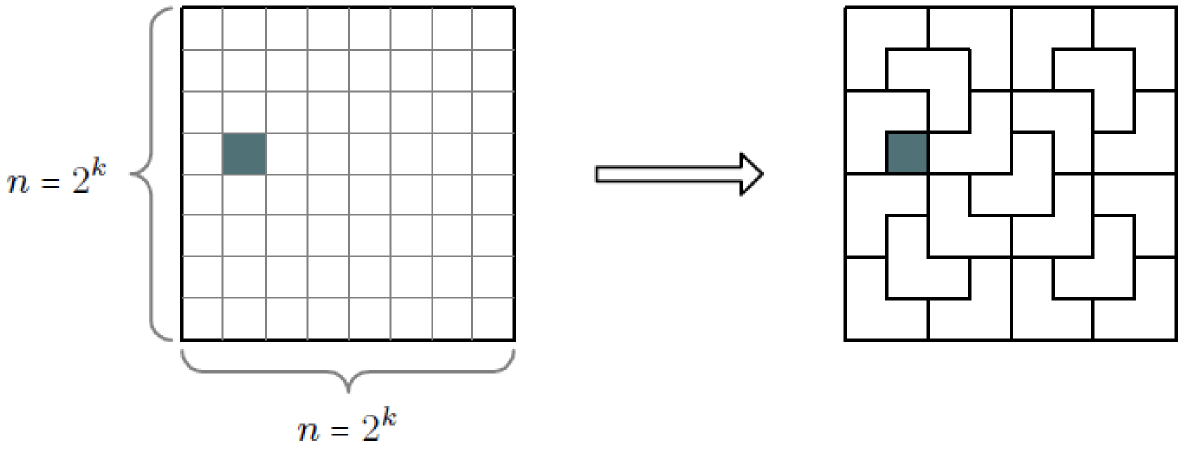
https://github.com/iamthememory/superquine

Ex: Tromino Tiling Problem



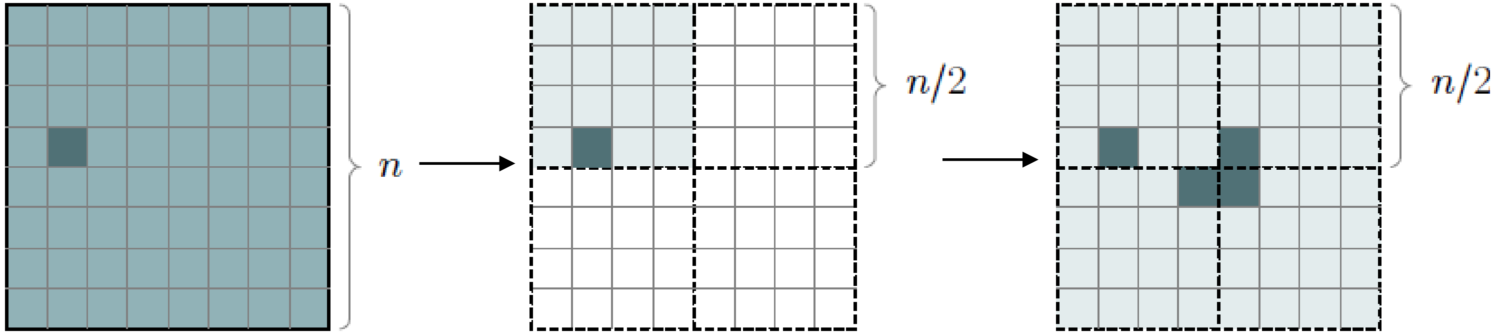
- A tromino is a shape created by joining three squares of equal size along their edges. Ignoring rotations and reflections, there are only two distinct types of trominoes: the "I" shaped tromino and the "L" shaped tromino.

- The problem involves covering an n×n square board, where n≥2 and is a power of two, with "L" shaped trominoes, except for a single "hole" square that cannot be covered. The goal is to place the trominoes so that they cover all other squares on the board without overlapping or covering the hole.



Ex: Tromino Tiling Problem

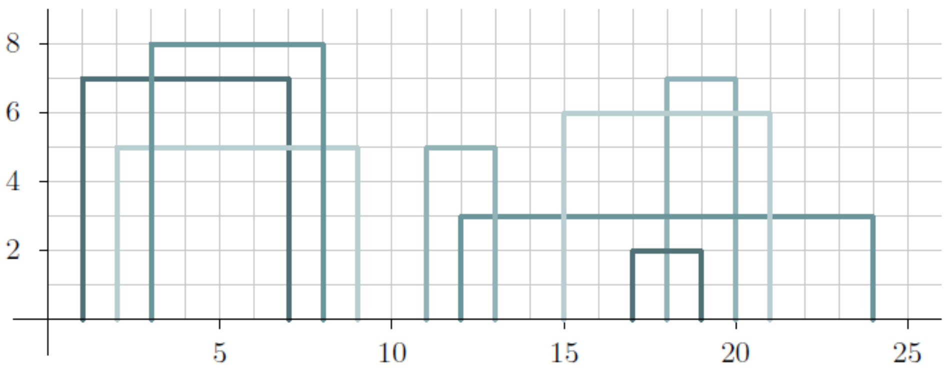
- Decomposition of the tromino tiling problem

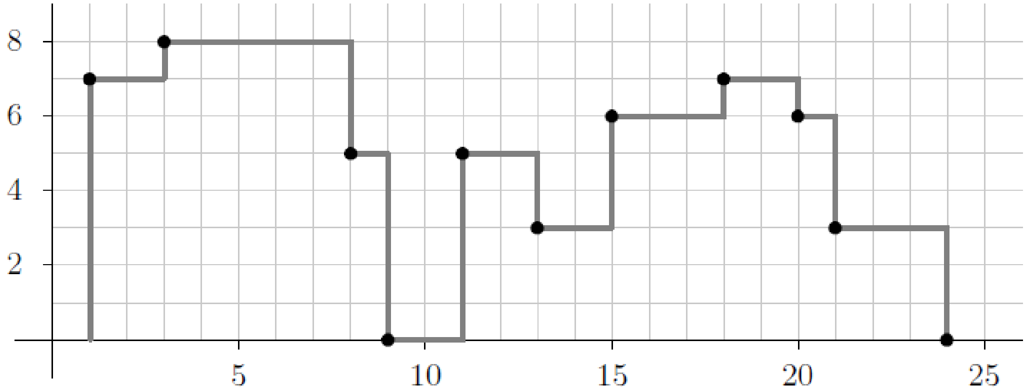


- The problem size is defined by n.
- The approach uses a divide-and-conquer strategy for larger boards.
- The board is divided into four smaller $\frac{n}{2}x\frac{n}{2}$ boards.
- However, only one of these smaller boards contains the initial hole.
- To address this, a single is used to effectively create "holes" in each of the other three smaller boards.

Ex: Skyline Problem

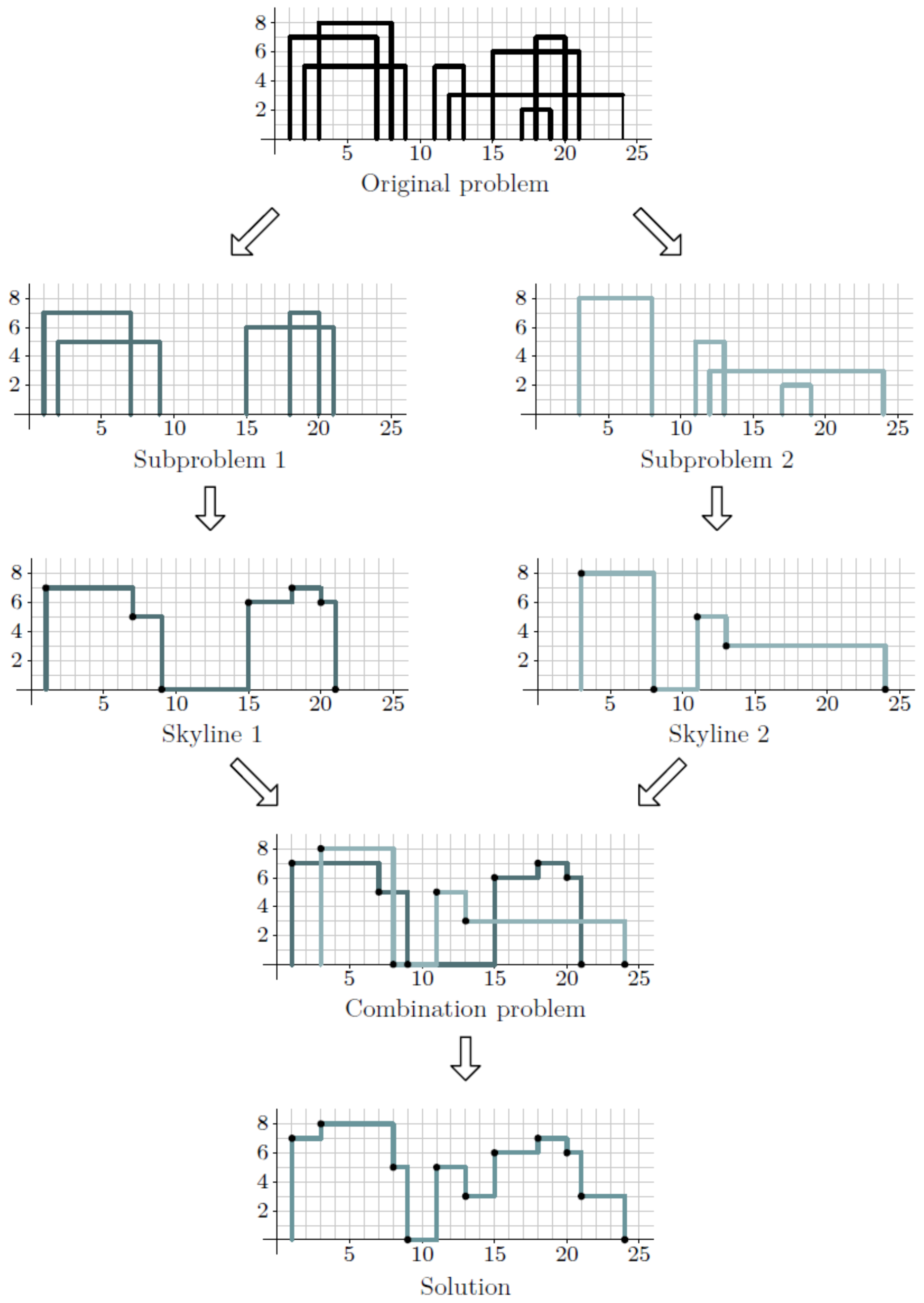
- The skyline problem involves determining the silhouette created by a group of rectangular buildings.
- Each building is represented as (x_1, x_2, h): left, right, height





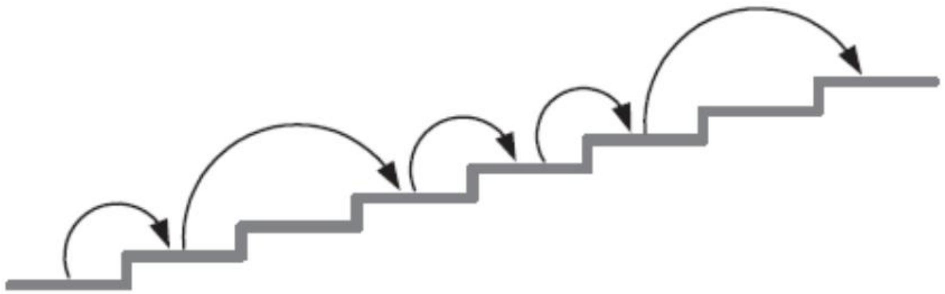
Ex: Skyline Problem

```python
def compute_skyline(buildings):
    n = len(buildings)
    if n == 1:
        return ([(buildings[0][0], buildings[0][2]),
                 (buildings[0][1], 0)])
    else:
        skyline1 = compute_skyline(buildings[0:n // 2])
        skyline2 = compute_skyline(buildings[n // 2:n])
        return merge_skylines(skyline1, skyline2, 0, 0)
```



Ex: Staircase Climbing

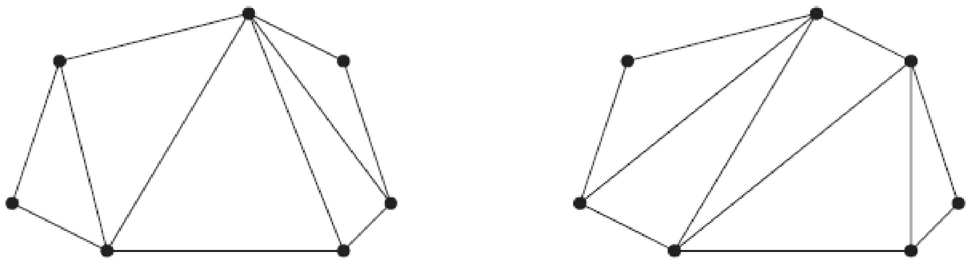
Suppose that, to climb a staircase of n steps, we have two different actions: taking an individual step or leaping two steps. What is the function f(n) which counts the number of ways to reach the top?



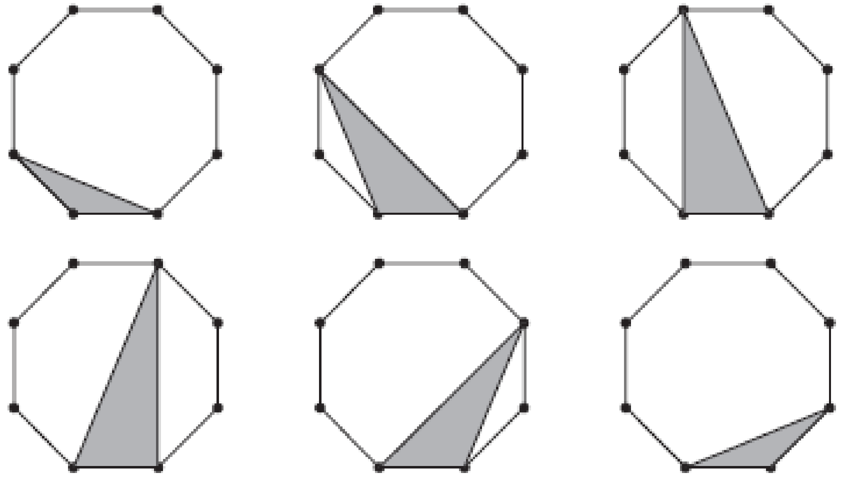
$$f(n) = \begin{cases} 1, & n = 1 \\ 2, & n = 2 \\ f(n-1) + f(n-2), & n = 3 \end{cases}$$

Ex: Convex Polygon Triangulations

A triangulation of a polygon is a division of the polygon into triangles that fully cover its area without overlapping, using only the polygon's vertices as the triangle corners. How can we define a function to calculate the number of possible ways to triangulate a convex polygon with n vertices?



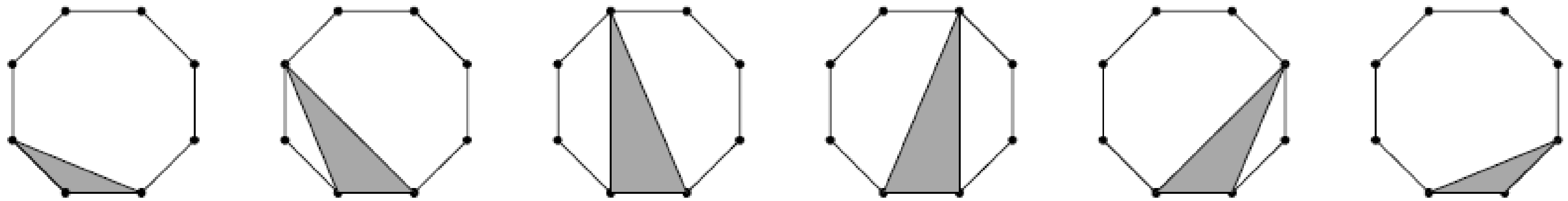
Two possible triangulations of the same polygon



Six possible triangles for the same edge of an octagon

Ex: Convex Polygon Triangulations

Total number of triangulations of an octagon:



$$f(8) = f(2)f(7) + f(3)f(6) + f(4)f(5) + f(5)f(4) + f(6)f(3) + f(7)f(2)$$

$$f(n) = \begin{cases} 1 & \text{if n=2,} \\ \sum_{i=2}^{n-1} f(i) \cdot f(n+1-i) & \text{if n>2.} \end{cases}$$

The STL Structure, again

A formal definition for a 3D mesh

- A collection of triangles.

```
solid Mesh
facet
outer loop
vertex 0.0666225 -0.00713973 -0.0520612
vertex 0.0695272 -0.00912108 -0.0509354
vertex 0.0659653 -0.00814601 -0.052367
endloop
endfacet
facet
outer loop
vertex 0.0762163 -0.00201969 -0.0587023
vertex 0.0769302 -0.00441556 -0.0564184
vertex 0.0760299 -0.00791856 -0.0610091
endloop
endfacet
```

```c
typedef struct point{
    float x, y, z;
} Point;

typedef struct Triangle{
    Point *point1, *point2, *point3;
} Triangle;

typedef struct mesh{
    DoublyList triangle_array;
} Mesh;
```

The STL Structure, again

We could generate Koch's snowflake and Sierpinski's triangle in 3D by setting Z=0.