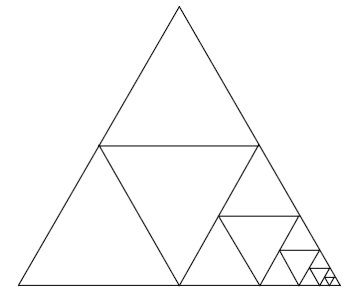


Recursion

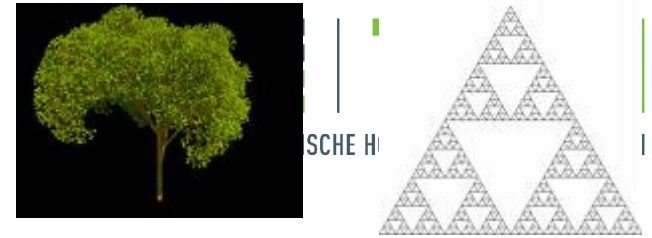
Educational Objective

- ability to apply recursion
- ability to assess the complexity of recursive algorithms



[aus Schiemenz]

Recursion



An algorithm is **recursive**, if it invokes itself or contains parts that invoke themselves directly or indirectly.

A non-recursive algorithmus is called **iterative**.

Primitive recursions can be replaced with iterations (and vice versa). Basically this are functions, which call themself only one time.

If the call takes place in the end, we speak of **tail recursion**, else of **head recursion**.

Without break condition the consequence is a **infinite regress** (results in endless loop, StackOverflow, ...).

Example calculation of factorial

$$n! = n * (n - 1)! \quad \leftarrow$$

recursive formulation

...

$$5! = 5 * 4! = 120$$

$$4! = 4 * 3! = 24$$

$$3! = 3 * 2! = 6$$

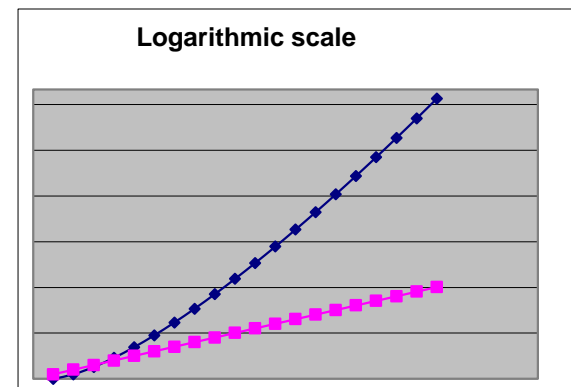
$$2! = 2 * 1! = 2$$

$$1! = 1 * 0! = 1$$

$$0! = 1 \quad \leftarrow$$

end of recursion!

power of two
vs. factorial



Factorial iterative

```
public static long fakul(int n) {  
    long fakul=1;  
    for (int i=1;i<=n;i++)  
        fakul=fakul*i;  
    return fakul;  
}
```

The multiplication is executed n-times: $O(n)$

Factorial recursive

```
public static long fakul(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * fakul(n-1));  
}
```

We see, that the complexity $T(n)$ for $n!$ is approx. $2 + T(n - 1)$.

This results in the difference equation: $T(n) = 2 + T(n - 1)$

The solution is (with starting value $T(1) = 2$): $2n + 2$

So for the recursive solution we also get: $T(n) \triangleq O(n)$

Further examples

exponentiation: $x^n = x * x^{n-1}, x^0 = 1$

harmonic series

iterative: $S_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$

recursive: $S_n = S_{n-1} + \frac{1}{n}, S_1 = 1$

Exercise

Create a JUnit testcases for the exponentiation as well as for the harmonic series!

Implement a recursive solution!

A non-primitive-recursive example

The Ackermann-Function is an example for a recursive function, that cannot be replaced by an iteration directly. For us this function is an exception!

Simplified definition:

For $m, n \in \mathbb{N}$ let be:

$$\text{ack}(0, m) = m + 1$$

$$\text{ack}(n, 0) = \text{ack}(n - 1, 1)$$

$$\text{ack}(n, m) = \text{ack}(n - 1, \text{ack}(n, m - 1))$$



Wilhelm Ackermann
1896 - 1962

What is the result of the Ackermann-Function for $n = 2, m = 1$?

Result

$$\begin{aligned}ack(2,1) &= \\ack(1, ack(2, 0)) &= \\ack(1, ack(1, 1)) &= \\ack(1, ack(0, ack(1, 0))) &= \\ack(1, ack(0, ack(0, 1))) &= \\ack(1, ack(0, 2)) &= \\ack(1, 3) &= \\ack(0, ack(1, 2)) &= \\ack(0, ack(0, ack(1, 1))) &= \\ack(0, ack(0, ack(0, ack(1, 0)))) &= \\ack(0, ack(0, ack(0, ack(0, 1)))) &= \end{aligned}$$

$$\begin{aligned}ack(0, ack(0, ack(0, 2))) &= \\ack(0, ack(0, 3)) &= \\ack(0, 4) &= \\5\end{aligned}$$

Fibonacci



Fibonacci (medieval italian mathematician, lived approx. 1170-1250 in Pisa) observed - among other things - the reproduction behaviour of rabbits. Out of this he derived the famous Fibonacci numbers.

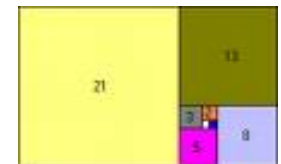
Reproduction-rules:

- rabbits get infinitely old
- rabbits mature after 2 month
- each adult rabbit-pair bears another pair every month
- start population comprises 1 newborn rabbit-pair

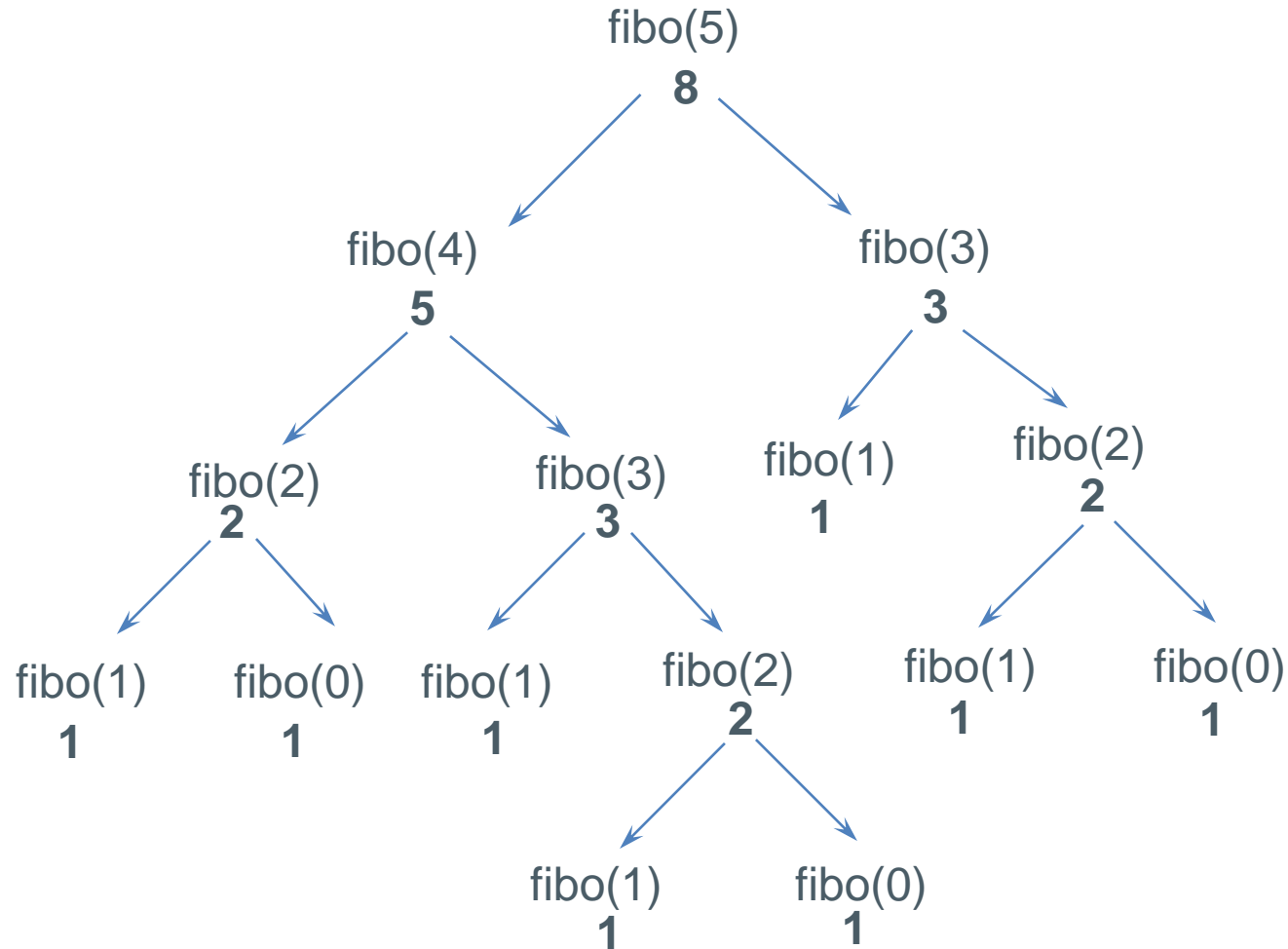
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$f(n) := f(n - 1) + f(n - 2)$$

Monate (n)	Geschlechtsreife Paare	Paare 1 Monat	Paare 0 Monat	Anzahl Paare (a_n)
0	0	0	1	1
1	0	1	0	1
2	1	0	1	2
3	1	1	1	3
4	2	1	2	5
5	3	2	3	8
6	5	3	5	13
7	8	5	8	21
8	13	8	13	34



Fibonacci numbers



Fibonacci numbers iterative

```
public static int fibo (int n) {  
    int z = 0;  
    int z_1 = 1;  
    int z_2 = 1;  
    for (int i = 1; i < n; i++) {  
        z = z_1 + z_2;  
        z_2 = z_1;  
        z_1 = z;  
    }  
    return z;  
}
```

Diagram illustrating the complexity analysis of the iterative Fibonacci algorithm:

- The initialization of `z`, `z_1`, and `z_2` is grouped with a brace and labeled **3**.
- The loop body (assignment and update statements) is grouped with a brace and labeled **4(n - 1)**.
- The loop condition and increment are grouped with a brace and labeled **1 + n + n - 1**.
- The return statement is labeled **1**.

$$3 + 2n + 4n - 4 + 1 = 6n = O(n)$$

Fibonacci numbers recursive



```
public static int fibo (int n) {  
    if (n == 0 || n == 1)  
        return 1;  
    else  
        return (fibo (n-1)+fibo (n-2)) ;  
}
```

$T(n) = T(n - 1) + T(n - 2)$ (difference equation of order 2)

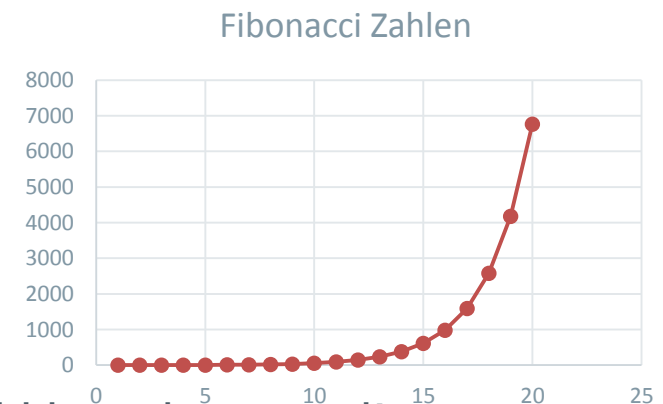
$T(0) = 2$

$T(1) = 2$

The solution leads to Binet's famous formula for closed calculation of the Fibonacci numbers.

We have: $T(n) \triangleq O(\Phi^n)$ mit $\Phi \approx 1,618$.

This means exponential growth! The worst that could have happened!



Iteration vs. Recursion

Generally recursions and iterations can be mutually replaced.
Which one should we prefer then?



Recursive solutions are more elegant and easier to read and understand.
However, a careless recursive implementation can be far more expensive!
But we have mechanisms at hand to solve this problem.



Fibonacci numbers revisited

```
static final int maxN=47;
static int knownF[]=new int[maxN];

public static int fibo(int n) {
    if (knownF[n]==0)
        if (n == 0 || n == 1)
            knownF[n]=1;
        else
            knownF[n]=fibo(n-1)+fibo(n-2);
    return knownF[n];
}
```

We call such an approach **dynamic top-down-programming** or simply **memoization**: calculated values are cached and do not have to be recalculated again. Complexity: $O(n)$.

Towers of Hanoi

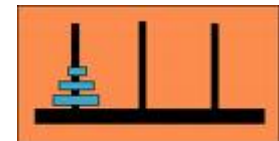
Problem:

- Given three locations A, B, C
- On location A there is a tower of n disks
- Disks are on a stack, smallest on top



Goal:

Move the tower to location C. But...



While respecting the following rules:

- You may only move disk at a time
- You may only move the topmost disk of a stack
- A disks must never be put on top of a smaller disk

Three disks...

Start



#1



#1



#3



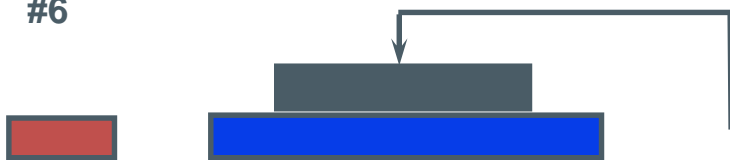
#4



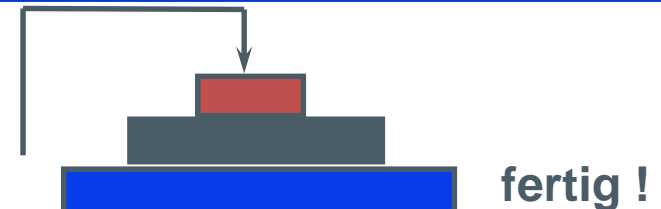
#5



#6



#7



A tower of n disks from A to C is moved by:

1. moving a tower of $(n-1)$ disks from A over B to C
2. moving a disk from A to B
3. moving the tower of $(n-1)$ disks from C over A to B

Recursive solution

```
public static void hanoi (int x, char p1, char p2, char p3){  
    if (x==1){  
        System.out.printf ("Klotz von %c nach %c \n",p1, p3);  
        return;  
    }  
    hanoi (x-1, p1, p3, p2);  
    hanoi (1, p1, p2, p3);  
    hanoi (x-1, p2, p1, p3);  
}
```

Question

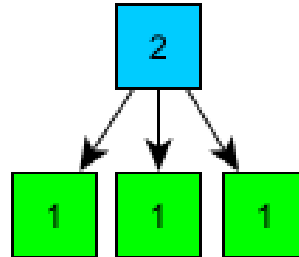
What is the complexity of the recursive solution for the towers of Hanoi?

Recursive method calls

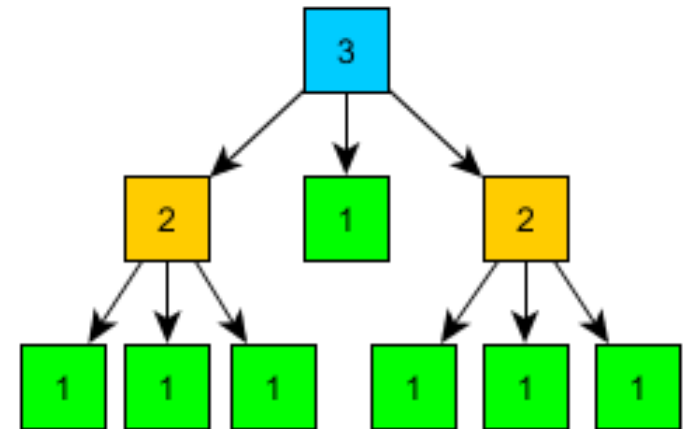
#Disks $x = 1$



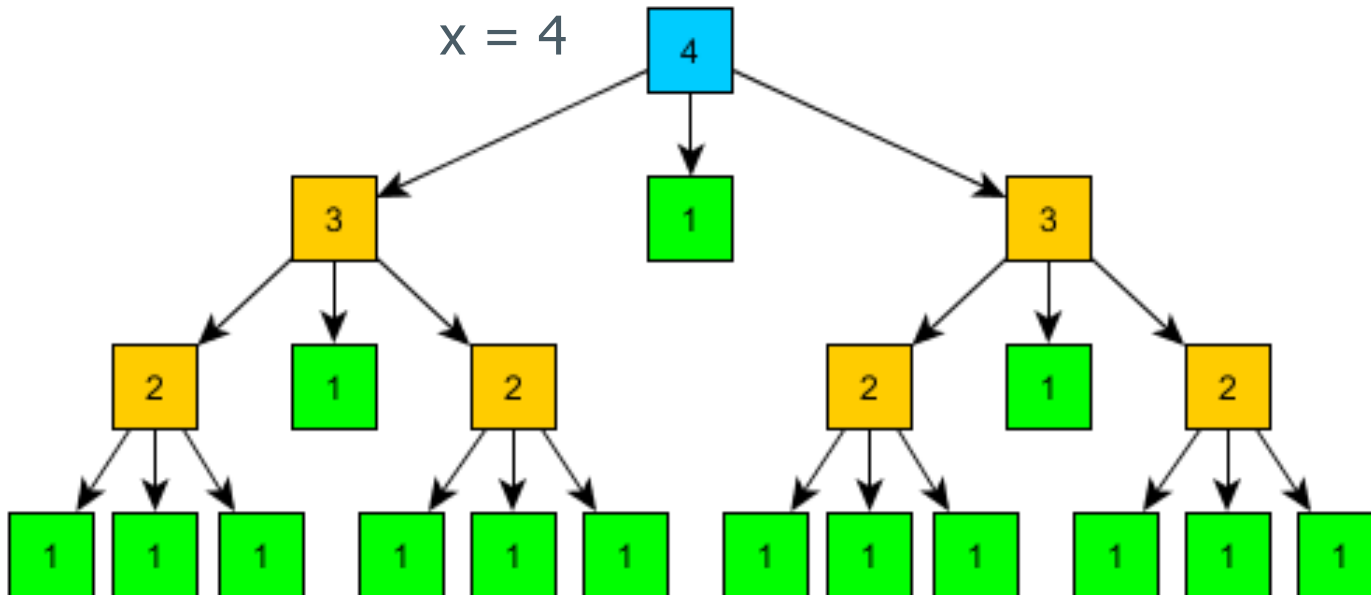
$x = 2$



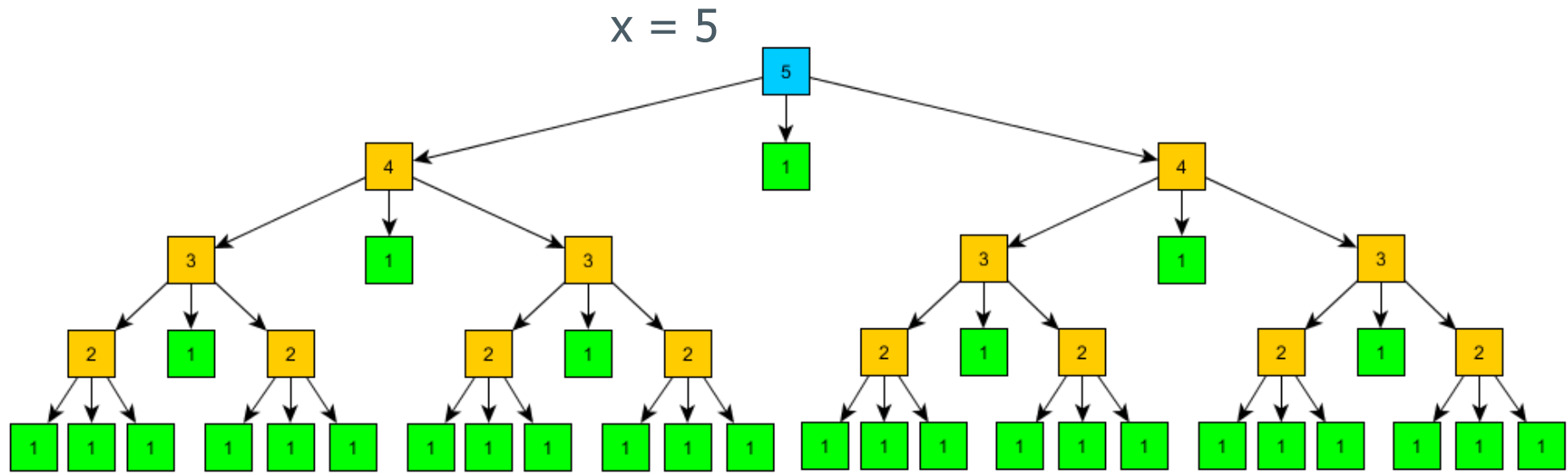
$x = 3$






$x = 4$



Recursive method calls



-  initial method call
-  move of a single disk
-  additional method call

Question

What is the complexity of the recursive solution for the towers of Hanoi?

$$T(n) = 2 * T(n - 1) + 1$$

$$T(1) = 1$$

$$T(n) = 2^n - 1$$

$$\triangleq O(2^n)$$

recursive method calls:

$$T(n) = 2 * T(n - 1) + 3$$

$$T(1) = 0$$

$$T(n) = 2^n + 2^{n-1} - 3$$

$$= 1 \frac{1}{2} (2^n - 2)$$

$$\triangleq O(2^n)$$

Divide and Conquer

You can solve big problems by splitting them into smaller ones, which are hopefully easier to solve and then construct the big solution using the small solutions.

That is the principle of **Divide-and-Conquer** (divide et impera in Latin).

The principle is a perfect fit for recursive solutions.



Maximal Subsequence Sum Recursive

We will now see a third, recursive solutions for *MaximalSubsequenceSum*
We will apply divide-and-conquer!

Idea:

The maximal subsequence sum will either be in the left half or the right half of the sequence or run right over the center.

We can further improve the complexity with that!

```
public static int maxSubSumRec(int[] a, int left, int right) {
    if (left==right)
        if (a[left]>0)
            return a[left];
        else
            return 0;

    // Divide the problem and solve the parts!
    int center=(left+right)/2;
    int maxLeftSum=maxSubSumRec(a,left, center);
    int maxRightSum=maxSubSumRec(a,center+1,right);

    // Build partial solutions!
    // Look for the biggest sum left, which goes to the
    // middle
    int maxLeftBorderSum=0, leftBorderSum=0;

    for (int i=center;i>=left;i--) {
        leftBorderSum+=a[i];
        if (leftBorderSum>maxLeftBorderSum)
            maxLeftBorderSum=leftBorderSum;
    }
```

```
// Look for the biggest sum right, which goes to the  
// middle
```

```
int maxRightBorderSum=0, rightBorderSum=0;
```

```
for (int i=center+1;i<=right;i++) {  
    rightBorderSum+=a[i];  
    if (rightBorderSum>maxRightBorderSum)  
        maxRightBorderSum=rightBorderSum;  
}
```

How could maxOf3 possibly be implemented?

```
return maxOf3(maxLeftSum,  
              maxRightSum,  
              maxRightBorderSum+maxLeftBorderSum);  
}
```

```
// entry point
```

```
public static int maxSubSum3(int a[]) {  
    return maxSubSumRec(a,0,a.length-1);  
}
```

Apply the recursive version of MaxSubSequenceSum to $-4, 5, 6, 7, 8, -8$.

$$T(1) = 1$$

$$T(n) = 2T(n/2) + O(n) \approx 2T(n/2) + n$$

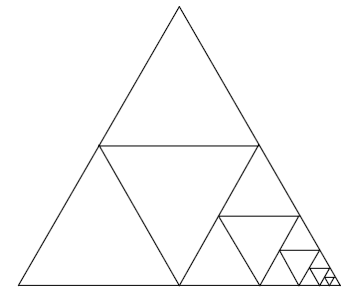
For $n = 2^k$ ($k = \log_2(n)$) :

$$\begin{aligned} T(n) &= n \cdot (k + 1) = n \cdot \log_2(n) + n \\ &\triangleq O(n \cdot \log_2(n)) \end{aligned}$$

$$O(n) < O(n \cdot \log_2(n)) < O(n^2)$$

Educational Objective

- ability to apply recursion
- ability to assess the complexity of recursive algorithms



[aus Schiemenz]