# intro to mathematics in software engineering

Fontys University of Applied Sciences

May 6, 2025

# objectives

- scary looking functions
- math related software engineering concepts
- translate math to programming

## fundamentals of mathematical and function notation

$$f(x) = x^2$$

# fundamentals of mathematical and function notation

$$a \cdot f(x)$$

$$f(x/b)$$

$$f(x - c)$$

$$f(x) + d$$

# fundamentals of mathematical and function notation

$a \cdot f(x) \Rightarrow$ multiplies the y-value by $a$

$f(x/b) \Rightarrow$ multiplies the x-value by $b$

$f(x-c) \Rightarrow$ shifts graph $c$ units to the right

$f(x)+d \Rightarrow$ shifts graph $d$ units upward

# breaking down a scary looking function

$$g(x) = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin(2\pi(2n-1)ft)}{2n-1}$$

(where $t =$ time, $f =$ frequency, $n =$ iterations)

# breaking down a scary looking function

$$g(x) = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin(2\pi(2n-1)ft)}{2n-1}$$

we are dealing with a function built from multiple smaller functions added together

# breaking down a scary looking function

$$g(x) = \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{\boxed{\sin}\,(2\pi(2n-1)ft)}{2n-1}$$

the $sin$ on the inside suggests we are dealing with waves or oscillations

# breaking down a scary looking function

$$g(x) = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin(2\pi(2n-1)ft)}{2n-1}$$

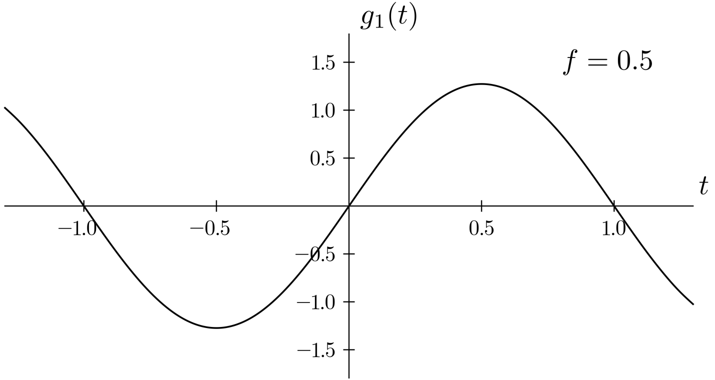
denominator $2n-1$ hints that terms get smaller as $n$ increases - later terms have less influence

# breaking down a scary looking function

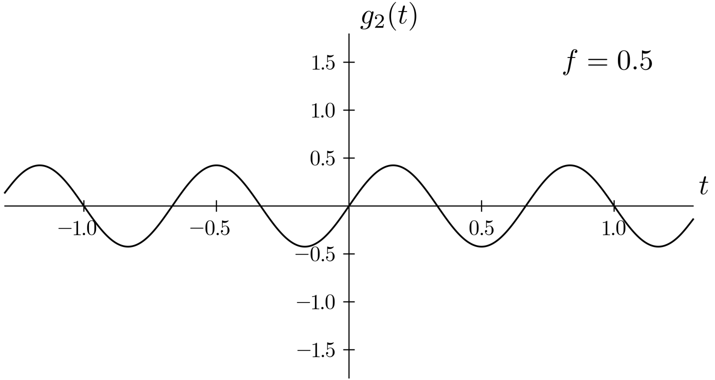
$$
\begin{aligned}
g_1(t) &= \frac{4}{\pi} \cdot \frac{\sin(2\pi(2(1)-1)ft)}{2(1)-1} \\
&= \frac{4}{\pi} \cdot \frac{\sin(2\pi(1)ft)}{1} \\
&= \frac{4}{\pi} \cdot \sin(2\pi ft)
\end{aligned}
$$

$$
g_1(t) = \boxed{\frac{4}{\pi}\sin(2\pi ft)}
$$

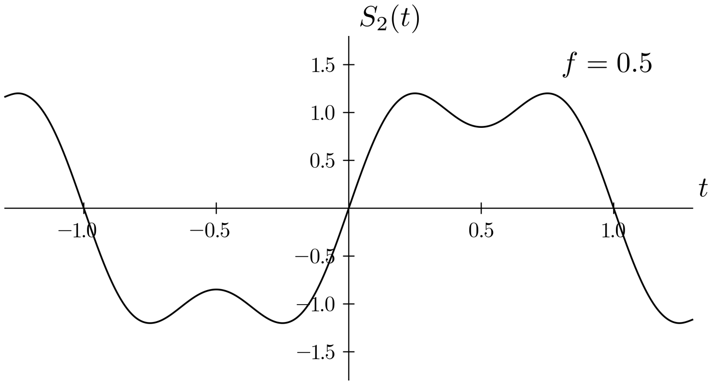
# breaking down a scary looking function



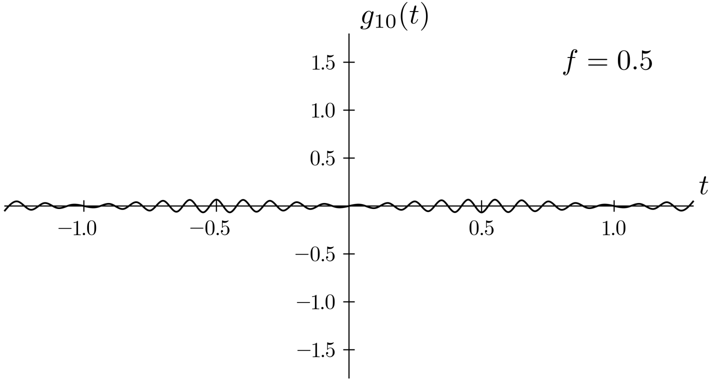
# breaking down a scary looking function



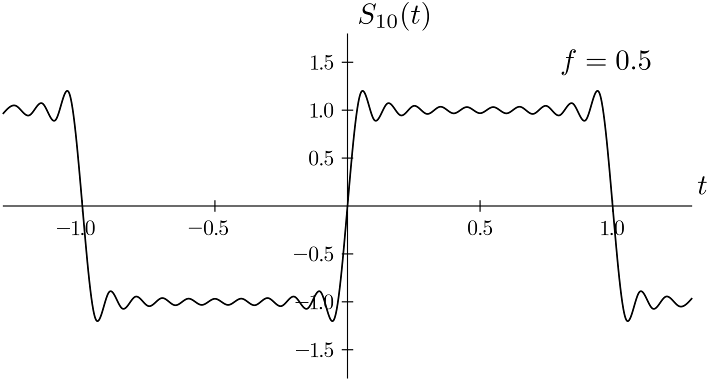
# breaking down a scary looking function



# breaking down a scary looking function



# breaking down a scary looking function



# vectors (not the math kind), i.e. arrays

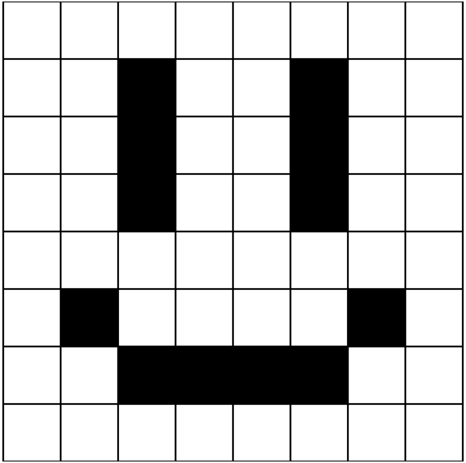
set of pairs $\Rightarrow$ index and value

elements (pairs) are conventionally of same memory size

# vectors (not the math kind), i.e. arrays

```
one_d_array = [0, 0, 1, 0, 1, 0, 1, 1]
```

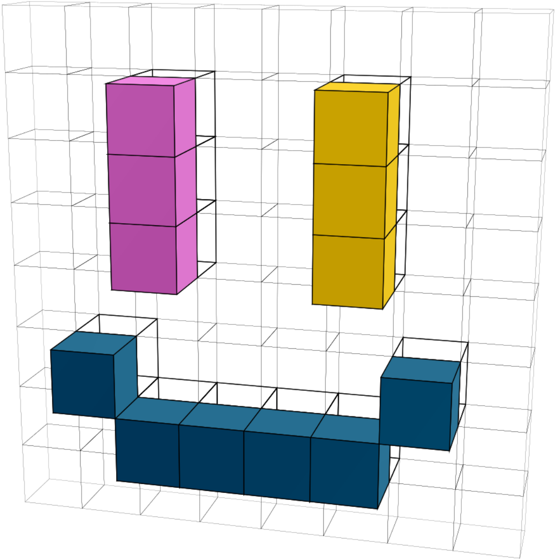
# vectors (not the math kind), i.e. arrays



# vectors (not the math kind), i.e. arrays

```
two_d_array = [
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 1, 0, 0],
[0, 0, 1, 0, 0, 1, 0, 0],
[0, 0, 1, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 0, 1, 0],
[0, 0, 1, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0]
]
```

# vectors (not the math kind), i.e. arrays



# binary tree

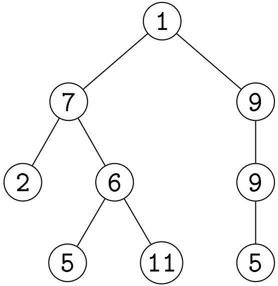
data structure expressed as a figurative tree

one root node

nodes can only have one parent node and at most two children nodes (left and right)

foundation for more complex data structures

# binary tree




an unbalanced and unsorted binary tree

height = 3 i.e., the number of edges from farthest node to root node (1)

size = 9, i.e., total number of nodes

# binary tree traversal using dfs

depth first search

starts at root, goes down as far as it can, backtracks, explores, etc.

to implement we need to create a) the structure and b) the algorithm

# binary tree traversal using dfs - the structure

```python
class Node:
  def __init__(self, val):
    self.data = val
    self.left = None
    self.right = None

def connect(parent, left=None, right=None):
  parent.left = left
  parent.right = right

```

# binary tree traversal using dfs - the structure

```
firstNode = Node(1)
secondNode = Node(7)
thirdNode = Node(9)

connect(firstNode, secondNode, thirdNode)
```

# binary tree traversal using dfs - the algorithm

```
def depth_first_search(root, value):
  if root is None:
    return False
  root.visited = True

  if root.data == value:
    return True

  l_res = depth_first_search(root.left, value)
  r_res = depth_first_search(root.right, value)

  return l_res or r_res
```

# matrices

math implementation of arrays, in a way

2d/3d transformations in graphics (scaling, rotation, translation)

$P = (1, 2)$ can also be expressed as $P = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

$$M = \begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix}$$

$$P_m = M \cdot P = \begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \cdot 1 + 4 \cdot 1 \\ 2 \cdot 2 + 1 \cdot 2 \end{bmatrix}$$

$$P_m = (7, 6)$$

# big o

o stands for ordnung, order of approximation

how different algorithms scale and grow in terms of complexity

time / input $\Rightarrow$ output

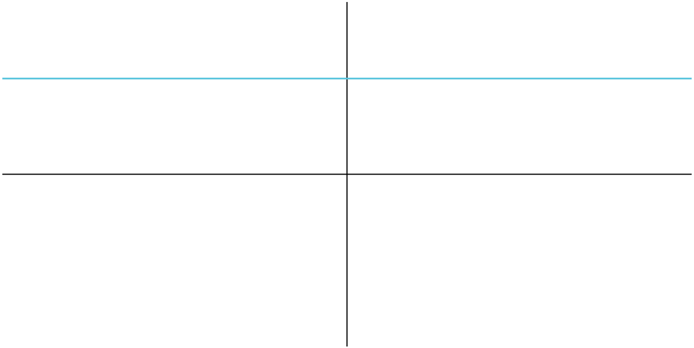
ignores terms but the one that "grows" the quickest

# big o

## constant $O(1)$

same amount of time, no matter the size of hte input

looking up value at index

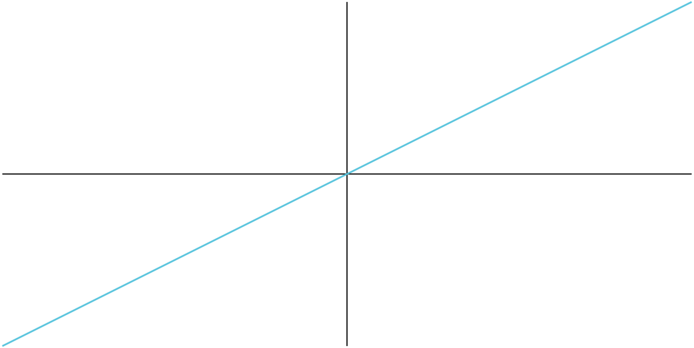


# big o

## linear $O(n)$

grows linearly with the size of the input

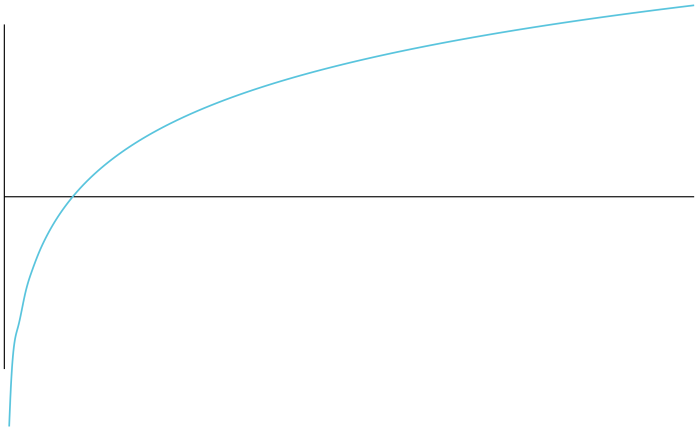
traversing a binary tree ;)



# big o

logarithmic $O(\log n)$

- proportional to the logarithm of the input size
- binary search algorithm



# big o

quadratic $O(n^2)$

proportional to the square of the input

for example bubble sort