Data Driven World
Week 2

Simon Perrault– Singapore University of Technology and Design

Week 2 – Binary Heap

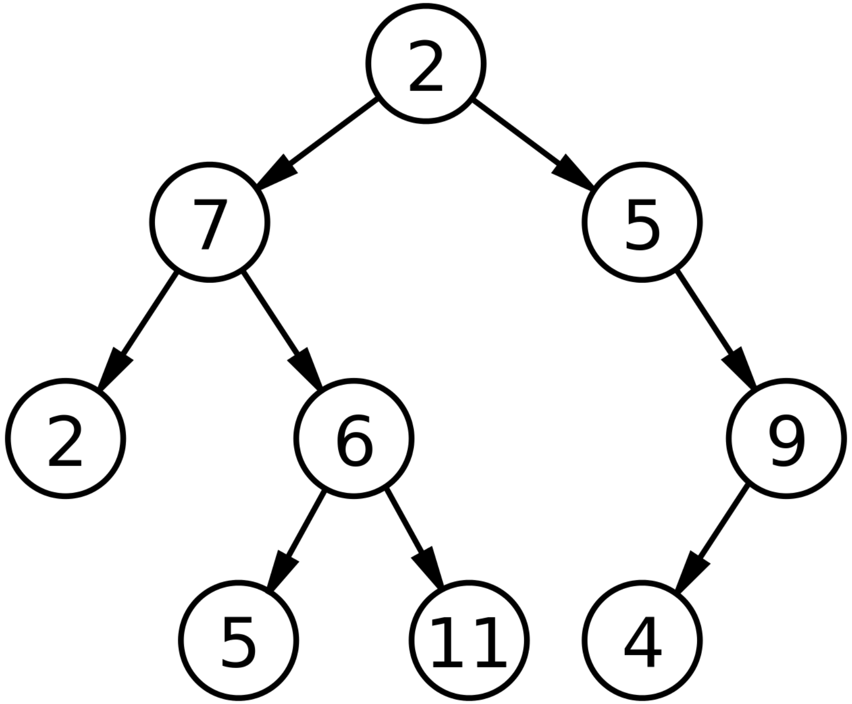
*"I accidentally swallowed a whole heap of Scrabble tiles last night. My next p** could spell disaster"* – 5th best heap joke

Binary Heap

- “A binary heap is a heap data structure that takes the form of a binary tree.”
- In computer science, a **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- TL;DR: **it’s a tree**

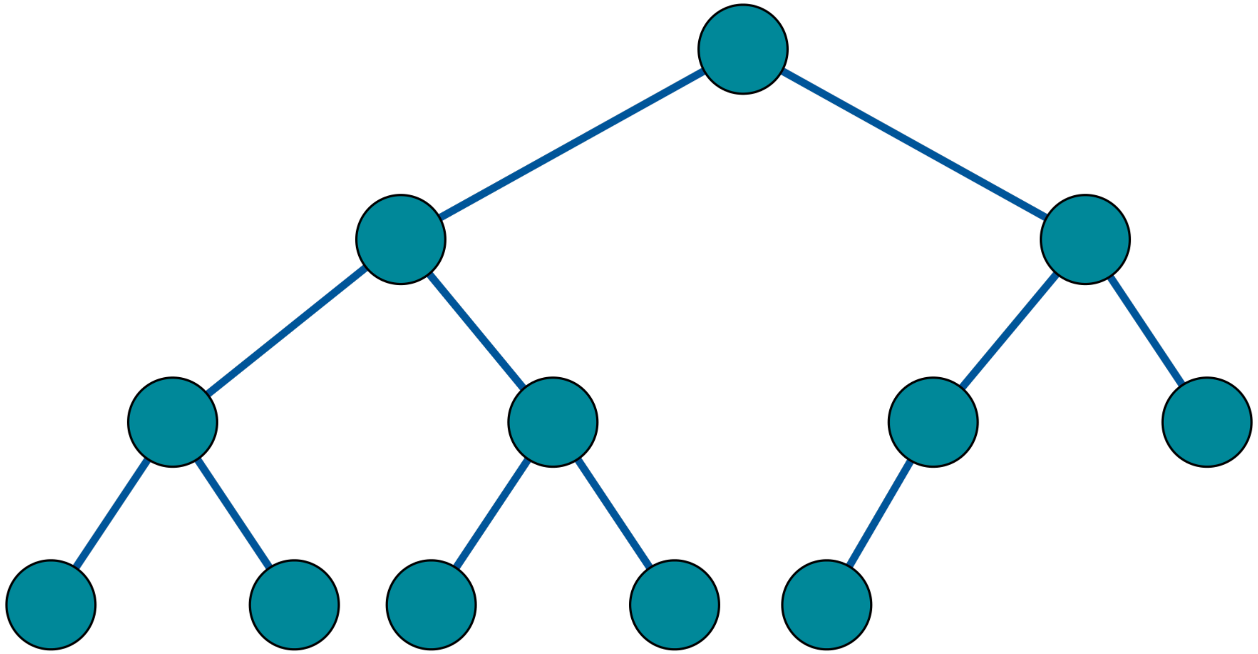
Binary Tree

- 1 root
- Each node has at most 2 children



Complete Binary Tree

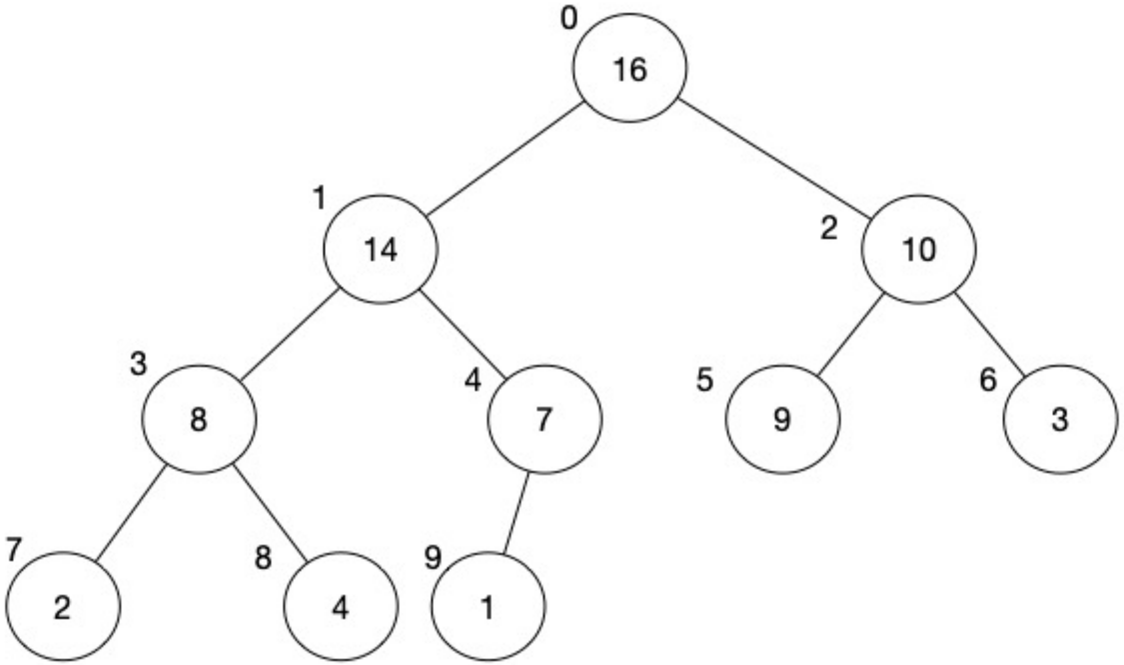
- In a **complete** binary tree every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left as possible.



Binary Heap Representation

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

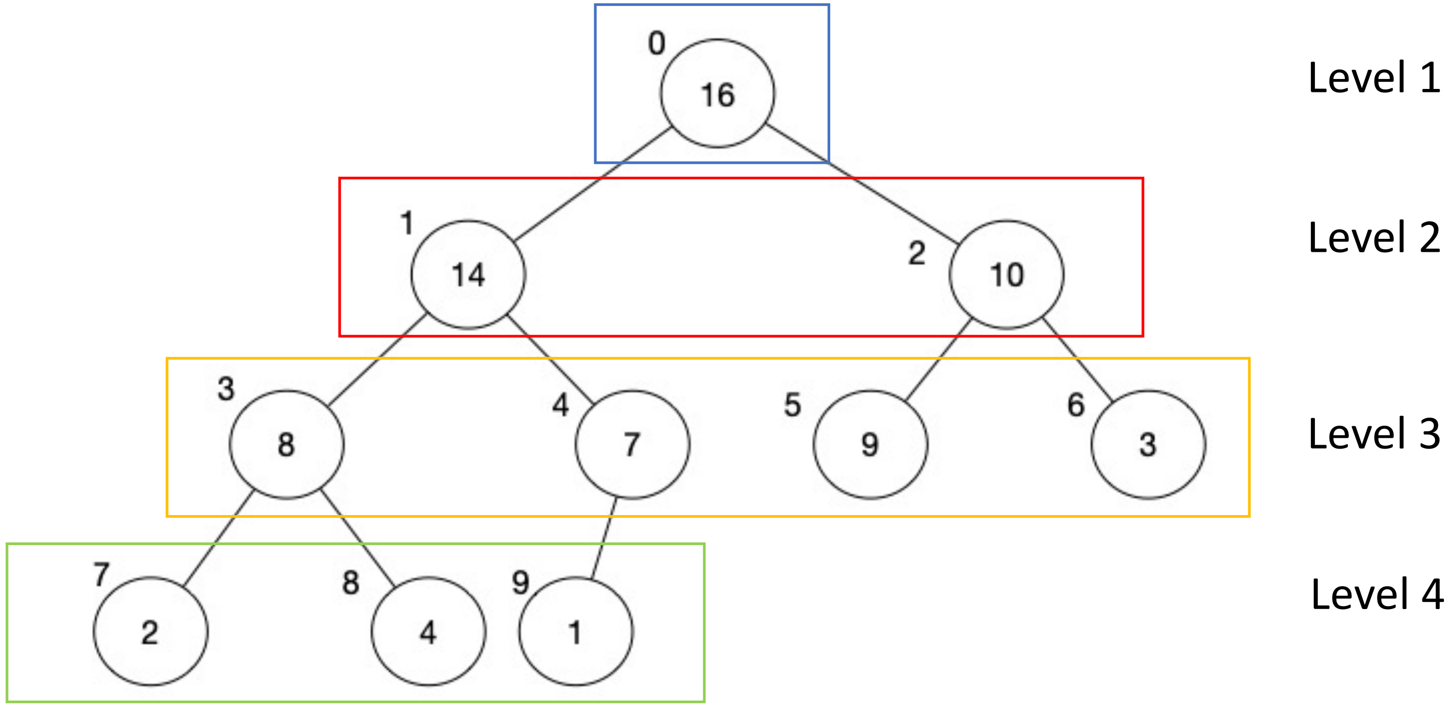
We have indicated the indices of each element in the array, which starts from 0. We can visualize the elements in this array in a form of a *tree* as shown below.



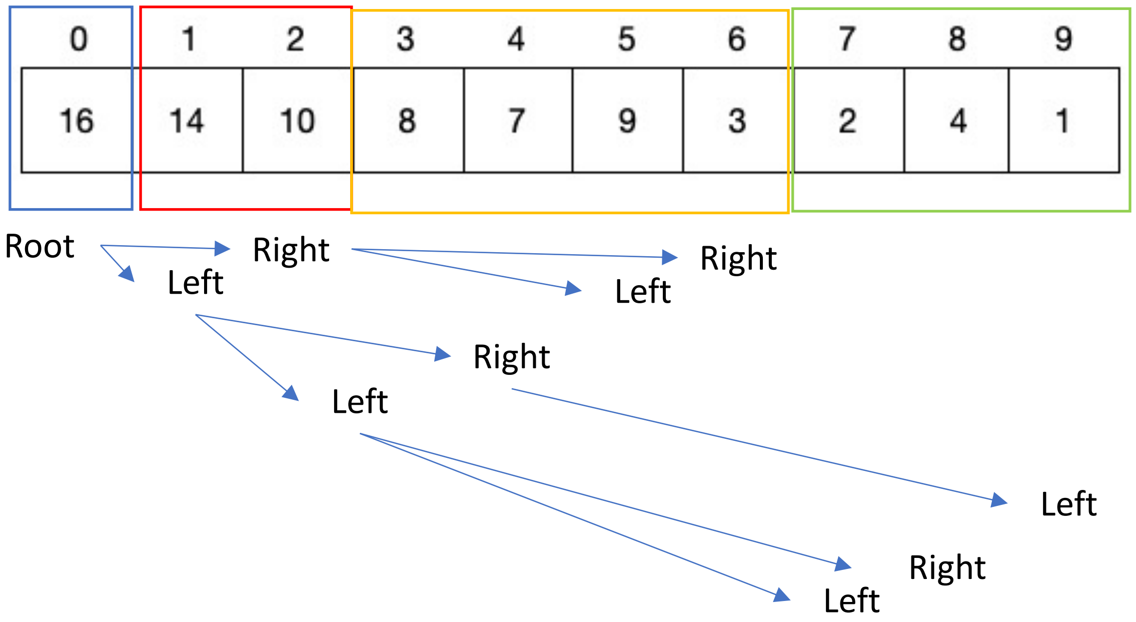
Binary Heap Representation

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

We have indicated the indices of each element in the array, which starts from 0. We can visualize the elements in this array in a form of a *tree* as shown below.



Finding Parents/Children Nodes



Finding Parent

```
def parent(index):
Input: index of current node
Output: index of the parent node
Steps:
1. return integer((index-1) / 2)
```

Finding Left/Right Children

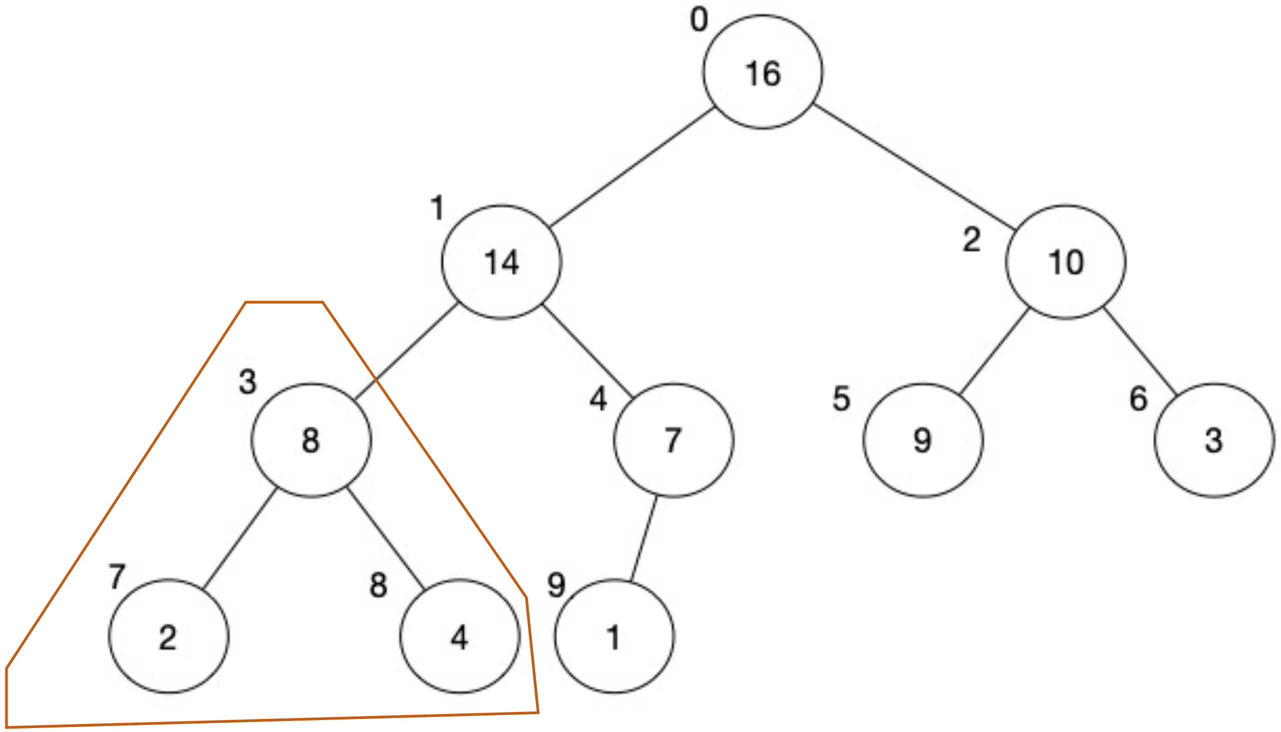
```
def left(index):
Input: index of current node
Output: index of the left child node
Steps: 1. return (index * 2) + 1

def right(index):
Input: index of current node
Output: index of the right child node
Steps: 1. return (index + 1) * 2
```

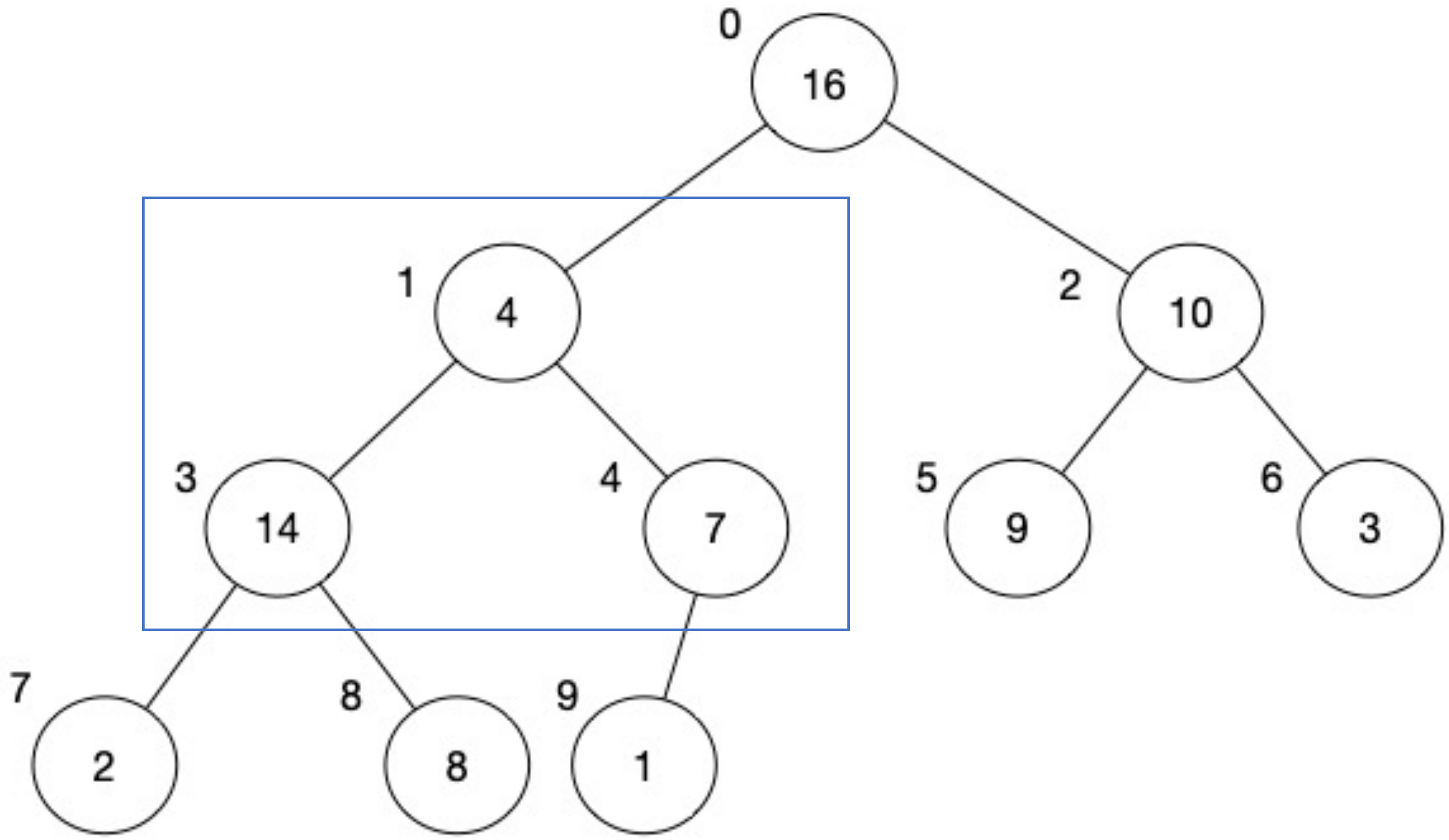
Note: what if there are no children?

Heap Property

- Given a node A and its parent P: `A < P`
- Largest value is the root
- Advantages:
  - Easy to find the max value of a binary heap
  - Insertion of a new element may become messy
- True for every tree and subtree (see red subtree)

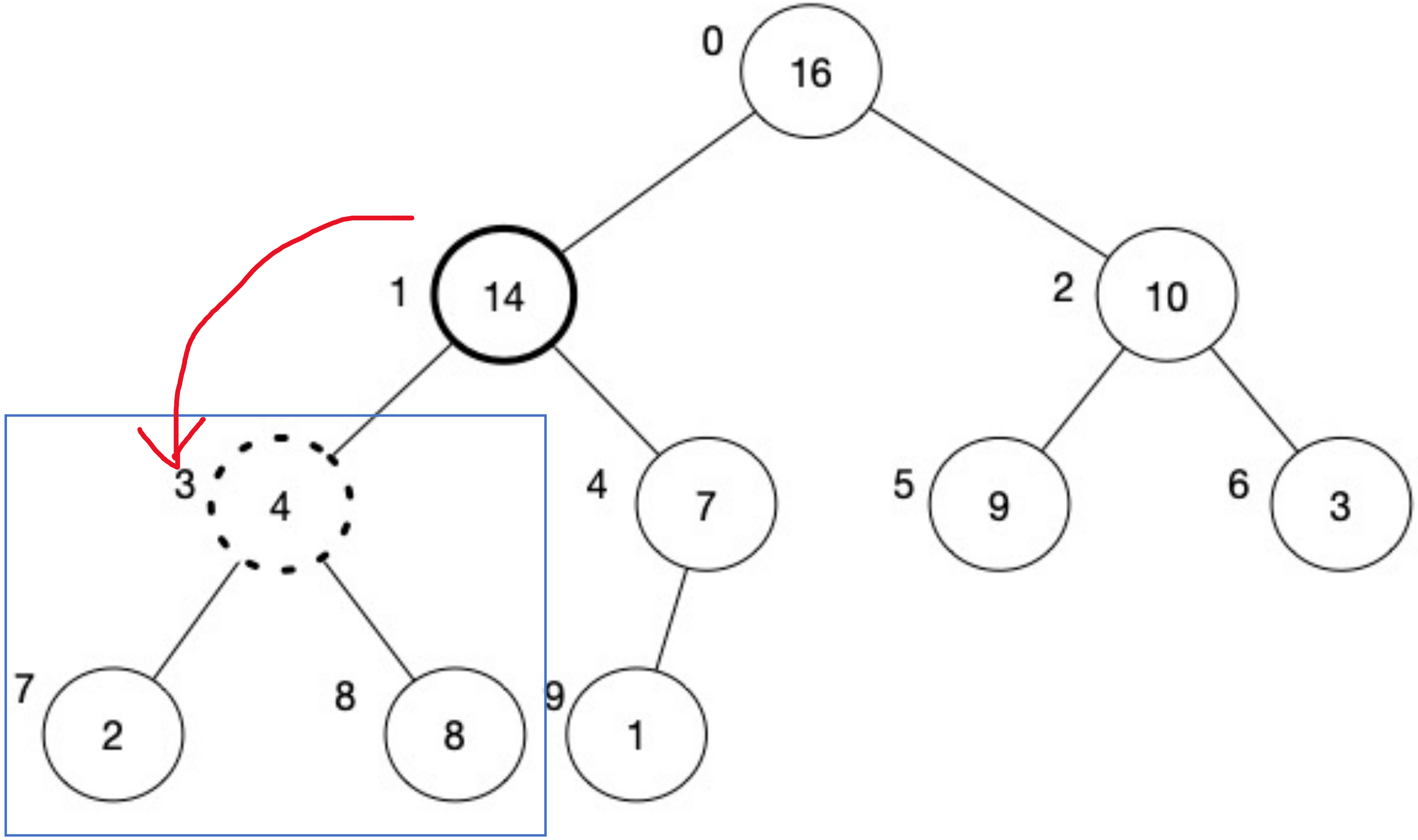


Maintaining the Heap Property (1/3)



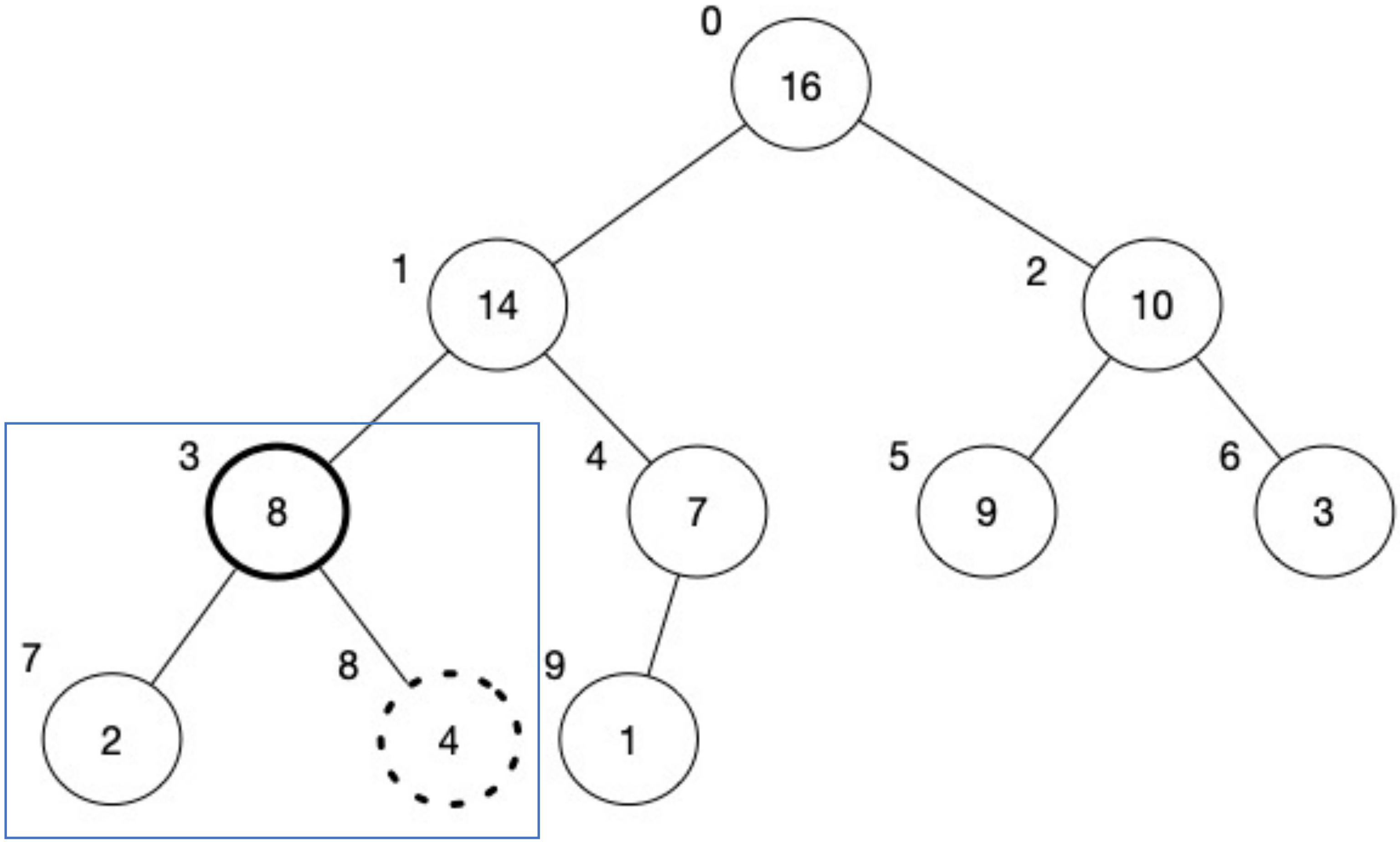
- Problem: Value at index 1 does not satisfy the property
- Let's swap!
- But first: find which child (left or right) has the highest value (so that the heap property will be verified after swapping)

Maintaining the Heap Property (2/3)



- Change index from 1 to 3
- Let's swap!
- But first: find which child (left or right) has the highest value (so that the heap property will be verified after swapping)

Maintaining the Heap Property (3/3)



- Change index from 3 to 8
- No children
- Subtree = the node at index 8
- Property satisfied!

Max-Heapify algorithm

Note: we COULD start implementing

```
def max-heapify(A, i):
version: 1
Input:
    - A = array storing the heap
    - i = index of the current node to restore max-heap property
Output: None, restore the element in place
Steps:
1. current_i = i # current index starting from input I
2. As long as ( left(current_i) < length of array), do:
2.1 max_child_i = get the index of largest child of the node
current_i
2.2 if array[max_child_i] > array[current_i], do:
2.2.1 swap( array[max_child_i], array[current_i])
2.3 current_i = max_child_i # move to the index of the largest
child
```

Helper Functions

```
def max_child(array, index, heap_size):
Assumption: there is a least one leaf (left one)
Input:
    • an array representing the heap
    • index of the current node
    • size of the heap
Output:
    • index of the child with the largest value
Steps:
1. If there is no right node:
        1.1 return index of the left node
2. Otherwise:
        2.1 if the value in the left node is higher
                2.1.1 return index of the left node
        2.2 otherwise:
                2.2.1 return index of the right node
```

Max-Heapify algorithm

```
def max-heapify(A, I, size):
version: 1
Input:
    - A = array storing the heap
    - i = index of the current node to restore max-heap property
    - size = length of the array
Output: None, restore the element in place
Steps:
1. current_i = i # current index starting from input I
2. As long as ( left(current_i) < length of array), do:
        2.1 max_child_i = get the index of largest child of the node
current_i # this part just got easier!
        2.2 if array[max_child_i] > array[current_i], do:
                2.2.1 swap( array[max_child_i], array[current_i])
        2.3 current_i = max_child_i # move to the index of the largest child
```

parent

Building a Heap from an Array

"Started from the bottom, now we're here
*Started from the bottom, now my whole team f*****' here"*

– Some canadian guy

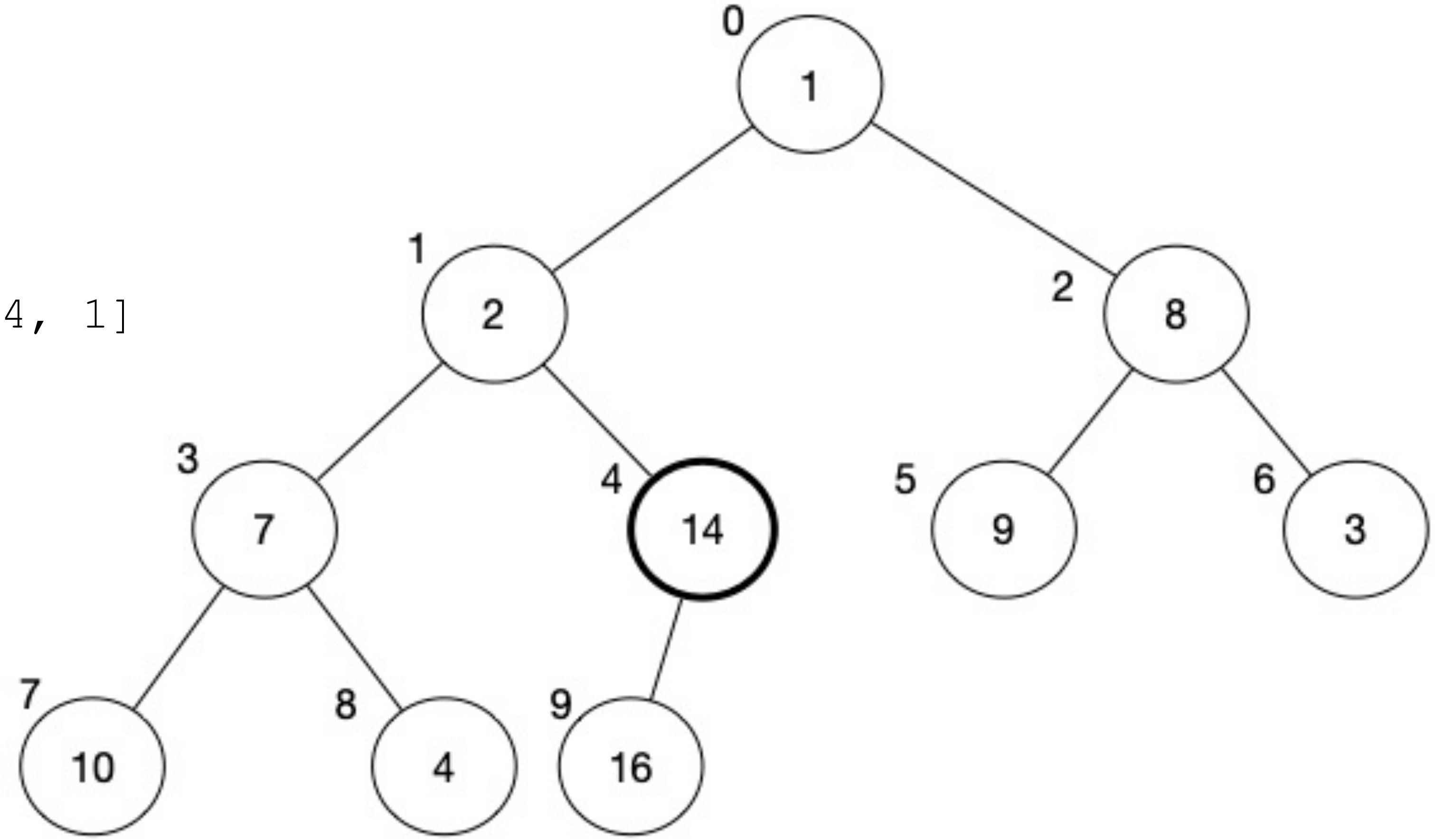
Check the Week 2 notes for more details

Actually, we start from the middle

- every element after the middle is a leaf, no need to reorder
- Go from the middle and max-heapify from there
- Once done, move to the element before
- Go until we reach the root (index 0)

Simple Version (Manually)

```
>>> heap = [1, 2, 8, 7, 14, 9, 3, 10, 4, 16]
>>> max-heapify(heap, 4)
>>> max-heapify(heap, 3)
>>> max-heapify(heap, 2)
>>> max-heapify(heap, 1)
>>> max-heapify(heap, 0)
>>> # now our h
>>> print(heap)
[16, 14, 9, 10, 2, 8, 3, 7, 4, 1]
```



Build-Max-Heap Algorithm

```
def build-max-heap(array):
Input:
     - array: arbitrary array of integers
Output:
None, sort the element in place
Steps:
1. n = length of array
2. starting_index = integer(n / 2) - 1 # start from the
middle or non-leaf node
3. For current_index in Range(from starting_index down
to 0), do:
     3.1 call max-heapify(array, current_index)
```

Heapsort

General Idea:

- Assuming a correctly formed heap:
  - Take the root (largest element)
  - Swap it with the last* element
  - Consider the last element to be excluded from the heap
  - **Max-heapify** the new array (excluding the last element)
  - Repeat

Illustration from the notes (TL;DR)

`heap = [16, 14, 9, 10, 2, 8, 3, 7, 4, 1]`

Iteration 1: `heap = [1, 14, 9, 10, 2, 8, 3, 7, 4], sorted = [16]`

End of iteration 1: `heap = [14, 10, 9, 7, 2, 8, 3, 1, 4], sorted = [16]`

Iteration 2: `heap = [4, 10, 9, 7, 2, 8, 3, 1], sorted = [14, 16]`

End of iteration 2: `heap = [10, 7, 9, 4, 2, 8, 3, 1], sorted = [14, 16]`

Iteration 3: `heap = [1, 7, 9, 4, 2, 8, 3], sorted = [10, 14, 16]`

End of iteration 3: `heap = [9, 7, 8, 4, 2, 1, 3], sorted = [10, 14, 16]`

Iteration 4: `heap = [3, 7, 8, 4, 2, 1], sorted = [9, 10, 14, 16]`

End of Iteration 4: `heap = [8, 7, 3, 4, 2, 1], sorted = [9, 10, 14, 16]`

[let’s skip a few iterations]

Iteration 8: `heap = [1, 2], sorted = [3, 4, 7, 8, 9, 10, 14, 16]`

End of iteration 8: `heap = [2, 1], sorted = [3, 4, 7, 8, 9, 10, 14, 16]`

Iteration 9: `heap = [1], sorted = [2, 3, 4, 7, 8, 9, 10, 14, 16]`

`result = [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]`

Testing Performance

```
def run_function(f, x):
    start = time.time()
    f(x)
    end = time.time()
    return end-start
```

- Two arguments:
    1. **f**, which can be any function defined earlier
    2. x, the argument needed to run the function

`run_function` counts the time it takes to run a given function **f**

Testing Performance

```python
def run_function(f, x):
    start = time.time()
    f(x)
    end = time.time()
    return end-start
```

1. Reuse `generate_random_int()` function from Week 1
2. Generate randomly shuffled arrays of size 10^n with a seed of 100
3. Use the `run_function()` with `sorted` (built-in) and array

Week 2 – Complexity

“*Great jokes don’t have to be complex, e.g.* ‘C'est l’histoire d’un pingouin qui respire par le derrière. Un jour, il s'assoit et il meurt.’” – Anonymous

Complexity Notation

- Complexity refers to the amount of resources required to run code
- Usually we focus on time resources
- Last week we covered two types of sorting algorithms
- There are dozens of them
- How to benchmark them?

Big-O Notation

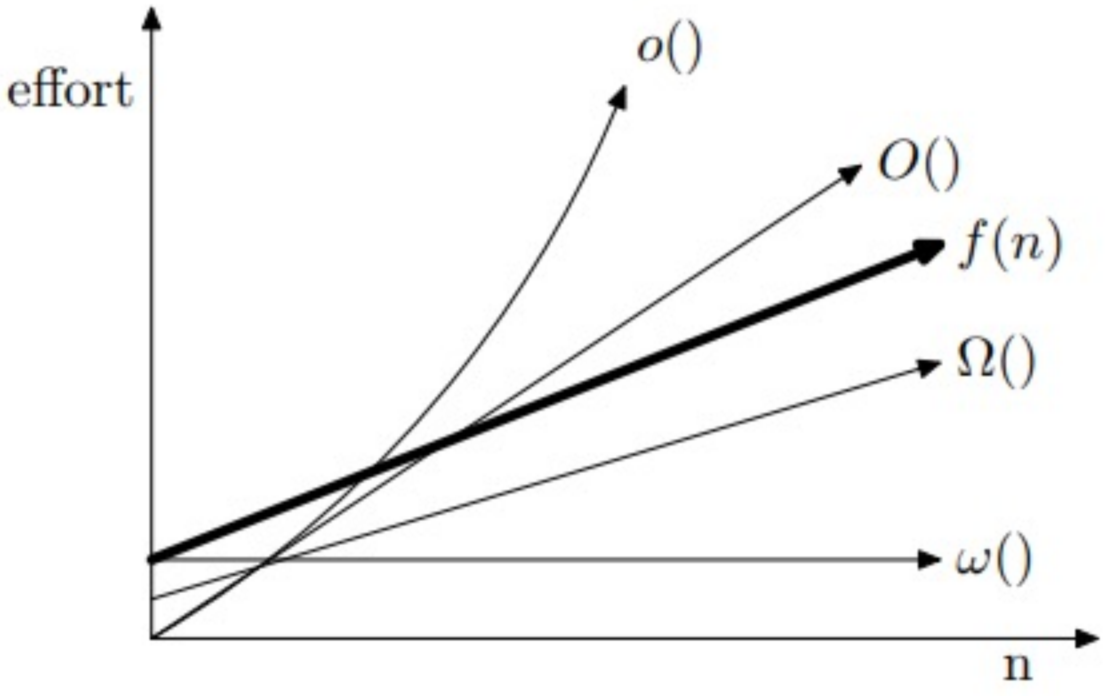
- Notation describing the limiting behavior of a function
- Describes the amount of resource (e.g. time) needed as a function of the size of the data
- From the notes for Week 2, we will discuss the following:
  - Big/Small Oh (Omicron): O
  - Big/Small Omega: Ω
  - Big Theta: Θ

Important

- Function used to describe the behavior of an algorithm are considered to be upper/lower bound above a specific data size x_0 large enough

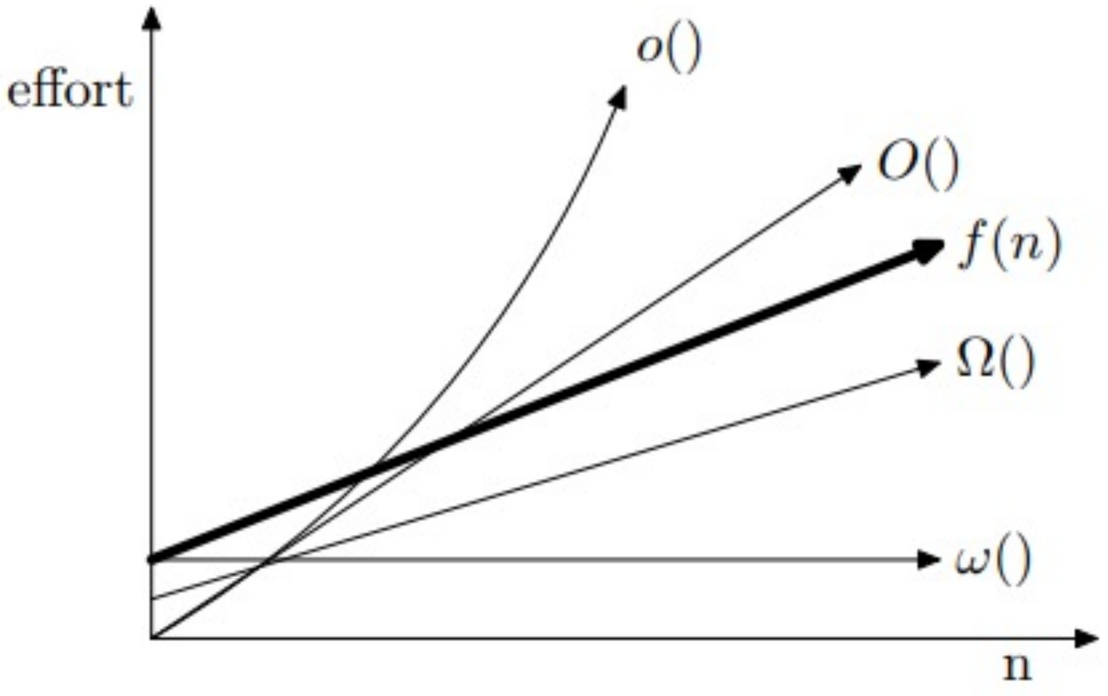
Omicron

- If f is a function describing the time complexity of your algorithm
- Omicron (big and small) refer to another function which is an upper bound to your own function
- Difference between small and big-Omicron: small-o is a “looser” bond



Omega

- If f is a function describing the time complexity of your algorithm
- Omega (big and small) refer to another function which is a lower bound to your own function
- Difference between small and big-Omega: small-Omega is a “looser” bond

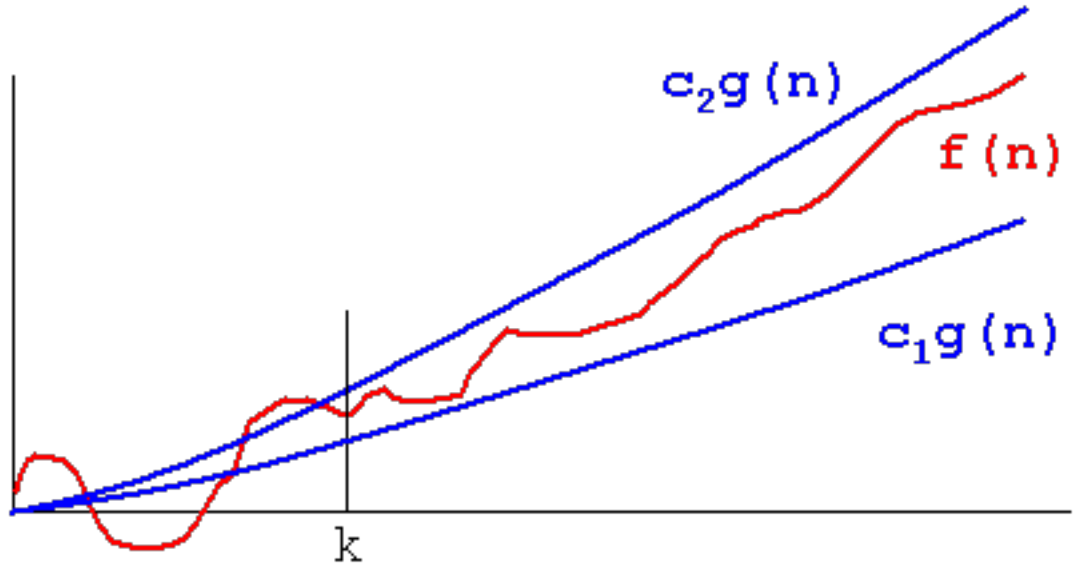


Theta

- If a function g can be used as both an upper and lower bound, then you have your theta
- In that example:

$c1\ g(n) \leq f(n) \leq c2\ g(n)$

For any $n \geq k$

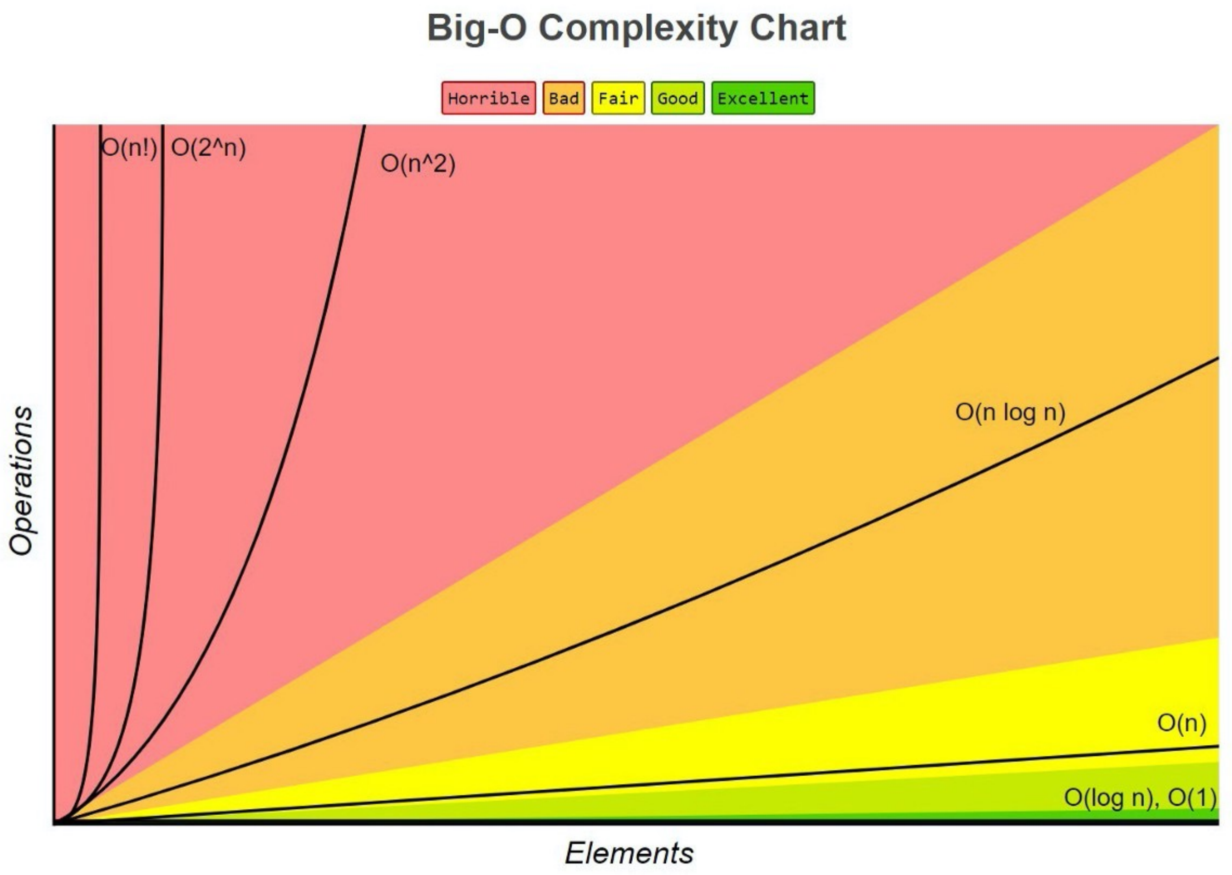


Summary

Big-O Notation	Comparison Notation	Limit Definition
f $\in o(g)$	f $<$ g	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
f $\in O(g)$	f $\leq$ g	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
f $\in \Theta(g)$	f $=$ g	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R}_{>0}$
f $\in \Omega(g)$	f $\geq$ g	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
f $\in \omega(g)$	f $>$ g	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Important Functions for Complexity

- Most algorithms will come with one of the given complexity:



Warning: the graph is not orthonormal. **See O(n)**

Best vs Worst case

- Consider a function trying to find whether a value can be found in a list:

```
def find_val(array, x):
    for element in array:
        if x == element:
            return True
    return False
```

n iterations

but may finish earlier

What are the best and worst case complexity?

Best: `find_val(1,16)`, 1 iteration, i.e. O(1)

`find_val(1,7)`, n iterations, i.e. O(n)

Examples – Bubble Sort

```python
def bubble_sort(array):
    n = len(array)

    for big_idx in range(1, n):
        for small_idx in range(1, n):
            if array[small_idx - 1] > array[small_idx]:
                array[small_idx - 1], array[small_idx] =
```

n-1 outer iterations

n-1 inner iterations

- Each outer iterations runs n-1 inner iterations
- Number of iterations needed = $(n-1)^2 = \mathbf{n^2} - 2n + 1$
- Upper bound ~ n^2, lower bound ~ n^2
- Bubble sort ~ $O(n^2)$

Measuring Computation Time

- Matplotlib

```
In [35]: import matplotlib.pyplot as plt
         import numpy as np
```

```
In [36]: nelements = [10, 100, 1000, 10000, 100000]
         bubbletime = [5.7220458984375e-06, 2.2649765014648438e-05, 0.0014679431915283203, 0.2126140594482422, 25.051520347595215]

         plt.title("Bubble Sort on Randomly Shuffled Array")
         plt.xlabel("log(number of input)")
         plt.ylabel("log(computation time (s))")
         plt.plot(np.log(nelements), np.log(bubbletime),'o-')
```

`plt.title()` = give a title to the graph

`plt.xlabel()` = show a label on the x axis

`plt.plot()` = needed to plot the graph

Plot arguments

```
plt.ylabel("log(computation time (s))")
plt.plot(np.log(nelements), np.log(bubbletime),'o-')
```

- Arguments:
  - x values
  - y vales
  - markers/drawing style (see https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html)

In this example, o is a circle markers for the data points, and - means using an “unbroken” line