FS12

Week 05
Asymptotic Complexity.

Mikhail Masyagin
Daniil Devyatkin

Intro to Asymptotic Complexity

- An **algorithm** is a formally described computational procedure that obtains the input or its argument, and giving the result
- The algorithm defines a mapping F : X → Y, where X set of initial data, Y set of values. (output);

```
output function() input {
    computing procedure
}
```

Intro to Asymptotic Complexity

- One of the main tasks in development is the analysis of algorithms;
- The analysis of the algorithm consists in predicting the amount of computing resources required for its operation. Most often, we are interested in the **running time** of the algorithm and the **amount of RAM used**.

Intro to Asymptotic Complexity

- Algorithm analysis is done to:
  - Algorithm Comparisons;
  - For a approximate estimate of programs efficiency in a new environment;
  - To identify practical applicability algorithm;

Intro to Asymptotic Complexity

- Often the initial data are characterized by natural number **n**;
- Then we can calculate the running time of the algorithm by function depending on **n** - T(n);
- Additional volume memory – M(n);
- Example:
  - Sorting an array. The size of the original array is important – n;

Intro to Asymptotic Complexity

Given a set of input data, we can calculate how many steps the algorithm takes to complete. However, in order to evaluate the quality of an algorithm, it is required to know how it behaves on all sets of input data.

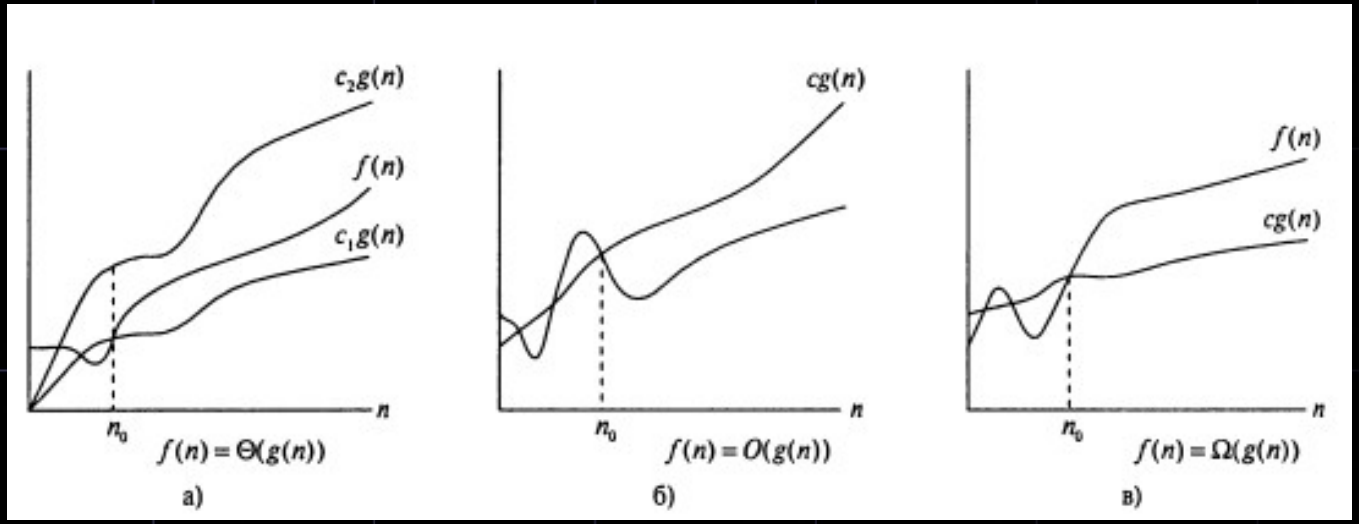
Definition. The complexity of the algorithm in the best (worst) case is a function that specifies the dependence of the min (max) number of steps of the random access memory machine exec the given algorithm, on the size of the input data.

Intro to Asymptotic Complexity

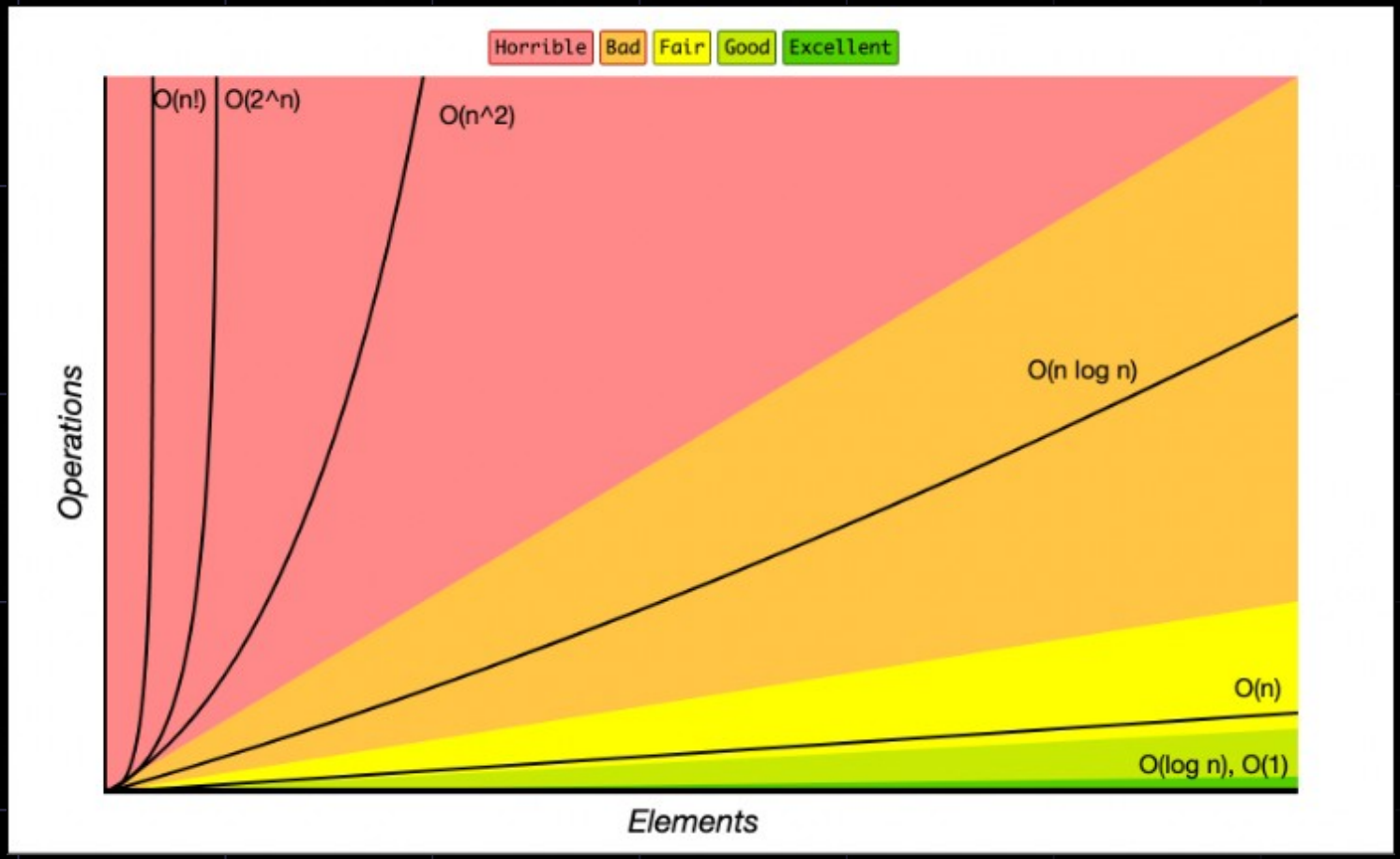
- *Definition.* The function $g(n)$ is an asymptotic lower bound for the function $f(n)$ if it is possible to find positive constants c and c_0 such that for any $n > n_0$ the expression $f(n) \geq c \cdot g(n)$ is true. For some function $g(n)$ designation is $\Omega(n)$. Also is named like a asymptotically lower bound;
- *Definition.* The function $g(n)$ is an asymptotic upper bound for the function $f(n)$ if it is possible to find positive constants c and c_0 such that for any $n > n_0$ the expression $f(n) \leq c \cdot g(n)$ is true. For some function $g(n)$ designation is $O(n)$. Also is named like a asymptotically upper bound.

Intro to Asymptotic Complexity

- If more formal:
  - $f(n) \in \Theta(g(n)) = \{ \exists\, c_1 > 0, c_2 > 0, n_0 > 0 : 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \ \forall\, n > n_0 \}$;
  - $f(n) \in O(g(n)) = \{ \exists\, c > 0, n_0 > 0 : 0 \le f(n) \le c\, g(n), \ \forall\, n > n_0 \}$;
  - $f(n) \in \Omega(g(n)) = \{ \exists\, c > 0, n_0 > 0 : 0 \le c\, g(n) \le f(n), \ \forall\, n > n_0 \}$;



Intro to Asymptotic Complexity



O(logn) – raising a number to a power;
O(n) – add element in array's beginning;
$O(n^2)$ – some sorts algorithms;

Meme-time!


big O
an orgasm, usually intense.
"He got me to the big O last night.
by Starla Pureheart June 29, 2002
313
60
Big O notation
Article
Talk
Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Big O is a member of a family of notations invented by Paul Bachmann,[1] Edmund Landau,[2] and others, collectively called Bachmann–Landau notation or asymptotic notation.
ProgrammerHumor.io

Complexity of operation

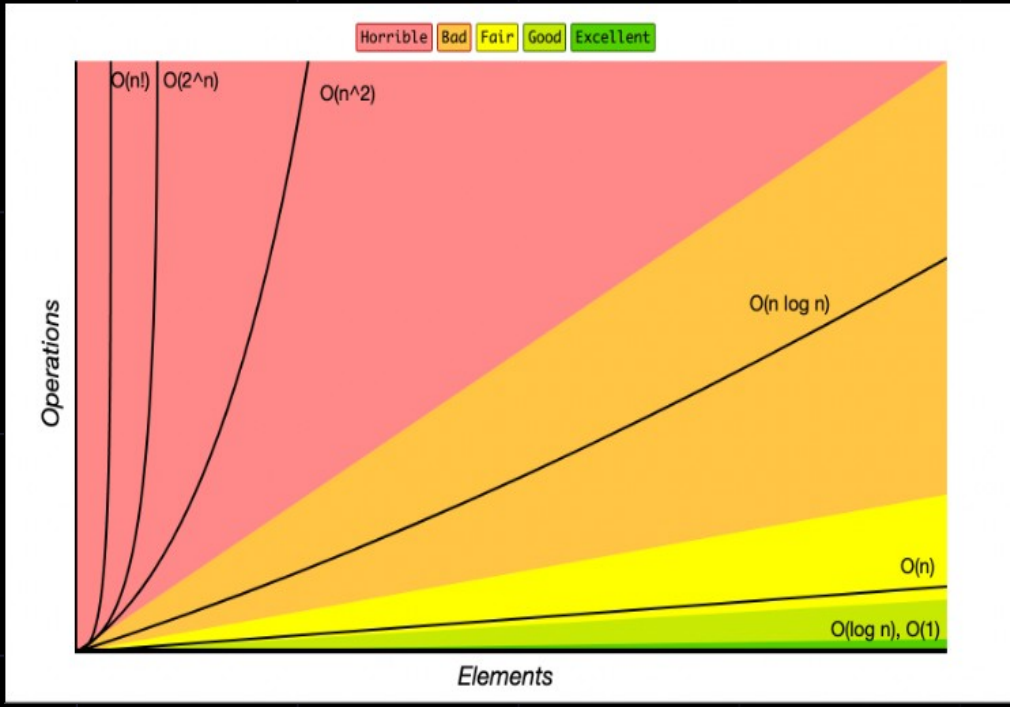
- Every simple operation (+, -, /, *, =, condition check, call sub-program) perform in one step;
- Cycles and programs are not considered simple operations: they are looking at how the compositions of the operations that make up their body;
- Each access to memory is performed in exactly one step.

Complexity of operation

- Every simple operation (+, -, /, *, =, condition check, call sub-program) perform in one step;
- Cycles and programs are not considered simple operations: they are looking at how the compositions of the operations that make up their body;
- Each access to memory is performed in exactly one step.

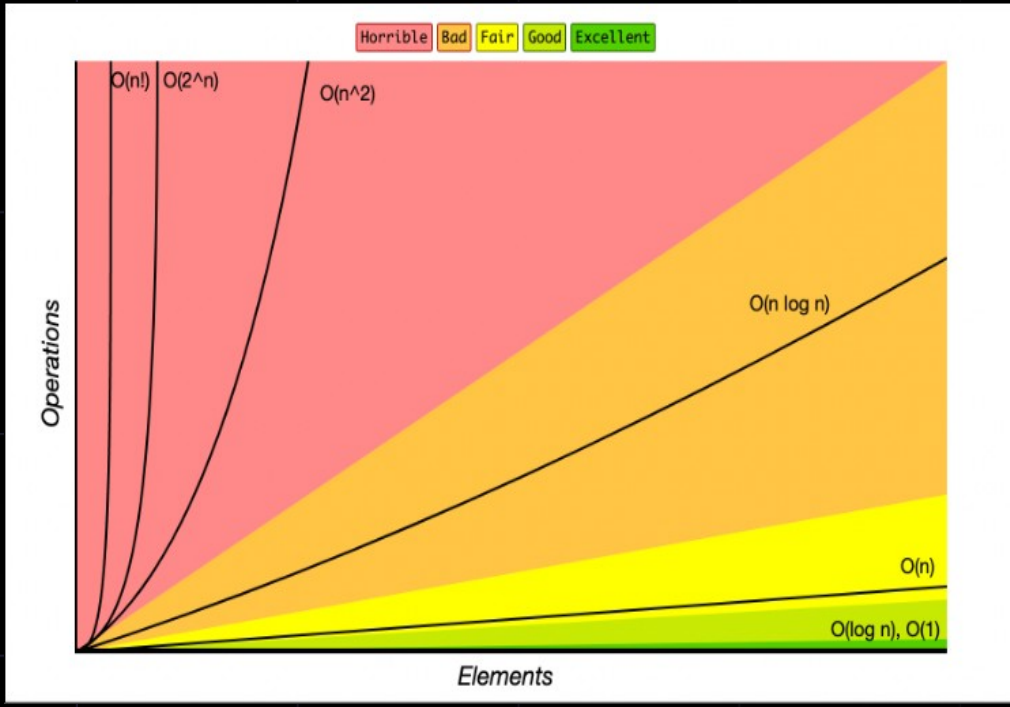
Complexity of cycles

```
def f(n) → None:
    for i in range(100):
        pass

# O(1)


def f(n: int) → None:
    for i in range(n):
        pass

# O(n)
```



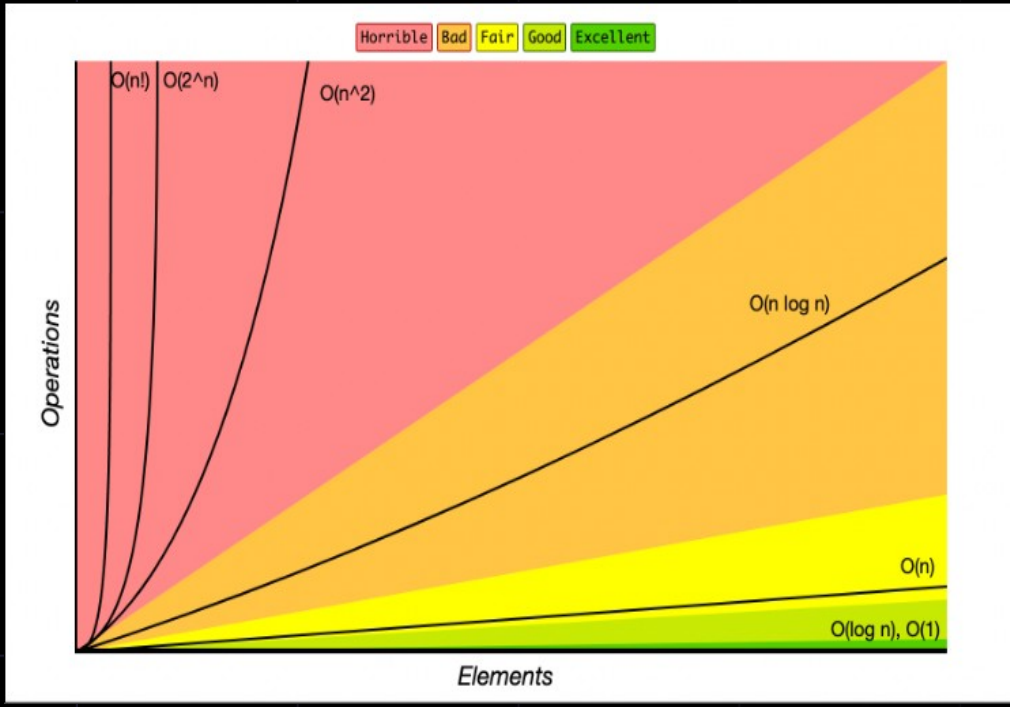
Complexity of cycles

```
def f(n: int) → None:
    a = 1
    for i in range(n):
        ...
    a += 1


# O(1 + n + 1) = O(n)
```



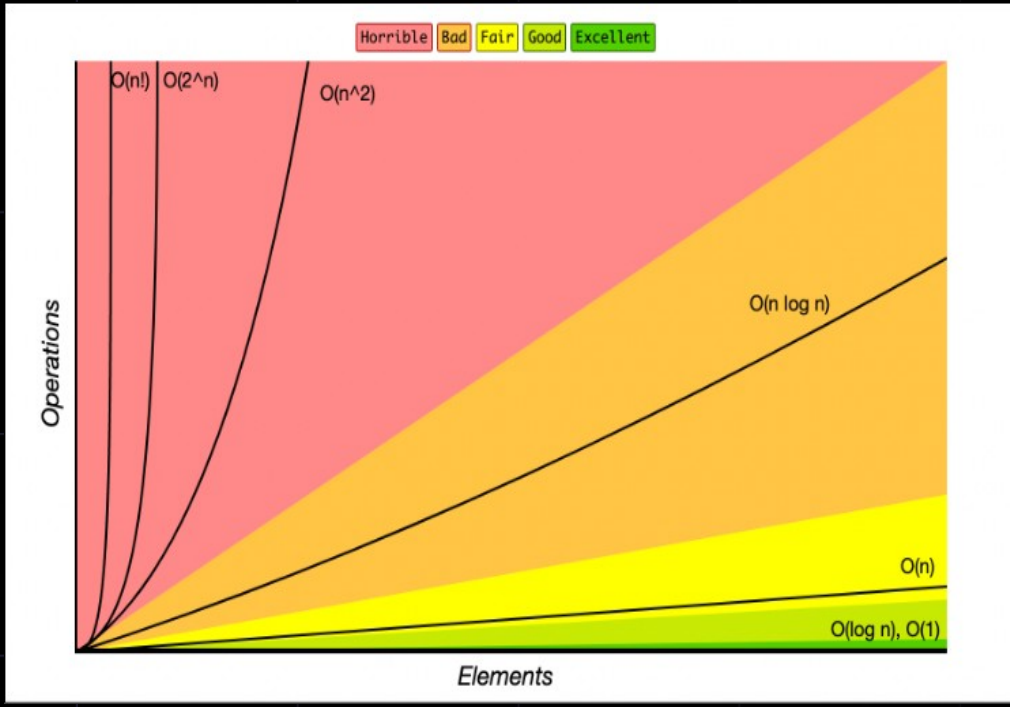
Complexity of cycles

```
def f(n: int, k: int) → None:
    a = 1
    for i in range(n):
        ...
    for i in range(k):
        ...
    a += 1
```



Complexity of cycles

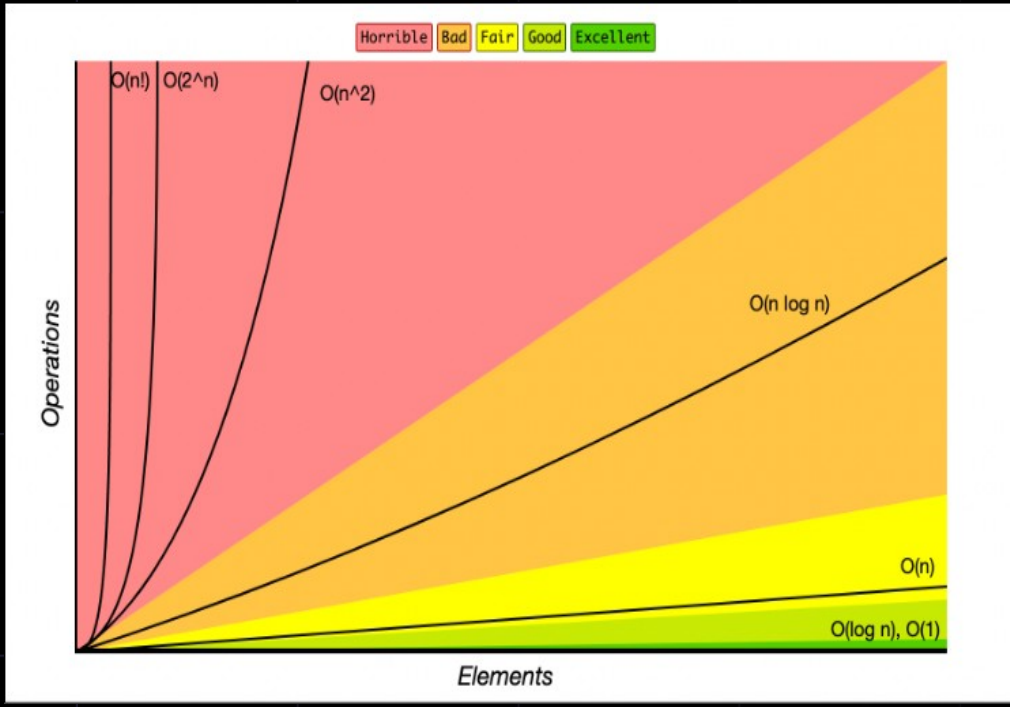
```
def f(n: int, k: int) → None:
    a = 1
    for i in range(n):
        ...
    for i in range(k):
        ...
    a += 1
```




```
# O(1 + n + k + 1) = O(n + k)
# if k = n, then O(2n) = O(n)
```

Complexity of cycles

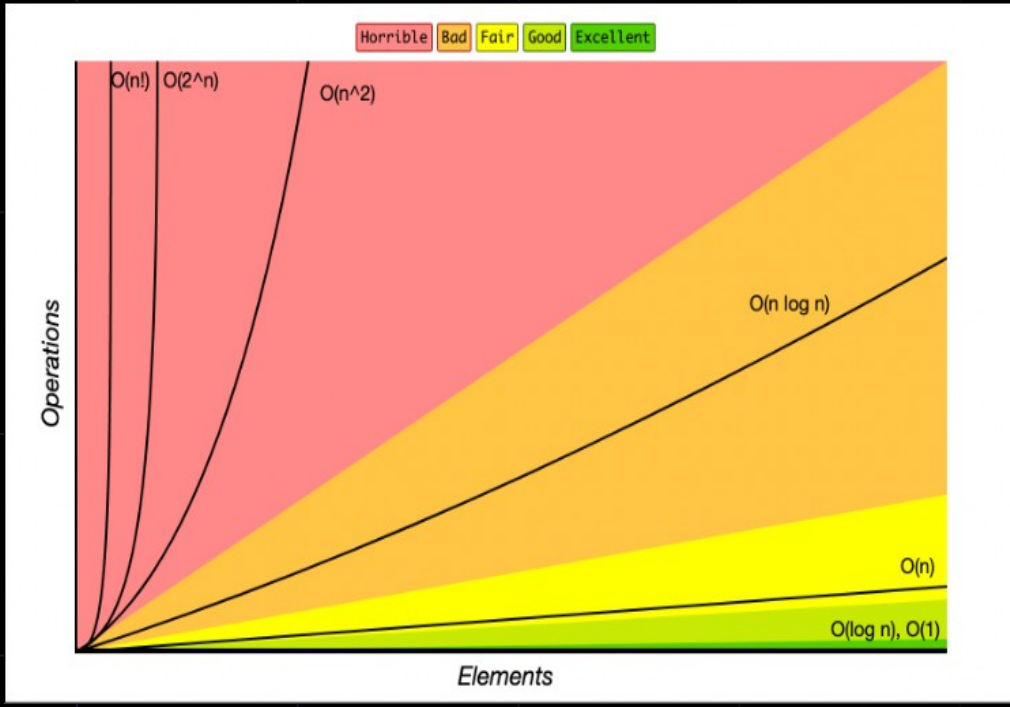
```
def f(n: int, k: int) → None:
    a = 1
    for i in range(n):
        for j in range(k):
            ...
    a += 1
```




```
# O(1 + n * k + 1) = O(n * k)
# if k = n, then O(n*n) = O(n²)
```

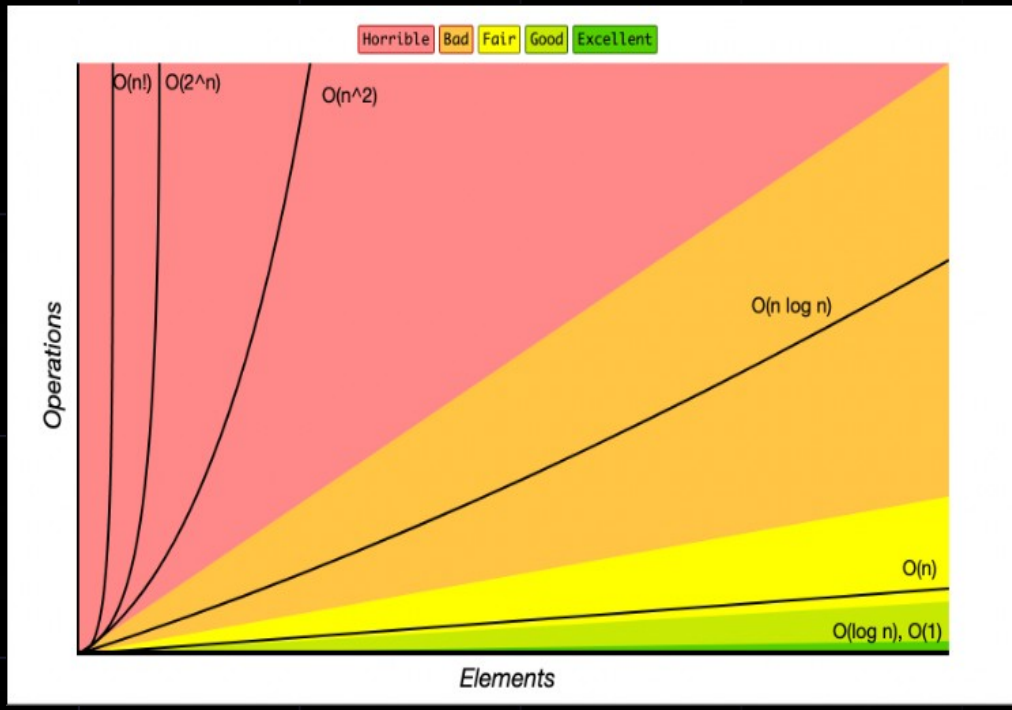
Complexity of cycles

```
def f(n: int, k: int) → None:
    a = 1
    for i in range(n):
        for j in range(k):
            ...
    for i in range(n):
        ...
    a += 1
```



Complexity of cycles

```
def f(n: int, k: int) → None:
    a = 1
    for i in range(n):
        for j in range(k):
            ...
    for i in range(n):
        ...
    a += 1
```




```
# O(1 + n * k + 1 + n) = O(n * k + n) = O(n * k)
# if k = n, then O(n*n) = O(n² + n) = O(n²)
```

Complexity of recursion

- In the general, it is necessary to calculate the recurrence relation;

Asymptotic Complexity - ???

```
def f(n):
    if n in (0, 1): return 1
    return n * f(n - 1)
```

Complexity of recursion

- In the general, it is necessary to calculate the recurrence relation;

Asymptotic Complexity - ???

```
def f(n):
    if n in (0, 1): return 1
    return n * f(n - 1)
```

O(n)

If n == 1 => 1 operations
If n == 2 => 1 + 1 operations

For k + 1 steps => k + 1 => $AC_k = k$

Complexity of recursion

- Substitute different argument values and calculate the amount of function calls;

Example:

Asymptotic Complexity - ???

```
def f(n: int) → int:
    if n == 1: return 1
    return f(n – 1) + f(n - 1)
```

Complexity of recursion

- Substitute different argument values and calculate the amount of function calls;

Example:

Asymptotic Complexity - ???

```
def f(n: int) → int:
    if n == 1: return 1
    if n == 0: return 0
    return f(n – 1) + f(n - 2)
```

If n == 1 => 1 operation
If n == 2 => 3 operations
If n == 3 => 7 operations
If n == 4 => 15 operations
If n == 5 => 31 operations

$O(1*(2^n - 1)) = O(2^n)$

Linear & binary search I: Linear

- Task 1. Check if the given element is in the array.

Linear & binary search I: Linear

- Task 1. Check if the given element is in the array.
- Solving. (Long) We sequentially check all elements of the array until we find the given element, or until the end of the array.
- Worst case run time???

Linear & binary search I: Linear

- Task 1. Check if the given element is in the array.
- Solving. (Long) We sequentially check all elements of the array until we find the given element, or until the end of the array.
- Worst case run time **O(n)**, where n is amount of array's elements. Proof: enumerate all elements.

Linear & binary search II: Binary

- Task 2. Check if the given element is in the **ordered** (!) array.
- Definition. Ascending array is array A whose elements are comparable, and ∀ i, j ∈ Z : i < j : A[i] =< A[j]. A descending sorted array is defined similarly.

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

Linear & binary search II: Binary

- Task 2. Check if the given element is in the **ordered** (!) array.
- Solving. Array is ordered – good condition!
  - Step. Compare median element with current element. Select the desired half of the array, depending on the result of the comparison. Repeat step while array's length is not equal 1.

Linear & binary search II: Binary

- More deeper. There are 3 variable: left (left bound’s idx), right (right bound’s idx) and mid (middle of array idx), array A and searching value - s. Check if A[mid] == s then cool! If A[mid] > s then left = left, right = mid, mid = (left + right) // 2. A[mid] < s then right = right, left = mid, mid = (left + right) // 2. Repeat.
- Run time **O(logn)**, where n is amount of array’s elements. Proof: split the array in half

Linear & binary search II: Binary

Example:

S = -12, left = 0, right = 9

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

-40	-12	0	1	2
0	1	2	3	4

-40	-12
0	1

Data structures: Array

Operation	Time
Find min (max)	O(n)
Delete min (max)	O(n)
Add value	O(1)
Get value by index	O(1)
Delete from beginning	O(n)

Heap: Definition

- Definition. Max (min) **Heap** is a abstract treelike data structure, that fulfills three properties:

  1) The value at any vertex is not less (greater) than the values of its children;

  2) The depth of the leaves (distance to the root) differs by no more than one;

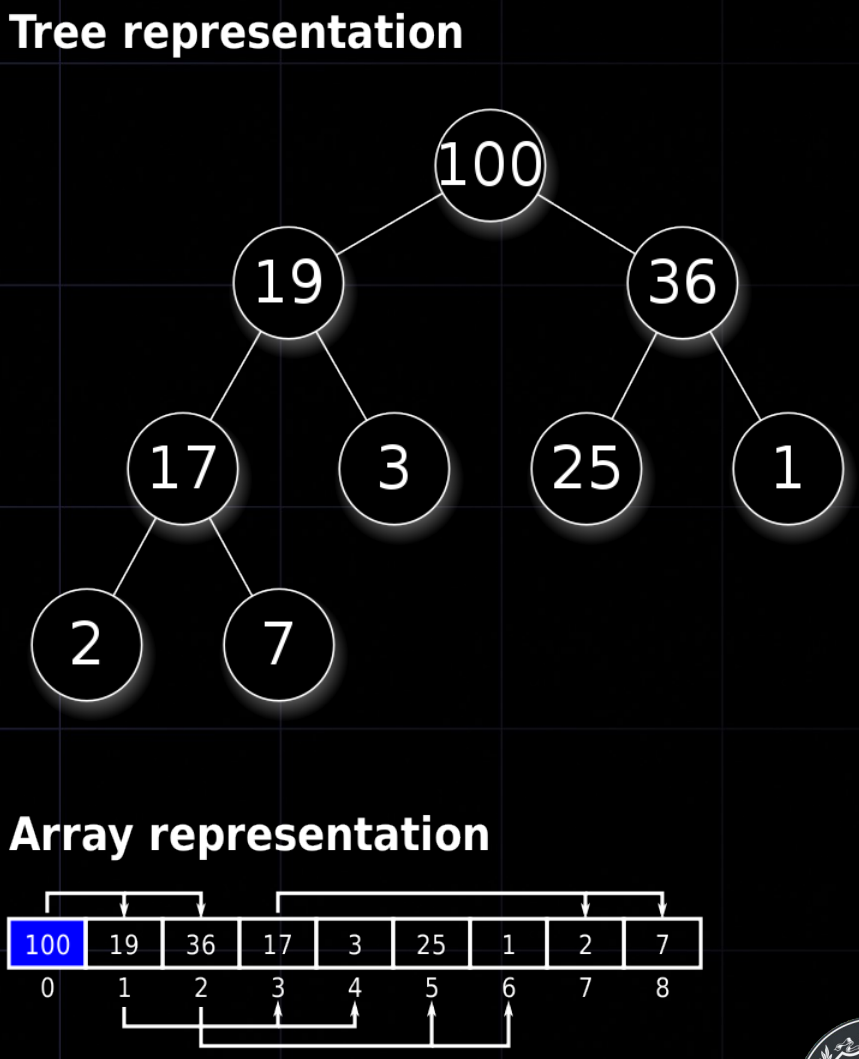
  3) The last layer is filled from left to right.

- We will consider max heap.

Heap: Visualization

- Heap's visualization like a tree, but in memory it keep as array with some rules!

- Also heap calls a binary heap, pyramid and sorting tree.



Heap I

- A convenient way to store a binary heap is an array;
- Sequentially store all the elements of the heap "in layers".
- The root is the first element of the array, the second and third elements are the child elements, and so on.

Heap I

- This store case is efficient for given an element; (O(1) for call by index)
- If array element indexing starts from 0 then:
  - A[0] is the element at the root;
  - The children of the element A[i] are the elements A[2i + 1] and A[2i + 2];
  - The parent of the element A[i] is the element A[(i – 1)/2].

Heap I

- If something element in heap was change, then heap can loss it properties;
- For recover heap's properties there are 2 methods: **sift up** and **sift down**:
  - Sift up – raise element, that is larger than parent;
  - Sift down – omit the element that is smaller than the children;

Heap: Sift Down I

- Sift Down (main idea):

  If the *i*-th element is greater than its children, the entire subtree is already a heap, and nothing needs to be done. Otherwise, we swap the *i*-th element with the largest of its sons, after which we perform Sift Down for this son;

  - Time complexity is O(???).

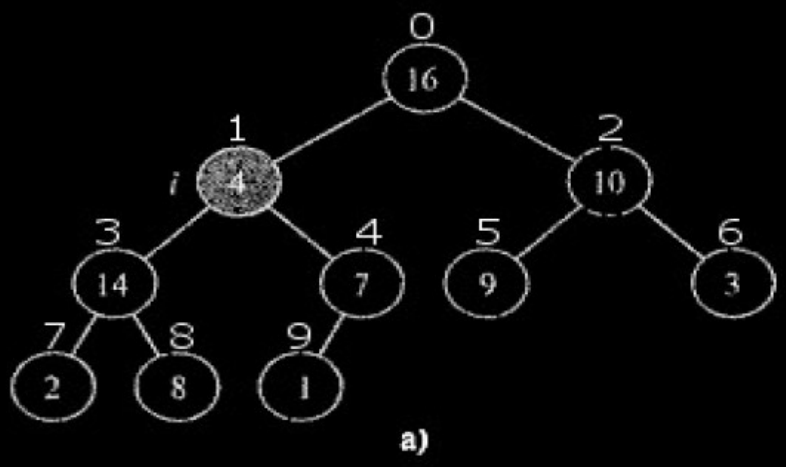
Heap: Sift Down I

- Sift Down (main idea):

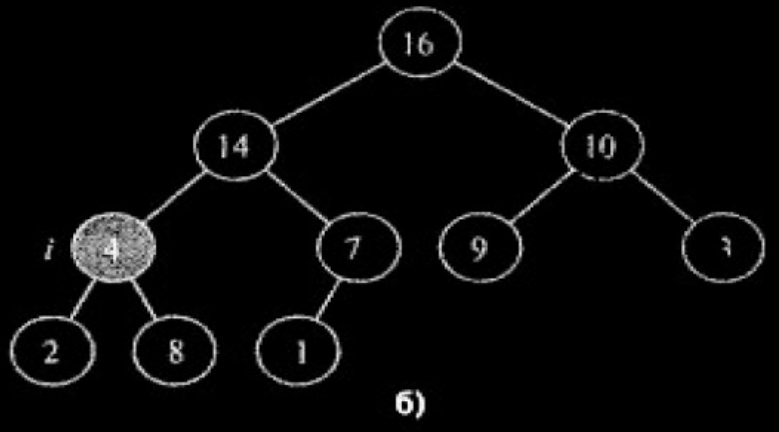
  If the *i*-th element is greater than its children, the entire subtree is already a heap, and nothing needs to be done. Otherwise, we swap the *i*-th element with the largest of its sons, after which we perform Sift Down for this son;

  - Time complexity is O(logn).

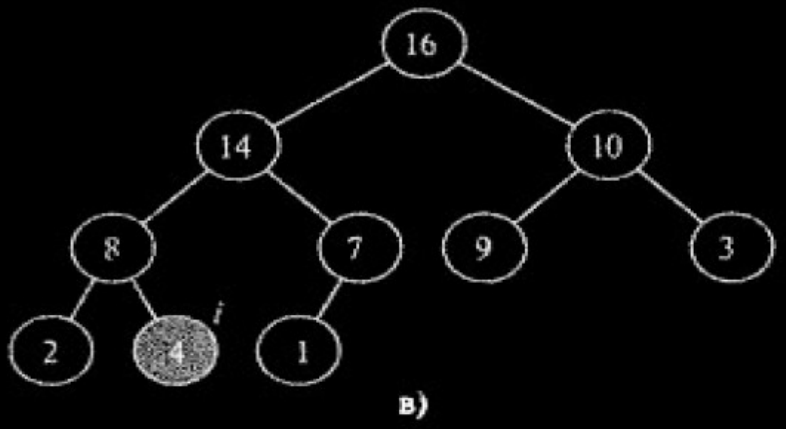
Heap: Sift Down II



а)



б)



в)

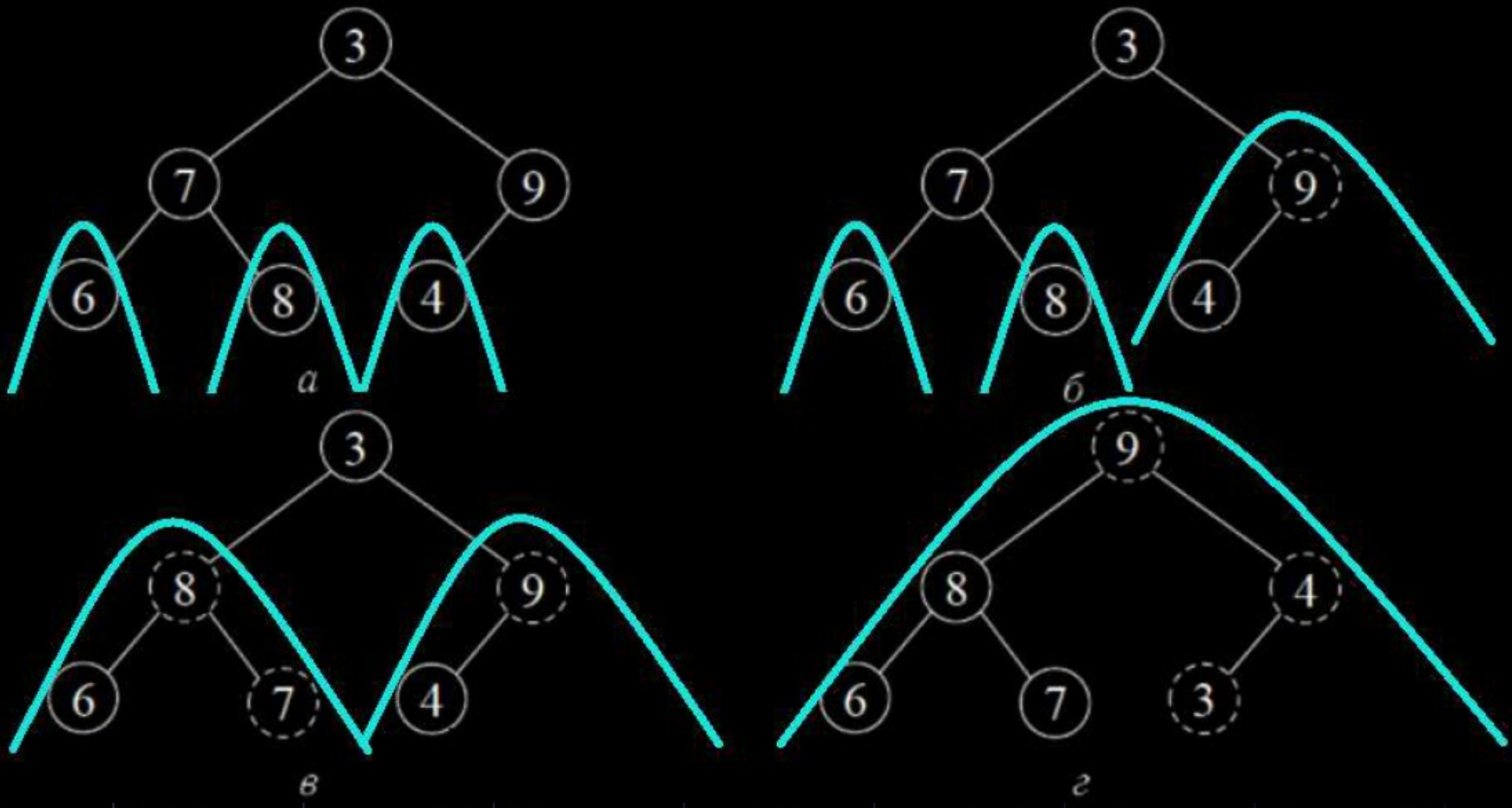
Heap: Heap building I

- It is also possible to create a heap from an unordered array:
  - If you perform Sift Down on all elements of array A, from the last to the first, it becomes a heap;
  - SiftDown(A, i) does nothing if;
  - It is enough to call SiftDown for all elements of array A from ([n/2] – 1)-th to 1-st;

Heap: Heap building I

- It is also possible to create a heap from an unordered array:
  - If you perform Sift Down on all elements of array A, from the last to the first, it becomes a heap;
  - SiftDown(A, i) does nothing if;
  - It is enough to call SiftDown for all elements of array A from ([n/2] – 1)-th to 1-st;
  - Time complexity is O(???).

Heap: Heap building II



Heap: Heap building III

- Time complexity for heap build is O(n).

  Proof. The SiftDown running time for working with a node that is at height h (bottom) is C * h. At level h, contains no more than $\lceil n/2^{h+1} \rceil$ nodes. Total operating time:

$$T(n)=\sum_{h=0}^{logn}\left\lceil \frac{n}{2^{h+1}} \right\rceil C\cdot h=O\left(n\sum_{h=0}^{logn}\frac{h}{2^h}\right)=O(2n)=O(n)$$

$$\sum_{h=0}^{\infty}\frac{h}{2^h}=\frac{1/2}{(1-1/2)^2}=2$$

Heap: Sift Up I

- Restores the ordering property by pushing the element to the top;
- If the element is larger than the parent, swaps it with the parent;
- If after that the parent is larger than the grandparent, the parent is swapped with the grandparent, and so on.

Heap: Sift Up I

- Restores the ordering property by pushing the element to the top;
- If the element is larger than the parent, swaps it with the parent;
- If after that the parent is larger than the grandparent, the parent is swapped with the grandparent, and so on.
- Time complexity is O(logn).

Heap: Add Element To Heap I

1) Let's add an element to the end of the heap;

2) Let's restore the ordering property by pushing the element to the top with SiftUp;

- The running time is O(???), if the buffer for the heap allows you to add an element without re-allocation.

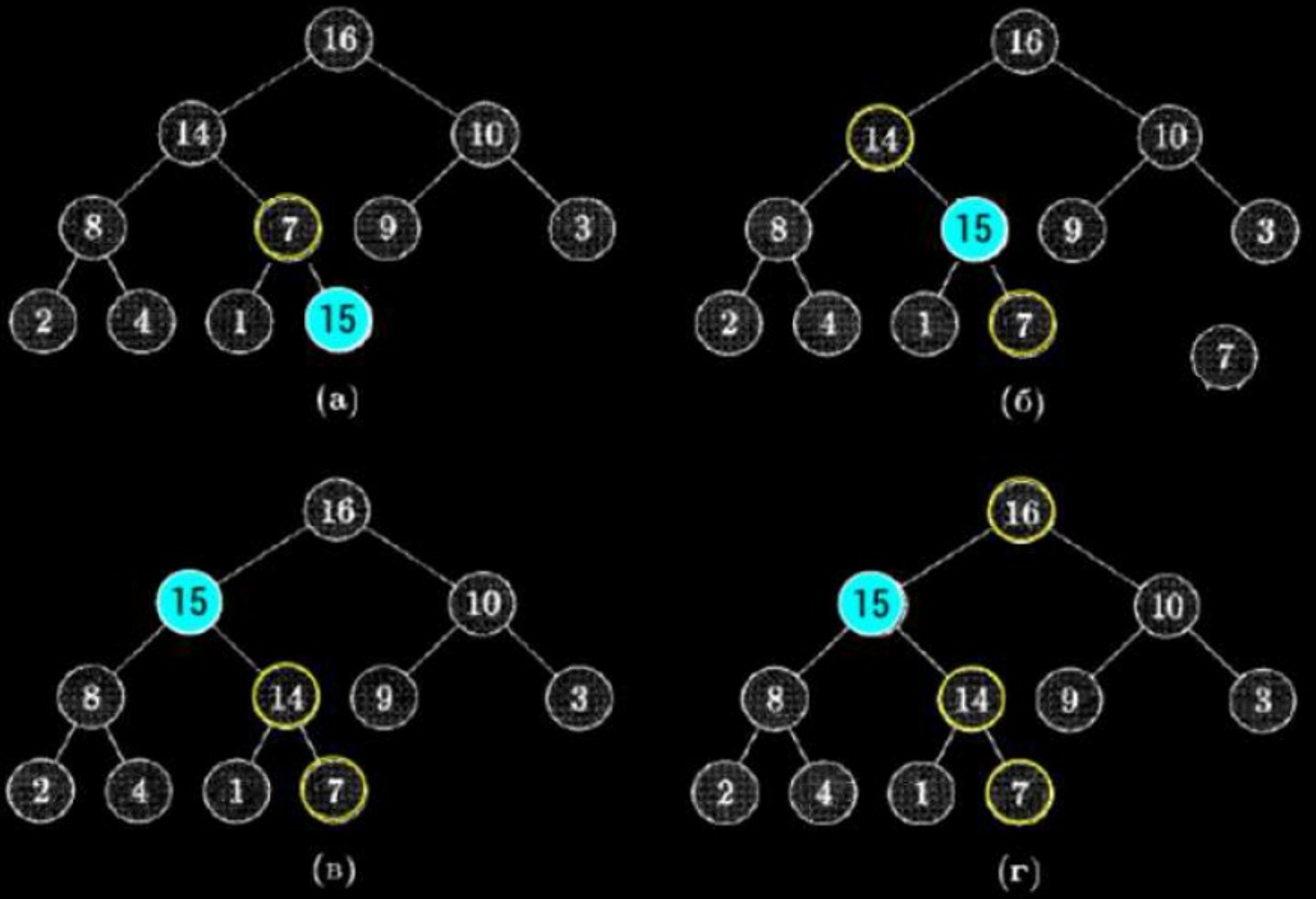
Heap: Add Element To Heap I

1) Let's add an element to the end of the heap;

2) Let's restore the ordering property by pushing the element to the top with SiftUp;

- The running time is O(logn), if the buffer for the heap allows you to add an element without re-allocation.

Heap: Add Element To Heap II



Heap: Give Element From Root I

- Algorithm:

  1) Store the value of the root element to return;

  2) Copy the last value to the root, delete the last position;

  3) Call SiftDown to root;

  4) Return stored element.

  - Time complexity: O(???)

Heap: Give Element From Root I

- Algorithm:

  1) Store the value of the root element to return;

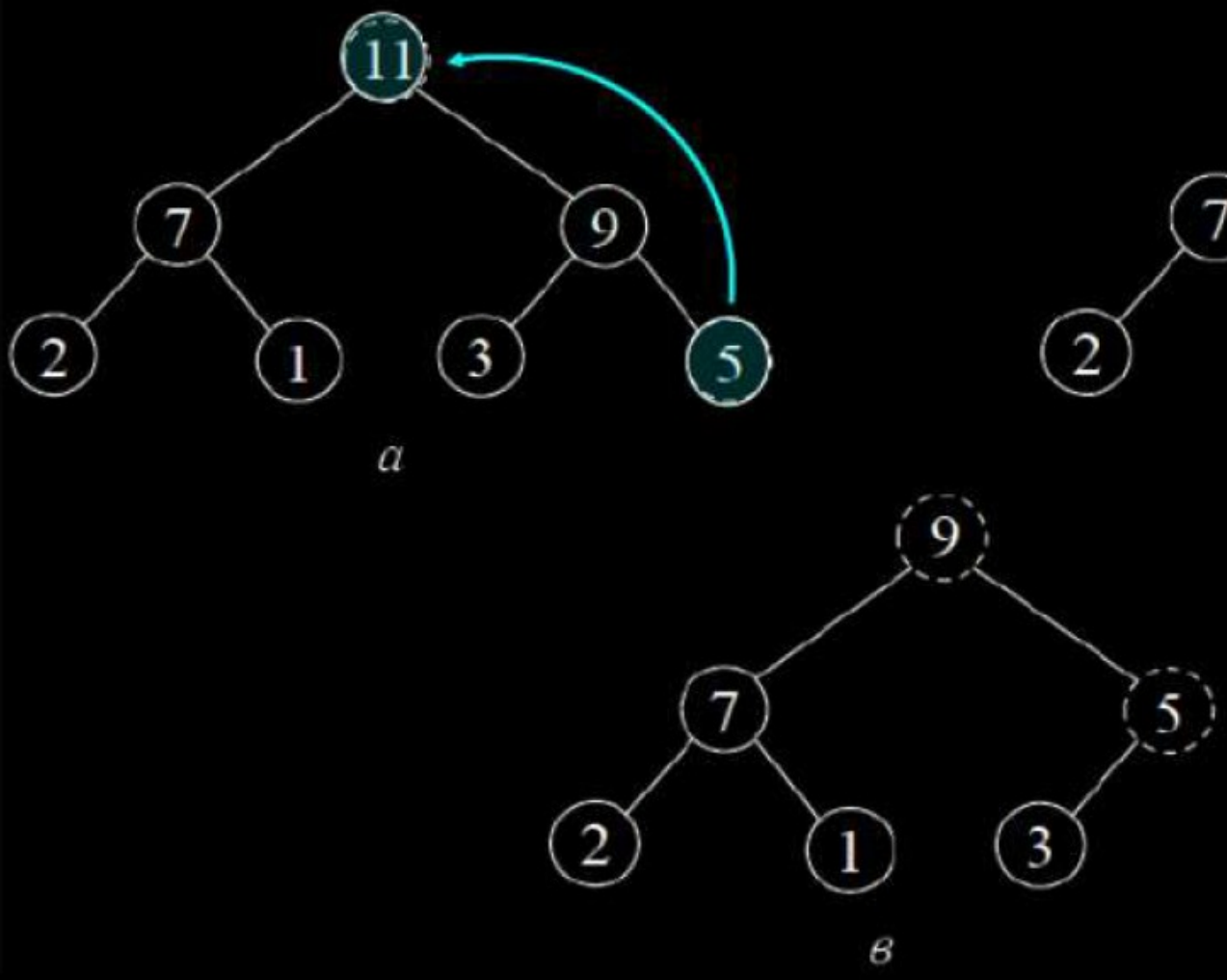
  2) Copy the last value to the root, delete the last position;

  3) Call SiftDown to root;

  4) Return stored element.

  - Time complexity: O(logn).

Heap: Give Element From Root II



Heap: Info

Operation	Time
Find min (max)	O(1)
Delete min (max)	O(n logn)
Add	O(n logn)
Get value by index	O(1)
Build heap	O(n)

Sorting

Sorting is element ordering process.

- Recap:
  - Definition. Ascending array is array A whose elements are comparable, and $\forall\ i, j \in Z : i < j : A[i] =< A[j]$. A descending sorted array is defined similarly.

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

Sorting: Types

Definition. A **stable** sort is one that preserves the order of equal elements.

Example: Sort numbers by high order.

10	25	30	31	24	21	36	32	11

10	11	25	24	21	30	31	36	32

Sorting: Types

Definition. A **local** sort is one that does not require additional memory.

Example:

- Heap sort;
- Merge sort.

Sorting: Types

Definition. A **local** sort is one that does not require additional memory.

Example:

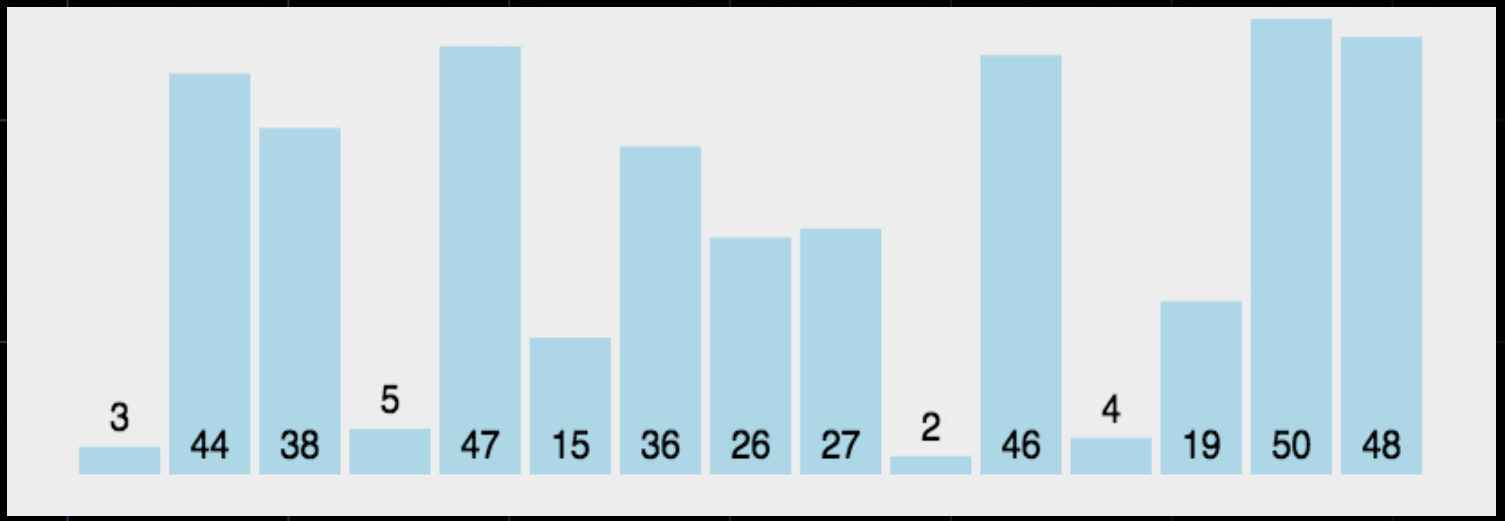
- Heap sort;
- Merge sort.

Sorting: Bubble Sort

- Sorted array is divided into 2 sides: sorted and unsorted.
- Main idea:
  - Here you need to sequentially compare the values of neighboring elements and swap numbers if the previous one is greater (less) than the next one.
  - Local, stable;

Sorting: Bubble Sort

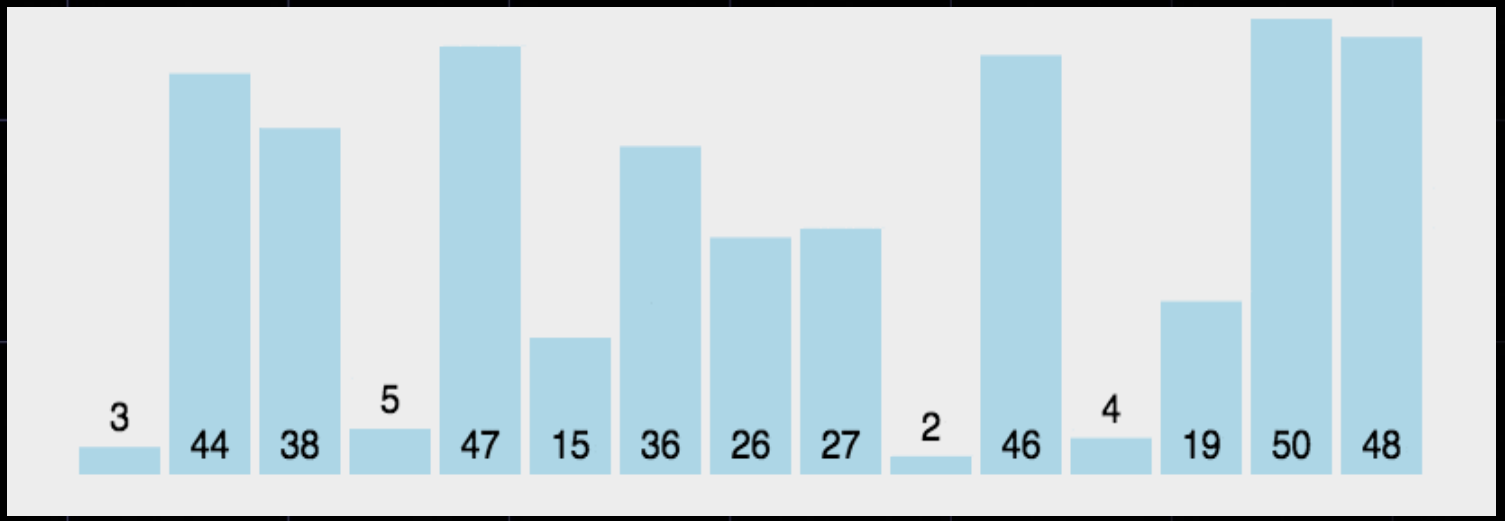
- Example visualization with array:



Time complexity is O(???).

Sorting: Bubble Sort

- Example visualization with array:



Time complexity is O(n) if array is sorted and $O(n^2)$ if unsorted.

Sorting: Insertion sort

- A simple algorithm often used on small volumes. (len < 60-90)
- Sorted array is divided into 2 sides: sorted and unsorted.
- Main idea:

  For *k* step:

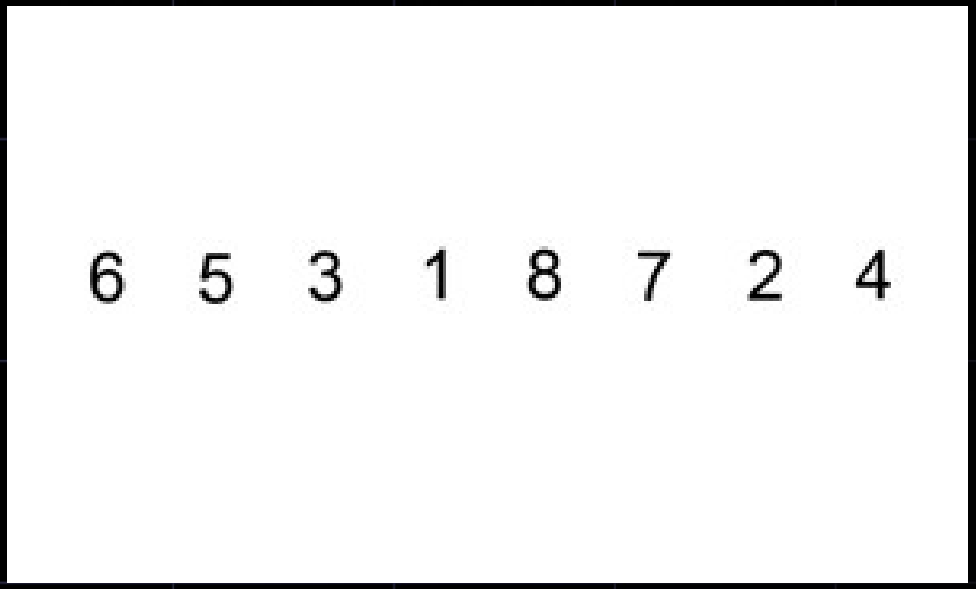
  1) take the first element of the right side;

  2) paste it in a suitable place on the left side;

  - Local, stable.

Sorting: Insertion sort

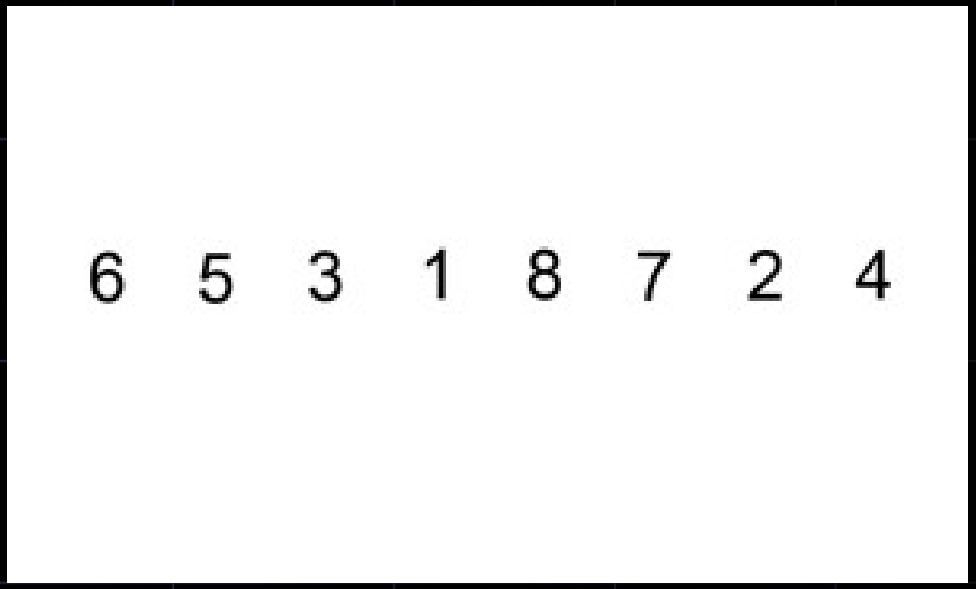
- Example visualization with array:



- Time complexity is O(???).

Sorting: Insertion sort

- Example visualization with array:



- Time complexity is O(n) if array is sorted and $O(n^2)$ if unsorted.

Sorting: Insertion sort

- Proof.

  If array is sorted: 2 * (n – 1) copying and (n – 1) comparison => O(n);

  if array is unsorted: 2 * (n – 1) + n * (n – 1) / 2 copying and n * (n – 1) / 2 comparison => $O(n^2)$.

- Remark: Use binary search in left side!!! => $O(n^2)$ for copy with small constant and O(nlogn) copying.

Sorting: Python

- There 2 ways of sorting iterable object in Python:
  - `sorted([...], reverse = ..., key = ...)` # return new object, for all iterable obj;
  - `[...].sort(reverse = ..., key = ...)` # inplace method, only for list.