Lecture 1.

Algorithm complexity estimation.
Sorting algorithms.

Algorithms and Data Structures
Ivan Solomatin
MIPT 2021

Outline

- Course description.
- Algorithm complexity basics
- Sorting problem statement
- Quadratic $O(N^2)$ sorting algorithms
  - Selection sort
  - Insertion sort
  - Bubble sort
- Linearithmic $O(N \log N)$ sorting algorithms
  - Merge sort
  - Quick sort
  - Heap sort (later)
- Greedy algorithms

Course description

Course goal:

- Get familiar with basic algorithms and data structures.
- Learn how to implement basic algorithms and data structures on python.
- Learn how to apply obtained knowledge in practice.
- Get experience and intuition in programming problems solving.

Sources:

- Thomas H. Cormen, et al. *Introduction to algorithms*. MIT press, 2009.
- www.geeksforgeeks.org/fundamentals-of-algorithms
- www.e-maxx.ru/algo/

Course description

Course content:

- Lectures (online)
- Homework (problem solving in contest.yandex.ru)
- Seminars (online, real time):
  - Brief repetition of lecture materials
  - Homework analysis
  - Additional topics / problems
  - Q&A

Final grade:

- Homework (~60%)
- Practical exam (problem solving) (~20%)
- Theoretical exam (~20%)

Algorithm complexity basics

Definition

Algorithm: a finite sequence of well-defined, computer-implementable instructions, which solves a class of problems.

$$A: X \to Y$$

Number of elementary operations (processor instructions) for executing A on input $x \in X$:

$$t(A, x)$$

Size of input x:

$$s(x)$$

$$\boldsymbol{N \to \infty}$$

Worst case complexity:

$$T(A, N) = \max_{\substack{x \in X \\ s(x) = N}} t(A, x)$$

$$50N + 100500 = \boldsymbol{O}(\boldsymbol{N})$$

$$10N^2 + 5N + 1 = \boldsymbol{O}(\boldsymbol{N^2})$$

$$N \log_2 N = N \frac{\log_c N}{\log_c 2} = \boldsymbol{O}(\boldsymbol{N} \mathbf{log} \boldsymbol{N})$$

$$42 = \boldsymbol{O}(\mathbf{1})$$

Algorithm complexity basics

Examples

Problem:

Calculate sum of two given numbers x, y:

$$x, y \in [-2^{63}; 2^{63})$$

Input:

- x – 64 bit integer = 8 bytes
- y – 64 bit integer = 8 bytes

input size: 16 bytes

Algorithm:

calculate sum, return result.

Complexity:

$$O(1)$$

Algorithm complexity basics

Examples

Problem:

Calculate sum of N numbers $x_1, \dots, x_N$.

$$x_i \in [-2^{31}; 2^{31})$$

Input:

- x_1 – 32bit integer = 4 bytes
- …
- x_N – 32bit integer = 4 bytes

size: $4N$ bytes

Algorithm:

Iterate over x_i and accumulate sum:

```
res = 0
for i in range(N):
    res += x[i]
```

Complexity:

$$O(1) * N = O(N)$$

Algorithm complexity basics

Examples

Problem:

Check if there is a pair of equal numbers between given

N numbers $x_1, \dots, x_N$.

$x_i \in [-2^{31}; 2^{31})$

Input:

- x_1 − 32bit integer = 4 bytes
- …
- x_N − 32bit integer = 4 bytes

size: $4N$ bytes

Algorithm:

Iterate over each pair and check equality:

```python
def f(x):
  N = len(x)
  for i in range(N):
    for j in range(i + 1, N):
      if x[i] == x[j]:
        return True
  return False
```

Complexity:

In worst case: $(\mathrm{N}-1)+(\mathrm{N}-2)+\cdots+1=\frac{N(\mathrm{N}-1)}{2}=O(N^2)$

Sorting problem statement

Given sequence of objects (let's suppose they're integer numbers for simplicity).

$$x_0, x_1, \dots, x_{N-1}: \ \ x_i \in X \tag{1}$$

Also given binary relation $\leq$ (transitive, reflexive) on X.

Task is to reorder elements:

$$x_{i_0}, x_{i_1}, \dots, x_{i_{N-1}} \tag{2}$$

$$x_{i_0} \leq x_{i_1} \leq \dots \leq x_{i_{N-1}} \tag{3}$$

Sequence of source indices used to reorder elements construct a permutation:

$$\pi = (i_0, i_1, \dots i_{N-1})$$

In other words, the task is to find a permutation that will satisfy (3).

Sorting problem statement

Example

	0	1	2	3	4	5	6	7
x	2	3	6	8	1	4	7	5

↓

$\pi = ($ 4, 0, 1, 5, 7, 2, 6, 3 $)$

$\text{sorted}(x)$	1	2	3	4	5	6	7	8

$\leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq$

$$x_{i_0} \leq x_{i_1} \leq \cdots \leq x_{i_{N-1}}$$

Sorting problem statement

Example

0	1	2	3	4	5	6	7
2	3	6	8	1	4	7	5

↓

(2, 0)	(3, 1)	(6, 2)	(8, 3)	(1, 4)	(4, 5)	(7, 6)	(5, 7)

↓

(1, **4**)	(2, **0**)	(3, **1**)	(4, **5**)	(5, **7**)	(6, **2**)	(7, **6**)	(8, **3**)

↓

$\pi = ($ 4, 0, 1, 5, 7, 2, 6, 3 $)$

Quadratic $O(N^2)$ sorting algorithms

- **Selection sort**
- Insertion sort
- Bubble sort

Selection sort
Idea

Let's sort array in range $[i, N)$ (initially: $i = 0$):

1. Let's find minimum value in range $[i, N)$: $x_{i_{min}}$.

2. We know the place $x_{i_{min}}$ should take in sorted array: i-th.

3. Let's swap $x_{i_{min}}$ with value on it's desired place.

4. Now, let's sort the rest array (`x[i + 1:]`) using the same approach (`i += 1` and go to 1.).

	0	1	2	3	4	5	6	7
x	4	2	5	6	3	1	7	8

Selection sort

Implementation

```
N = 8
i = 0:
   i_min = 5
i = 1:
   i_min = 1
i = 2:
   i_min = 4
i = 3:
   i_min = 5
i = 4:
   i_min = 4
i = 5:
   i_min = 5
i = 6:
   i_min = 6
```

```python
N = len(x)
for i in range(N - 1):
    i_min = i
    for j in range(i + 1, N):
        if x[j] < x[i_min]:
            i_min = j
    x[i], x[i_min] = x[i_min], x[i]
```

0	1	2	3	4	5	6	7
4	2	5	6	3	1	7	8

Complexity: $N - 1 + N - 2 + \ldots + 1 = N(N-1)/2 = O(N^2)$

Quadratic $O(N^2)$ sorting algorithms

- Selection sort
- **Insertion sort**
- Bubble sort

Insertion sort
Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
4	2	5	6	3	1	7	8

Insertion sort

Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
4	2	5	6	3	1	7	8

Insertion sort
Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
2	4	5	6	3	1	7	8

Insertion sort
Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
2	4	5	6	3	1	7	8

Insertion sort
Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
2	3	4	5	6	1	7	8

Insertion sort
Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
2	3	4	5	6	1	7	8

Insertion sort

Idea

This sorting algorithm works similar to the way you sort playing cards in your hands:

> You have a sorted part of array in the left, and for each new element you look for a place in sorted part to insert this element, and then insert it, shifting elements to the right if needed.

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Insertion sort

Implementation

```
N = 8
i = 4
```

```
N = len(x)
for i in range(1, N):
    key = x[i]
    j = i - 1
    while j >= 0 and key < x[j]:
        x[j + 1] = x[j]
        j -= 1
    x[j + 1] = key
```

0	1	2	3	4	5	6	7
2	4	5	6	3	1	7	8

Complexity: $1 + 2 + \ldots + N - 1 = N(N-1)/2 = O(N^2)$

Quadratic $O(N^2)$ sorting algorithms

- Selection sort
- Insertion sort
- **Bubble sort**

Bubble sort
Idea

Let's sort array in range $[0, N-i)$ (initially: $i=0$):

1. Let's iterate over j $\in [0, N-i)$ and for each j, check if it's more then next value $(j+1)$, swap j and $j+1$ elements.

2. After loop 1., maximum element will go right (float like a bubble).

3. Let's increase `i` and sort the rest of the array: `x[:N-i]`.

0	1	2	3	4	5
4	2	5	6	3	1

Bubble sort
Idea

Let’s sort array in range $[0, N - i)$ (initially: $i = 0$):

1. Let’s iterate over j $\in [0, N - i)$ and for each j, check if it’s more then next value $(j + 1)$, swap j and $j + 1$ elements.

2. After loop 1., maximum element will go right (float like a bubble).

3. Let’s increase `i` and sort the rest of the array: `x[:N-i]`.

0	1	2	3	4	5
2	4	5	3	1	6

Bubble sort

Idea

Let's sort array in range $[0, N - i)$ (initially: $i = 0$):

1. Let's iterate over j ∈ $[0, N - i)$ and for each j, check if it's more then next value $(j + 1)$, swap j and $j + 1$ elements.

2. After loop 1., maximum element will go right (float like a bubble).

3. Let's increase `i` and sort the rest of the array: `x[:N-i]`.

0	1	2	3	4	5
2	4	3	1	5	6

Bubble sort

Idea

Let’s sort array in range $[0, N - i)$ (initially: $i = 0$):

1. Let’s iterate over j ∈ $[0, N - i)$ and for each j, check if it’s more then next value $(j + 1)$, swap j and $j + 1$ elements.

2. After loop 1., maximum element will go right (float like a bubble).

3. Let’s increase `i` and sort the rest of the array: `x[:N-i]`.

0	1	2	3	4	5
2	3	1	4	5	6

Bubble sort

Idea

Let's sort array in range $[0, N - i)$ (initially: $i = 0$):

1. Let's iterate over j ∈ $[0, N - i)$ and for each j, check if it's more then next value $(j + 1)$, swap j and $j + 1$ elements.

2. After loop 1., maximum element will go right (float like a bubble).

3. Let's increase `i` and sort the rest of the array: `x[:N-i]`.

0	1	2	3	4	5
2	1	3	4	5	6

Bubble sort

Implementation

```
N = len(x)
for i in range(0, N - 1):
  for j in range(0, N - i - 1):
    if x[j] > x[j + 1]:
      x[j], x[j + 1] = x[j + 1], x[j]
```

Complexity:

$$1 + 2 + \ldots + N - 1 = N(N-1)/2 = O(N^2)$$

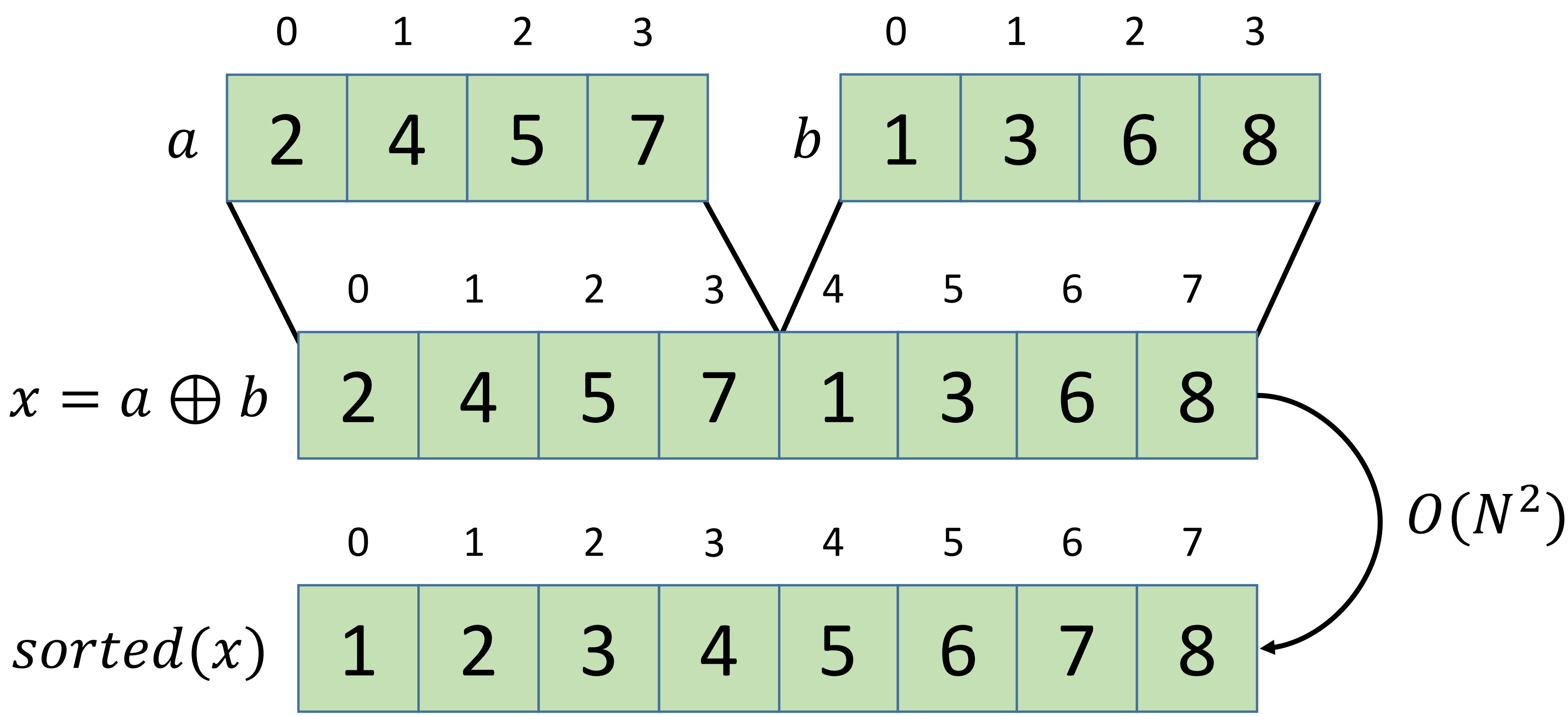
Linearithmic sorting algorithms $O(N \log N)$

- **Merge sort**
- Quick sort

Divide and conquer paradigm

Merge sort
Merging

Let’s suppose, we need to sort array which is a union of two sorted arrays, $\frac{N}{2}$ each. How fast can we sort it?



Merge sort
Merging

Let’s suppose, we need to sort array which is a union of two sorted arrays, $\frac{N}{2}$ each. How fast can we sort it?

	0	1	2	3
a	2	4	5	7
	$\uparrow$ i_1			

	0	1	2	3
b	1	3	6	8
	$\uparrow$ i_2			

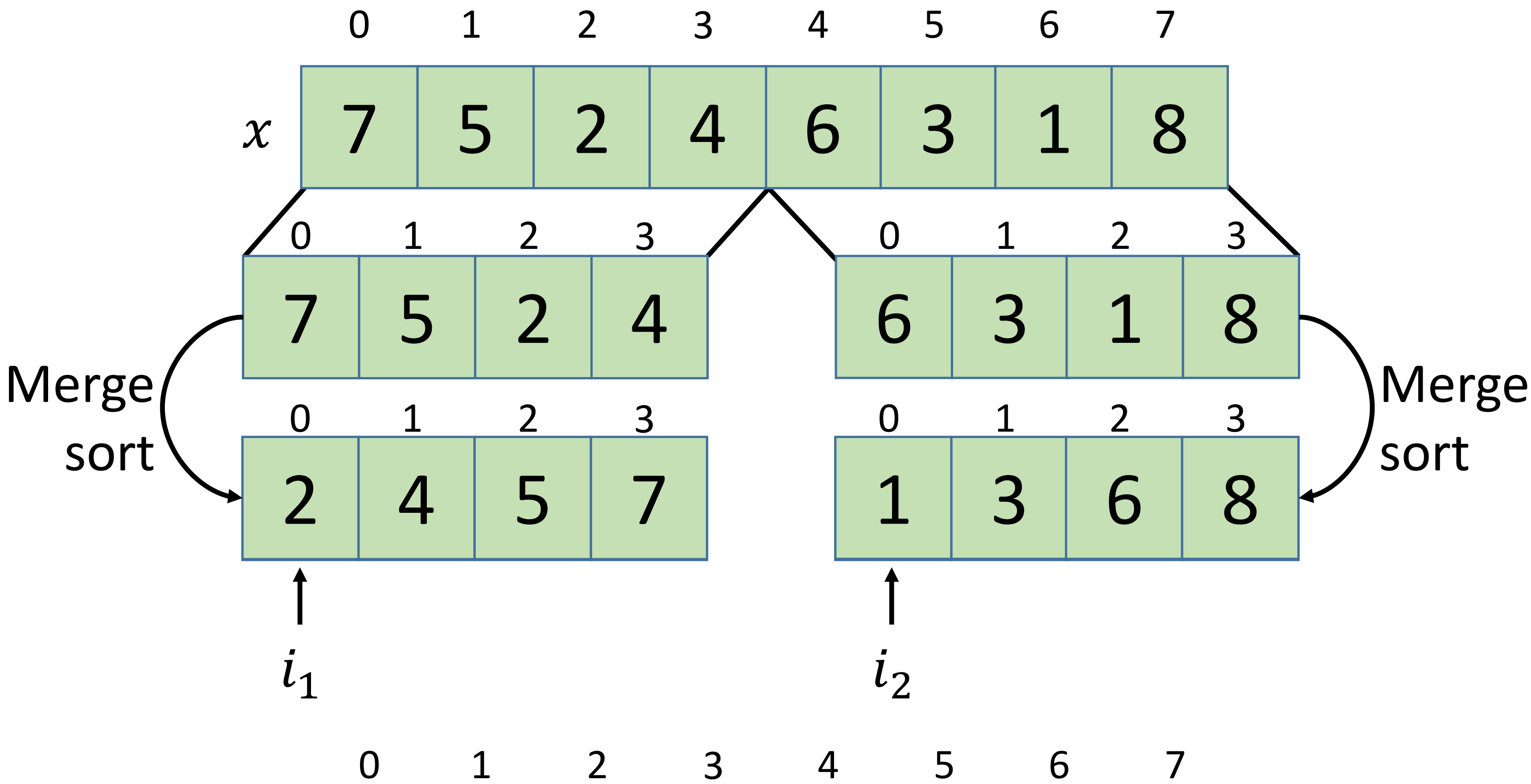
$sorted(a \oplus b)$

0	1	2	3	4	5	6	7

$$O(N)$$

Merge sort

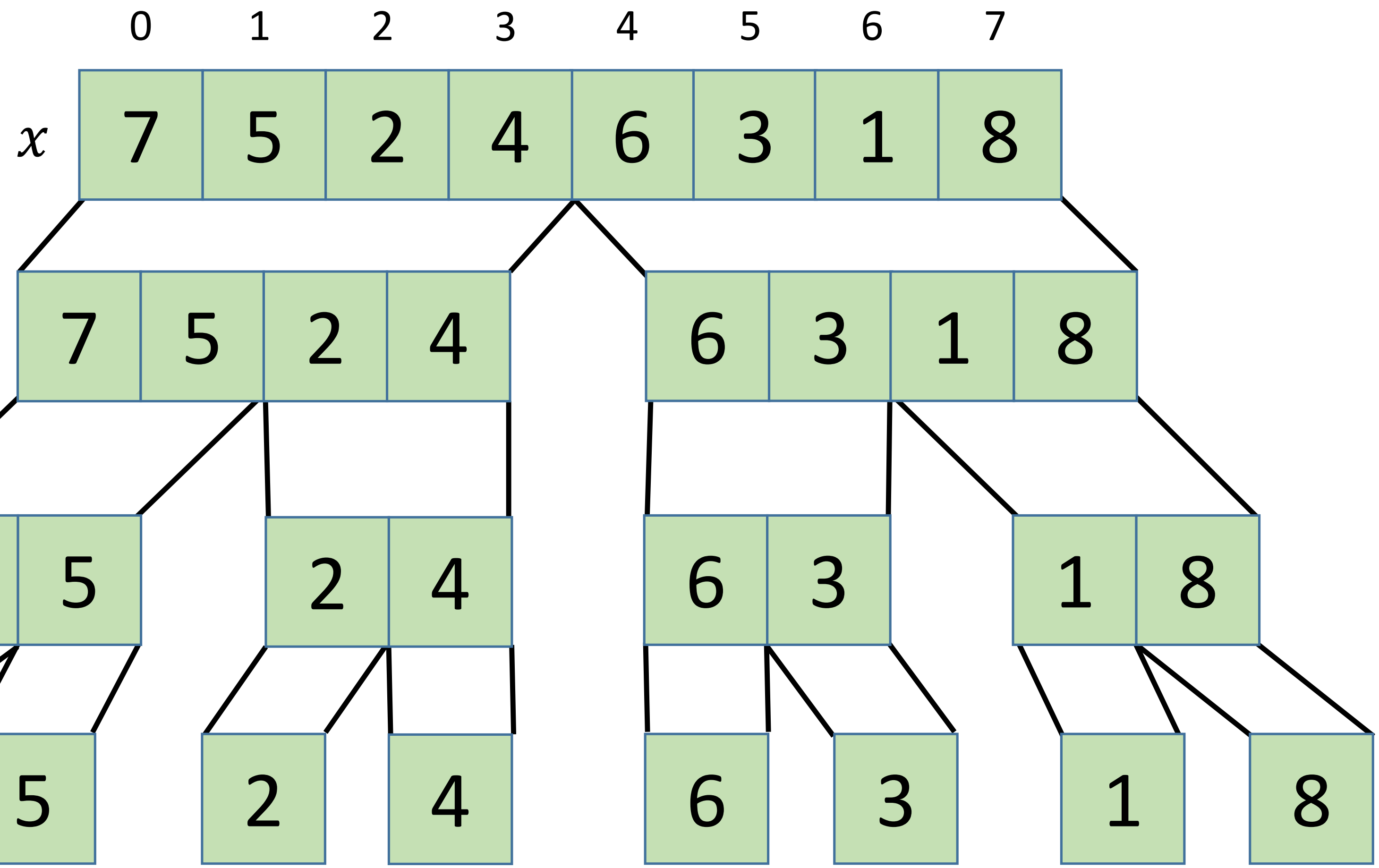
Divide



$sorted(x)$

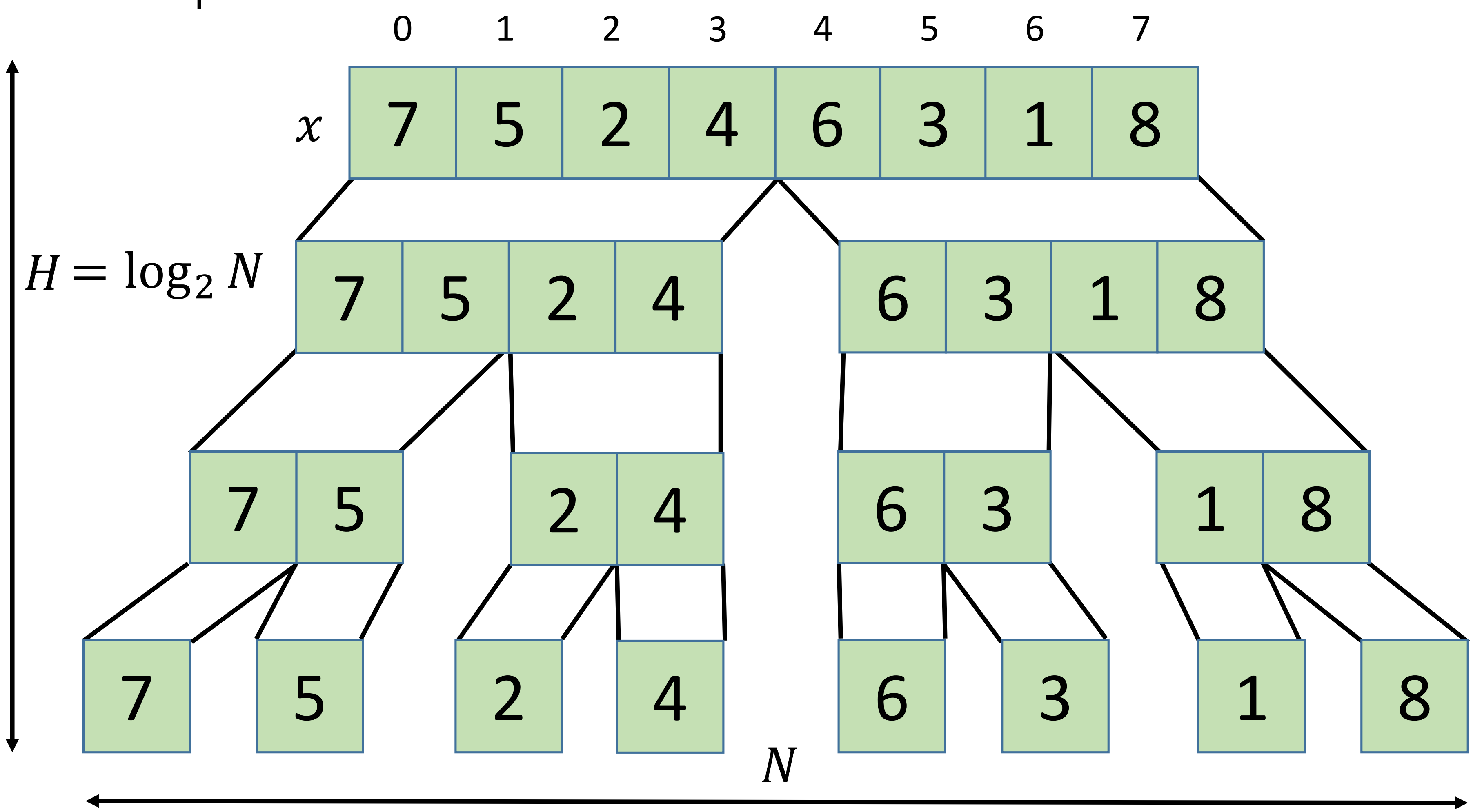
Merge sort

Conquer



Merge sort

Conquer



Complexity: $O(N \log N)$

Merge sort

Implementation

```
def merge(x, l, m, r):
    tmp = []
    i1 = l
    i2 = m
    while i1 < m or i2 < r:
        if (i2 >= r) or ((i1 < m) and
                         (x[i1] < x[i2])):
            tmp.append(x[i1])
            i1 += 1
        else:
            tmp.append(x[i2])
            i2 += 1
    x[l:r] = tmp
```

Merge sort

Implementation

```python
def merge_sort(x, l=0, r=None):
    if r is None:
        r = len(x)
    if r - l > 1:
        m = (l + r) // 2
        merge_sort(x, l, m)
        merge_sort(x, m, r)
        merge(x, l, m, r)
```

Linearithmic sorting algorithms $O(N \log N)$

- Merge sort
- **Quick sort**

Divide and conquer paradigm

QSort

Idea

QSort also uses Divide and Conquer approach, but dividing method is different.

1. Select pivot element (any element from array)
2. Divide by 3 parts: elements < pivot, == pivot, > pivot
3. Recursively sort 1st and 3rd parts.

0	1	2	3	4	5	6	7
7	5	2	4	6	3	1	8

1	2	3	4	5	6	7	8

QSort

Partition

Let's denote division into 3 parts with indices i_l, i_r.

Array to be partitioned is in range $[l, r)$.

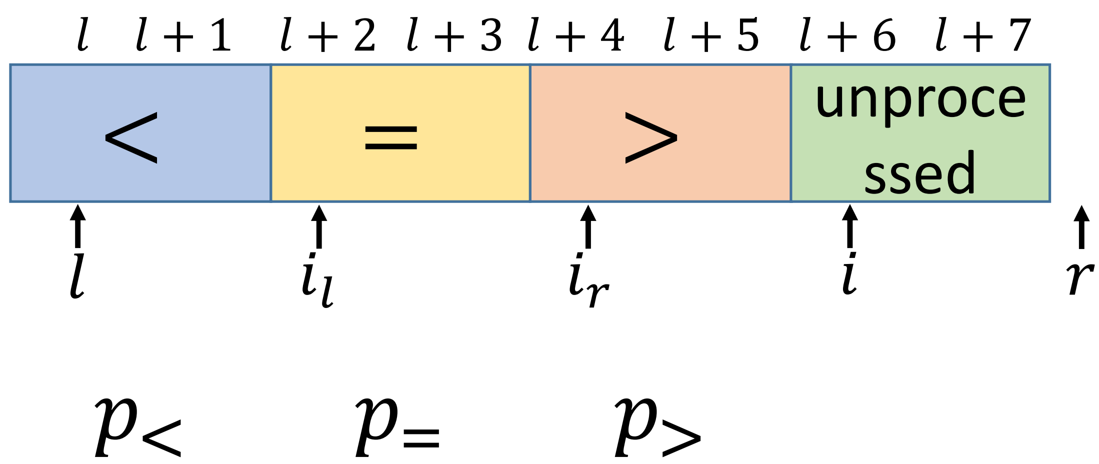
Division is correct in range $[l, i)$.

$x < pivot$: $[l, i_l)$

$x == pivot$: $[i_l, i_r)$

$x > pivot$: $[i_r, i)$

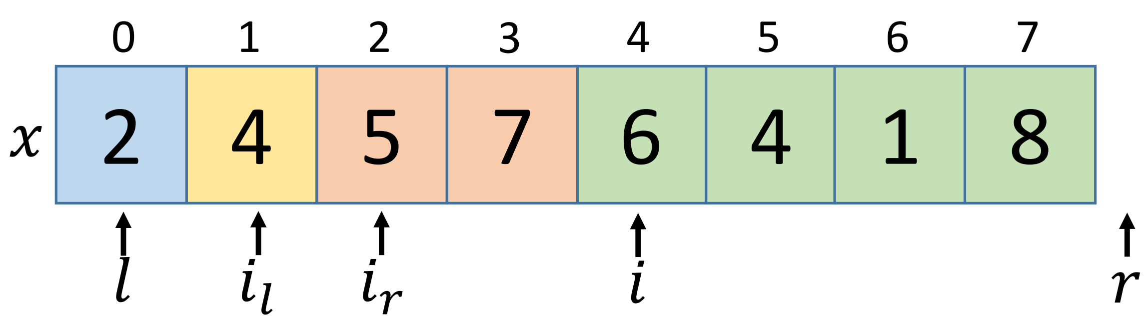
Unprocessed: $[i, r)$



QSort
Partition

Adding element to $p_>$.

1. Element already stands on it's place.

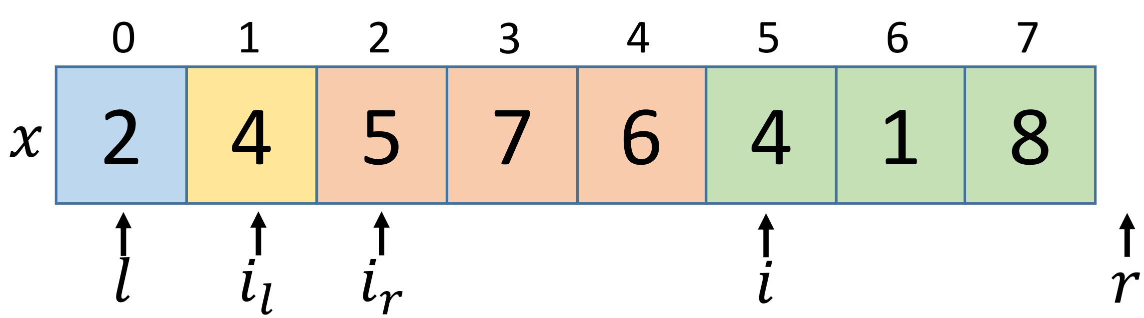


QSort

Partition

Adding element to $p_=$.

1. Swap $x[i_r]$ and $x[i]$.
2. Increase i_r.

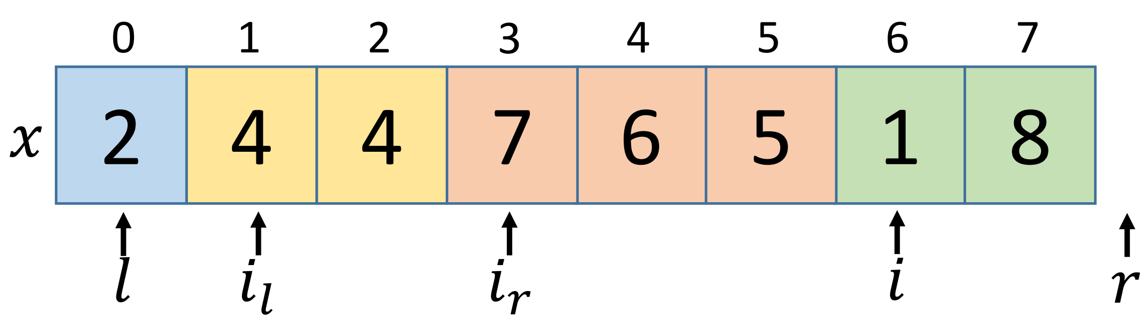


QSort

Partition

Adding element to $p_<$.

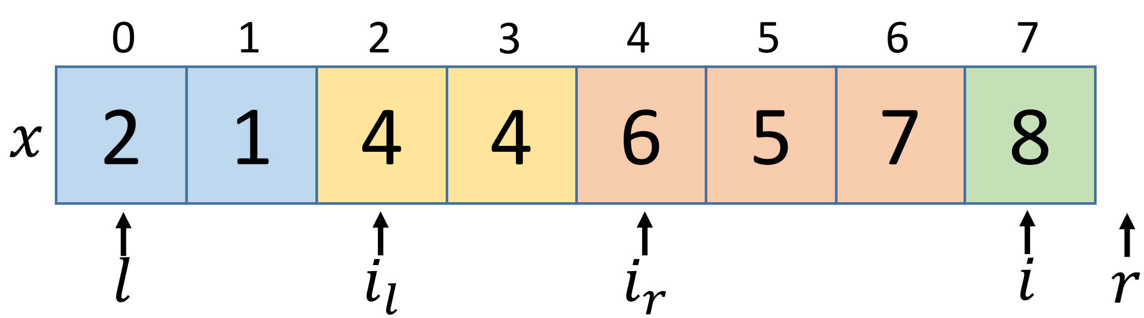
1. Swap $x[i_l]$ and $x[i]$.
2. If 2-nd part was not empty $(i_l < i_r)$, $x[i]$ is from $p_=$ and we need to return it (as on previous slide).
   Otherwise, $x[i]$ is from $p_>$, and it stands on it's place
3. Increase i_l and i_r.



QSort
Partition

Adding element to $p_<$.

1. Swap $x[i_l]$ and $x[i]$.
2. If 2-nd part was not empty $(i_l < i_r)$, $x[i]$ is from $p_=$ and we need to return it (as on previous slide).
   Otherwise, $x[i]$ is from $p_>$, and it stands on it's place
3. Increase i_l and i_r.



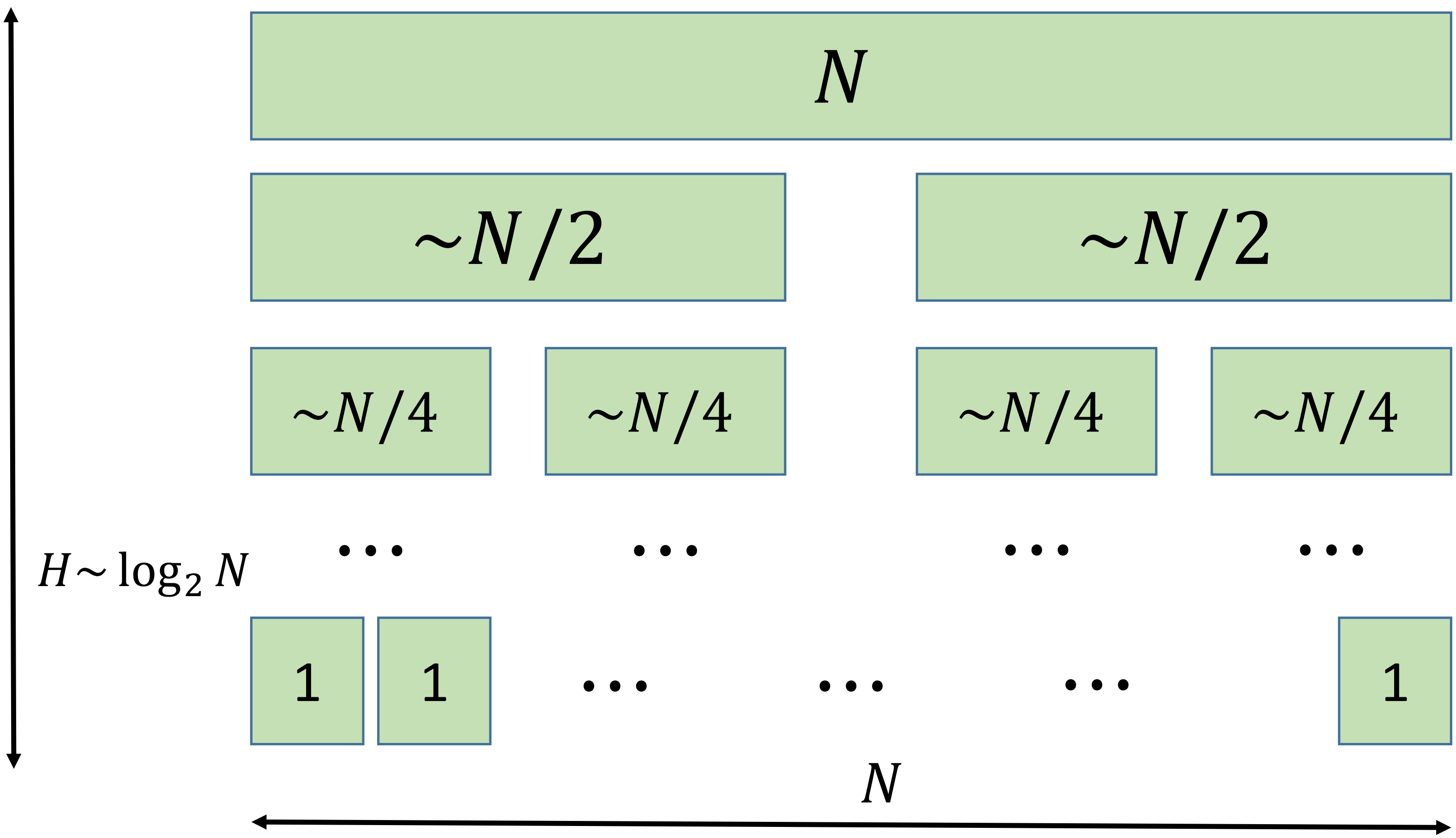
QSort

Implementation

```
def qsort(x, l=0, r=None):
  if r is None:
    r = len(x)
  if (r - l) > 1:
    pivot = x[(l + r) // 2]
    il, ir = partition(x, l, r, pivot)
    qsort(x, l, il)
    qsort(x, ir, r)
```

QSort

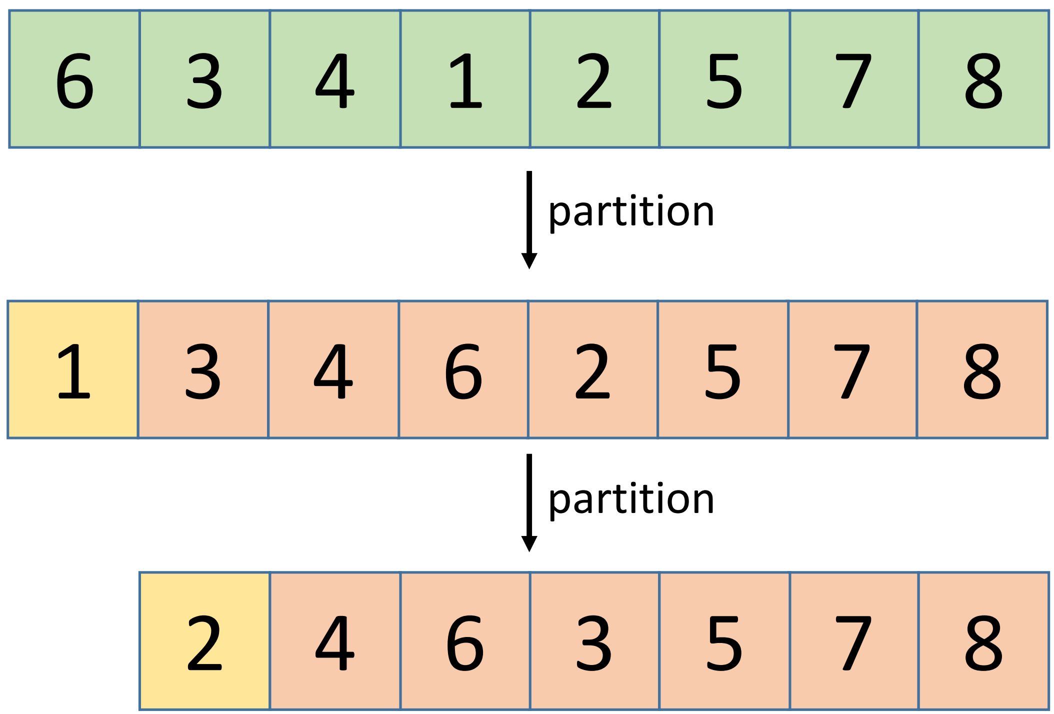
Complexity



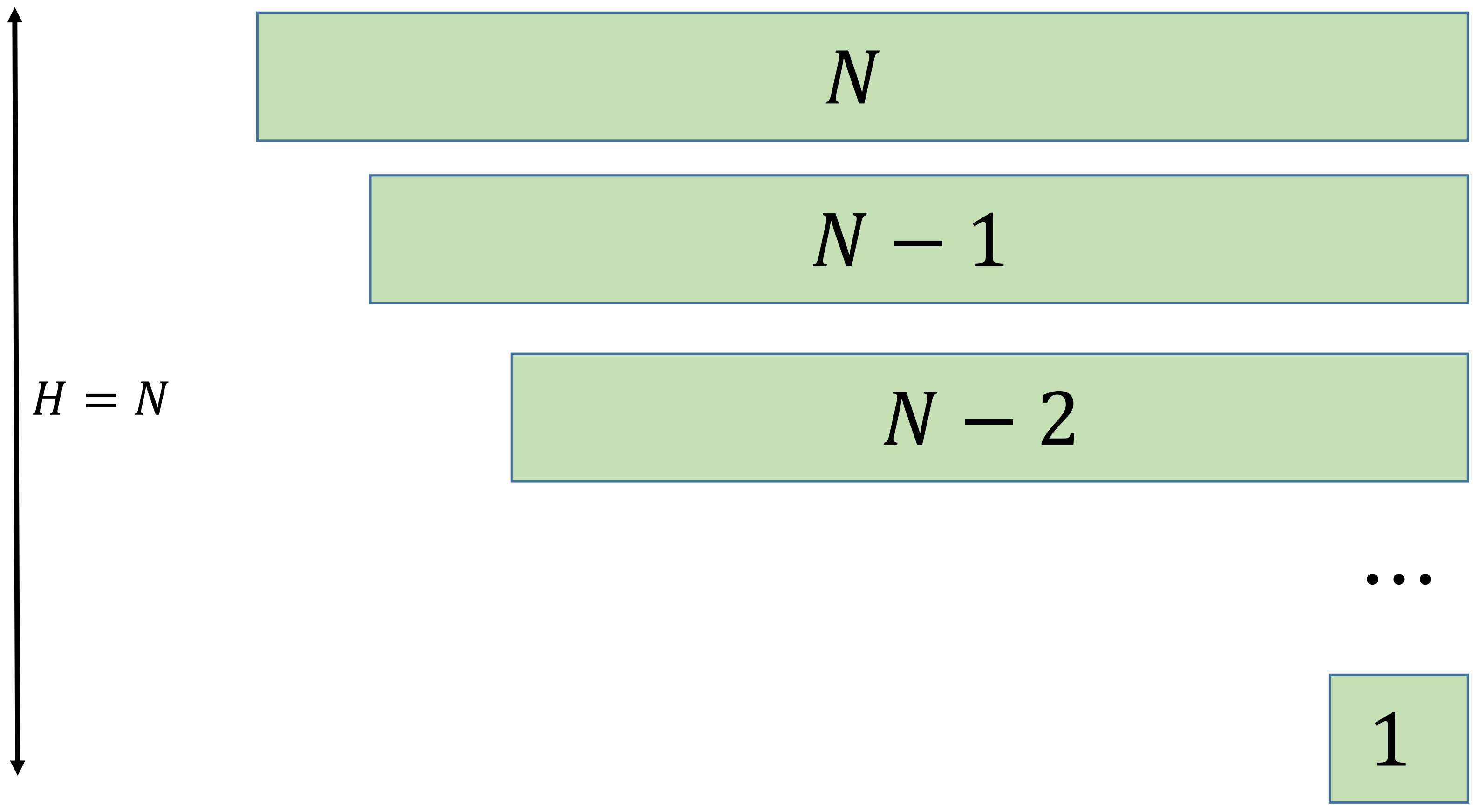
Complexity: $O(N \log N)$? (Actually, that's not correct proof. See the lecture.)

QSort

Complexity



QSort
Complexity



$$N + N - 1 + \ \dots + 1 = O(N^2)$$

QSort

Implementation

```python
import random
def qsort(x, l=0, r=None):
  if r is None:
    r = len(x)
  if (r - l) > 1:
    pivot = x[random.randint(l, r - 1)]
    il, ir = partition(x, l, r, pivot)
    qsort(x, l, il)
    qsort(x, ir, r)
```

Expectation of complexity: $O(N \log N)$

Greedy algorithm

Definition

Greedy algorithm builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

So on each step it chooses locally optimal solution.

Greedy algorithm
Example

Problem: Ali Baba 1

Ali-baba entered the cave with lot's of treasures. He can hold only N items in his hands. You are given list of all items in the cave with their costs. Help Ali Baba take out items with maximum total cost.

Solution:

Let's sort elements in non-increasing cost order and take top N elements.

Proof (informal):

Let's suppose, our solution A is not optimal. That means that exists a better solution B. A and B differs at least by 1 item, but if we replace any item in A with another, total sum will not increase, because our solution contains top-cost items. Contradiction.

Conclusion

$$O(N^2): N \leq 1000$$

$$O(N \log N): N \leq 100000$$

Python built-ins

```
import random
x = [random.randint(0, 100000) for i in range(100000)]
y = sorted(x)
x.sort()
```

Visualizers

- http://sorting.at/
- www.youtube.com/user/AlgoRythmics

Thank you for watching!