# ADS: lab session #1

Kamil Salakhiev

August 18, 2015

# Who am I?

**Kamil Salakhiev**
**Master student on Data Science program**
**Telegram nickname:** @kamil_sa
**E-Mail:** k.salakhiev@innopolis.ru

# Second TA

**Ahmed**
**PHd student in Machine Learning lab**
**Telegram nickname:** ???
**E-Mail:** ???

# Grading

- 4 Graded Tutorials (Lab Sessions): 40 %
- Midterm Exam: 20 %
- Final Exam: 40 %

# Graded Tutorial

In Graded tutorial I am going to give you task to evaluate your skills that was acquired during previous ungraded tutorials according to the next parameters:

- Completeness of your program
- Correctness of your program
- Codestyle
- Your activity during lab sessions

Questions?

# Algorithm

Example:

- **Task:** Sort sequence of numbers
- **Input:** Shuffled sequence of numbers
- **Output:** Sorted sequence

**Algorithm** – procedure that takes any possible allowed input, processes it and returns output according to given conditions.

# Algorithm Complexity

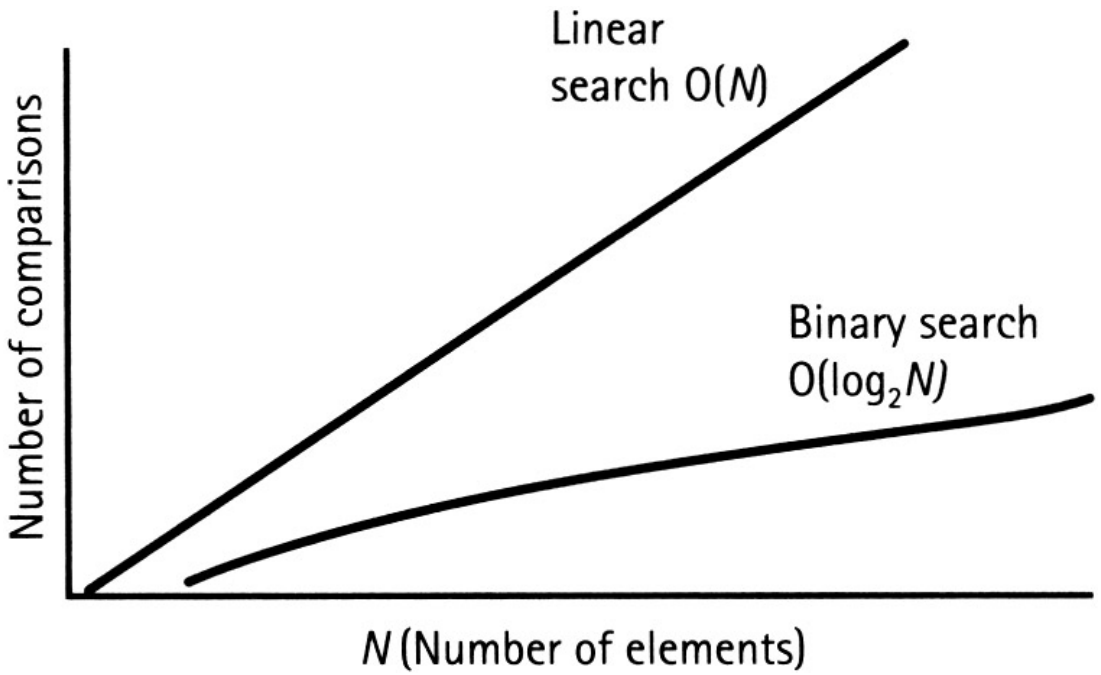
Model RAM:

- for performing any "atomic" operation(+,*,-,=,if,call) exactly one time step is required
- loops and procedures costs as couple of atomic operations, not as one
- each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Big O notation:
$f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* of $f(n)$. Thus there exist some constant $c$ such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough $n$ (i.e., $n \geq n_0$ for some constant $n_0$)

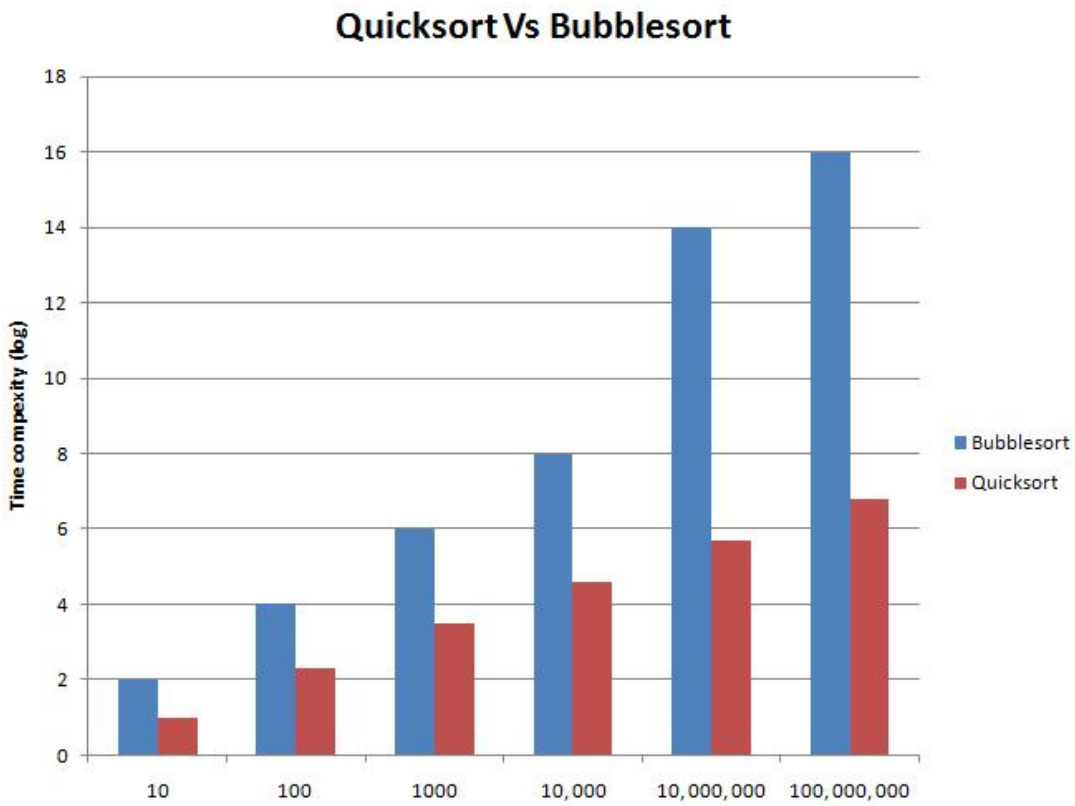
# Simple search algorithms

- Linear search
- Binary search



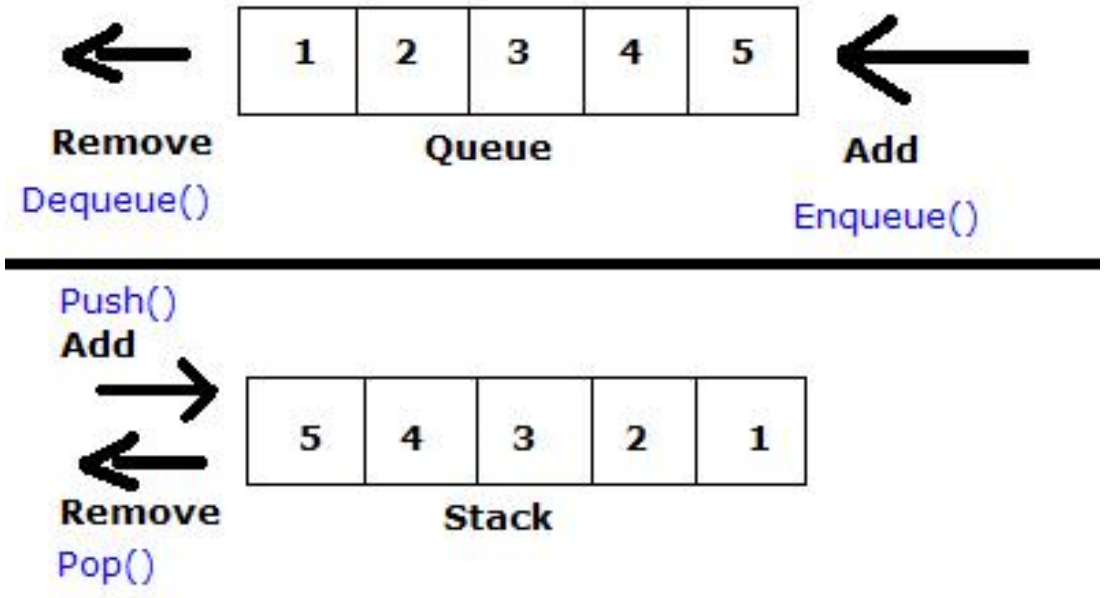
# Simple sorting algorithms

- Bubblesort
- Quicksort



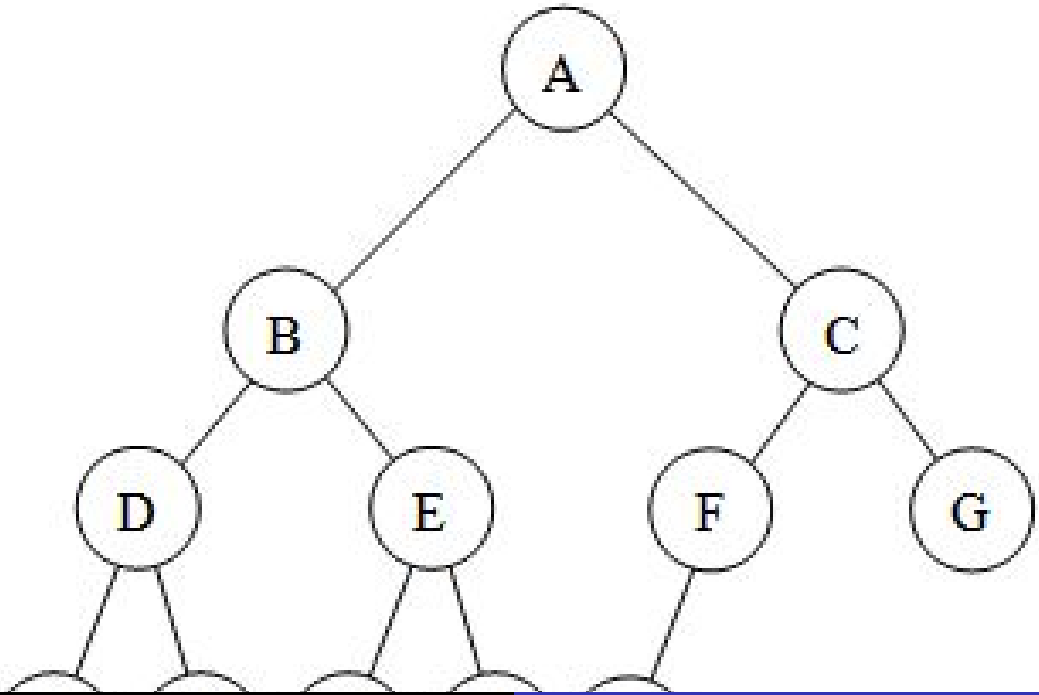
# Abstract data types

- List
- Stack
- Queue



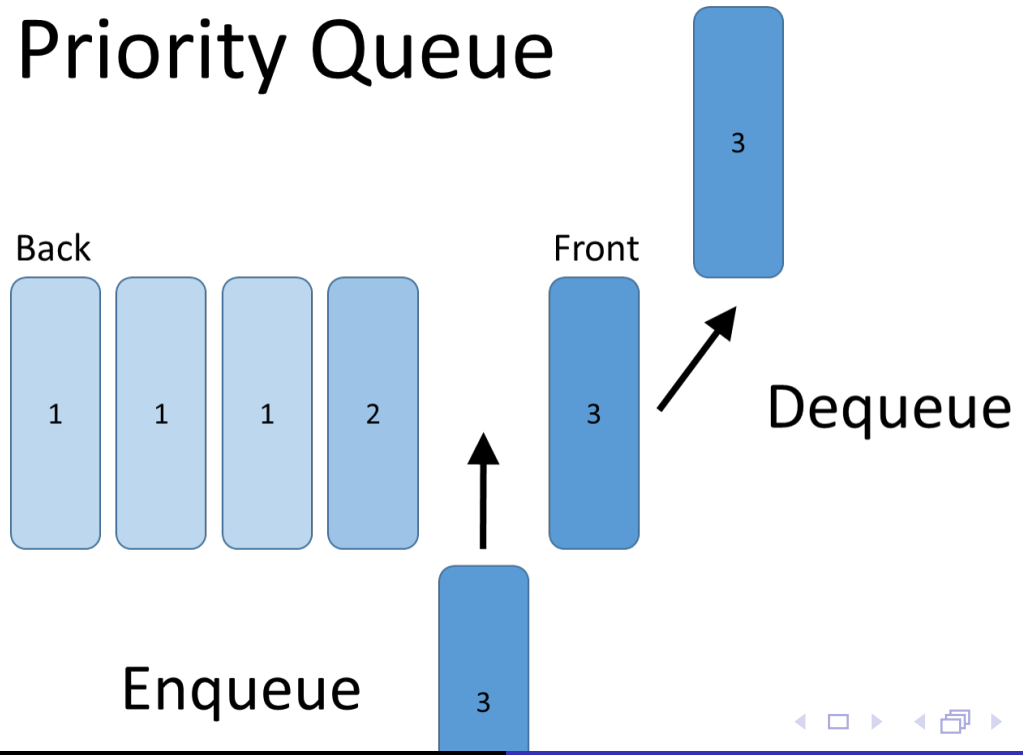
# Trees

- Binary trees
- Binary search trees
- Huffman coding
- Height-balanced trees(AVL, Red-Black Trees)



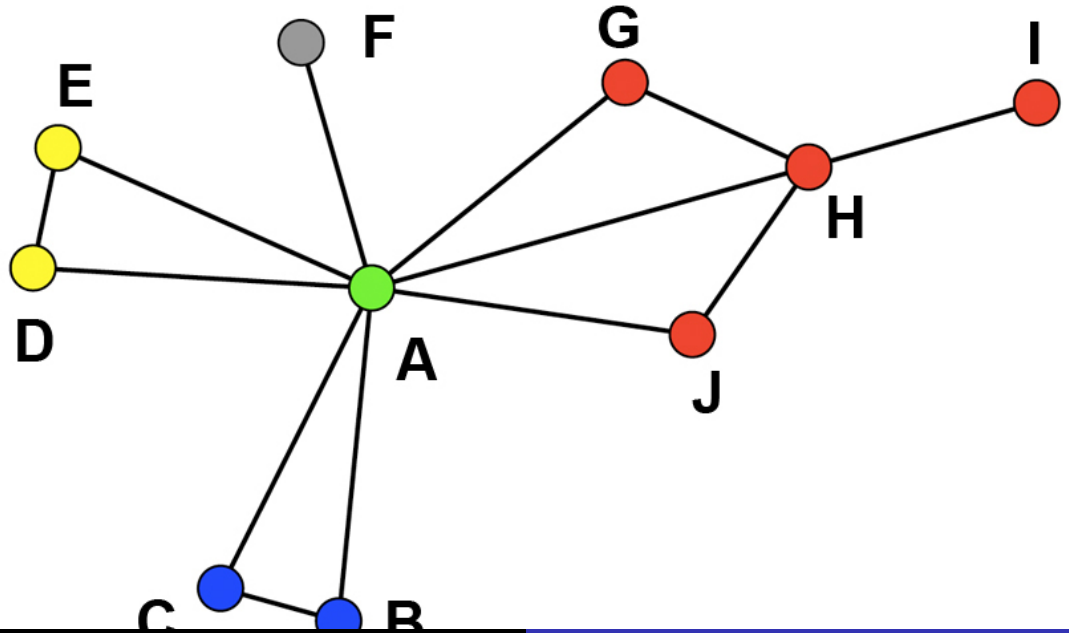
# Priority queues

- Priority queue data structure
- Binary heap
- Heapsort



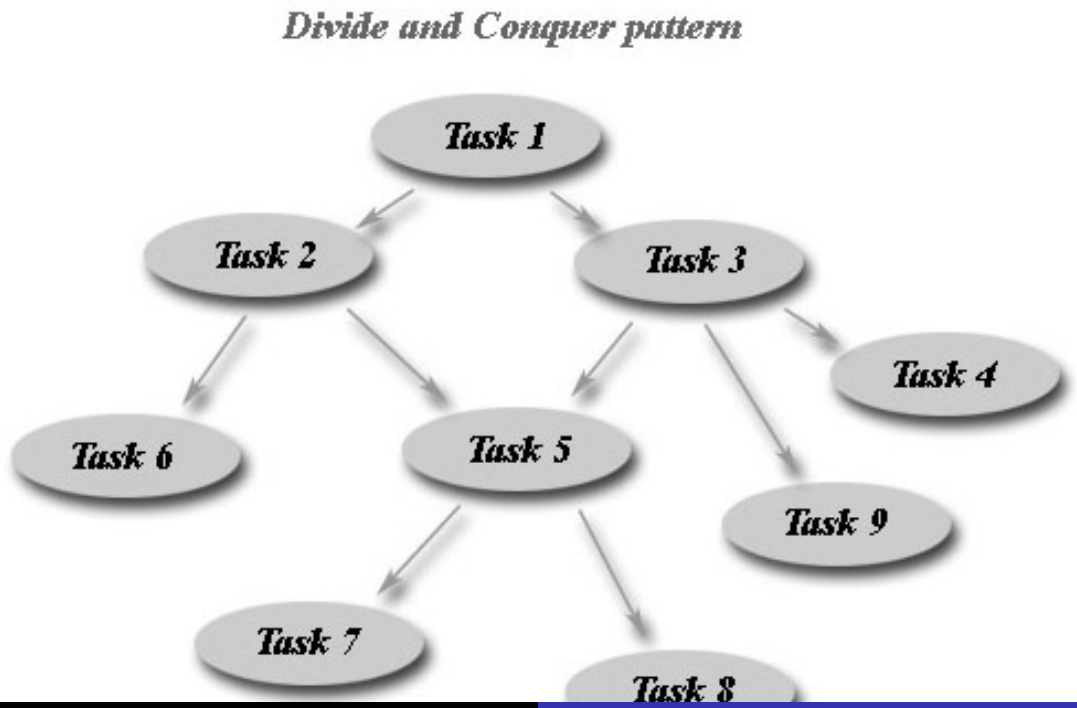
# Graphs

- Graph data structure
- Breadth-first search traversal
- Minimum spanning tree (e.g. Prims and Kruskals algorithms)
- Shortest-path algorithms (e.g. Dijkstras and Floyds algorithms)



# Algorithmic strategies

- Brute-Force
- Divide-and-conquer
- Greedy
- Combinatorial Search & Backtracking



# Data type

For the most commonly used **data types**, Java provides the following **primitive types**:

- **boolean** a boolean value: true or false
- **char** 16-bit Unicode character
- **byte** 8-bit signed twos complement integer
- **short** 16-bit signed twos complement integer
- **int** 32-bit signed twos complement integer
- **long** 64-bit signed twos complement integer
- **float** 32-bit floating-point number (IEEE 754-1985)
- **double** 64-bit floating-point number (IEEE 754-1985)

# Data type

You may create your own data types through defining classes:

- Consumer
- Bank
- Branch
- Stack
- List

# Abstract Data Type (ADT)

- A type in which the internal structure and the internal working of the objects of the type are unknown to users of the type!
- Users can only see the effects of the operations!

For example **Stack ADT** has the following operations:

- Push
- Pop

But we do not care how they are implemented.

# Datastructure

Physical representation of an ADT. Here the implementation matters for us.

- Arrays
- Linked Structures
- Stacks, Queues
- Trees
- Graphs

# Arrays: Pros and Cons

Pros:

1. *Constant-time access given the index* - Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index
2. *Space efficiency* - Arrays consist purely of data, so no space is wasted with links or other formatting information. Further, end-of-record information is not needed because arrays are built from fixed-size records

Cons:

1. *Restricted memory access* – it is impossible to get access to $(n+1)th$ element of array
2. We have to keep in the memory additional variable to know the tail of array

# ArrayList in Java

ArrayList is a built-in datastructure in Java which serves to be a wrapper for common arrays and simplify users interaction with them. ArrayList class has add, get, delete functions implementations.
So, how ArrayList implemented in Java?

# Initialization

You create ArrayList as follows:

```
public ArrayList(int initialCapacity) {
super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
this.elementData = new Object[initialCapacity];
}



public ArrayList() {
this(10);
}
```

# Adding new elements

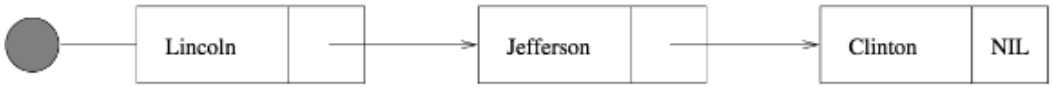
How does size increase when new element is included?

```
1       public boolean add(E e) {
2       ensureCapacity(size + 1);  // Increments modCount!!
3       elementData[size++] = e;
4       return true;
5       }
6
7
8
9       public void ensureCapacity(int minCapacity) {
10      modCount++;
11      int oldCapacity = elementData.length;
12      if (minCapacity > oldCapacity) {
13          Object oldData[] = elementData;
14          int newCapacity = (oldCapacity * 3)/2 + 1;
15              if (newCapacity < minCapacity)
16          newCapacity = minCapacity;
17              // minCapacity is usually close to size, so this is a win:
18              elementData = Arrays.copyOf(elementData, newCapacity);
19      }
20      }
```

# Operations in array

- Getting access to element
- Adding new element into certain place in array

# Linked Structures




```java
/**
 * Created by kamil on 17/08/15.
 */
public class ListNode {
    private int key;
    private ListNode next;

    public ListNode(){}
    public ListNode(int key, ListNode next) {
        this.key = key;
        this.next = next;
    }

    public int getKey() { return key; }

    public void setKey(int key) { this.key = key; }

    public ListNode getNext() { return next; }

    public void setNext(ListNode next) { this.next = next; }
}
```

# Pros and Cons

Pros:

1. Overflow on linked structures can never occur
2. Insertions and deletions are simpler than for adjacent (array) lists
3. With large records, moving pointers is easier and faster than moving the items themselves

Cons:

1. Linked structures require extra space for storing pointer fields.
2. Linked lists do not allow efficient random access to items
3. Arrays allow better memory locality and cache performance than random pointer jumping

# Comparison of arrays and linked structures

What structure is more suitable if the most frequently used operation is:

1. Insertion?
2. Deletion?
3. Getting access?

# ADT operations

The main ADT operations are:

- Insert
- Delete
- Getting access
- ???

# In Java

In Java there is an interface called List, that has similar methods:

- int size(): Returns the number of elements in the list
- boolean isEmpty(): Returns a boolean indicating whether the list is empty
- E get(i): Returns the element of the list having index i; an error condition occurs if i is not in range [0, size() - 1]
- E set(i,e): Replaces the element at index i with e,and returns the old element that was replaced; an error condition occurs if i is not in range [0, size( ) - 1]
- boolean add(i, e): Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range [0,size()]
- boolean remove(i): Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range [0, size() - 1]

# Task

Fill out the form that given below:

| Method | Return Value | List Contents |
|---|---|---|
| add(0, A) | – | (A) |
| add(0, B) | – | (B, A) |
| get(1) | | |
| set(2, C) | | |
| add(2, C) | | |
| add(4, D) | | |
| remove(1) | | |
| add(1, D) | | |
| add(1, E) | | |
| get(4) | | |
| add(4, F) | | |
| set(2, G) | | |
| get(2) | | |

# Algorithm

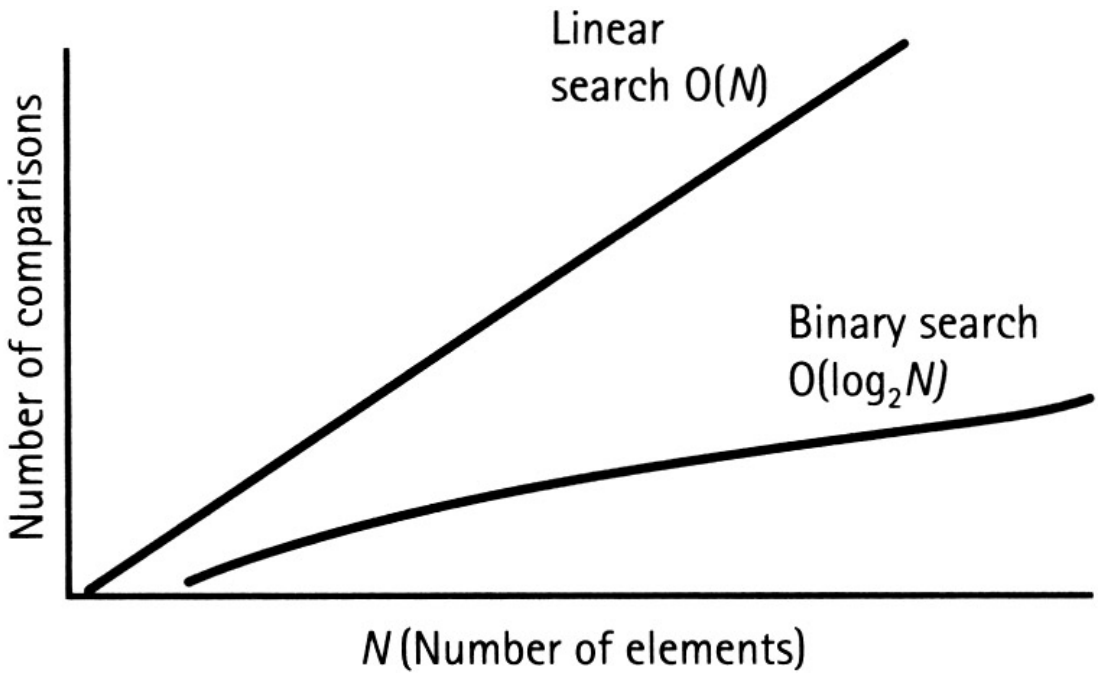
Example:

- **Task:** Sort sequence of numbers
- **Input:** Shuffled sequence of numbers
- **Output:** Sorted sequence

**Algorithm** – procedure that takes any possible allowed input, processes it and returns output according to given conditions.

# Importance of right algorithm choosing

- Linear search
- Binary search



# Tic-Tac-Toe game

Implement makeStep function for Tic-Tac-Toe game. In pseudo-code or in Java. Function has to satisfy the following conditions:

- As an input you have an array 3 by 3 representing the board, where each value is from the set {null, me, enemy}, and coordinates of the shoot
- As output you have to define the array that was derived after step performing
- Algorithm should process exceptional situations, when step performing is impossible

# Collection

Interfaces:

- List ADT
- Set ADT
- Map ADT
- ...

# Array based collection

Initialization:

```
private int arr[];
private int capacity;
private int currSize = 0;
public MyList(){
    capacity = 10;
    arr = new int[capacity];
}

public MyList(int capacity){
    this.capacity = capacity;
    arr = new int[capacity];
}
```

# Array based collection

Adding:

```
public void add(Integer integer) {
    arr[currSize++] = integer;
}
```

Removing:

```
public void remove(int e){
    int index = 0;
    for(int i = 0; i < currSize; i++){
        if(arr[i] == e){
            index = i;
            break;
        }
    }
    //shift array elements to the previous position
    for(int i = currSize -1; i > index; i--){
        arr[i-1] = arr[i];
    }
    currSize--;
}
```

# Iterator interface

```
public class MyList implements Iterator<Integer>{

    @Override
    public boolean hasNext() {
        if(nextCounter < currSize)
            return true;
        else{
            nextCounter = 0;
            return false;
        }
    }

    @Override
    public Integer next() {
        return arr[nextCounter++];
    }

    public int size() {
        return currSize;
    }

    public boolean isEmpty() {
        return currSize == 0;
```

# Iterator using

```
public static void main(String args[]){
    MyList list = new MyList(10);
    list.add(1);
    list.add(2);
    list.add(3);

    Iterator<Integer> it = list.getIterator();
    while (it.hasNext()){
        System.out.println(it.next());
    }

}
```

# Your own ADT

Implement your own ADT which is going to be:

- Based on array
- Iterable
- Contains insert, delete, get method
- Maintain resizing when it is required(as it demonstrated in ArrayList example)