

# Algorithms & Data Structures

## Introduction

Jacob Trier Frederiksen & Anders Kalhauge (assisting)



Spring 2019

## Introduction

- Your Instructors
- Course Overview

## Warming Up

- Basic Sorting
- Shuffling Cards: Yet Another Exercise

## Analysis of Algorithmic Complexity

- Sum of Array Elements
- Three-Sum Over an Array
- Computational Complexity
- $\mathcal{O}$  (order) notation

## Sorting

- Exercise - Yet Another Card Shuffle

## Introduction

- Your Instructors
- Course Overview

## Warming Up

- Basic Sorting
- Shuffling Cards: Yet Another Exercise

## Analysis of Algorithmic Complexity

- Sum of Array Elements
- Three-Sum Over an Array
- Computational Complexity
- $\mathcal{O}$  (order) notation

## Sorting

- Exercise - Yet Another Card Shuffle

# Jacob Trier Frederiksen

`jtf@cphbusiness.dk` (+4536154500, front desk)

- PhD in Computational Astrophysics
- 10 years experience in research, higher learning
- 3 years experience in business intelligence & machine learning
- Main interests
  - Computational modelling
  - Machine learning
  - Artificial Intelligence

# Anders Kalhauge

aka@cphbusiness.dk (21 72 44 11)

- 26 years experience as IT consultant in the private sector
- 16 years teaching computer science for students and private companies
- Main interests
  - Programming and programming languages
  - Development of large scale systems
  - Software architecture

We have decided on four major topics:

**Introduction** **2 weeks** Introduction to algorithms and complexity.  
Basic sorting algorithms.

**Data Structures** **4 weeks** Basic data structures and searching algorithms. Heaps, heap sorting, and priority queues.

**Graphs** **4 weeks** Graph types, directed and weighted graphs. Implementation of graph data structures. Algorithms for graphs including searching for spanning trees and shortest path.

**Application** **5 weeks** Application of algorithms including scheduling, text mining, and big data.

At the end of the course the student will:

- Have experience with a representative selection of algorithms and data structures
- Know what is inside the abstract data types of the programming framework
- Know how to compare algorithm time complexity, and associated data structures
- Know how to use algorithms on large data sets

At the end of the course the students can:

- Select and utilize relevant algorithms in own applications
- Calculate time and space complexity (big  $\mathcal{O}$ )
- Have basic knowledge of data compression
- Handle big and faulty data

The exam is oral but as part of the exam a written test is performed in the end of the course. For the oral part, the student will prepare a (app. ten minutes) presentation of the solution of one of the major assignments. Further discussions will be based on the presentation, but can include all aspects of the curriculum.

In order to be approved for the exam:

- ❑ Major hand-in assignments must be completed (no less than 80% as minimum)!
- ❑ Participation in the written test (no study points, but mandatory for exam eligibility)
- ❑ Participation in giving a lesson in small groups.
- ❑ At least 80% of the study points must be obtained



- Hand in of five major assignments (20 per assignment): 100
- Giving a lesson on an assigned or chosen/approved topic in small groups: 10

## Introduction

Your Instructors  
Course Overview

## Warming Up

Basic Sorting  
Shuffling Cards: Yet Another Exercise

## Analysis of Algorithmic Complexity

Sum of Array Elements  
Three-Sum Over an Array  
Computational Complexity  
 $\mathcal{O}$  (order) notation

## Sorting

Exercise - Yet Another Card Shuffle

Here follows two basic sorting algorithms that we will be using in just a few minutes...

We will find the recipe for another basic sorting algorithms, and test it manually.

1. Team up in groups of three.
2. Get your deck of cards/suit.
3. Shuffle **really** well (TRUE random is very hard, by the way)!
4. Google the Tree-Sort algorithm and adopt it for card games.
5. Get ready, set, go sort ...
6. ...leading to a manual test of simple application of a Tree-based algorithm. How many moves, how much time?

We will find the recipe for another basic sorting algorithms, and test it manually.

1. Team up in groups of three.
2. Get your deck of cards/suit.
3. Shuffle **really** well (TRUE random is very hard, by the way)!
4. Google the Tree-Sort algorithm and adopt it for card games.
5. Get ready, set, go sort . . .
6. . . .leading to a manual test of simple application of a Tree-based algorithm. How many moves, how much time?

**Q: what did you find out? Discuss in class...**

## Introduction

- Your Instructors
- Course Overview

## Warming Up

- Basic Sorting
- Shuffling Cards: Yet Another Exercise

## Analysis of Algorithmic Complexity

- Sum of Array Elements
- Three-Sum Over an Array
- Computational Complexity
- $\mathcal{O}$  (order) notation

## Sorting

- Exercise - Yet Another Card Shuffle

*// We will measure the following in time units only,  
// neglecting any movement of data and excess storage  
// needs.*

```
int sumOfarray(int[] list) {  
    int total = 0;                // ~ 1  
    for (int i = 0; i < list.length; i++) // ~ 2*N  
        total = total + list[i];    // ~ 2*N  
    return total;                 // ~ 1  
}
```

Total units of time consumption:

$$\mathcal{O}(1 + 2N + 2N + 1) \approx \mathcal{O}(1 + N) \approx \mathcal{O}(N)$$

```
// We will measure the following in time units only,  
// neglecting any movement of data and excess storage  
// needs.
```

```
int count(int[] a) {  
    int n = a.length;           // ~ 1  
    int count = 0;              // ~ 1  
    for (int i = 0; i < n; i++)  // ~ 2*N  
        for (int j = i + 1; j < n; j++) // ~ 2*N  
            for (int k = j + 1; k < n; k++) // ~ 2*N  
                if (a[i] + a[j] + a[k] == 0) count++ // ~ 1  
    return count;  
}
```

Total units of time consumption:

$$\mathcal{O}(1 + 1 + 2N + 2N^2 + 2N^3 + 1) \approx \mathcal{O}(1 + N + N^2 + N^3) \approx \mathcal{O}(N^3)$$



When accessing an array of length  $N$  while testing for the  $K$  Zero-sum, what is the complexity of that operation in algorithmic terms?

**Task:** We need to test whether the sum of  $M$  values, chosen from  $N$  values, sums to zero (fx). ***Btw; should that be with / without replacement – does it matter(?) – discuss in plenum.***

**Discussion:** go to the web and find a formula which expresses this particular complexity.

When accessing an array of length  $N$  while testing for the  $K$  Zero-sum, what is the complexity of that operation in algorithmic terms?

**Task:** We need to test whether the sum of  $M$  values, chosen from  $N$  values, sums to zero (fx). ***Btw; should that be with / without replacement – does it matter(?) – discuss in plenum.***

**Discussion:** go to the web and find a formula which expresses this particular complexity.

The binomial coefficient holds that answer, namely:

$$\frac{N!}{(N-K)!K!} = \binom{N}{K},$$

where  $N! \equiv N(N-1)(N-2)(N-3)\dots(N-N+1)$ .

**Q: So what about the Two-sum – what is the complexity?**

To measure the efficiency of algorithms, We calculate their time and space complexity.

The notation for that is  $\mathcal{O}$ , or order of growth:

$$\mathcal{O}(N^q),$$

where  $q$  is a constant,  $q \in \mathbb{R}$ .

$\mathcal{O}$  is a measure of complexity, not of actual running time or space consumption.

In other words,  $\mathcal{O}$  is a measure of the cost of the algorithm, and has no bearing on the size or speed of the computer.

$\mathcal{O}$  gives a picture of what happens if we scale the problem by running the algorithm on larger number (normally  $N$ ) of data.

Therefore the running time (a constant  $c$ ) of the individual steps are not relevant:

$$\mathcal{O}(c N^q) \approx \mathcal{O}(N^q)$$

The algorithm uses the same time or space no matter how much data is involved.

- Pushing an element on a stack
- Returning the size of an array
- Calculating a single step

$$\mathcal{O}(1)$$

Remember:

$$\mathcal{O}(cx) \approx \mathcal{O}(x) \rightarrow \mathcal{O}(7) \approx \mathcal{O}(1000000) \propto \mathcal{O}(1)$$

The algorithm separates the problem in two (or more) equally sized problems and solve those.

- Binary search
- Binary tree insertions and deletions

$$\mathcal{O}(\log N)$$

No matter what log you choose:

$$\mathcal{O}(\log_{10} N) \sim \mathcal{O}(\log_2 N) \sim \mathcal{O}(\log N)$$

**Q: prove, or dig out (i.e. from the web) a proof of this.**

The algorithm uses one step per data element. Remember: each computational step can take as much time as needed, as long as that step requires a constant time, i.e.  $10^{-9}$ s (a GHz machine).

- Search for maximum in unordered data
- Copying an array
- Bucket Sort
- Sorting using a Trie (project!)

$$\mathcal{O}(N)$$

Again:

$$\mathcal{O}(7N) \approx \mathcal{O}(6,022 \times 10^{23} N) \sim \mathcal{O}(N)$$

The algorithm separates the problem in two (or more) equally sized problems and solves these. The algorithm involves doing something with each element in the data.

- Tree Sort
- Merge Sort and Heap Sort
- Quick Sort (best case)

$$\mathcal{O}(N \log N)$$

And of course

$$\mathcal{O}(1000N \log_{10} N) \sim \mathcal{O}(N \log N)$$



The algorithm works on (almost) any pair of elements in the data.  
Normally done by a double nested loop.

- Selection Sort
- Insertion Sort
- Bubble sort

$$\mathcal{O}(N^2)$$

Consider:

$$\begin{aligned} N + (N - 1) + (N - 2) + \dots + 1 + 0 &= \\ (N + 0) + (N - 1 + 1) + \dots + \left(\frac{N}{2} + \frac{N}{2}\right) &= \frac{N(N - 1)}{2} \\ \mathcal{O}\left(\frac{N(N - 1)}{2}\right) &= \mathcal{O}(N^2) \end{aligned}$$

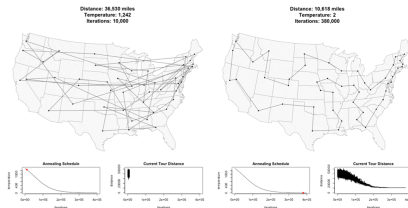
The algorithm works on (almost) any triplet of elements in the data. Normally done by a triple nested loop.

- Three-sum Algorithm

$$(N^3)$$

The algorithm checks does an exhaustive search.

- Matrix chain multiplication (brute force, but for AI CNNs, case is much better ...)
- Travelling salesman ( $(N \cdot 2^N) \sim 2^N$ )



$$\mathcal{O}(2^N)$$



## Introduction

- Your Instructors
- Course Overview

## Warming Up

- Basic Sorting
- Shuffling Cards: Yet Another Exercise

## Analysis of Algorithmic Complexity

- Sum of Array Elements
- Three-Sum Over an Array
- Computational Complexity
- $\mathcal{O}$  (order) notation

## Sorting

- Exercise - Yet Another Card Shuffle

We will find the recipe for another basic sorting algorithms, and test it manually.

1. Team up in groups of three.
2. Get your deck of cards/suit.
3. Shuffle **really** well (TRUE random is very hard, by the way)!
4. Google the Tree-Sort algorithm and adopt it for card games.
5. Get ready, set, go sort ...
6. ...leading to a manual test of simple application of a Tree-based algorithm. How many moves, how much time?

We will find the recipe for another basic sorting algorithms, and test it manually.

1. Team up in groups of three.
2. Get your deck of cards/suit.
3. Shuffle **really** well (TRUE random is very hard, by the way)!
4. Google the Tree-Sort algorithm and adopt it for card games.
5. Get ready, set, go sort . . .
6. . . .leading to a manual test of simple application of a Tree-based algorithm. How many moves, how much time?

**Q: what did you find out? Discuss in class...**

Create a class called `SortingAlgorithms`. This class should have an array of integer as a datafield and array size. The constructor should be used to create an array of given size.

`SortingAlgorithms` class should have three methods.

- ❑ One method for filling the array with random integers which you can call from the constructor.
- ❑ One method for implementing insertion sort,
- ❑ One method for selection sort.

In the main method you create three objects with array size of 100, 1000, and 10000. Identify the elapsed times using `Stopwatch` <sup>1</sup> class when selection and insertion methods are called.

---

<sup>1</sup>see page 175 in Algorithms book