# Algorithmns and Datastructures
## Runtime Complexity, Associative Arrays

Albert-Ludwigs-Universität Freiburg

UNI
FREIBURG

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science
Algorithmns and Datastructures, March 2016

# Structure

# Feedback from the exercises

- Some people enjoyed the absence of programming tasks, some not :)

- Difficulty was anything between underwhelmed and overwhelmed, but mostly manageable

# Feedback from the lecture

- Q: Can I use a calculator in the exam?

- A: No sorry. On the other hand, you can bring

    - handwritings

    - printed-out course material

    - books

# Runtime Complexity

- The runtime does not entirely depend on the size of the problem, but also on the type of input
- This results in:
  - Best runtime:
    Lowest possible runtime complexity for an input of size *n*
  - Worst runtime:
    Highest possible runtime complexity for an input of size *n*
  - Average / Expected runtime:
    The average of all runtime complexities for an input of size *n*

- Input: Field *a* with *n* elements
  $a[i] \in \mathbb{N}$, $0 \leq a[i] \leq n$, $0 \leq i < n$
- Output: Field *a* with *n* elements $a[0] \neq 1$

```
if  a[0] == 0:
    a[0] = 2
else:
    for i in range(0, n):
        a[i] = 2
```

$$\left.\begin{array}{c} \mathscr{O}(1) \\ \hline \mathscr{O}(1) \end{array}\right\} \quad \mathscr{O}(1)$$

$$\left.\begin{array}{c} \mathscr{O}(n) \\ \hline \mathscr{O}(1) \end{array}\right\} \quad \begin{array}{c} \mathscr{O}(n) \cdot \mathscr{O}(1) \\ = \mathscr{O}(n) \end{array}$$

$$\Bigg\} \quad \mathscr{O}(?)$$

- Best runtime: $\mathscr{O}(1) + \mathscr{O}(1) = \mathscr{O}(1)$
- Worst runtime: $\mathscr{O}(1) + \mathscr{O}(n) = \mathscr{O}(n)$

# Runtime Complexity
Example 1 - Average Runtime

- The average runtime is determined by the average runtime for all input instances of size $n$
- Every element of $a$ can have $n$ different values
  $\Rightarrow n \cdot \ldots \cdot n = n^n$ different input instances of size $n$
  - a[i] == 1 in $n^{n-1}$ instances
  - a[i] != 1 in $n^n - n^{n-1} = n^{n-1} \cdot (n-1)$ instances
- Sum of all runtime complexities:

$$\underbrace{n^{n-1} \cdot \mathscr{O}(1)}_{\text{a[i] == 1}} + \underbrace{n^{n-1} \cdot (n-1) \cdot \mathscr{O}(n)}_{\text{a[i] != 1}}$$

- Average runtime:

$$\frac{n^{n-1} + n^{n-1} \cdot (n-1) \cdot n}{n^n} = \frac{1}{n} + n - 1 \in \mathscr{O}(n)$$

# Runtime Complexity
## Example 2 - Binary Addition

- Input: $n$ digit dual number $a$
- Output: $n$ digit dual number $a + 1$
- Runtime of the algorithm is determined by the number of bits getting changed (steps)
    1. "0" $\rightarrow$ "1"
    2. "1" $\rightarrow$ "0"
- Best runtime: 1 step = $\mathscr{O}(1)$
- Worst runtime: $n$ steps = $\mathscr{O}(n)$

Table: Binary addition

| Digits ($n$) | Input | Output | Steps |
|---|---|---|---|
| 10 | 1011100100 | 1011100101 | 1 |
| 4 | 1011 | 1100 | 3 |
| 8 | 11111111 | 00000000 | 8 |

# Runtime Complexity
## Example 2 - Average Steps

Table: Binary addition with $n = 1$

| Input | Output | Steps |
|-------|--------|-------|
| 0 | 1 | 1 |
| 1 | 0 | 1 |

Table: Binary addition with $n = 2$

| Input | Output | Steps |
|-------|--------|-------|
| 00 | 01 | 1 |
| 01 | 10 | 2 |
| 10 | 11 | 1 |
| 11 | 00 | 2 |

$$\overline{\text{steps}} = \frac{1 + 1}{2} = 1$$

$$= 2 - \frac{1}{1} = 2 - \frac{1}{2^{n-1}}$$

$$\overline{\text{steps}} = \frac{1 + 2 + 1 + 2}{4} = \frac{3}{2}$$

$$= 2 - \frac{1}{2} = 2 - \frac{1}{2^{n-1}}$$

# Runtime Complexity
Example 2 - Average Steps

Table: Binary addition with $n = 3$

| Input | Output | Steps |
|-------|--------|-------|
| 000   | 001    | 1     |
| 001   | 010    | 2     |
| 010   | 011    | 1     |
| 011   | 100    | 3     |
| 100   | 101    | 1     |
| 101   | 110    | 2     |
| 110   | 111    | 1     |
| 111   | 000    | 3     |

$$\overline{\text{steps}} = \frac{1+2+1+3+1+2+1+3}{8} = \frac{7}{4}$$

$$= 2 - \frac{1}{4} = 2 - \frac{1}{2^{n-1}}$$

$\Rightarrow$ Average runtime:

$$2 - \frac{1}{2^{n-1}} \in \mathscr{O}(1)$$

# Runtime Complexity
Example 2 - Average Steps

Table: Case analysis for instances of size $n$

| Input | Output | Instances | Steps |
|-------|--------|-----------|-------|
| $\_\_\_\cdots\_\_\_0$ | $\_\_\_\cdots\_\_\_1$ | $2^{n-1}$ | 1 |
| $\_\_\_\cdots\_\_01$ | $\_\_\_\cdots\_\_10$ | $2^{n-2}$ | 2 |
| $\_\_\_\cdots\_011$ | $\_\_\_\cdots\_100$ | $2^{n-3}$ | 3 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $\_01\ldots1111$ | $\_10\ldots0000$ | $2^1$ | n-1 |
| $011\ldots1111$ | $100\ldots0000$ | $2^0$ | n |
| $111\ldots1111$ | $000\ldots0000$ | 1 | n |

Average steps:

$$\frac{1\cdot 2^{n-1} + 2\cdot 2^{n-2} + \cdots + (n-1)\cdot 2^1 + n\cdot 2^0 + n\cdot 1}{2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0 + 1} = \frac{\sum_{i=1}^{n}\left(i\cdot 2^{n-i}\right) + n}{\sum_{i=0}^{n-1} 2^i + 1}$$

# Runtime Complexity
Example 2 - Average Steps

- Denominator:

$$\sum_{i=0}^{n-1} 2^i + 1 \overset{\substack{\text{geometric} \\ \text{series}}}{=} 2^n - 1 + 1 = 2^n$$

- Counter:

$$\sum_{i=1}^{n} \left( i \cdot 2^{n-i} \right) + n \overset{a=2a-a}{=} 2 \sum_{i=1}^{n} \left( i \cdot 2^{n-i} \right) - \sum_{i=1}^{n} \left( i \cdot 2^{n-i} \right) + n$$

$$= 1 \cdot 2^n + 2 \cdot 2^{n-1} + 3 \cdot n^{n-2} + \cdots + (n-1) \cdot 2^2 + n \cdot 2^1$$
$$\quad - 1 \cdot 2^{n-1} - 2 \cdot 2^{n-2} - \cdots - (n-2) \cdot 2^2 - (n-1) \cdot 2^1 - n \cdot 2^0 + n$$

$$= \underbrace{2^n + 2^{n-1} + \cdots + 2^1 + 2^0}_{2^{n+1}-1} - 1 = 2^{n+1} - 2$$

Average steps:

$$\overline{steps} = \frac{\sum\limits_{i=1}^{n} \left( i \cdot 2^{n-i} \right) + n}{\sum\limits_{i=0}^{n-1} 2^i + 1} = \frac{2^{n+1} - 2}{2^n} = 2 - \frac{1}{2^{n-1}}$$

$$\lim_{n \to \infty} \left( 2 - \frac{1}{2^{n-1}} \right) = 2 \in \mathscr{O}(1)$$

**Normal array:**

$$A = [0 \Rightarrow A_0, \, 1 \Rightarrow A_1, \, 2 \Rightarrow A_2, \, 3 \Rightarrow A_3, \, \ldots]$$

- Searching elements by index
- Lookup of element with index "3":
  $\Rightarrow A[3] = A_3$

**Associative array:**

$$A = \left[ \begin{array}{l} \text{"}Europa\text{"} \Rightarrow A_0, \text{ "}Amerika\text{"} \Rightarrow A_1, \\ \text{"}Asien\text{"} \Rightarrow A_2, \text{ "}Afrika\text{"} \Rightarrow A_3, \\ \cdots \end{array} \right]$$

- Searching elements by key
- The keys can be of any type with unique elements
- Lookup of element with key "*Afrika*":
    $\Rightarrow A[\text{"}Afrika\text{"}] = A_3$

# Associative Arrays
Practical Example

Table: Country data query from http://geonames.org

| ISO | ISO3 | Country | Continent | ... |
|-----|------|---------|-----------|-----|
| AD | AND | Andorra | EU | ... |
| AE | ARE | United Arab Emirates | AS | ... |
| AF | AFG | Afghanistan | AS | ... |
| AG | ATG | Antigua and Barbuda | NA | ... |
| AI | AIA | Anguilla | NA | ... |
| AL | ALB | Albania | EU | ... |
| AM | ARM | Armenia | AS | ... |
| AO | AGO | Angola | AF | ... |
| AQ | ATA | Antarctica | AN | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋱ |

# Associative Arrays
Practical Example
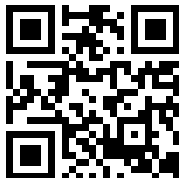
**Task:** How many countries belong to each continent?

- We are interested in column 2 (country) and 3 (continent)
- There are two typical ways to solve this:
  - Using Sorting
  - Using an associative array

**Idea using sorting:**

- We sort the columns 2 and 3 by continent, so that all countries with the same continent are grouped in one block
- We count the size of the blocks

   **Disadvantage:**
   - Runtime of $\Theta(n \log n)$
   - We have to iterate the array twice (sort and then count)

   **Advantage:**
   - Easy to implement (even with simple linux / unix commands)

## Associative Arrays
Practical Example - Sorting With Linux / Unix Commands

**Input:**

- The data is saved as tab seperated text (countryInfo.txt)
- Comments begin with a hash sign #

**Commands:**

- **grep**: Selects a specific set of lines (filter by ...)
  `grep -v 'ˆ#' countryInfo.txt`
  - -v: not
  - ˆ#: # at start of line
- **cut**: Selects specific columns of each line (tab separated)
  `cut -f5,9`
  - -f5,9: columns 5 and 9 (columns 2, 3 of Table 6)

**Commands:**

- **sort**: Sorts lines by a key
  sort -t ' ' -k2,2
    -t ' ': Separator: Tab (Insert with CTRL-V TAB)
    -k2,2: Key from column 2 to 2
- **uniq**: Finds or counts unique keys
  uniq -c
    -c: count occurences of keys
- **head**: Displays a provided number of lines
  head -n30
    -n30: Displays the first 30 lines
- **less**: Displays the file page wise

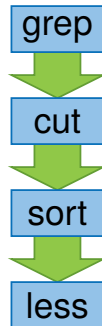**Sort countries by continent:**

```
grep -v 'ˆ#' countryInfo.txt | cut -f5,9 \
    | sort -t ' ' -k2,2 | less
```

Table: Resulting data

| | |
|---|---|
| Algeria | AF |
| Angola | AF |
| Benin | AF |
| Botswana | AF |
| Burkina Faso | AF |
| Burundi | AF |
| Cameroon | AF |
| Cape Verde | AF |
| ⋮ | ⋮ |

Figure: Data pipeline

**Count countries per continent:**

```
grep -v '^#' countryInfo.txt | cut -f9 \
   | sort | uniq -c | sort -nr
```

Table: Resulting data

| | |
|---|---|
| 58 | AF |
| 54 | EU |
| 52 | AS |
| 42 | NA |
| 27 | OC |
| 14 | SA |
| 5 | AN |

Figure: Data pipeline

# Associative Arrays
Practical Example - Associative Array

**Idea using associative arrays:**

- Take the continent as key
- Use a counter (occurences) or a list with all countries found belonging to this continent as value

**Advantage:**

- Runtime $\mathcal{O}(n)$, implied we can find an element in $\mathcal{O}(1)$ like in an normal array

# Associative Arrays
Python

**Python:**

```python
# creates a new map (called dictionary)
countries = {"DE" : "Deutschland", \
    "EN" : "England"}

# check if element exists
if "EN" in countries:
    print("Found %s!" % countries["EN"])

# map key "DE" to value 0
countries["DE"] = "Germany"

# delete key "DE"
del countries["DE"]
```

# Associative Arrays
Efficiency

**Efficiency:**

- Depends on implementation
- Two typical implementations:
    - **Hashing:** Calculates a checksum of the key and uses as key of a normal array
      search: $\mathcal{O}(1)\ldots\mathcal{O}(n)$
      insert: $\mathcal{O}(1)\ldots\mathcal{O}(n)$
      delete: $\mathcal{O}(1)\ldots\mathcal{O}(n)$
    - **(Binary-)Tree:** Creates a sorted (binary) tree
      search: $\mathcal{O}(\log n)\ldots\mathcal{O}(n)$
      insert: $\mathcal{O}(\log n)\ldots\mathcal{O}(n)$
      delete: $\mathcal{O}(\log n)\ldots\mathcal{O}(n)$

# Associative Arrays
Efficiency

Table: Map implementions of programming languages

|           | Hashing              | (Binary-)Tree       |
|-----------|----------------------|---------------------|
| Python    | all dictionaries     |                     |
| Java      | `java.util.HashMap`  | `java.util.TreeMap` |
| C++11/14  | `std::unordered_map` | `std::map`          |
| C++98     | `__gnu_cxx::hash_map`| `std::map`          |

# Further Literature

- **General**

  [CRL01] Thomas H. Cormen, Ronald L. Rivest, and
  Charles E. Leiserson.
  **Introduction to Algorithms**.
  MIT Press, Cambridge, Mass, 2001.

  [MS08] Kurt Mehlhorn and Peter Sanders.
  Algorithms and data structures, 2008.
  https://people.mpi-inf.mpg.de/~mehlhorn/
  ftp/Mehlhorn-Sanders-Toolbox.pdf.

# Further Literature

- **Map - Implementations / API**

  [Java] Java - HashMap
  https://docs.oracle.com/javase/7/docs/api/
  java/util/HashMap.html

  [Javb] Java - TreeMap
  https://docs.oracle.com/javase/7/docs/api/
  java/util/TreeMap.html

  [Pyt] Python - Dictionaries (Hash table)
  https://docs.python.org/3/tutorial/
  datastructures.html#dictionaries

# Further Literature

**Map - Implementations / API**

[Cppa] C++ - hash_map
http://www.sgi.com/tech/stl/hash_map.html

[Cppb] C++ - map
http://www.sgi.com/tech/stl/Map.html