

Algorithms and Datastructures

Runtime Complexity, Associative Arrays

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science
Algorithms and Datastructures, November 2017

Runtime Complexity

Associative Arrays

- Introduction

- Practical Example

 - Sorting

 - Associative Array

Runtime Complexity

Associative Arrays

Introduction

Practical Example

Sorting

Associative Array

- 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 $a[0] \neq 1$

| | |
|--|---|
| <pre>if a[0] == 0: a[0] = 2 else: for i in range(0, n): a[i] = 2</pre> | $\left. \begin{array}{l} \frac{\mathcal{O}(1)}{\mathcal{O}(1)} \\ \frac{\mathcal{O}(n)}{\mathcal{O}(1)} \end{array} \right\} \left. \begin{array}{l} \mathcal{O}(1) \\ \mathcal{O}(n) \cdot \mathcal{O}(1) \\ = \mathcal{O}(n) \end{array} \right\} \mathcal{O}(?)$ |
|--|---|

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

- 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
 $\Rightarrow n \cdot \dots \cdot n = n^n$ 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:

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

- **Average runtime:**

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

- 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 = $\mathcal{O}(1)$
- **Worst runtime:** n steps = $\mathcal{O}(n)$

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 |

$$\begin{aligned}\overline{\text{steps}} &= \frac{1+1}{2} = 1 \\ &= 2 - \frac{1}{1} = 2 - \frac{1}{2^{n-1}}\end{aligned}$$

Table: Binary addition with $n = 2$

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

$$\begin{aligned}\overline{\text{steps}} &= \frac{1+2+1+2}{4} = \frac{3}{2} \\ &= 2 - \frac{1}{2} = 2 - \frac{1}{2^{n-1}}\end{aligned}$$

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 |

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

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

⇒ Average runtime:

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

Table: Case analysis for instances of size n

| Input | Output | Instances | Steps |
|--------------|--------------|-----------|----------|
| ___...___0 | ___...___1 | 2^{n-1} | 1 |
| ___...___01 | ___...___10 | 2^{n-2} | 2 |
| ___...___011 | ___...___100 | 2^{n-3} | 3 |
| \vdots | \vdots | \vdots | \vdots |
| _01...1111 | _10...0000 | 2^1 | $n-1$ |
| 011...1111 | 100...0000 | 2^0 | n |
| 111...1111 | 000...0000 | 1 | n |

Average steps:

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

■ Denominator:

$$\sum_{i=0}^{n-1} 2^i + 1 \quad \begin{array}{c} \text{geometric} \\ \text{series} \end{array} = 2^n - 1 + 1 = 2^n$$

■ Counter:

$$\begin{aligned} & \sum_{i=1}^n \left(i \cdot 2^{n-i} \right) + n \stackrel{a=2^{a-a}}{=} 2 \sum_{i=1}^n \left(i \cdot 2^{n-i} \right) - \sum_{i=1}^n \left(i \cdot 2^{n-i} \right) + n \\ &= 1 \cdot 2^n + 2 \cdot 2^{n-1} + 3 \cdot 2^{n-2} + \dots + (n-1) \cdot 2^2 + n \cdot 2^1 \\ & \quad - 1 \cdot 2^{n-1} - 2 \cdot 2^{n-2} - \dots - (n-2) \cdot 2^2 - (n-1) \cdot 2^1 - n \cdot 2^0 + n \\ &= \underbrace{2^n + 2^{n-1} + \dots + 2^1 + 2^0}_{2^{n+1} - 1} - 1 = 2^{n+1} - 2 \end{aligned}$$

Average steps:

$$\overline{\text{steps}} = \frac{\sum_{i=1}^n (i \cdot 2^{n-i}) + n}{\sum_{i=0}^{n-1} 2^i + 1} = \frac{2^{n+1} - 2}{2^n} = 2 - \frac{1}{2^{n-1}}$$

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

Runtime Complexity

Associative Arrays

- Introduction

- Practical Example

 - Sorting

 - Associative Array

Normal array:

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

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

Associative array:

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

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

Runtime Complexity

Associative Arrays

Introduction

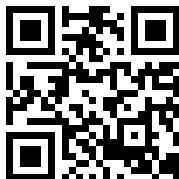
Practical Example

Sorting

Associative Array

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 | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |



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

Runtime Complexity

Associative Arrays

Introduction

Practical Example

Sorting

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)

Input:

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

Commands:

- **grep**: Selects a specific set of lines (filter by ...)
`grep -v '^#' countryInfo.txt`
 - v: not
 - ^#: # 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 occurrences 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 '^#' 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



Runtime Complexity

Associative Arrays

Introduction

Practical Example

Sorting

Associative Array

Idea using associative arrays:

- Take the continent as **key**
- Use a counter (occurrences) or a list with all countries associated with this continent as **value**

Advantage:

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

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"]
```

Efficiency:

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

Table: Map implementations of programming languages

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

■ 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>.

■ Map - Implementations / API

[Java] [Java - HashMap](#)

`https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html`

[Java] [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`

■ 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>