Algorithms and Data Structures

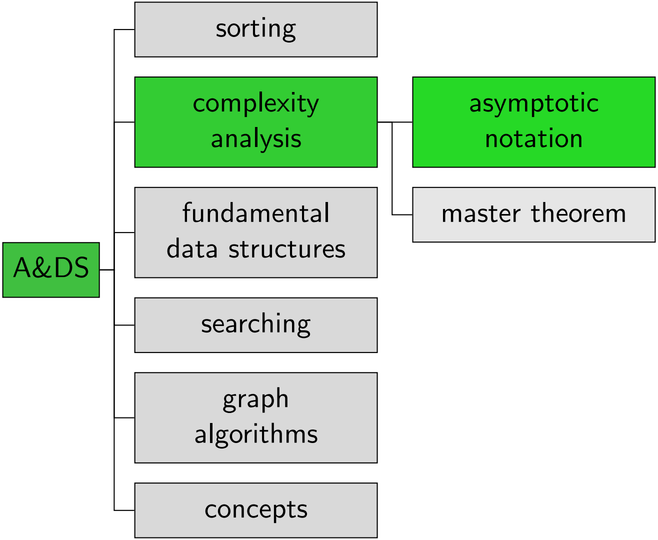
A9. Runtime Analysis: Application

Gabriele Röger and Patrick Schnider

University of Basel

March 5, 2025

Content of the Course



Recap

Symbols

- "f grows asymptotically as fast as g"

$$\Theta(g) = \{f \mid \exists c > 0\ \exists c' > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \\ c \cdot g(n) \leq f(n) \leq c' \cdot g(n)\}$$

Symbols

- "f grows asymptotically as fast as g"

$$\Theta(g) = \{f \mid \exists c > 0\ \exists c' > 0\ \exists n_0 > 0\ \forall n \geq n_0 : c \cdot g(n) \leq f(n) \leq c' \cdot g(n)\}$$

- "f grows no faster than g"

$$O(g) = \{f \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

Symbols

- "f grows asymptotically as fast as g"

$$\Theta(g) = \{f \mid \exists c > 0\ \exists c' > 0\ \exists n_0 > 0\ \forall n \geq n_0 : c \cdot g(n) \leq f(n) \leq c' \cdot g(n)\}$$

- "f grows no faster than g"

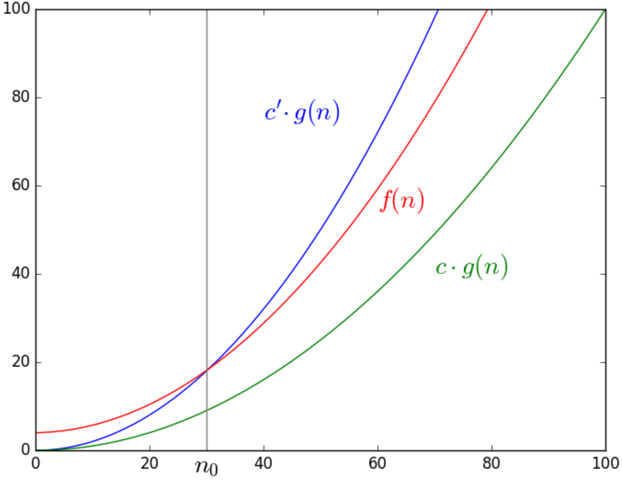
$$O(g) = \{f \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

- "f grows no slower than g"

$$\Omega(g) = \{f \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : c \cdot g(n) \leq f(n)\}$$

Symbol Theta: Illustration

$$f \in \Theta(g)$$



Some Relevant Classes of Functions

In increasing order (except for the general n^k):

g	growth
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	linearithmic
n^2	quadratic
n^3	cubic
n^k	polynomial (constant k)
2^n	exponential

Connections

It holds that:

- $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^k) \subset O(2^n)$ (for $k \geq 2$)

Connections

It holds that:

- $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^k) \subset O(2^n)$ (for $k \geq 2$)
- $O(n^{k_1}) \subset O(n^{k_2})$ for $k_1 < k_2$
  e.g. $O(n^2) \subset O(n^3)$

Calculation Rules

- Product
  $f_1 \in O(g_1)$ and $f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$

Calculation Rules

- Product
  $f_1 \in O(g_1)$ and $f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$
- Sum
  $f_1 \in O(g_1)$ and $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$

Calculation Rules

- Product
  $f_1 \in O(g_1)$ and $f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$
- Sum
  $f_1 \in O(g_1)$ and $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$
- Multiplication with a constant
  $k > 0$ and $f \in O(g) \Rightarrow kf \in O(g)$
  $k > 0 \Rightarrow O(kg) = O(g)$

Application

Quick O-Analysis for Common Code Patterns I

- Constant-time operation:

`var = 4`	$O(1)$

Quick O-Analysis for Common Code Patterns I

- Constant-time operation:

`var = 4`	$O(1)$

- Sequence of constant-time operations:

`var1 = 4`	$O(1)$	$O(123 \cdot 1) = O(1)$
`var2 = 4`	$O(1)$	
`...`	$\cdots$	
`var123 = 4`	$O(1)$	

Quick O-Analysis for Common Code Patterns II

- Loop:

`for i in range(n):`	$O(n)$	$O(n \cdot 1) = O(n)$
`  res += i * m`	$O(1)$	

Quick O-Analysis for Common Code Patterns II

- Loop:

`for i in range(n):`	$O(n)$	$O(n \cdot 1) = O(n)$
`  res += i * m`	$O(1)$	

`for i in range(n):`	$O(n)$	$O(n)$	$O(n^2)$
`  for j in range(i):`	$O(n)$	$O(n)$	
`    res += i * (m - j)`	$O(1)$		

i depends on n.

Quick O-Analysis for Common Code Patterns III

- if-then-else

Code			
`if var < bound:`	$O(1)$	$O(1)$	$O(1 + \max\{1, n\}) = O(n)$
`  res += var`	$O(1)$	$O(1)$	
`else:`			
`  for i in range(n):`	$O(n)$	$O(n \cdot 1) = O(n)$	
`    res += i * n`	$O(1)$		

Quick O-Analysis for Common Code Patterns III

- if-then-else

`if var < bound:`	$O(1)$	$O(1)$	$O(1 + \max\{1, n\})$ $= O(n)$
`  res += var`	$O(1)$	$O(1)$	
`else:`			
`  for i in range(n):`	$O(n)$	$O(n \cdot 1)$ $= O(n)$	
`    res += i * n`	$O(1)$		

Attention: Can lead to unnecessarily loose bound if the expensive case only occurs with small n (bound by a constant).

Example: Worst Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

Example: Worst Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Worst case: `break` never happens.

Example: Worst Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Worst case: `break` never happens.
- $O(1 + n \cdot n \cdot 1) = O(n^2)$

Example: Worst Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Worst case: `break` never happens.
- $O(1 + n \cdot n \cdot 1) = O(n^2)$
- Over-estimated?
  No, each of the two loops has $\Omega(n)$ iterations.

Example: Best Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

Example: Best Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Best case: `break` always immediately with $j = i$

Example: Best Case for Insertion Sort

```
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Best case: `break` always immediately with $j = i$
- $O(1 + n \cdot 1 \cdot 1) = O(n)$

Example: Best Case for Insertion Sort

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Best case: `break` always immediately with $j = i$
- $O(1 + n \cdot 1 \cdot 1) = O(n)$
- Over-estimated?
  No, the outer loop has $\Omega(n)$ iterations.

Exam Question from 2019

Consider the following code fragment.
Specify the asymptotic running time (depending on $n \in \mathbb{N}$) in Θ notation and justify your answer (1-2 sentences).

```
int result = 0;
if (n > 23) {
    return result;
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        result += j;
    }
}
return result;
```

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!
- Example: for several years, GTA online took several minutes to load.

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!
- Example: for several years, GTA online took several minutes to load.
  - Several minutes for parsing 10 megabyte of JSON data!

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!
- Example: for several years, GTA online took several minutes to load.
  - Several minutes for parsing 10 megabyte of JSON data!
  - Probably bad library for parsing

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!
- Example: for several years, GTA online took several minutes to load.
  - Several minutes for parsing 10 megabyte of JSON data!
  - Probably bad library for parsing
  - Unsuitable data structure for duplication check

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!
- Example: for several years, GTA online took several minutes to load.
  - Several minutes for parsing 10 megabyte of JSON data!
  - Probably bad library for parsing
  - Unsuitable data structure for duplication check
  - After fix: 70% less loading time

Why are we Interested in All This?

- Because algorithms/data structures with bad runtime complexity strike back!
- Example: for several years, GTA online took several minutes to load.
  - Several minutes for parsing 10 megabyte of JSON data!
  - Probably bad library for parsing
  - Unsuitable data structure for duplication check
  - After fix: 70% less loading time
  - `https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/index.html`

Summary

Summary

- In practice, we quite quickly can get an impression of the running time of an algorithm with simple “cookbook recipes”.
- Insertion sort has
  - in the best case running time $\Theta(n)$.
  - in the worst case running time $\Theta(n^2)$.