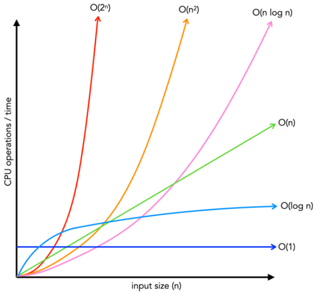
Algorithms and Data Structures

Big O
Properties & Examples of Big O



Learning goals

- Properties of Big O
- Know how to determine the runtime
- Complexity classes

PROPERTIES

Be $f, g, h, f_i, g_i : X \to \mathbb{R}$, $c \geq 0$.

1. Constants: $f \in \mathcal{O}(cg)$ is equivalent to $f \in \mathcal{O}(g)$. In particular: $f \in \mathcal{O}(c)$ is equivalent to $f \in \mathcal{O}(1)$ (Constant runtime)
2. Transitivity: If $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(h)$ then $f \in \mathcal{O}(h)$
3. Products: $f_1 \in \mathcal{O}(g_1)$ and $f_2 \in \mathcal{O}(g_2) \Rightarrow f_1 f_2 \in \mathcal{O}(g_1 g_2)$
4. Sums: $f_1 \in \mathcal{O}(g_1)$ and $f_2 \in \mathcal{O}(g_2) \Rightarrow f_1 + f_2 \in \mathcal{O}(|g_1| + |g_2|)$

PROPERTIES / 2

Particularly important for determining the runtime of an algorithm:

- If a function is the sum of several functions, the fastest growing function determines the order of the sum of functions.
- If f is a product of several factors, constants can be neglected.

Example 1:
The complexity of the function $f(n) = n \log n + 3 \cdot n^3$ can be determined quickly: the fastest growing function is $3 \cdot n^3$, multiplicative constants can be neglected. So

$$f(n) \in \mathcal{O}(n^3)$$

OTHER EXAMPLES

Example 2:

$$f(n) = 10\log(n) + 5(\log(n))^3 + 7n + 3n^2 + 6n^3$$

- The fastest growing summand is $6n^3$
- Constants can be neglected
- $\Rightarrow f(n) \in \mathcal{O}(n^3)$

Example 3:

$$g(n) = n^2 \cdot \exp(n)$$

- $\Rightarrow g(n) \in \mathcal{O}(n^2 \cdot \exp(n))$

DETERMINING THE RUNTIME

How fast a function runs depends on the different statements that are executed.

$$\mathit{total\_time} = \mathit{time}(\mathit{statement}_1) + \mathit{time}(\mathit{statement}_2) + \ldots + \mathit{time}(\mathit{statement}_k)$$

If each statement is a simple base operation, the time for each statement is constant and the total runtime is also constant: $\mathcal{O}(1)$.

DETERMINING THE RUNTIME / 2

If-else

```
if (cond) {
  block1 # sequence of statements
} else {
  block2 # sequence of statements
}
```

- Either `block1` **or** `block2` is executed
- The worst case is the slower one of the two options:

$$\max(\textit{time}(\textit{block}1), \textit{time}(\textit{block}2))$$

DETERMINING THE RUNTIME / 3

Loops

```
for (i in 1:n) {
  block # sequence of statements
}
```

- We consider *n* as part of our input size (e.g., number of elements in a list).
- The loop is executed *n* times.
- If we assume that the statements are $\mathcal{O}(1)$, then the total runtime is: $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

DETERMINING THE RUNTIME / 4

Nested loops

```
for (i in 1:n) {
  for (j in 1:m) {
    block # sequence of statements
  }
}
```

- Let m, n be part of our input size (e.g. number of rows/columns of a matrix).
- The outer loop is executed n times.
- At each iteration of i the inner loop is executed m times.
- Thus the statements are executed $n \cdot m$ times in total and the complexity is $\mathcal{O}(n \cdot m)$.

DETERMINING THE RUNTIME / 5

Statements with function calls

- When a statement calls a function, the complexity of the function must be included in the calculation.
- This also holds for loops:

```
for (i in 1:n) {
  g(i)
}
```

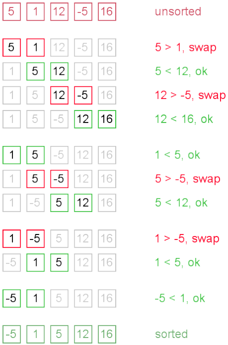
If $g \in \mathcal{O}(n)$, the runtime of the loop is $\mathcal{O}(n^2)$.

EXAMPLES (CONTINUED)

Example 4: Bubble sort algorithm

The bubble sort is an algorithm that sorts the elements of a (numeric) vector of length *n* in ascending order.

```
for (k in n:2) {
  for (i in 1:(k - 1)) {
    if (x[i] > x[i + 1]) {
      # swap elements
      s = x[i]
      x[i] = x[i + 1]
      x[i + 1] = s
    }
  }
}
```



http://teerexie.blogspot.com/

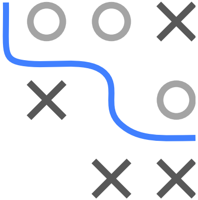
EXAMPLES (CONTINUED) / 2

- The inner loop depends on the outer loop and is executed $i = n-1$, then $i = n-2$, ... and finally $i = 1$ times.
- According to the sum of natural numbers (Carl Friedrich Gauss) the inner loop is executed $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ times.
- The operations in the `if` statement are operations with constant runtime.

The total runtime is therefore

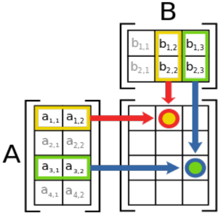
$$\frac{n^2 - n}{2} \cdot \mathcal{O}(1) = \mathcal{O}\left(\frac{n^2 - n}{2}\right) = \mathcal{O}(n^2)$$

EXAMPLES (CONTINUED) / 3



Example 5: The multiplication of two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{n \times p}$ has a runtime of $\mathcal{O}(mpn)$:

- $m \cdot p$ scalar products
- For each scalar product: n multiplications and $n - 1$ additions
- $\rightarrow m \cdot p \cdot (n + (n - 1))$ operations



https://commons.wikimedia.org/wiki/File:Matrix_multiplication_diagram_2.svg

EXAMPLES (CONTINUED) / 4

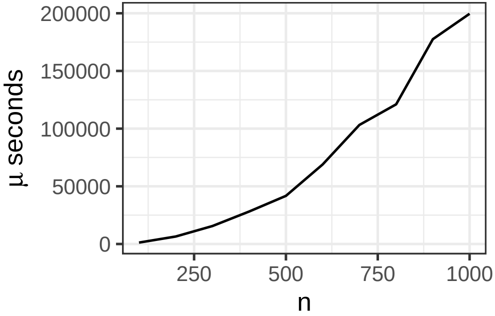
The Coppersmith-Winograd algorithm allows matrix multiplication of two $n \times n$ matrices in $\mathcal{O}(n^{2.373})$. A lower bound for the complexity of the matrix multiplication is n^2, since each of the n^2 elements of the output matrix must be generated.

More about [Computational complexity of mathematical operations]

EXAMPLES (CONTINUED) / 5

```
multiplyMatrices = function(n) {
  A = matrix(runif(n^2), n, n)
  B = matrix(runif(n^2), n, n)

  return(A %*% B)
}
```



EXAMPLES (CONTINUED) / 6

If possible: Avoid matrix multiplication!

```
n = 1000
A = matrix(runif(n), n, n)
B = matrix(runif(n), n, n)
y = c(runif(n))

system.time(A %*% B %*% y)
## user system elapsed
## 0.72 0.00 0.73

system.time(A %*% (B %*% y))
## user system elapsed
## 0.00 0.00 0.03
```

EXAMPLES (CONTINUED) / 7

```
n = 1000
A = matrix(rnorm(n), n, n) + diag(1, nrow = n)
b = rnorm(n)

# solving Ax = b
system.time(solve(A) %*% b)  # A^{-1} %*% b
## user system elapsed
## 0.96 0.01 0.05

system.time(solve(A, b)) # direct solution of the LES
## user system elapsed
## 0.0 0.2 0.0
```

EXAMPLES (CONTINUED) / 8

Example 6:
In mathematics one is interested in the estimation of error terms for approximations.
Using Taylor's theorem a *m*-times differentiable function f at point $x = x_0$ can be defined as follows:

$$\begin{aligned} f(x) &= f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f'(x_0)}{2!}(x - x_0)^2 + \dots + \frac{f^{(m)}(x_0)}{m!}(x - x_0)^m \\ &+ \mathcal{O}(|x - x_0|^{m+1}), \quad x \to x_0. \end{aligned}$$

- The more x approaches x_0, the better the Taylor polynomial approximates f at point x.
- The higher the order m of the Taylor polynomial, the better the approximation for $x \to x_0$.

EXAMPLES (CONTINUED) / 9

For example, consider the exponential function as **Taylor series**

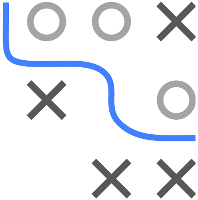
$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

$\exp(x)$ approximated at the point $x = 0$

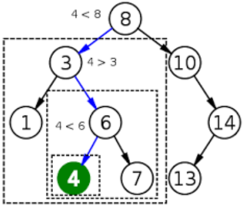
$$\exp(x) = 1 + x + \frac{x^2}{2!} + \mathcal{O}(x^3) \text{ for } x \to 0$$

In this way, it becomes clear that the error does not become greater than $M \cdot x^3$ when x approaches 0.

EXAMPLES (CONTINUED) / 10



Example 7:
The complexity of the **binary search** is visualized by a tree representation.



For an array of length n, the search tree has a height of $\log_2(n)$. After a maximum of $\log_2(n)$ comparisons, the searched element is found. The complexity of the binary search is $\mathcal{O}(\log n)$.

EXAMPLES (CONTINUED) / 11

Example 8:

The **Fibonacci sequence** is a series of numbers where each number is the sum of the two preceding ones, starting with 1. The sequence thus begins as:
$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

```
fib = function(n) {
  if (n <= 2L)
    return(1L)
  return(fib(n - 2) + fib(n - 1))
}

fib_table = microbenchmark(fib(5), fib(10), fib(20), fib(21), times = 500L)
print(xtable(summary(fib_table), digits = 0), booktabs=TRUE,
    caption.placement="top", size="\\fontsize{8pt}{9pt}\\selectfont")
```

	expr	min	lq	mean	median	uq	max	neval
1	fib(5)	2	2	99	3	4	47817	500
2	fib(10)	27	29	32	30	33	88	500
3	fib(20)	3611	3733	4164	3842	4052	10118	500
4	fib(21)	5861	6047	6926	6227	6476	49636	500

EXAMPLES (CONTINUED) / 12

Fibonacci$(n) \in \mathcal{O}(2^n)$ (exponential runtime)

Informal proof:

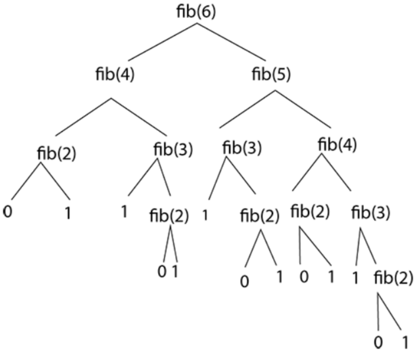
$$\texttt{Fibonacci(n)} = \underbrace{\texttt{Fibonacci(n - 1)}}_{T(n-1)} \underbrace{+}_{\mathcal{O}(1)} \underbrace{\texttt{Fibonacci(n - 2)}}_{T(n-2)}$$

This results in a runtime of $T(n) = T(n-1) + T(n-2) + \mathcal{O}(1)$ for $n > 1$.

The function is executed twice in each step.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &= T(n-2) + T(n-3) + T(n-3) + T(n-4) = \ldots \end{aligned}$$

EXAMPLES (CONTINUED) / 13



By simply "counting" the nodes of this recursion tree you can determine the exact number of operations.
$\rightarrow$ Worst case runtime $\mathcal{O}(2^n)$.

EXAMPLES (CONTINUED) / 14

Variations of Fibonacci(n): Iterative

```
fib2 = function(n) {
  a = 0; b = 1
  if (n <= 2)
    return(1)
  for (i in seq_len(n-1L)) {
    tmp = b;  b = a + b; a = tmp
  }
  return(b)
}
```

This is $\mathcal{O}(n)$ (if we, incorrectly, assume addition is constant in n).

EXAMPLES (CONTINUED) / 15

```
fib2_table = microbenchmark(fib2(10), fib2(20), fib2(40), fib2(80),
                            fib2(160), times = 5000L)

print(xtable(summary(fib2_table), digits = 0), booktabs=TRUE,
    caption.placement="top", size="\\fontsize{8pt}{9pt}\\selectfont")
```

	expr	min	lq	mean	median	uq	max	neval
1	fib2(10)	1	1	2	1	2	37	5000
2	fib2(20)	1	2	2	2	2	24	5000
3	fib2(40)	2	2	4	2	3	6755	5000
4	fib2(80)	3	3	4	3	4	26	5000
5	fib2(160)	5	5	7	6	7	49	5000

Time measurement becomes imprecise since “for loops” are not that slow in R due to JIT compilation. Hence we are using doubles here as a lazy trick to generate large fibonacci numbers. An alternative to generate large integers would be to use the int64 package.

EXAMPLES (CONTINUED) / 16

Variations of Fibonacci(n): In C

```
library(inline)
fib3 = cfunction(signature(n="integer"), language="C",
         convention=".Call", body = '
         int nn = INTEGER(n)[0];
         SEXP res;
         PROTECT(res = allocVector(INTSXP, 1));
         INTEGER(res)[0] = 1;
         int a = 0; int b = 1;
         for (int i=0; i<nn-1; i++) {
         int tmp = b;
         b = a + b;
         a = tmp;
         }
         INTEGER(res)[0] = b;
         UNPROTECT(1);
         return res;
         ')
```

See how ugly the C interface is?

EXAMPLES (CONTINUED) / 17

```
fib3_table = microbenchmark(fib3(20L), fib3(40L),times = 5000L)

print(xtable(summary(fib3_table), digits = 0), booktabs=TRUE,
    caption.placement="top", size="\\fontsize{8pt}{9pt}\\selectfont")
```

	expr	min	lq	mean	median	uq	max	neval
1	fib3(20L)	300	400	465	400	500	13900	5000
2	fib3(40L)	300	400	479	400	500	21200	5000

This is both $\mathcal{O}(n)$ ... See the difference? Actually, you do not see anything as the function is so fast, we would need to calculate with bigints to really see the $\mathcal{O}(n)$!

EXAMPLES (CONTINUED) / 18

Variations of Fibonacci(n): `C++`-version

```
library(Rcpp)
fib4 = cppFunction('int fibonacci(const int x) {
         if (x <= 2) return(1);
         return (fibonacci(x - 1)) + fibonacci(x - 2);
}
' )
```

Much nicer `C++`-Interface with Rcpp.

EXAMPLES (CONTINUED) / 19

Variations of Fibonacci(n): Matrix power-exponentiation

```
library(expm)
fib5 = function(n) {
  A = matrix(c(1, 1, 1, 0), 2, 2)
  B = A%^%n
  B[1, 2]
}
```

How does **fib5()** work?

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad \boldsymbol{A}^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \quad \boldsymbol{A}^3 = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix} \quad \boldsymbol{A}^4 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}$$

$$\boldsymbol{A}^5 = \begin{pmatrix} 8 & 5 \\ 5 & 3 \end{pmatrix} \quad \boldsymbol{A}^6 = \begin{pmatrix} 13 & 8 \\ 8 & 5 \end{pmatrix} \quad \boldsymbol{A}^7 = \begin{pmatrix} 21 & 13 \\ 13 & 8 \end{pmatrix} \quad \cdots$$

EXAMPLES (CONTINUED) / 20

Matrix power-exponentiation
What does `A %^% n` do?
Computes the n-th power of a matrix corresponding to $n-1$ matrix multiplications (`A^n` only computes element wise powers).
The algorithm uses $\mathcal{O}(log_2(k))$ matrix multiplications.

Exponentiation by squaring:

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}} & \text{if n is odd} \\ (x^2)^{\frac{n}{2}} & \text{if n is even} \end{cases}$$

EXAMPLES (CONTINUED) / 21

Exponentiation by squaring

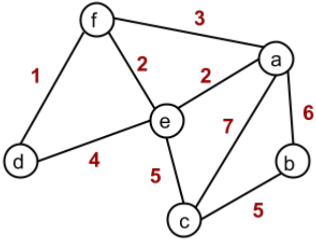
Implemented as a recursive algorithm:

```
exp.by.squaring = function(x, n) {
  if(n<0) {
    return(exp.by.squaring(1 / x, -n))
  } else if(n==0){
    return(1)
  } else if(n==1){
    return(x)
  } else if(n%%2 == 0){
    return(exp.by.squaring(x^2, n/2))
  } else {
    return(x * exp.by.squaring(x^2, (n-1)/2))
  }
}
exp.by.squaring(2,5)
## [1] 32
```

EXAMPLES (CONTINUED) / 22

Example 9: The **Traveling Salesman Problem** (TSP) is the problem of planning a route through all locations in such a way that

- The entire route is as short as possible,
- The first location is equal to the last location.



Left: Route through places in Germany
(`https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden`)
Right: Weighted graph (`https://www.chegg.com/`)

EXAMPLES (CONTINUED) / 23

Exact algorithms with long runtime exist

- Brute force search (Calculate lengths of all possible round trips and choose shortest): $\mathcal{O}(n!)$
- Dynamic Programming (Held-Karp algorithm): $\mathcal{O}(n^2 2^n)$

and heuristic algorithms with shorter runtime, which do not guarantee an optimal solution, e.g.

- Nearest-Neighbor heuristics: $\mathcal{O}(n^2)$

The TSP problem is **NP-complete**.

COMPLEXITY CLASSES

In theoretical computer science, problems are divided into complexity classes. For an input size n a distinction is made between

- **P**: Problems solvable in polynomial runtime ($\mathcal{O}(n^k), k \geq 1$)
- **NP** (**N**on-deterministic **P**olynomial time): Problems from **P** and problems that cannot be solved in polynomial time;
  NP problems can only be solved with a non-deterministic turing machine in an acceptable time (hence the name)
- **NP-complete**: All problems from NP can be traced back to this problem

It has not yet been proven that P $\neq$ NP holds.