Week 12: Complexity and Performance

POP77001 Computer Programming for Social Scientists

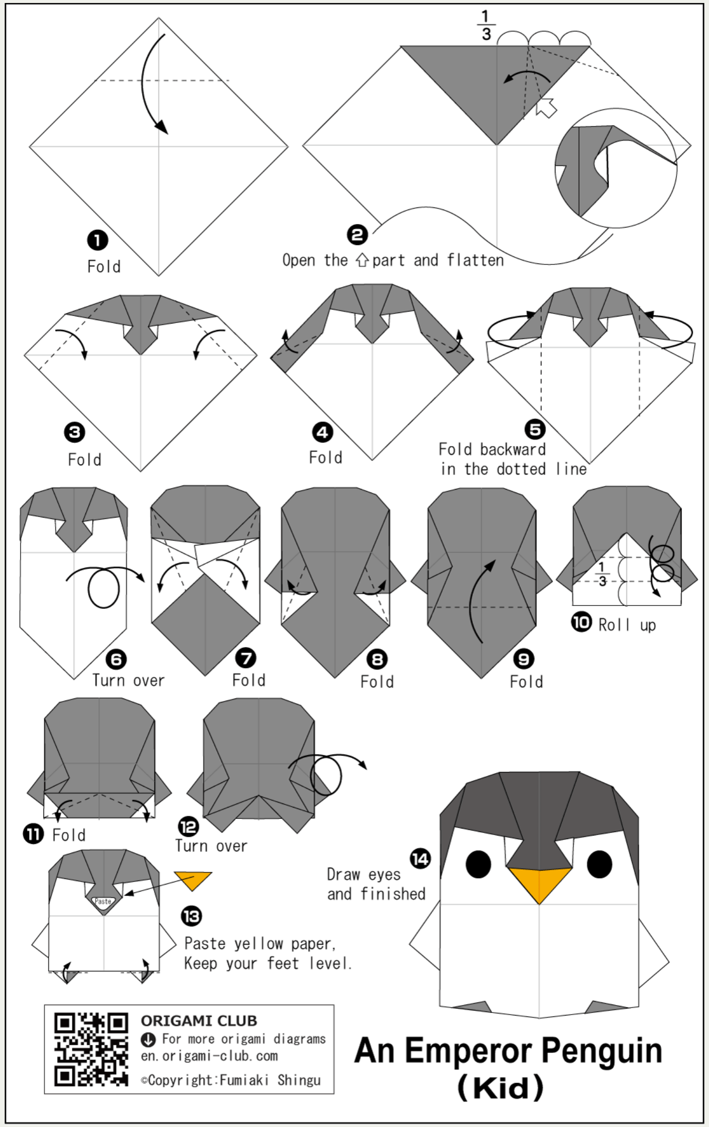
Tom Paskhalis

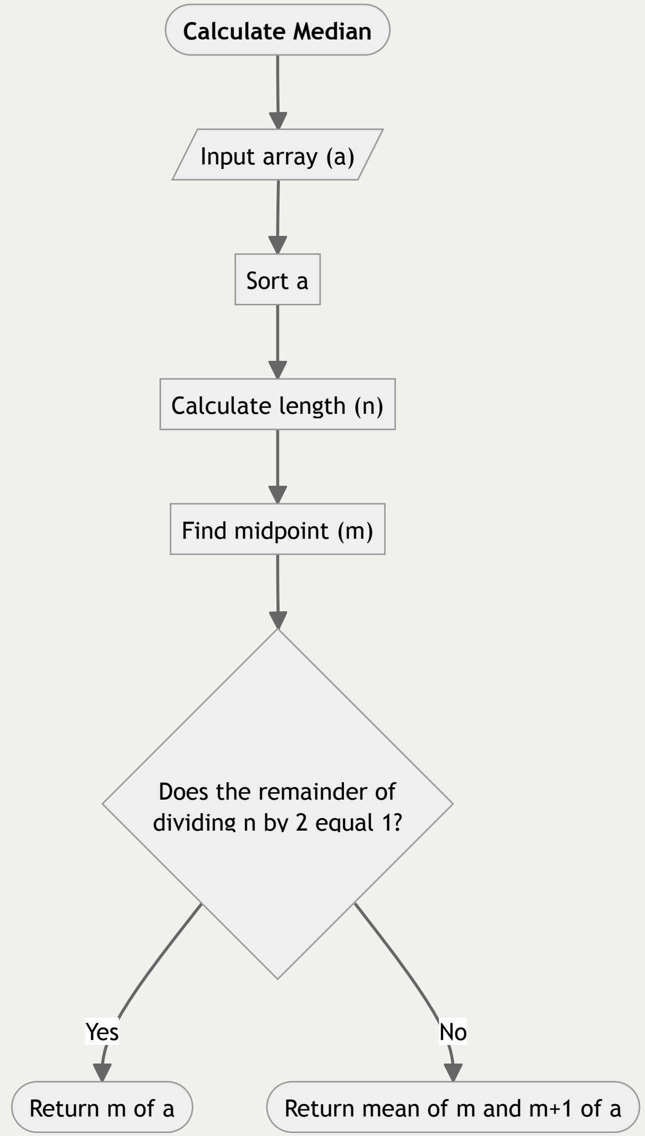
Overview

- Algorithms
- Computational complexity
- Big-O notation
- Code optimisation
- Benchmarking

Complexity

Algorithms





(Origami Club)

Algorithm

- *Finite list of well-defined instructions that take input and produce output.*
- Consists of a sequence of simple steps that start from input, follow some control flow and have a stopping rule.

Complexity

- **Conceptual complexity**
  - Structural sophistication of a program
- **Computational complexity**
  - Resources (time/space) required to finish a program
- Often there is some trade-off between the two
- Reducing computational complexity results in increased conceptual complexity

How Long Does It Take to Run?

Linear search using exhaustive enumeration.

In R:

```r
linear_search <- function(v, x) {
  n <- length(v)
  for (i in seq_len(n)) {
    if (v[i] == x) {
      return(TRUE)
    }
  }
  return(FALSE)
}
```

In Python:

```python
def linear_search(v, x):
    n = len(v)
    for i in range(n):
        if v[i] == x:
            return True
    return False
```

Benchmarking

Benchmarking in R

```r
# Best case (running time is independent of the length of vector)
system.time(linear_search(1:1e6, 1))
```

```
 user  system elapsed 
0.003   0.000   0.003 
```

```r
# Average case (middle element for vector of length 1M)
system.time(linear_search(1:1e6, 5e5))
```

```
 user  system elapsed 
0.016   0.000   0.015 
```

```r
# Worst case (last element for vector of length 1M)
system.time(linear_search(1:1e6, 1e6))
```

```
 user  system elapsed 
 0.03    0.00    0.03 
```

```r
# Even worse case (last element for vector of length 1B)
system.time(linear_search(1:1e9, 1e9))
```

```
  user  system elapsed 
31.545   0.017  31.606 
```

Limitations of Benchmarking

- Depends on many factors:
  - Computer hardware
  - Input size
  - Programming language used
- Which of the benchmarking cases is the most useful?

More General Approach

Worst-case Scenario

> Anything that can go wrong will go wrong.
>
> Edward Murphy

- In **defensive design** it is often helpful to think about the worst-case scenario.
- This sets an **upper bound** on the execution time of a program.
- However, this still depends on the input size.

Number of Steps

- A useful heuristic is the number of steps that a program takes.

```r
linear_search <- function(v, x) {
  n <- length(v) # 2 steps (call to length() function and assignment `<-`)
  # n steps + 1 step (for seq_len(n) call)
  for (i in seq_len(n)) {
    # 3n steps (n steps for `[`(v, i), another n steps for `==` and n calls to `if`)
    if (v[i] == x) {
      return(TRUE) # 1 step
    }
  }
  return(FALSE) # 1 step
}
```

- $4n + 4$
- If length of input vector v is 1000 ($n = 1000$),
- This function will execute roughly 4004 steps.

Calculating Number of Steps

- Consider the previous example: $4n + 4$.
- As n grows larger, these extra 4 steps can be ignored.
- Multiplicative constants can certainly make a difference within implementation.
- But across several algorithms the difference between $2n$ and $4n$ is usually negligible.

Alternative Algorithms

```r
# Binary search for sorted sequences
binary_search <- function(v, x) {
  low <- 1
  high <- length(v)
  while (low <= high) {
    # Calculate mid-point (similar to median)
    m <- (low + high) %/% 2
    if (v[m] < x) {
      low <- m + 1
    } else if (v[m] > x) {
      high <- m - 1
    } else {
      return(TRUE)
    }
  }
  return(FALSE)
}
```

Comparing Algorithms: Benchmarking

```r
system.time(linear_search(1:1e6, 1e6))
```

```
 user  system elapsed 
 0.03    0.00    0.03 
```

```r
system.time(binary_search(1:1e6, 1e6))
```

```
 user  system elapsed 
0.005   0.000   0.005 
```

```r
system.time(1e6 %in% 1:1e6)
```

```
 user  system elapsed 
0.001   0.004   0.005 
```

Comparing Algorithms: N Steps

```r
linear_search <- function(v, x) {
  n <- length(v)
  for (i in seq_len(n)) {
    if (v[i] == x) {
      print(paste0("Number of iterations: ", as.character(i)))
      return(i) # return(TRUE)
    }
  }
  return(i) # return(FALSE)
}
```

```r
binary_search <- function(v, x) {
  low <- 1
  high <- length(v)
  iters <- 1
  while (low <= high) {
    m <- (low + high) %/% 2
    if (v[m] < x) {
      low <- m + 1
    } else if (v[m] > x) {
      high <- m - 1
    } else {
      print(paste0("Number of iterations: ", as.character(iters)))
      return(iters) # return(TRUE)
    }
    iters <- iters + 1
  }
  return(iters) # return(FALSE)
}
```

Comparing Algorithms: N Steps

```r
ls_1e3 <- linear_search(1:1e3, 1e3)
```

```
[1] "Number of iterations: 1000"
```

```r
ls_1e6 <- linear_search(1:1e6, 1e6)
```

```
[1] "Number of iterations: 1000000"
```

```r
bs_1e3 <- binary_search(1:1e3, 1e3)
```

```
[1] "Number of iterations: 10"
```

```r
bs_1e6 <- binary_search(1:1e6, 1e6)
```

```
[1] "Number of iterations: 20"
```

Big-O Notation

- Performance on small inputs and in best-case scenarios is usually of limited interest.
- What matters is the worst-case performance on progressively larger inputs.
- In other words, **upper bound** (or **order of growth**).
- Big O (**O**rder of growth) is an *asymptotic notation* for describing such growth.
- For example, in case of linear search $O(n)$ (running time increases linearly in the size of input).
- The most important question is the growth rate of the largest term.
- All constants can be ignored.

Common Complexity Cases

Varieties of Big-O

Big-O notation	Running time
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	log-linear
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial

Constant Complexity: $O(1)$

- Running time of a program is bounded by a value, which is independent of the input size

```r
get_len <- function(v) {
    # Internally, length just returns the 'length' attribute of an R object
    n <- length(v)
    return(n)
}
```

```r
system.time(get_len(1:1e3))
```

```
   user  system elapsed 
      0       0       0 
```

```r
system.time(get_len(1:1e6))
```

```
   user  system elapsed 
  0.001   0.000   0.001 
```

```r
system.time(get_len(1:1e9))
```

```
   user  system elapsed 
      0       0       0 
```

Logarithmic Complexity: $O(\log n)$

- E.g. for binary search $O(\log(n))$ (running time increases as a logarithm of the input size)
- Base of logarithm is irrelevant as it can be easily re-arranged $\log_2(n) = \log_2(10) \times \log_{10}(n)$ and constants are ignored
- For base 2 we can say that each time the size of input doubles, the algorithm performs one additional step

```r
bs_10 <- binary_search(1:10, 10)
```

```
[1] "Number of iterations: 4"
```

```r
bs_20 <- binary_search(1:20, 20)
```

```
[1] "Number of iterations: 5"
```

```r
bs_100 <- binary_search(1:100, 100)
```

```
[1] "Number of iterations: 7"
```

```r
bs_200 <- binary_search(1:200, 200)
```

```
[1] "Number of iterations: 8"
```

```r
bs_1000 <- binary_search(1:1000, 1000)
```

```
[1] "Number of iterations: 10"
```

Linear Complexity: $O(n)$

- For linear search $O((n))$ (running time increases as a linear function of the input size).
- Generally, iteration over all elements of a sequence would be in $O(n)$.
- But it doesn't have to be a loop (e.g. recursive factorial implementation).

```
ls_10 <- linear_search(1:10, 10)
```

```
[1] "Number of iterations: 10"
```

```
ls_20 <- linear_search(1:20, 20)
```

```
[1] "Number of iterations: 20"
```

```
ls_100 <- linear_search(1:100, 100)
```

```
[1] "Number of iterations: 100"
```

```
ls_200 <- linear_search(1:200, 200)
```

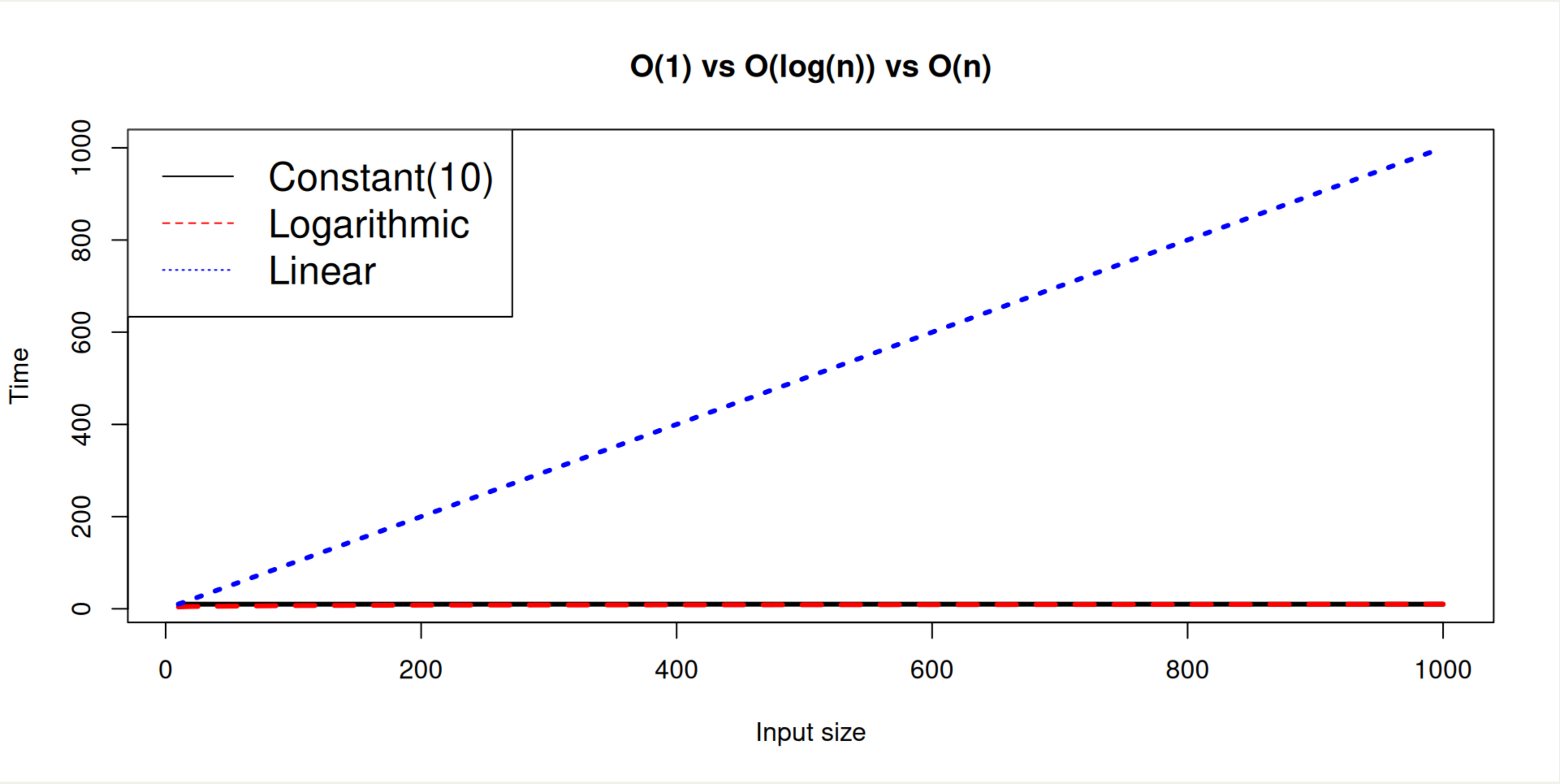
```
[1] "Number of iterations: 200"
```

```
ls_1000 <- linear_search(1:1000, 1000)
```

```
[1] "Number of iterations: 1000"
```

Comparing $O(1)$, $O(\log n)$, $O(n)$

Plot Code



Log-Linear Complexity: $O(n \log n)$

- More complicated complexity case.
- Involves a product of 2 terms.
- Important complexity case as many practical problems are solved in log-linear time.
- Many sorting algorithms (e.g. merge sort, timsort, built-in `sorted()` in Python).

Polynomial Complexity: $O(n^c)$

- Most common case is quadratic complexity: $O(n^2)$
- Nested loops typically result in polynomial complexity.

```r
# Check whether one vector is contained within another vector
is_subset <- function(v1, v2) {
  n <- length(v1)
  m <- length(v2)
  iters <- 1
  for (i in seq_len(n)) {
    matched <- FALSE
    for (j in seq_len(m)) {
      iters <- iters + 1
      if (v1[i] == v2[j]) {
        matched <- TRUE
        break
      }
    }
    if (isTRUE(matched)) {
      print(paste0("Number of iterations: ", as.character(iters)))
      return(iters) # return(TRUE)
    }
  }
  print(paste0("Number of iterations: ", as.character(iters)))
  return(iters) # return(FALSE)
}
```

Polynomial Complexity

```
# For simplicity of analysis let's assume that
# lengths of 2 input vectors are about the same
is_10 <- is_subset(11:21, 1:10)
```

```
[1] "Number of iterations: 111"
```

```
is_20 <- is_subset(21:41, 1:20)
```

```
[1] "Number of iterations: 421"
```

```
is_100 <- is_subset(101:201, 1:100)
```

```
[1] "Number of iterations: 10101"
```

```
is_200 <- is_subset(201:401, 1:200)
```

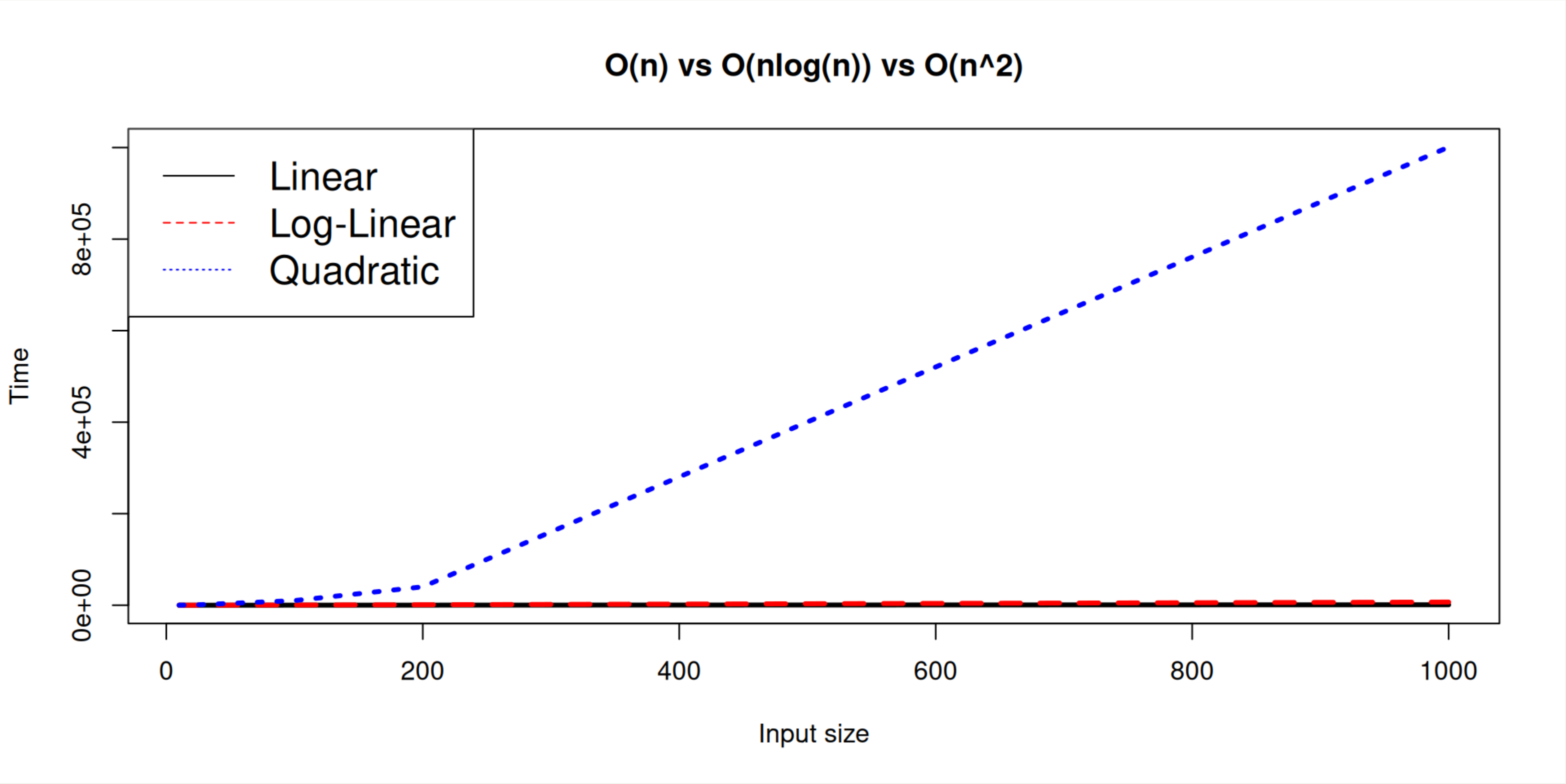
```
[1] "Number of iterations: 40201"
```

```
is_1000 <- is_subset(1001:2001, 1:1000)
```

```
[1] "Number of iterations: 1001001"
```

Comparing $O(n)$, $O(n \log n)$, $O(n^2)$

Plot | Code

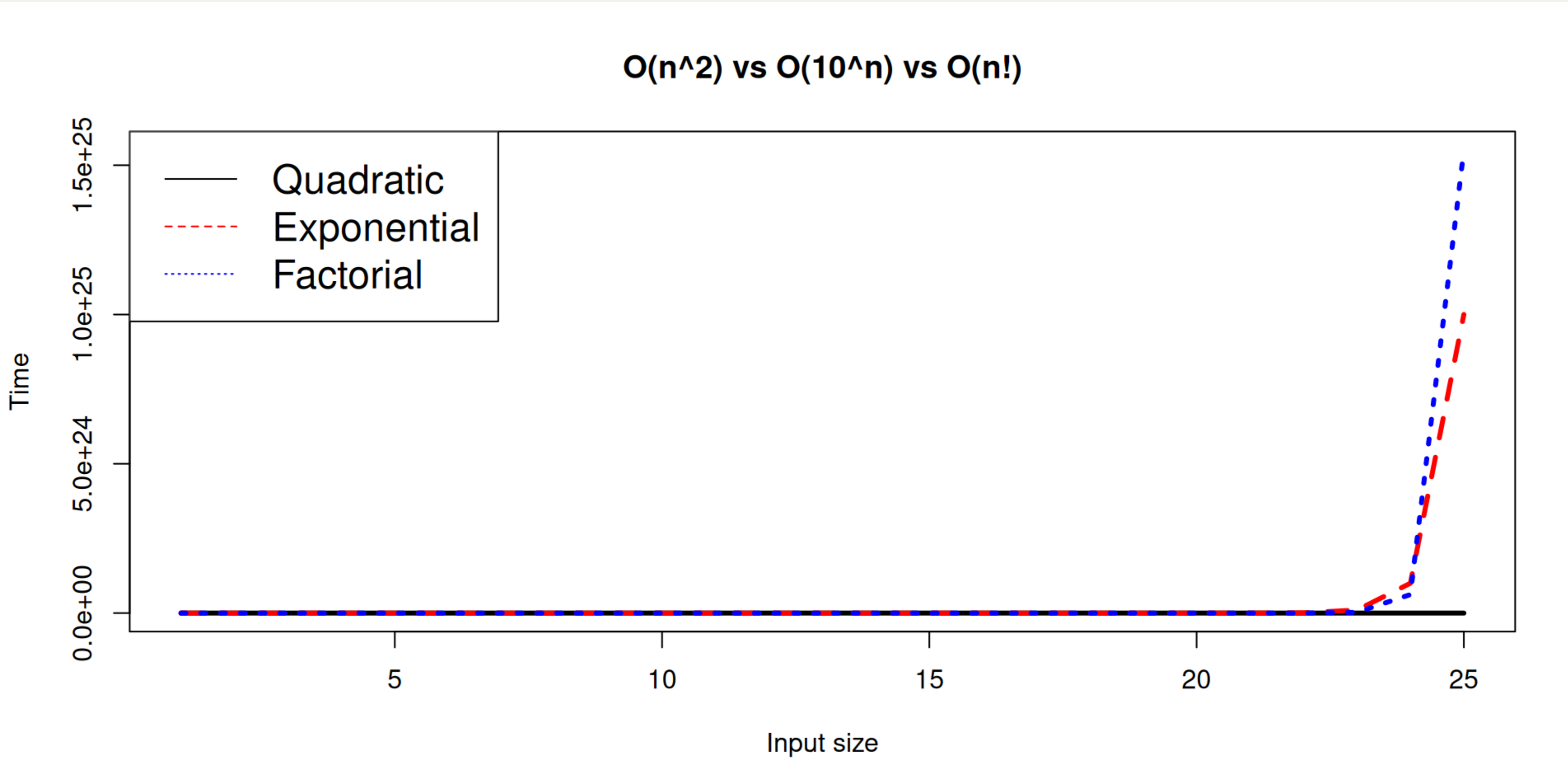


Exponential and Factorial Complexity: $O(c^n)$ and $O(n!)$

- Many real-life problems have exponential or factorial solutions
- E.g. travelling salesman problem (given the list of points and distances, what is the shortest path through all of them)
- However, the general solutions with such complexity are usually impractical
- For these problems either approximate solutions can be used

Comparing $O(n^2)$, $O(10^n)$, $O(n!)$

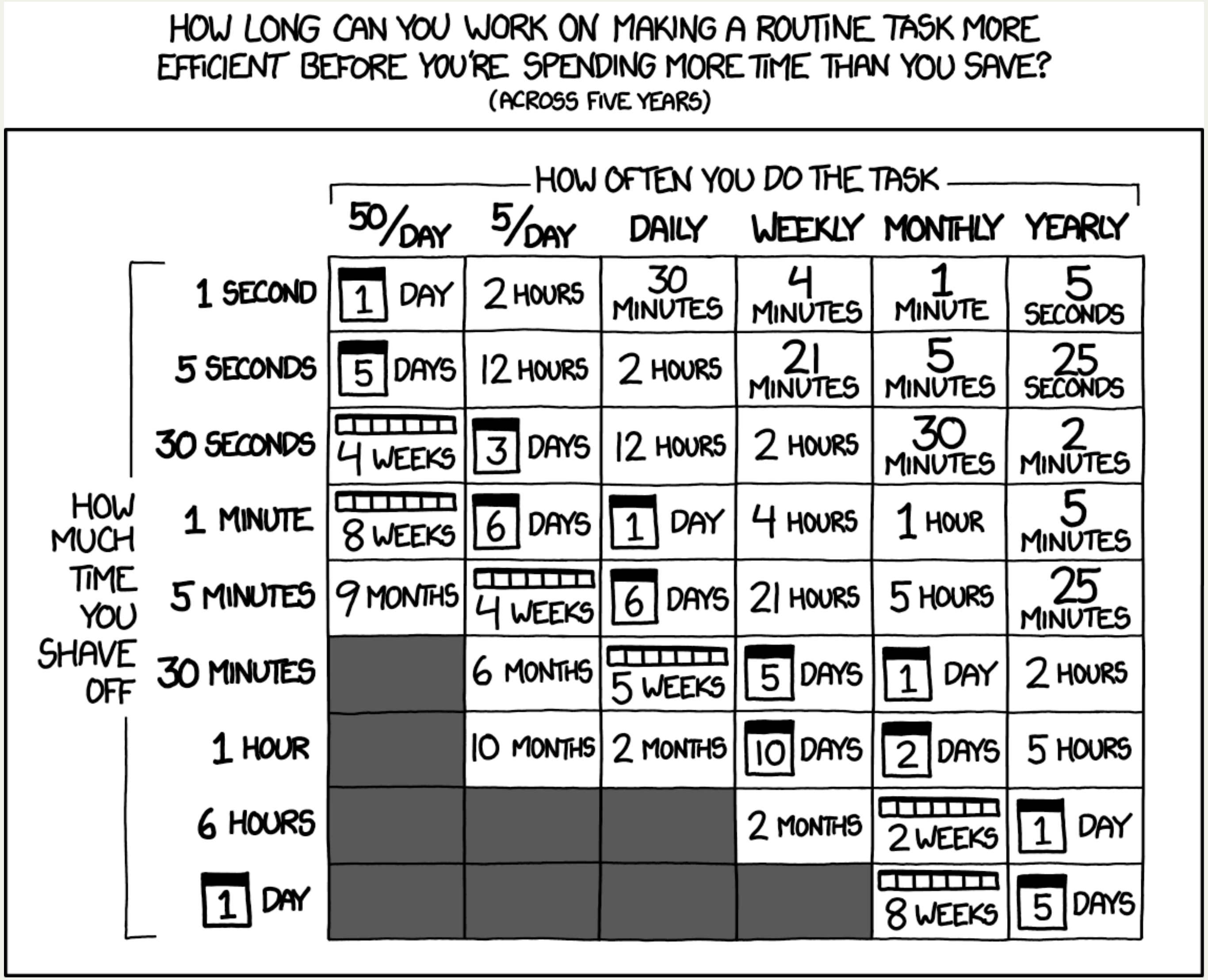
Plot Code



Time Complexity in Practice

- Check for loops, recursion.
- Think about function calls, what is the complexity of underlying implementations?
- The presented complexity cases are not exhaustive and are often 'idealised' cases.
- Complexity can take any form (e.g. one of `sort()` methods in R is $O(n^{4/3})$).
- Running time can be a function of more than one input (e.g. `is_subset()` function above).
- Big-O ignores constants (multiplicative and additive), but they matter within implementation.
- There is a tradeoff between conceptual and computational complexity (more efficient solutions are often harder to read and debug).

Code Optimisation Tradeoff



(xkcd)

Next

- Tutorial: Benchmarking, analysis of function complexity and performance
- Final project: Due by 23:59 on Friday, 13th December (submission on Blackboard)

The End


:(
Your PC ran into a problem that it couldn't handle, and now it needs to restart.
You can search for the error online: HAL_INITIALIZATION_FAILED