Week 12: Complexity and Performance

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

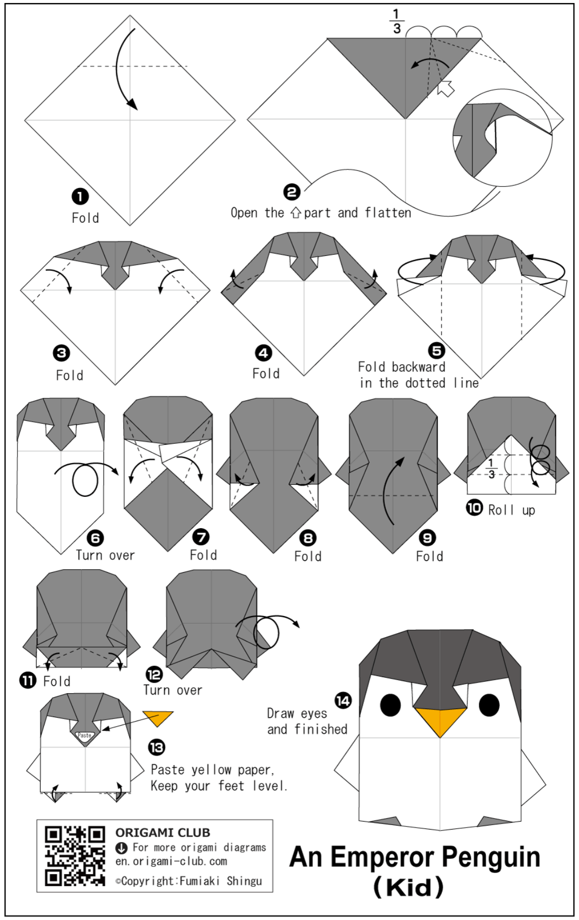
28 November 2022

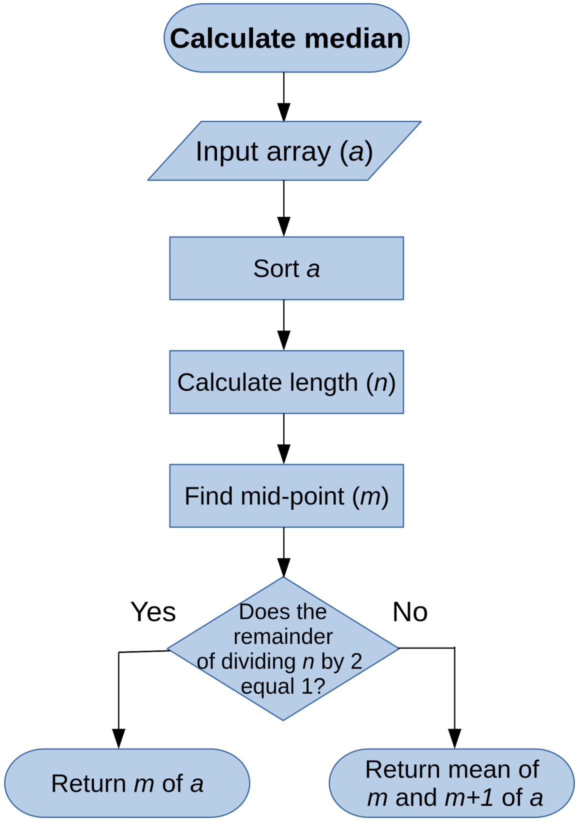
Module website: tinyurl.com/POP77001

Overview

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

Algorithms





Source: 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 a program take to run?

How long does a program take to run?

In [2]:

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

Benchmarking

Benchmarking

In [3]:

```
# 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   0.002 
```

Benchmarking

In [3]:
```
# 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   0.002 
```

In [4]:
```
# Average case (middle element for vector of length 1M)
system.time(linear_search(1:1e6, 5e5))
```

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

Benchmarking

In [3]:
```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   0.002 
```

In [4]:
```r
# Average case (middle element for vector of length 1M)
system.time(linear_search(1:1e6, 5e5))
```

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

In [5]:
```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 
```

Benchmarking

In [3]:
```
# 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   0.002 
```

In [4]:
```
# Average case (middle element for vector of length 1M)
system.time(linear_search(1:1e6, 5e5))
```

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

In [5]:
```
# 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 
```

In [6]:
```
# Even worse case (last element for vector of length 1B)
system.time(linear_search(1:1e9, 1e9))
```

```
   user  system elapsed 
 31.747   0.063  31.883 
```

Limitations of benchmarking

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

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.

Number of steps

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

In [7]:

```
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 (
    if (v[i] == x) {
      return(TRUE) # 1 step
    }
  }
  return(FALSE) # 1 step
}
```

Number of steps

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

In [7]:
```
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 c
    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: $4n + 2$.
- As n grows larger, these extra 2 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

Alternative algorithms

In [8]:

```
# 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)

Comparing algorithms (benchmarking)

In [9]:

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

```
   user  system elapsed 
  0.034   0.000   0.035 
```

Comparing algorithms (benchmarking)

In [9]:
```
system.time(linear_search(1:1e6, 1e6))
```

```
   user  system elapsed 
  0.034   0.000   0.035 
```

In [10]:
```
system.time(binary_search(1:1e6, 1e6))
```

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

Comparing algorithms (benchmarking)

In [9]:
```
system.time(linear_search(1:1e6, 1e6))
```

```
   user  system elapsed 
  0.034   0.000   0.035 
```

In [10]:
```
system.time(binary_search(1:1e6, 1e6))
```

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

In [11]:
```
system.time(1e6 %in% 1:1e6)
```

```
   user  system elapsed 
  0.000   0.004   0.004 
```

Comparing algorithms (n steps)

Comparing algorithms (n steps)

In [12]:

```
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)
}
```

Comparing algorithms (n steps)

In [12]:

```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)
}
```

In [13]:

```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) continued

Comparing algorithms (n steps) continued

In [14]:

```
ls_1e3 <- linear_search(1:1e3, 1e3)
ls_1e6 <- linear_search(1:1e6, 1e6)
```

```
[1] "Number of iterations: 1000"
[1] "Number of iterations: 1000000"
```

Comparing algorithms (n steps) continued

In [14]:
```
ls_1e3 <- linear_search(1:1e3, 1e3)
ls_1e6 <- linear_search(1:1e6, 1e6)
```

```
[1] "Number of iterations: 1000"
[1] "Number of iterations: 1000000"
```

In [15]:
```
bs_1e3 <- binary_search(1:1e3, 1e3)
bs_1e6 <- binary_search(1:1e6, 1e6)
```

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

Important computational complexity cases

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

Constant complexity: $O(1)$

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

In [16]:
```
get_len <- function(v) {
    # Internally, length just returns the 'length' attribute of an R ol
    n <- length(v)
    return(n)
}
```

Constant complexity: $O(1)$

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

In [16]:
```
get_len <- function(v) {
    # Internally, length just returns the 'length' attribute of an R ob
    n <- length(v)
    return(n)
}
```

In [17]:
```
system.time(get_len(1:1e3))
```

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

Constant complexity: $O(1)$

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

In [16]:
```
get_len <- function(v) {
    # Internally, length just returns the 'length' attribute of an R ob
    n <- length(v)
    return(n)
}
```

In [17]:
```
system.time(get_len(1:1e3))
```

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

In [18]:
```
system.time(get_len(1:1e6))
```

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

Constant complexity: $O(1)$

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

In [16]:
```
get_len <- function(v) {
    # Internally, length just returns the 'length' attribute of an R ob
    n <- length(v)
    return(n)
}
```

In [17]:
```
system.time(get_len(1:1e3))
```

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

In [18]:
```
system.time(get_len(1:1e6))
```

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

In [19]:
```
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

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

In [20]:
```r
bs_10 <- binary_search(1:10, 10)
bs_20 <- binary_search(1:20, 20)
bs_100 <- binary_search(1:100, 100)
bs_200 <- binary_search(1:200, 200)
bs_1000 <- binary_search(1:1000, 1000)
```

```
[1] "Number of iterations: 4"
[1] "Number of iterations: 5"
[1] "Number of iterations: 7"
[1] "Number of iterations: 8"
[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)

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)

In [21]:

```
ls_10 <- linear_search(1:10, 10)
ls_20 <- linear_search(1:20, 20)
ls_100 <- linear_search(1:100, 100)
ls_200 <- linear_search(1:200, 200)
ls_1000 <- linear_search(1:1000, 1000)
```

```
[1] "Number of iterations: 10"
[1] "Number of iterations: 20"
[1] "Number of iterations: 100"
[1] "Number of iterations: 200"
[1] "Number of iterations: 1000"
```

Graphic comparison of time complexity: $O(1)$, $O(\log n)$, $O(n)$

Graphic comparison of time complexity: $O(1)$, $O(\log n)$, $O(n)$

In [22]:

```
x <- c(10, 20, 100, 200, 1000)
constant_complexity <- rep(10, length(x))
log_complexity <- c(bs_10, bs_20, bs_100, bs_200, bs_1000)
linear_complexity <- c(ls_10, ls_20, ls_100, ls_200, ls_1000)
```

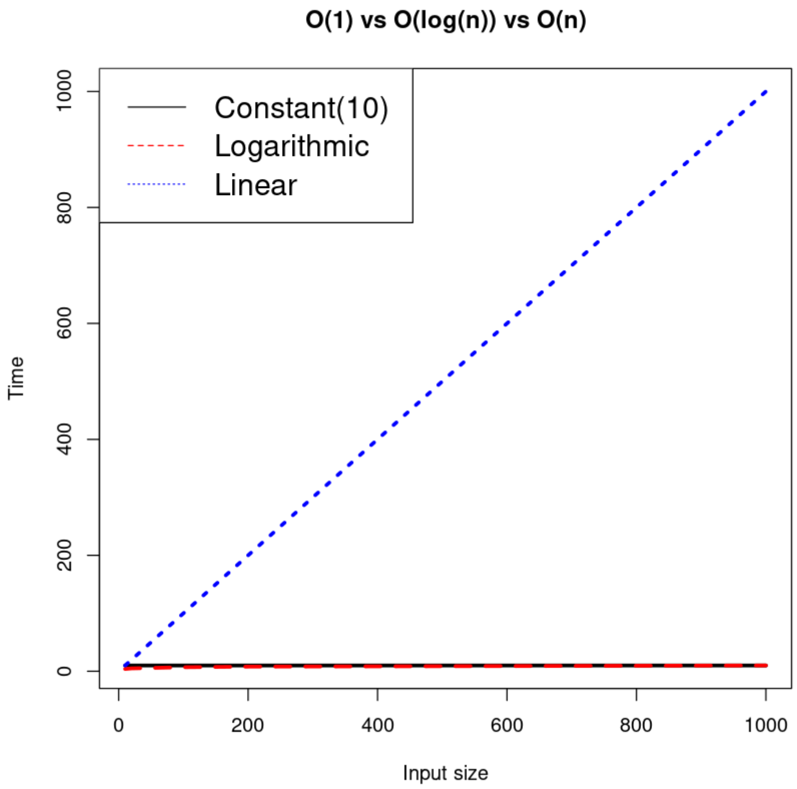
Graphic comparison of time complexity: $O(1)$, $O(\log n)$, $O(n)$

In [22]:

```
x <- c(10, 20, 100, 200, 1000)
constant_complexity <- rep(10, length(x))
log_complexity <- c(bs_10, bs_20, bs_100, bs_200, bs_1000)
linear_complexity <- c(ls_10, ls_20, ls_100, ls_200, ls_1000)
```

In [23]:

```
plot(x, linear_complexity, type = "n", main = "O(1) vs O(log(n)) vs O(r
lines(x, constant_complexity, type = "l", col = "black", lty = 1, lwd =
lines(x, log_complexity, type = "l", col = "red", lty = 2, lwd = 3)
lines(x, linear_complexity, type = "l", col = "blue", lty = 3, lwd = 3)
legend("topleft", legend = c("Constant(10)", "Logarithmic", "Linear"),
       col = c("black", "red", "blue"), lty = 1:3, cex = 1.5)
```



O(1) vs O(log(n)) vs O(n)
Constant(10)
Logarithmic
Linear
Time
0
200
400
600
800
1000
Input size

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

Polynomial complexity: $O(n^c)$

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

In [24]:

```
# 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: $O(n^c)$

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

In [24]:

```
# 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)
}
```

In [25]:

```r
# 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)
is_20 <- is_subset(21:41, 1:20)
is_100 <- is_subset(101:201, 1:100)
is_200 <- is_subset(201:401, 1:200)
is_1000 <- is_subset(1001:2001, 1:1000)
```

```
[1] "Number of iterations: 111"
[1] "Number of iterations: 421"
[1] "Number of iterations: 10101"
[1] "Number of iterations: 40201"
[1] "Number of iterations: 1001001"
```

Graphic comparison: $O(n)$, $O(n \log n)$, $O(n^2)$

Graphic comparison: $O(n)$, $O(n \log n)$, $O(n^2)$

In [26]:
```
log_linear_complexity <- x * log(x)
quadratic_complexity <- c(is_10, is_20, is_100, is_200, is_1000)
```

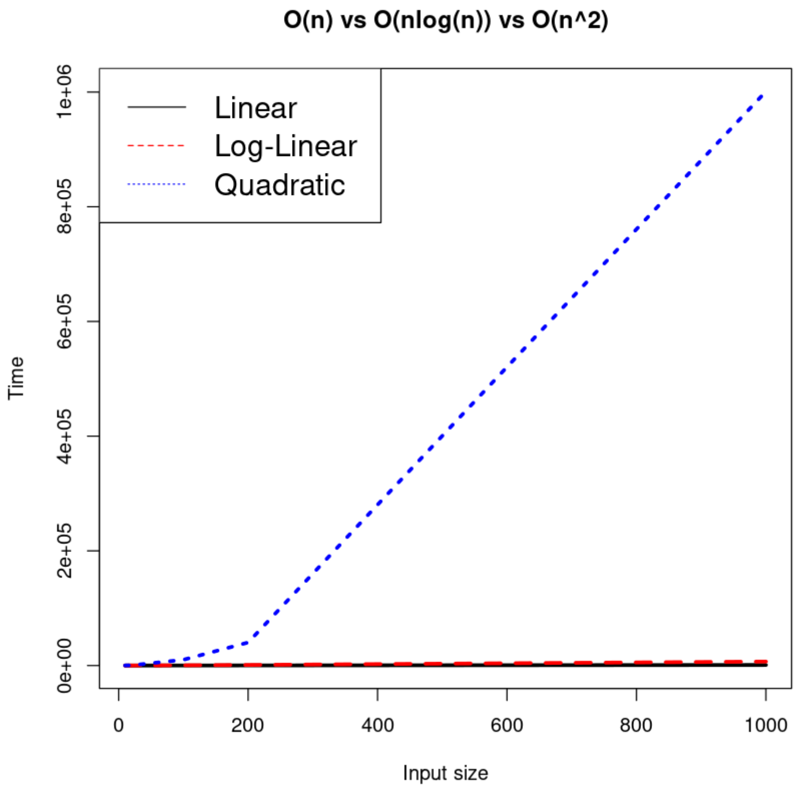
Graphic comparison: $O(n)$, $O(n \log n)$, $O(n^2)$

In [26]:

```
log_linear_complexity <- x * log(x)
quadratic_complexity <- c(is_10, is_20, is_100, is_200, is_1000)
```

In [27]:

```
plot(x, quadratic_complexity, type = "n", main = "O(n) vs O(nlog(n)) vs
lines(x, linear_complexity, type = "l", col = "black", lty = 1, lwd = 3
lines(x, log_linear_complexity, type = "l", col = "red", lty = 2, lwd =
lines(x, quadratic_complexity, type = "l", col = "blue", lty = 3, lwd =
legend("topleft", legend = c("Linear", "Log-Linear", "Quadratic"),
       col = c("black", "red", "blue"), lty = 1:3, cex = 1.5)
```


O(n) vs O(nlog(n)) vs O(n^2)
Linear
Log-Linear
Quadratic
Time
0e+00
2e+05
4e+05
6e+05
8e+05
1e+06
0
200
400
600
800
1000
Input size

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
- Or a problem is solved for specific cases

Graphic comparison: $O(n^2)$, $O(10^n)$, $O(n!)$

Graphic comparison: $O(n^2)$, $O(10^n)$, $O(n!)$

In [28]:

```
x <- 1:25
quadratic_complexity <- x ^ 2
exponential_complexity <- 10 ^ x
factorial_complexity <- factorial(x)
```

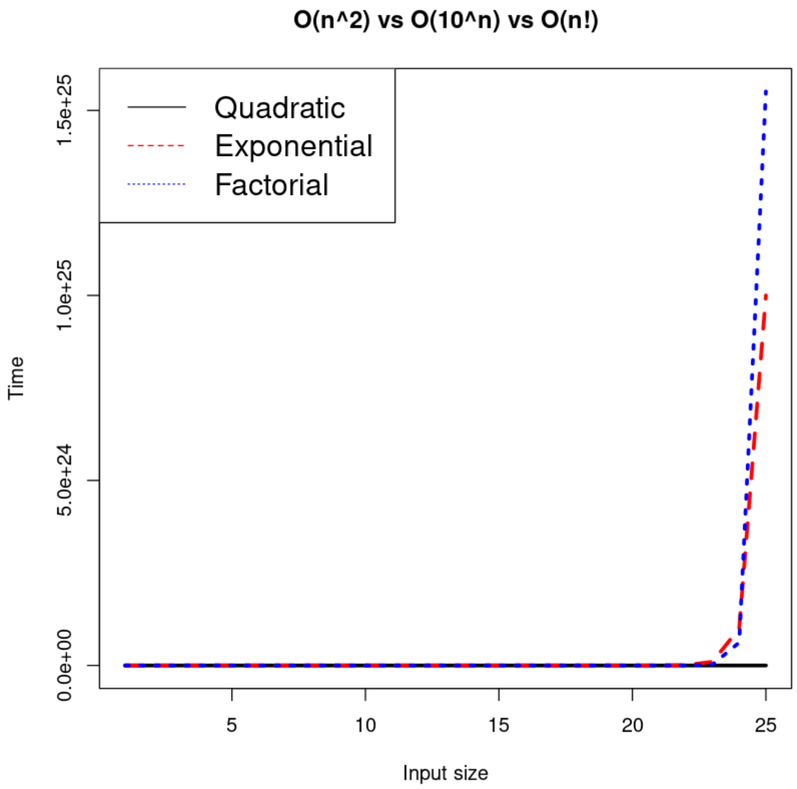
Graphic comparison: $O(n^2)$, $O(10^n)$, $O(n!)$

In [28]:

```
x <- 1:25
quadratic_complexity <- x ^ 2
exponential_complexity <- 10 ^ x
factorial_complexity <- factorial(x)
```

In [29]:

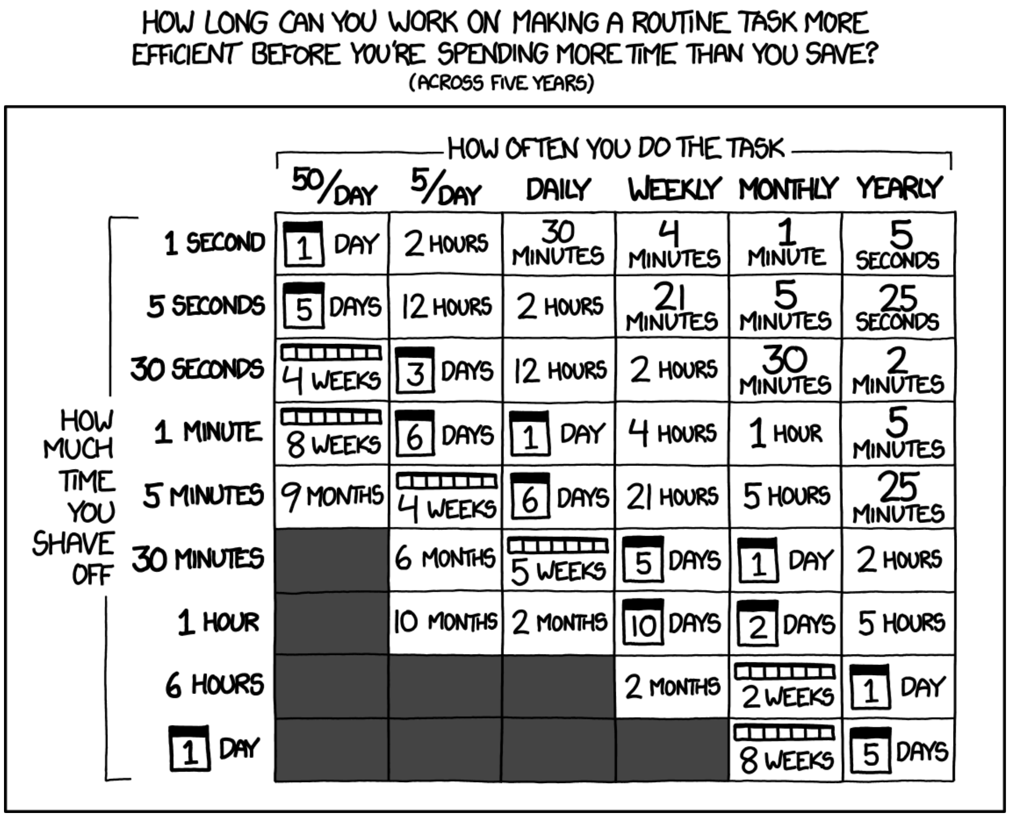
```
plot(x, factorial_complexity, type = "n", main = "O(n^2) vs O(10^n) vs
lines(x, quadratic_complexity, type = "l", col = "black", lty = 1, lwd
lines(x, exponential_complexity, type = "l", col = "red", lty = 2, lwd
lines(x, factorial_complexity, type = "l", col = "blue", lty = 3, lwd =
legend("topleft", legend = c("Quadratic", "Exponential", "Factorial"),
       col = c("black", "red", "blue"), lty = 1:3, cex = 1.5)
```


O(n^2) vs O(10^n) vs O(n!)
Quadratic
Exponential
Factorial
Time
Input size
0.0e+00
5.0e+24
1.0e+25
1.5e+25
5
10
15
20
25

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



Source: xkcd

Benchmarking and profiling in R

- Built-in function `system.time()` provides basic benchmarking functionality
- One problem is it runs each code snippet once, which can be sensitive to other factors
- Check `microbenchmark` for more advanced features (e.g. averaging over many runs)
- *Profiling* allows more granular instruction-specific timing
- RStudio has a built-in profiling functionality (through `profvis` package)

Extra: Profiling code with RStudio IDE

Code optimisation in R

- Find the biggest bottlenecks (parts of code that take most time to evaluate)
- Look for existing solutions to your problem (including built-in R functions)
- Avoid unnecessary copying
- Vectorise your code (more below)
- Parallelise your code
- Do not start optimising prematurely

Vectorised operations in R

- Many operations in R are *vectorized*
- They take vectors as input and produce vectors as output
- This is not just convenient, but often speeds up your code
- It eliminates extra function calls by relying on built-in routines
- Many base R functions are implemented in fast compiled C/Fortran code

Vectorised operations in R example

Vectorised operations in R example

In [30]:

```
# 1M can also be written as 1e6 in scientific notation (see above)
# We draw a random sample of 1M elements from a uniform distribution
# With min = 0 and max = 1
x <- runif(1000000)
y <- runif(1000000)
# And a vector for storing the results
z <- vector("double", length = 1000000)
```

Vectorised operations in R example

In [30]:

```r
# 1M can also be written as 1e6 in scientific notation (see above)
# We draw a random sample of 1M elements from a uniform distribution
# With min = 0 and max = 1
x <- runif(1000000)
y <- runif(1000000)
# And a vector for storing the results
z <- vector("double", length = 1000000)
```

In [31]:

```r
# Using built-in vectorized summation with `+` operator
system.time(z <- x + y)
```

```
   user  system elapsed 
  0.003   0.001   0.003 
```

Vectorised operations in R example

In [30]:

```r
# 1M can also be written as 1e6 in scientific notation (see above)
# We draw a random sample of 1M elements from a uniform distribution
# With min = 0 and max = 1
x <- runif(1000000)
y <- runif(1000000)
# And a vector for storing the results
z <- vector("double", length = 1000000)
```

In [31]:

```r
# Using built-in vectorized summation with `+` operator
system.time(z <- x + y)
```

```
   user  system elapsed 
  0.003   0.001   0.003 
```

In [32]:

```r
# Using an excplicit loop for summing up individual elements
system.time(for (i in seq_along(x)) z[i] <- x[i] + y[i])
```

```
   user  system elapsed 
  0.052   0.000   0.053 
```

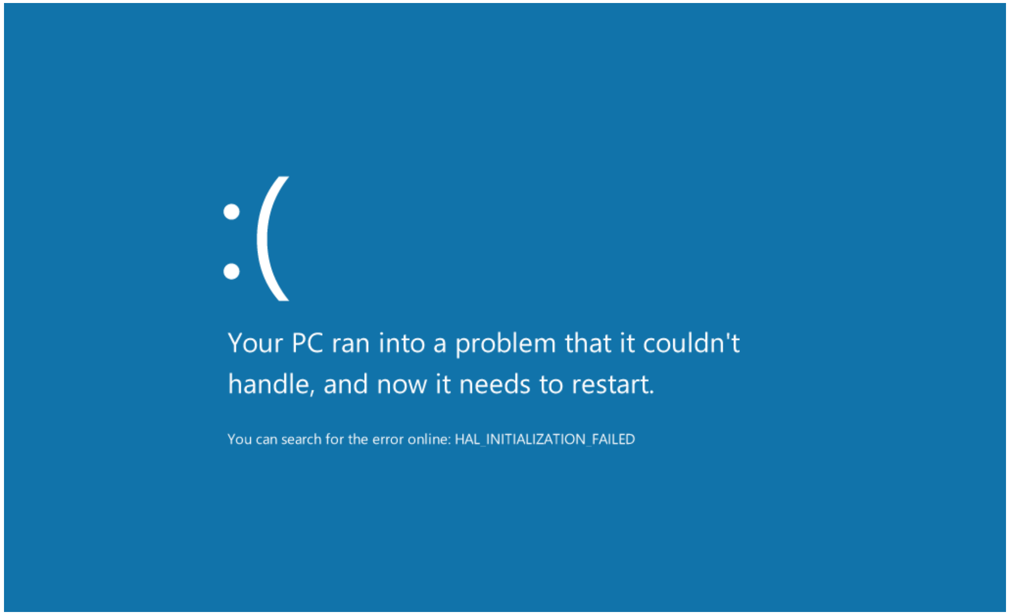
Vectorised functions in R

- All core arithmetic operators are vectorized (`+`, `-`, `*`, `^`, `/`, `%/%`, `%%`)
- Relational operators (`==`, `!=`, `>`, `>=`, `<`, `<=`)
- Matrix/data.frame summaries (`rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`)
- Some functionals (particularly, `lapply()`)
- Some other functions for common operations (e.g. `ifelse()` , `which()`, `cumsum()`)

Next

- Tutorial: Benchmarking, analysis of function complexity and performance
- Final project: Due at 12:00 on Monday, 19th 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