

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

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:

```
1 linear_search <- function(v, x) {  
2   n <- length(v)  
3   for (i in seq_len(n)) {  
4     if (v[i] == x) {  
5       return(TRUE)  
6     }  
7   }  
8   return(FALSE)  
9 }
```

In Python:

```
1 def linear_search(v, x):  
2   n = len(v)  
3   for i in range(n):  
4     if v[i] == x:  
5       return True  
6   return False
```

Benchmarking

Benchmarking in R

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

user	system	elapsed
0.003	0.000	0.003

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

user	system	elapsed
0.018	0.000	0.018

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

user	system	elapsed
0.035	0.000	0.036

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

user	system	elapsed
36.234	0.007	36.242

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.

```
1 linear_search <- function(v, x) {  
2   n <- length(v) # 2 steps (call to length() function and assignment `<-`)  
3   # n steps + 1 step (for seq_len(n) call)  
4   for (i in seq_len(n)) {  
5     # 3n steps (n steps for `[`(v, i), another n steps for `==` and n calls to `if`)  
6     if (v[i] == x) {  
7       return(TRUE) # 1 step  
8     }  
9   }  
10  return(FALSE) # 1 step  
11 }
```

- $4n + 4$
- 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

```
1 # Binary search for sorted sequences
2 binary_search <- function(v, x) {
3   low <- 1
4   high <- length(v)
5   while (low <= high) {
6     # Calculate mid-point (similar to median)
7     m <- (low + high) %/% 2
8     if (v[m] < x) {
9       low <- m + 1
10    } else if (v[m] > x) {
11      high <- m - 1
12    } else {
13      return(TRUE)
14    }
15  }
16  return(FALSE)
17 }
```

Comparing Algorithms: Benchmarking

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

user	system	elapsed
0.036	0.000	0.036

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

user	system	elapsed
0.006	0.000	0.005

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

user	system	elapsed
0.003	0.003	0.007

Comparing Algorithms: N Steps

```
1 linear_search <- function(v, x) {  
2   n <- length(v)  
3   for (i in seq_len(n)) {  
4     if (v[i] == x) {  
5       print(paste0("Number of iterations: ", as.character(i)))  
6       return(i) # return(TRUE)  
7     }  
8   }  
9   return(i) # return(FALSE)  
10 }
```

```
1 binary_search <- function(v, x) {  
2   low <- 1  
3   high <- length(v)  
4   iters <- 1  
5   while (low <= high) {  
6     m <- (low + high) %/% 2  
7     if (v[m] < x) {  
8       low <- m + 1  
9     } else if (v[m] > x) {  
10      high <- m - 1  
11    } else {  
12      print(paste0("Number of iterations: ", as.character(iters)))  
13      return(iters) # return(TRUE)  
14    }  
15    iters <- iters + 1  
16  }  
17  return(iters) # return(FALSE)  
18 }
```

Comparing Algorithms: N Steps

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

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

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

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

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

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

```
1 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**rders 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

```
1 get_len <- function(v) {  
2   # Internally, length just returns the 'length' attribute of an R object  
3   n <- length(v)  
4   return(n)  
5 }
```

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

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

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

```
user  system elapsed  
0.002  0.000  0.001
```

```
1 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

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

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

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

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

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

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

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

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

```
1 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).

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

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

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

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

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

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

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

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

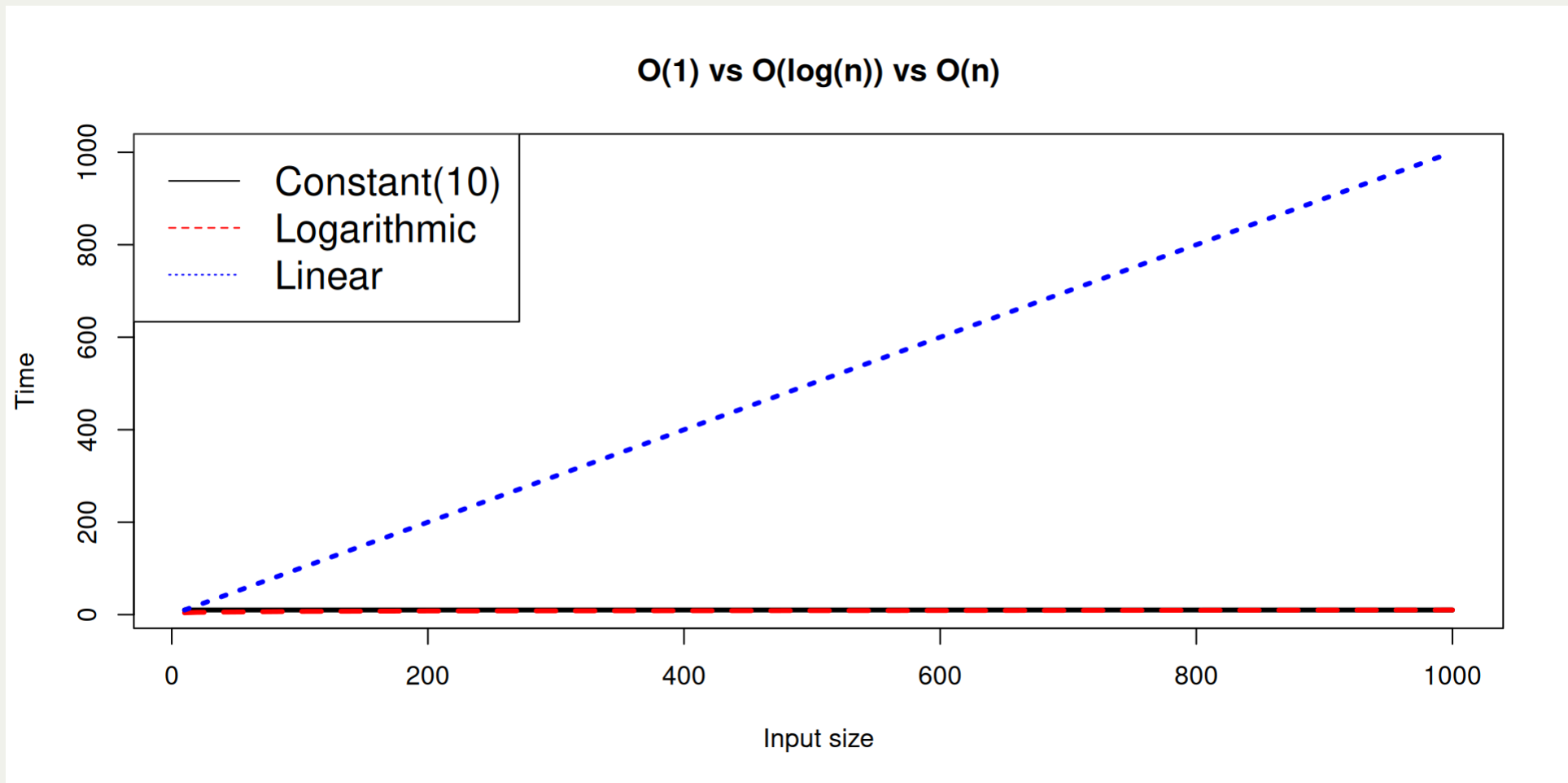
```
1 ls_1000 <- linear_search(1:1000, 1000)
```

```
[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.

```
1 # Check whether one vector is contained within another vector
2 is_subset <- function(v1, v2) {
3   n <- length(v1)
4   m <- length(v2)
5   iters <- 1
6   for (i in seq_len(n)) {
7     matched <- FALSE
8     for (j in seq_len(m)) {
9       iters <- iters + 1
10      if (v1[i] == v2[j]) {
11        matched <- TRUE
12        break
13      }
14    }
15    if (isTRUE(matched)) {
16      print(paste0("Number of iterations: ", as.character(iters)))
17      return(iters) # return(TRUE)
18    }
19  }
20  print(paste0("Number of iterations: ", as.character(iters)))
21  return(iters) # return(FALSE)
22 }
```

Polynomial Complexity

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

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

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

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

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

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

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

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

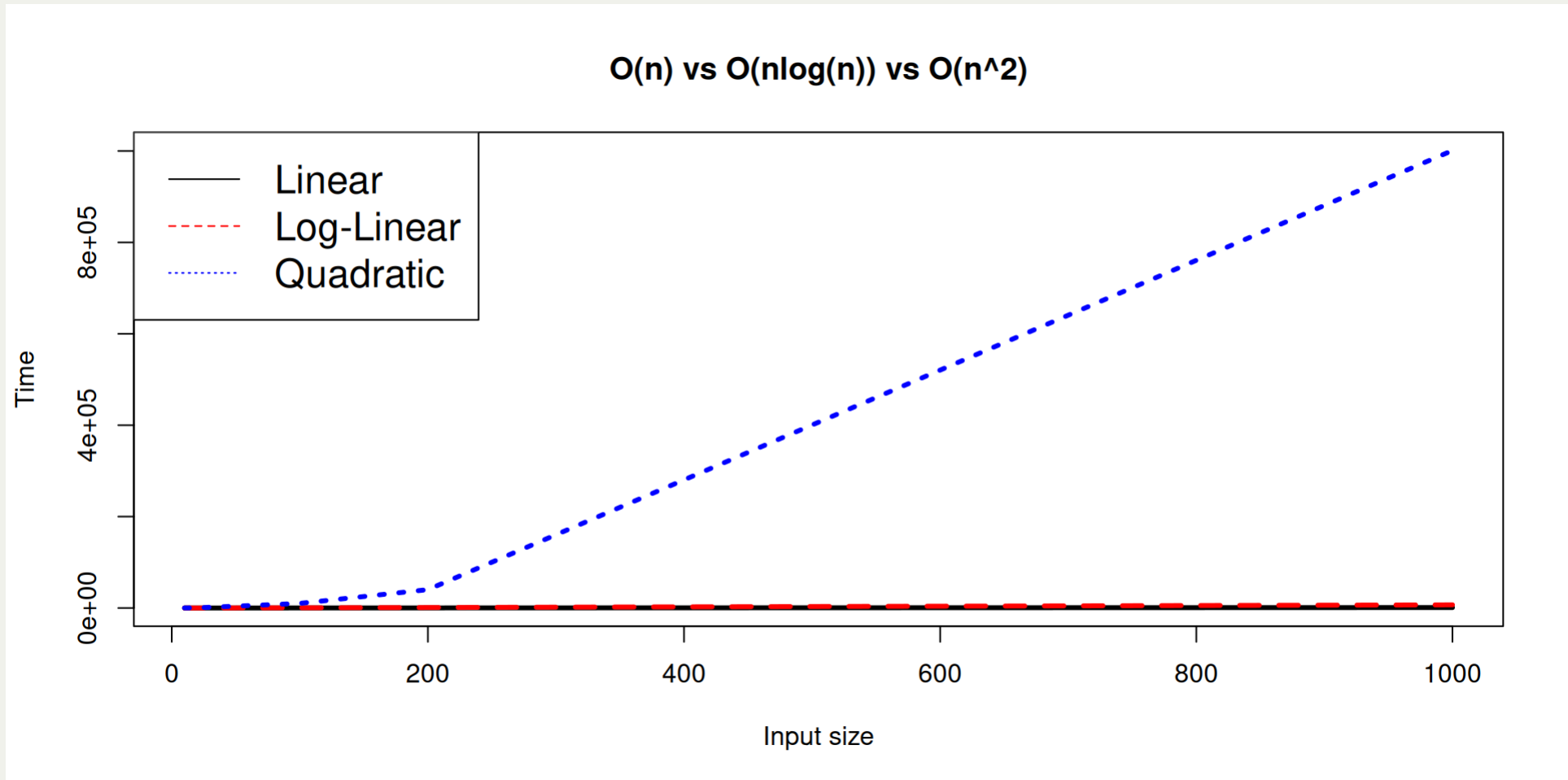
```
1 is_1000 <- is_subset(1001:2001, 1:1000)
```

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

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

Plot

Code



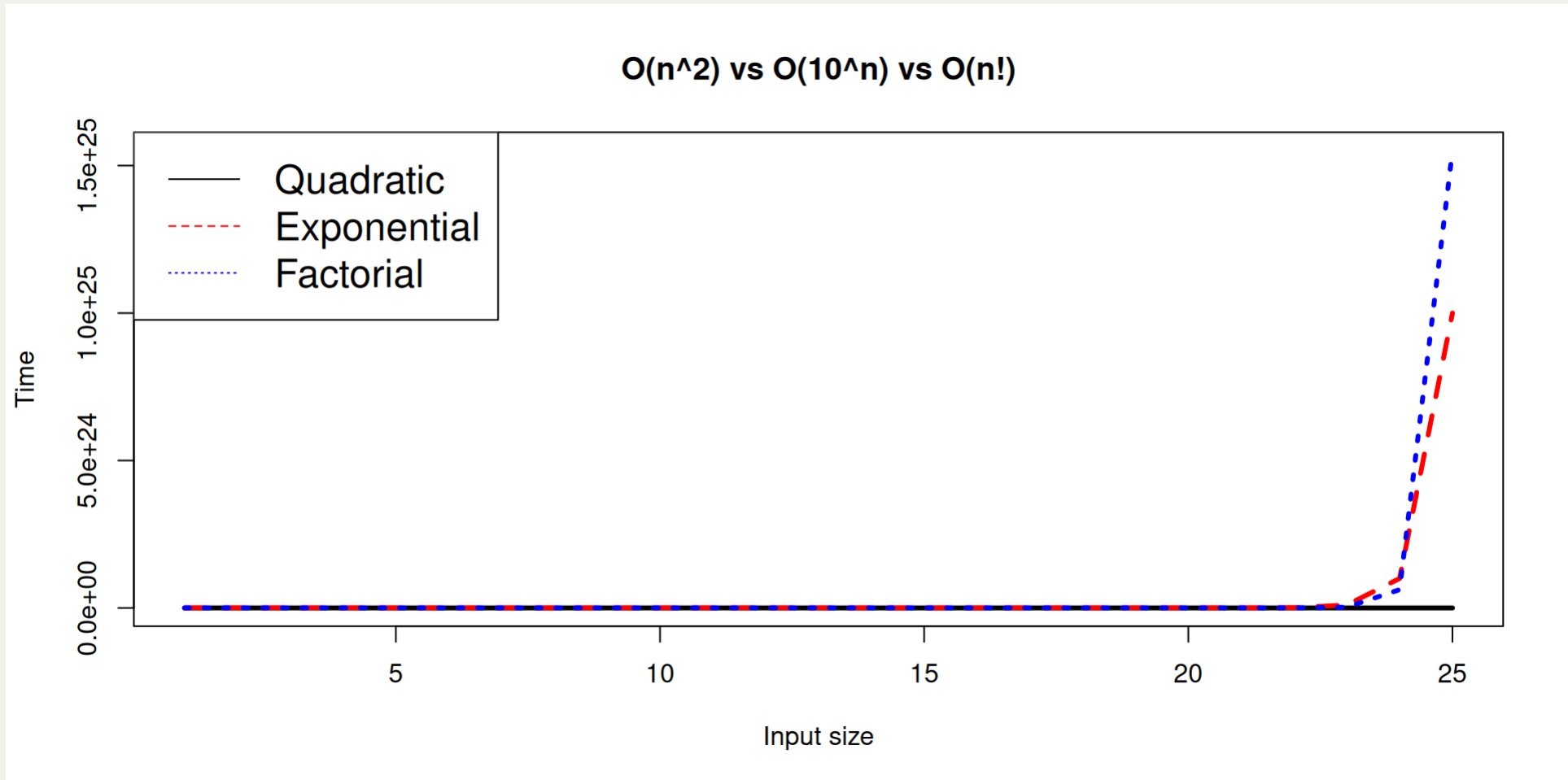
Exponential and Factorial Complexity: $O(c^n)$ and $O(n!)$

- 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
- Or a problem is solved for specific cases

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

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	1 DAY
	1 DAY					8 WEEKS	5 DAYS

(xkcd)

Next

- Tutorial: Benchmarking, analysis of function complexity and performance
- Final project: Due by 23:59 on Friday, 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`