High Performance Computing in Julia

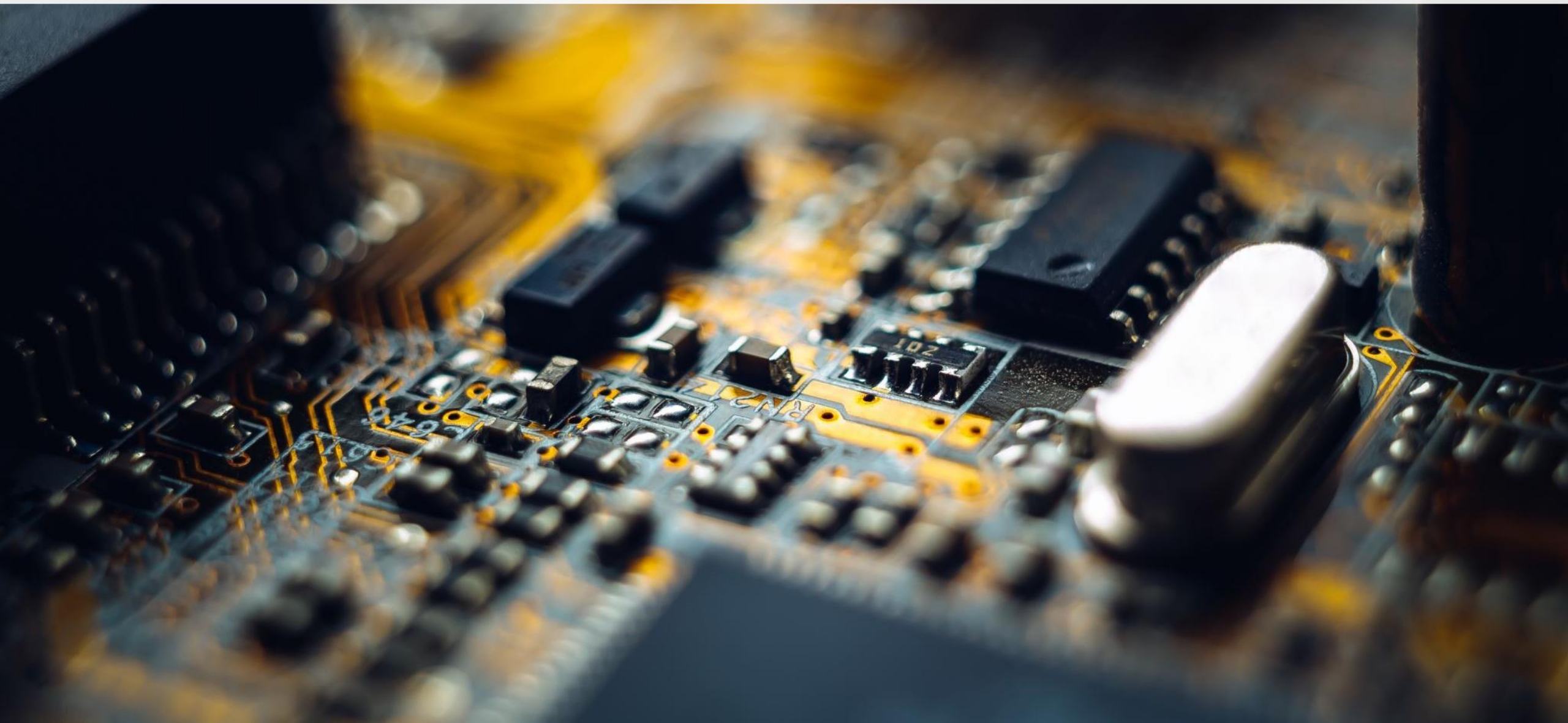
from the ground up.

Measuring Performance & Optimisation

Aims

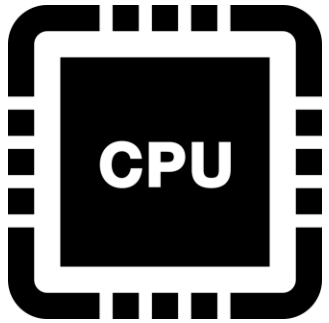
- To be able to **profile** and **benchmark** Julia code
- To understand the basics of **computational complexity**
- To begin learning some **optimisation** techniques (in Julia)

Measuring Performance



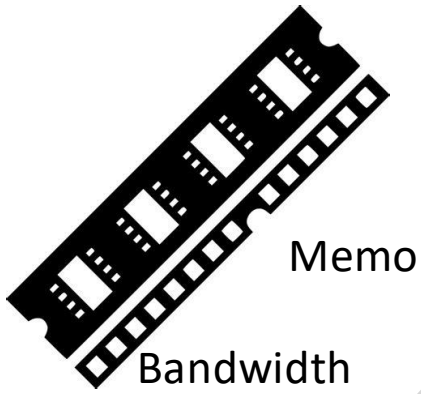
Which resources can a program consume?

Cache



Cores

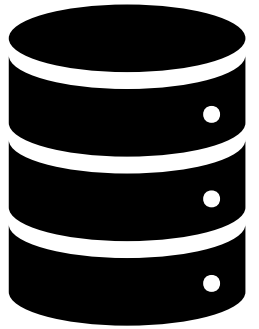
Clock Cycles



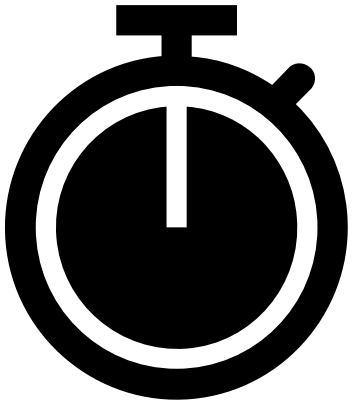
Memory

Bandwidth

Storage



Disk I/O



Wall Time

Network I/O

Measuring Wall Time

- The CPU has an internal clock – used to synchronise action across the CPU
- Can be used to measure how long a piece of code takes by comparing the clock before and after execution
- Can use the `@time` macro for simplicity, but this is not very accurate

Example: Sum of cubes

```
julia> f(arr) = sum(x->x^3, arr);
julia> arr = rand(1024);
julia> @time f(arr)
  0.042326 seconds (56.16 k allocations: 3.067 MiB, 99.96% compilation time)
251.11661067703318
julia> @time f(arr)
  0.000005 seconds (1 allocation: 16 bytes)
251.11661067703318
```

- First usage includes **compile time** - always measure twice

BenchmarkTools.jl

$ for **interpolation**

```
julia> using BenchmarkTools

julia> @benchmark f($arr)
BenchmarkTools.Trial: 10000 samples with 960 evaluations.
 Range (min … max):   86.071 ns … 117.006 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):      88.076 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):    88.893 ns ±   1.341 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  86.1 ns        Histogram: log(frequency) by time        93.8 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

<u>More **accurate** results</u>

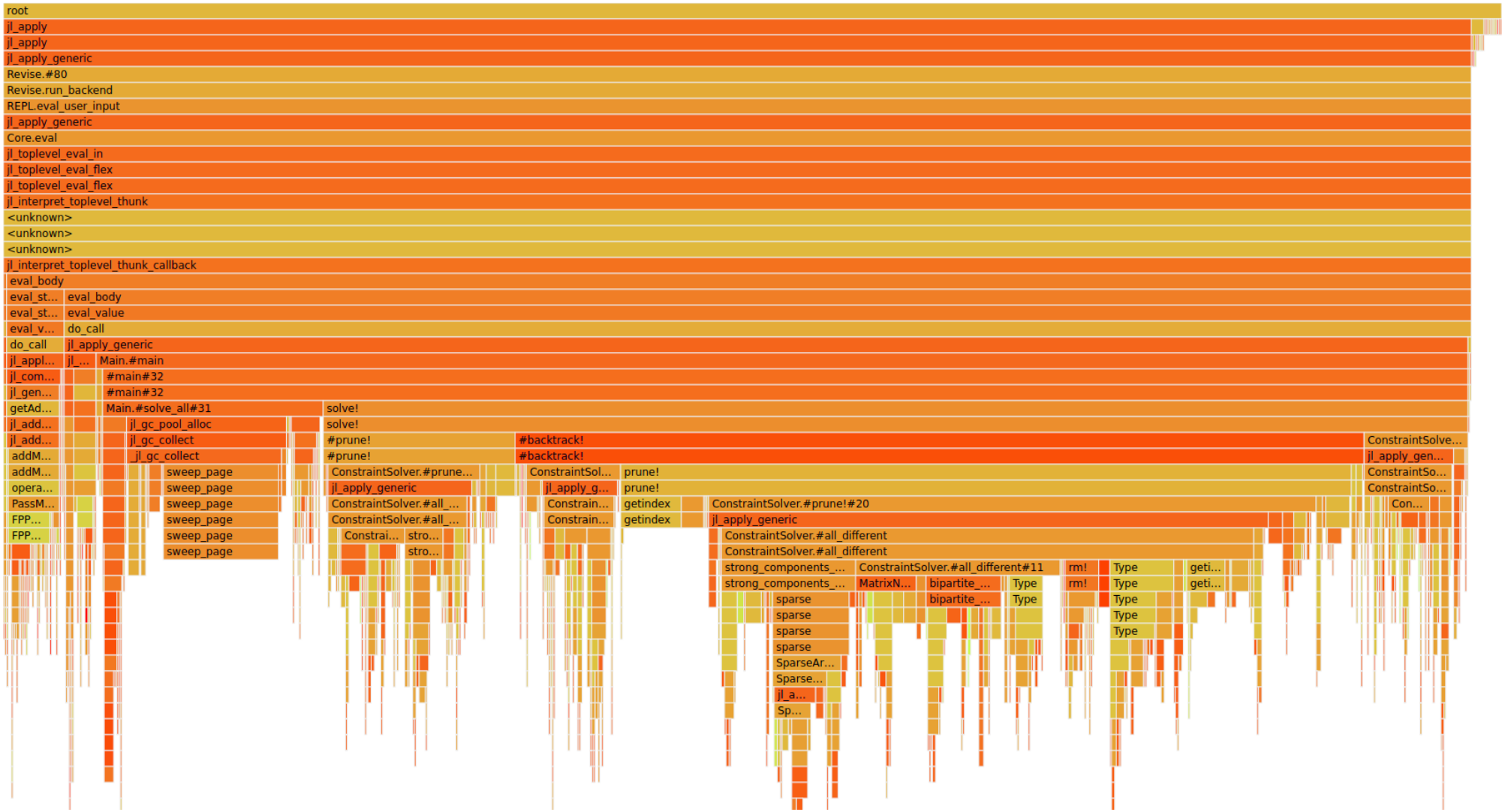
Why Benchmark?

- Time taken to execute code can be **highly** variable
- Some variability can be due to **scheduling** on the CPU, tasks may be **interrupted** while processing
- CPU thermals may cause it to **lower the clock speed** to avoid damage
- **Boost clocks** are common on modern CPUs, which is turned off when multiple cores are used

Profiling

- **Statistical profilers** sample your program **during** execution
- Interrupts the program and takes note of where in a stack frame (function) the program is
- Can infer which pieces of code take the **longest**, as they have the **most samples**
- Can show results as a **flame graph**

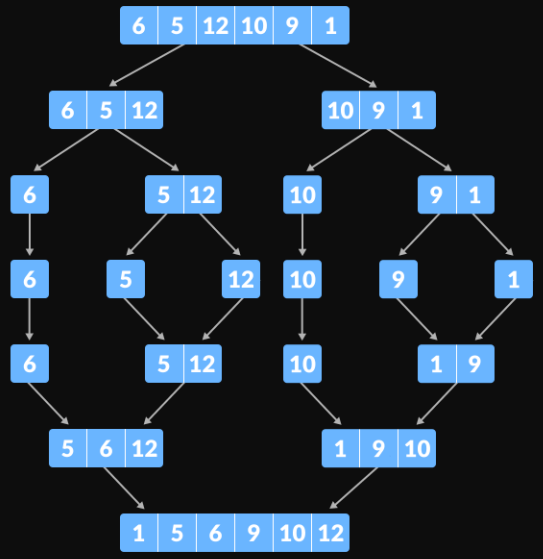
Profiling: Flame Graph

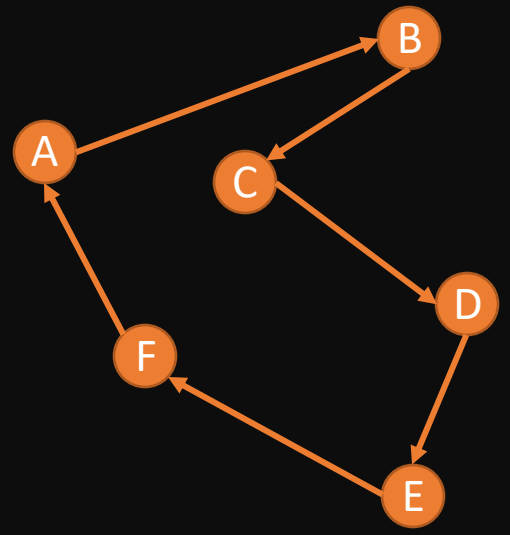


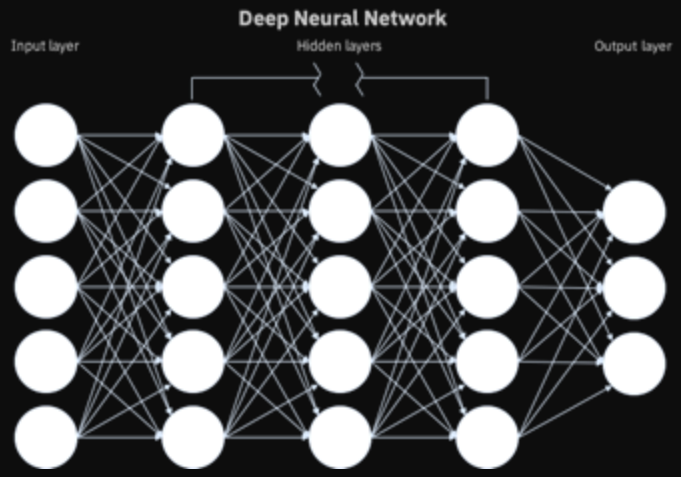
Profiling + Optimising Technique

- Use profiling to **identify** the slow functions in your code
- Use macros from ***BenchmarkTools.jl*** to measure the performance of the slow functions
- Only optimise the part of your code that is **slow**!
- Keep the old code for reference to compare benchmarks
- Keep hardware the **same**!
- Minimise number of **concurrent tasks**

Computational Complexity – Big $\mathcal{O}$ notation







Computational Complexity

- All computers have different hardware, with varying speed
- How do you compare algorithms on different hardware?
- We compare the **computational complexity** of the algorithms, which measures how execution time will **scale** with varying input sizes
- The **complexity** classifies the number of resources required to run it, mostly focusing on computation time (or memory storage).

Computational Complexity

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

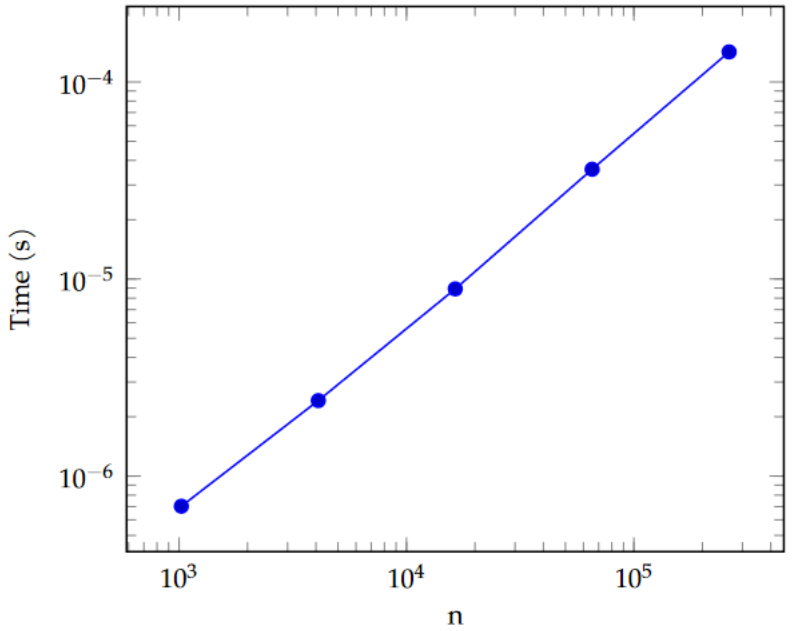


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

Computational Complexity

$$\log(t - t_0) = m \log n + c$$

$$t = e^c n^m + t_0$$

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

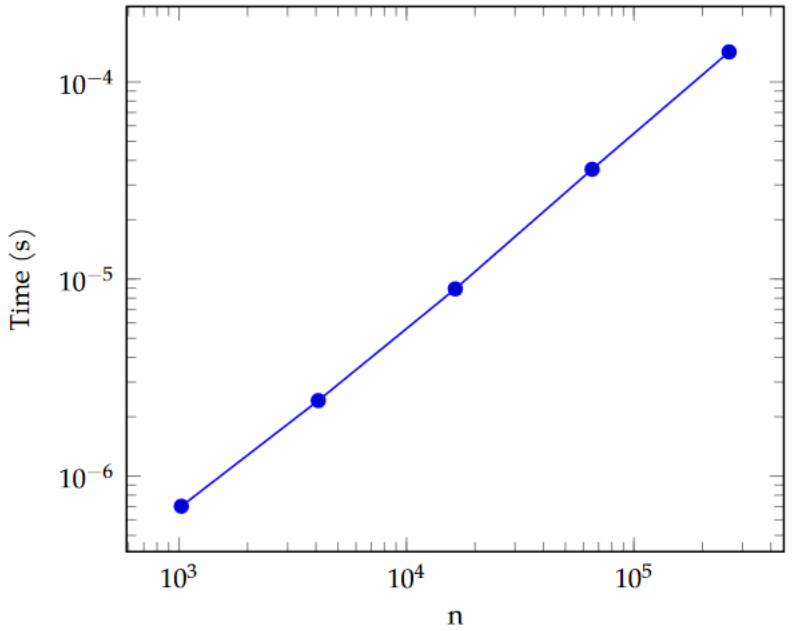


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

Computational Complexity

$$\log(t - t_0) = m \log n + c$$

$$t = e^c n^m + t_0$$

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

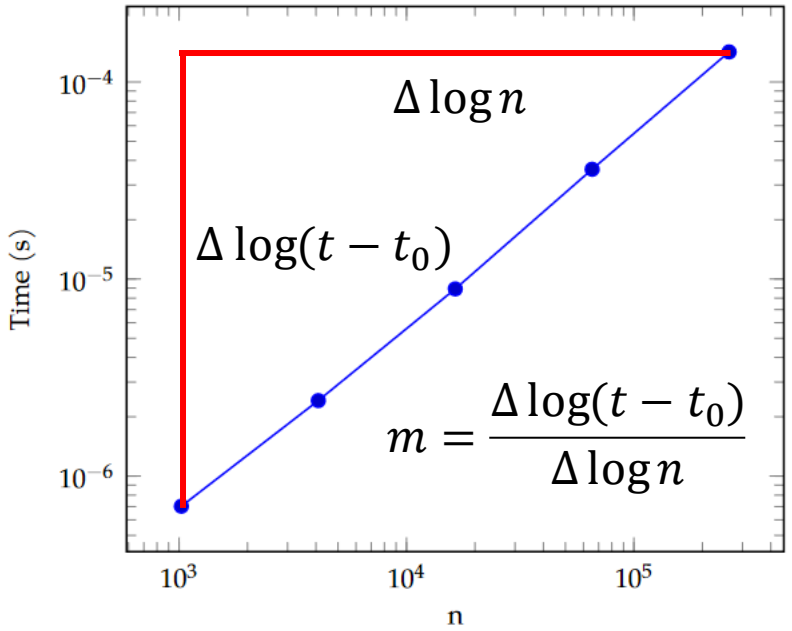


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

Computational Complexity

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

$$\log(t - t_0) = m \log n + c$$

$$t = e^c n^m + t_0$$

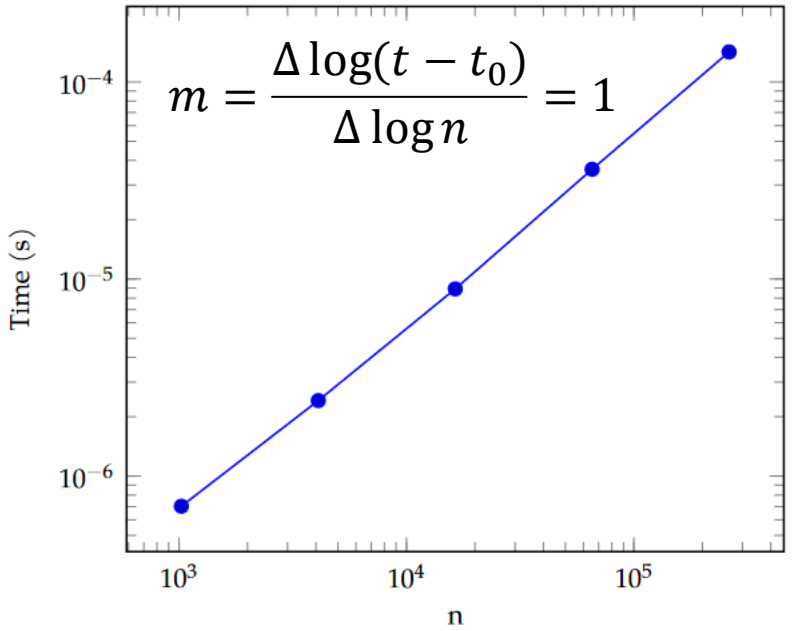


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

Computational Complexity

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

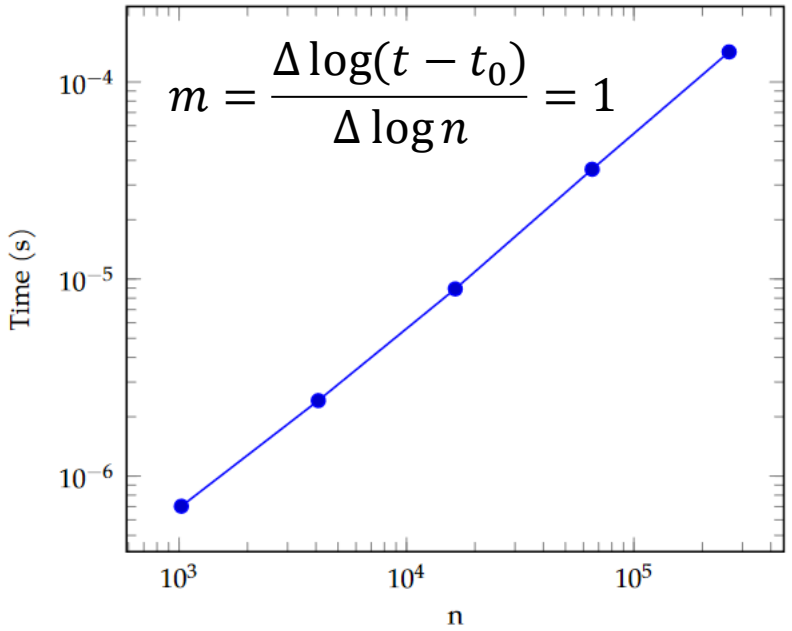


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

$$t(n) = e^c n^m + t_0$$

Computational Complexity

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

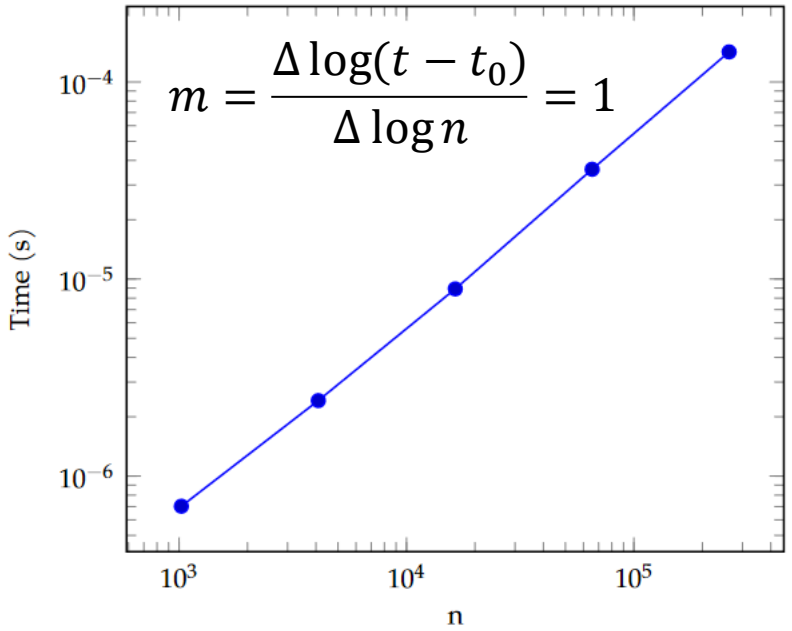


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

$$t(n) = an + b$$

Computational Complexity

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```

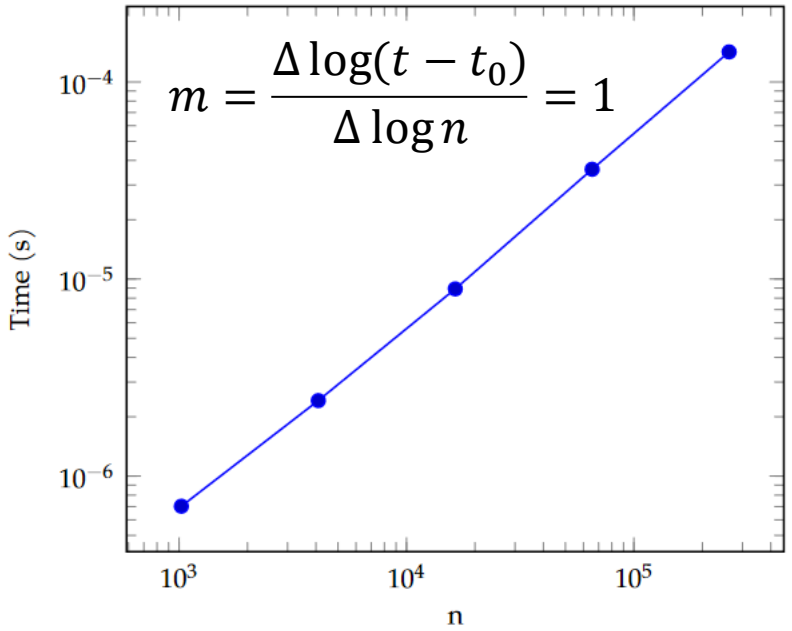


Figure 5.4. Time taken when using `elementmul` for several values of `n`. The time taken is measured several times for each value of n using the `@belapsed` macro from *BenchmarkTools.jl*.

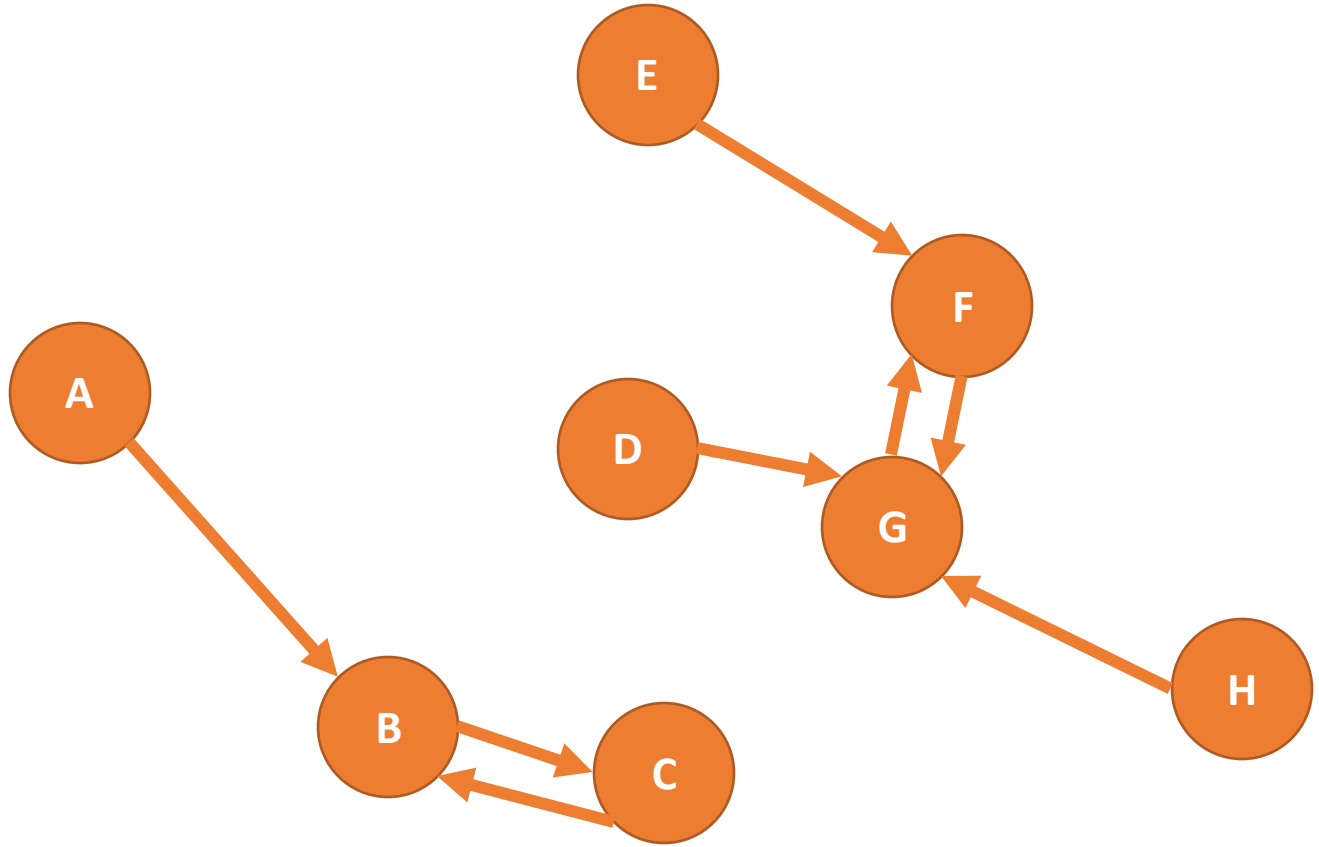
Big $\mathcal{O}$ notation

$$t(n) = \mathcal{O}(n)$$

Big $\mathcal{O}$ notation

- **Asymptotic** Computational Complexity – only keep largest growing factors, and throw away constants
- Usually deals with the **worst-case** complexity, but sometimes **average-case** is also interesting
- Is interesting for problems which **scale**, if problem size is small, benchmarking is often preferred
- Helps to **choose** the right algorithm for a given problem size

Example – Nearest Neighbour



Node	Neighbour
A	B
B	C
C	B
D	G
E	F
F	G
G	F
H	G

Example – Nearest Neighbour

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(1)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n)$
(assumption)

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n)$

$\mathcal{O}(1)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n)$

$\mathcal{O}(1)$

$\mathcal{O}(n)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n)$

$\mathcal{O}(n)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n)$

$\mathcal{O}(n^2)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n^2)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Nearest Neighbour

$\mathcal{O}(n^2)$

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Example – Recursive calls

```
function recursivecalc(n, a)
    if n==1
        return a*a
    end

    s = 0.0
    for i in 1:n
        s += recursivecalc(n-1, i)
    end

    return s
end
```

$$\mathcal{O}(n!)$$

Example – Sorted Insert

```
function insertintosorted!(numbers, num)
    n = length(numbers)
    startpoint = 1
    endpoint = n
    while startpoint-endpoint > 1
        midpoint = (startpoint+endpoint)÷2
        if num < numbers[midpoint]
            endpoint = midpoint
        elseif num > numbers[midpoint]
            startpoint = midpoint
        else
             insert!(numbers, midpoint, num)
             return
        end
    end
    insert!(numbers, startpoint, num)
    nothing
end
```

Algorithm 5.6. A simple algorithm which inserts a number into an already sorted list, making sure the list is still sorted after insertion. This algorithm assumes the list is sorted in ascending order.

$$\mathcal{O}(\log n)$$

Summary

- Choose the **best algorithm** for your use case
- **Constants matter** when the problem size is smaller
- Should always benchmark the code to get an idea of the constants
- Can use a **combination** of complexity and benchmarking to predict how long execution will take for a larger input / work out the maximum input size

Optimisation

Why optimise our code?

Get results **faster** – quickly iterate on our experiments

Increase of your own **productivity**

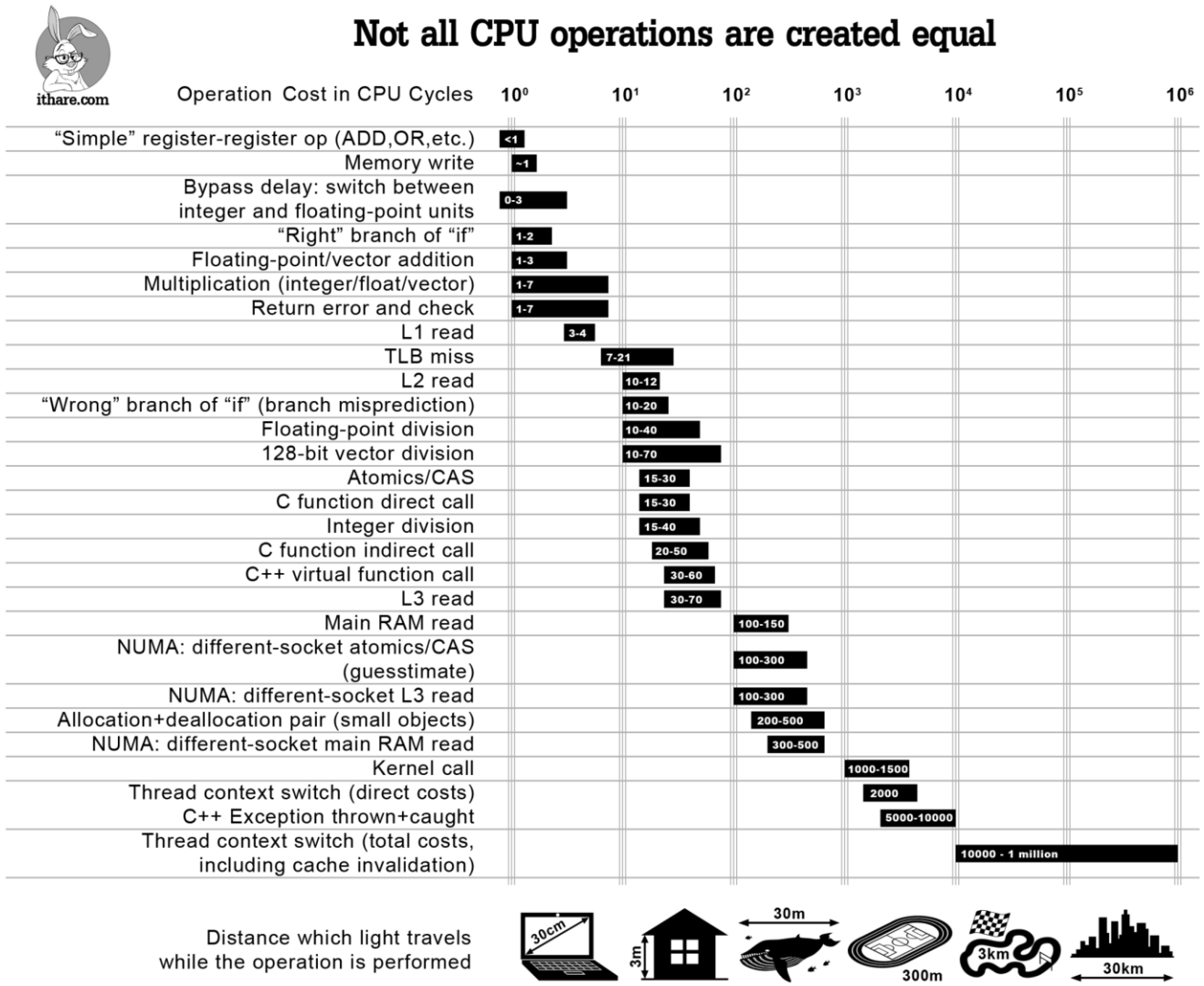
Scale up the experiments to **larger** sizes and **longer** times

Use fewer compute resources on the HPC

Why is code **slow** in the first place?

- **Incorrect** algorithms are chosen
- Contains **unnecessary** operations
- Program has to spend time working out what to do, instead of having the instructions ready (not compiled)
- Choices are made to cooperate with language design, but cause poor performance (i.e. must vectorise code with numpy)

Speed of operations



Example: Vectorisation

Native Python

```
import math

def f_native(x_values):
    y_values = []
    for x in x_values:
        y = math.exp(x)*math.sin(x)*x + x**4 + 5 * math.sqrt(3*x)
        y_values.append(y)
    return y_values
```

```
52.5 ms ± 446 µs per loop
(mean ± std. dev. of 7
runs, 10 loops each)
```

Numpy

```
import numpy as np

def f_np(x):
    return np.exp(x)*np.sin(x)*x + x**4 + 5 * np.sqrt(3*x)
```

```
3.16 ms ± 29.2 µs per loop
(mean ± std. dev. of 7
runs, 10 loops each)
```

$\sim 16x$ speedup

Example: Vectorisation

Native Julia

```
function f_vectorised(x)
    y = similar(x)
    @. y = exp(x)*sin(x)*x + x^4 + 5 * sqrt(3*x)
    return y
end
function f_native(x_array)
    y = similar(x_array)
    @inbounds for i in eachindex(x_array)
        x = x_array[i]
        y[i] = exp(x)*sin(x)*x + x^4 + 5 * sqrt(3*x)
    end
    return y
end
```

f_vectorised -> 942.500 µs (2 allocations: 512.11 KiB)

f_native -> 900.000 µs (2 allocations: 512.11 KiB)

$\sim 3.5x$ speedup

Numpy

```
import numpy as np
def f_np(x):
    return np.exp(x)*np.sin(x)*x + x**4 + 5 * np.sqrt(3*x)
```

```
3.16 ms ± 29.2 µs per loop
(mean ± std. dev. of 7
runs, 10 loops each)
```

Example: Vectorisation

Julia

- Julia will **fuse** broadcast operations together
- Broadcasting does not allocate **intermediate** results
- Vectorised vs for loop is aesthetic
- Compiler can **automatically** simd the code
- Does not need to pass the array through the language barrier – all native Julia

Numpy

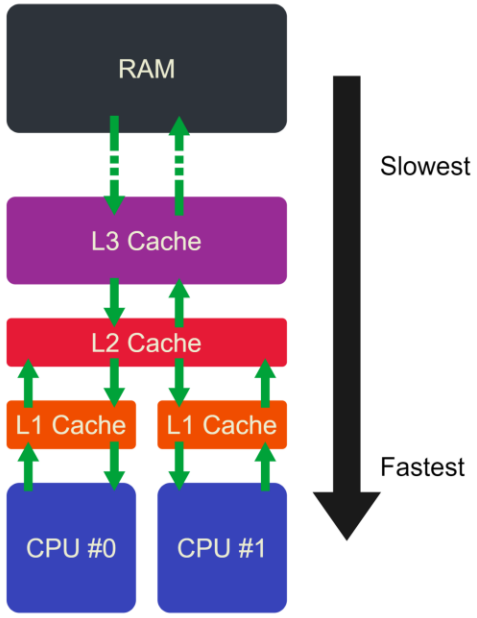
- Requires **rewriting** the code into a vectorised loop
- Moves the for loop into a compiled function (in C)

Hardware SIMD vs Vectorisation

- **@simd** just gives the compiler more leeway to use hardware level vector instructions (SSE and AVX) on operations that may change results
- Compiler will **automatically** hardware vectorise code in specific cases
- Read the help for more info: help?> @simd

Cache and Memory Locality

- All arrays are stored in a **contiguous** block of memory
- Operations on an array done in order are much faster due to **cache**
- **Cache lines** store an entire line of memory (~128 bits)



Address (Hex)	Data
…	…
5F19	01001001
5F1A	00001000
5F1B	10000001
5F1C	00011110
…	…

Multidimensional Arrays

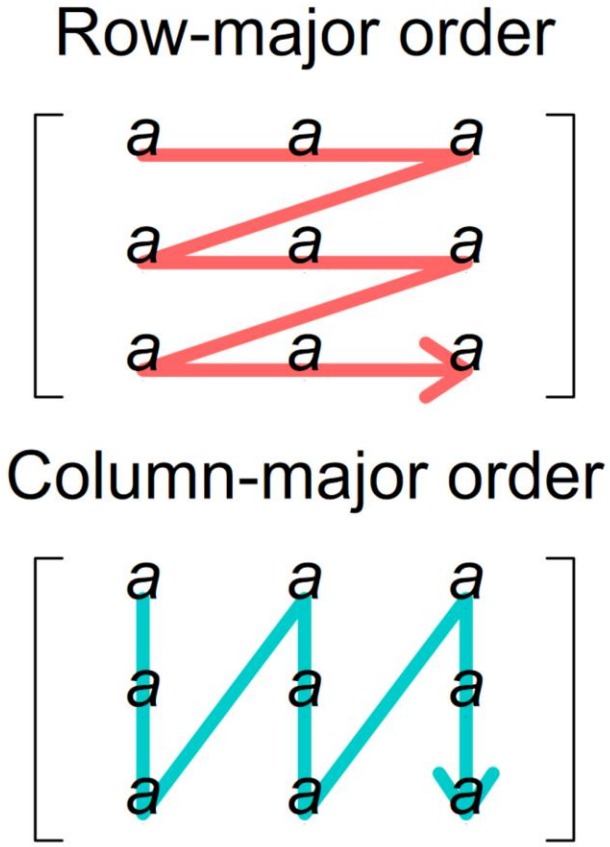
- Even multidimensional arrays are stored linearly in memory
- Indexing scheme must be used to calculate linear index from cartesian index
- **Row-major:**

$$k = jN_y + i$$

- **Column-major:**

$$k = iN_x + j$$

- Julia uses **column-major**



Multidimensional Arrays

- Order of iteration makes a **big** performance difference

```
julia> @benchmark row_major_matrix_add!($C, $A, $B)
BenchmarkTools.Trial: 184 samples with 1 evaluation.
 Range (min … max):  26.970 ms …  27.820 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     27.200 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   27.218 ms ± 133.593 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
 Memory estimate: 0 bytes, allocs estimate: 0.

julia> @benchmark column_major_matrix_add!($C, $A, $B)
BenchmarkTools.Trial: 4000 samples with 1 evaluation.
 Range (min … max):  1.187 ms …  1.447 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.215 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.230 ms ± 36.353 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
function row_major_matrix_add!(C, A, B)
    @assert size(C)==size(A)==size(B)
    @inbounds for i in axes(A, 2)
        for j in axes(A, 1)
            C[i, j] = A[i, j] + B[i, j]
        end
    end
    nothing
end
```

```julia
function column_major_matrix_add!(C, A, B)
    @assert size(C)==size(A)==size(B)
    @inbounds for j = axes(A, 2)
        for i in axes(A, 1)
            C[i, j] = A[i, j] + B[i, j]
        end
    end
    nothing
end
```

Multidimensional Arrays

- Julia arrays can be **linearly** indexed – same performance

```
julia> @benchmark vector_add!($C, $A, $B)
BenchmarkTools.Trial: 3984 samples with 1 evaluation.
 Range (min … max):  1.193 ms …  1.599 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.215 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.235 ms ± 47.159 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
function vector_add!(C, A, B)
    @inbounds for i in eachindex(C, A, B)
        C[i] = A[i] + B[i]
    end
    nothing
end
```

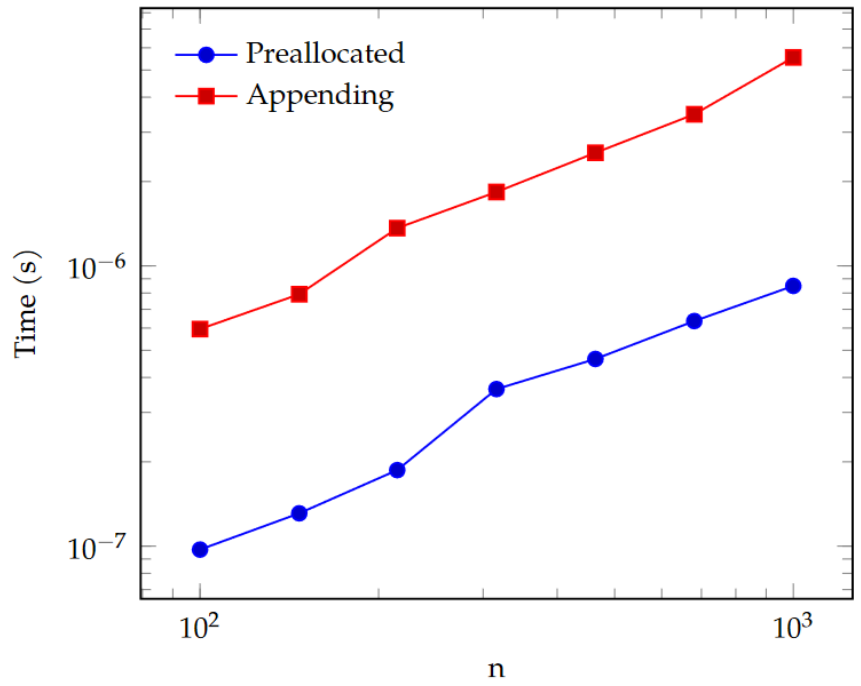
Heap Allocations

- When we say **“allocations”**, we refer to **heap** allocations
- Allocating memory on the heap is costly
- Memory needs to be cleaned up by the **Garbage Collector (GC)**
- The GC will interrupt processing to clean up memory – performance hit!

Preallocating vs Appending

```julia
function cumulative_sum_preallocated(numbers)
    results = similar(numbers)
    total_sum = zero(eltype(numbers))
    for i in eachindex(numbers)
        total_sum += numbers[i]
        results[i] = total_sum
    end
    return results
end
```

```julia
function cumulative_sum_appending(numbers)
    results = (eltype(numbers))[]
    total_sum = zero(eltype(numbers))
    for i in eachindex(numbers)
        total_sum += numbers[i]
        push!(results, total_sum)
    end
    return results
end
```



Reusing Memory: Caching

- Can pre-allocate storage for the results
- Still need to **allocate** the memory, but only needs to be done **once**
- Can make a difference on speed, but also memory

```
function example_equation_no_cache(x)
    numerator = 5 .* x .^ 5 .* sin.(x.^2) .+ 20
    denominator = exp.(-4 .* x) .- x .^ 2
    y = numerator ./ denominator
    return y
end
```

```
function example_equation_cache!(y, x)
    # Set y to the value of the numerator
    y .= 5 .* x .^ 5 .* sin.(x.^2) .+ 20
    # Divide out the denominator
    y ./= exp.(-4 .* x) .- x .^ 2
    return y
end
```

Reusing Memory: Avoid Copying

- Slicing an array allocates new memory for that array
- We can instead create a **view** into that data to point at the same memory
- Changing values in a view will propagate back to the original array
- *Sometimes* copying may be **faster** due to memory locality

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point_i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min_distance_squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance_squared = sum((point_i .- point_j).^2)
            if min_distance_squared < distance_squared
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Avoiding the Heap – Stack Allocating

- Can use packages like ***StaticArrays.jl*** to create small arrays on the stack
- Can be used to make the code more readable, and avoiding allocations
- These are **immutable** by default
- Useful for small vectors representing positions/velocities etc

Profiling & Reducing Allocations

Live Demonstration

Workshop – Thursday 26/01/2023

Assignment

Link will be on the website

Tasks:

- Optimise the nearest neighbour algorithm
- Read through the README for assignment details