Unit 3: On Tools and Algorithms

Table of contents

1 Unit 3: On Tools and Algorithms

In the previous unit we got straight into general programming basics as well as specifics with Julia. We now know basic syntax, variables, conditional statements, loops, functions, and more. We spoke about bits, bytes, and representation of integers on a computer. In terms of tools we used Jupyter and the REPL.

In this unit we'll get a taste for basic mathematical analysis of algorithms and quantification of behavior and performance. We'll also deal with more extensive tools including Git and GitHub, Visual Studio Code (IDE), and basic Unix command line.

We'll start with tooling and then move onto a basic demonstrations of the fact that the performance of algorithms can be analyzed mathematically.

1.1 More tools

We now spend more time on "tools", namely,

- Unix command line
- Git and GitHub
- The Visual Studio Code IDE

1.1.1 Unix command line

An operating system is software that manages the other software and hardware of the computer. It manages:

- Files
- Computer memory (RAM)
- Running programs, processes (tasks), and sometimes threads within a task
- Input/Output

- Networking
- More …

This includes low level interfaces between the running programs and hardware, as well as its own "programs" for interfacing and managing the system (e.g. graphical user interface for files, and importantly for us the **shell**).

Unix is a classic operating system still very much in use today. Mac's operating system macOS is based on Unix. Similarly, Linux is a Unix based system. The most popular desktop family of operating systems, Windows is not Unix based.

Our focus is on the **shell**, or **command line interface** which allows to execute simple and complicated tasks via command line. In Windows this is called the "command line". You can watch this (one of many) video tutorial series about the windows command line. However, you are not required to build up "command line" knowledge as part of this course. Instead if you are a Windows user, we recommended you use GitBash which gives you "Unix command line emulation". If you are a Mac user, you will simply run "Terminal", and on Linux you will run "Shell".

Note that when you run a Julia REPL, hitting `;` puts you in "shell mode" where the system's default shell is present.

We won't become shell experts but only learn to carry out a few basic tasks. Our main use case will be using Git source control software (see more below). For this, we'll also learn about basic file manipulation commands and a few more tid bits will appear as needed. We assume you are familiar with the fact your system has files, and folders (directories). Here are more things to be aware of:

- `~` stands for the folder of your user.
- `/` stands for the root directory.
- `.` stands for the current directory (there is always the notion of the current directory for our process/program).
- `..` stands for the parent directory.

Key file navigation commands are:

- `ls` show files.
- `cd` change current directory.
- `pwd` print working directory.

Key file manipulation commands are (be careful!):

- `rm` remove.
- `rmdir` remove (empty directory).
- `cp` copy.
- `mv` move (also rename).

A few more useful commands (apps): * `echo` print to screen (can be used as part of a shell script). * `cat` print a text file. * `head` print with first 10 lines of a text file * `tail` print with last 10 lines of a text file * `diff` compare two files line by line * `grep` search within a file. * `vim` and `emacs` text editors

Commands work well with wildcards/regular expressions. A very common one is `*`.

Output of commands can be **piped** into other commands using `|`, similarly it can be redirected into file with `>` or `>>` for appending.

There are some commands dealing with users and permissions (we won't cover unless needed):

- `chmod` change permissions of a file.
- `whoami` who you are just in case you aren't sure.
- `sudo` super user do.

There are some commands dealing with processes:

- `top` a task manager, shows running processes with cpu and memory utilization
- `kill` a process.
- `ps` see running processes.

Here are some tools for dealing with useful networking tasks:

- `curl` transfer data from a network with a variety of protocols supported
- `wget` transfer data from a network (recursive downloads supported)
- `ssh` cryptographic network protocol. Allows remote access to other machines.

Finally some useful development tools:

- `gcc` GNU compiler collection
- `make` useful tool for building applications
- `gdb` debugging tools

1.1.2 A Note on the Unix Philosophy

It can be hard to quantify good software design. An informative approach is the "Unix Philosophy". This is summarized by Doug McIlroy (the inventor of Unix pipes) as:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
- Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
- Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

1.1.3 Git and GitHub

Working with software requires keeping track of versions and modifications. For this there is a set of best working practices that often depend on the task at hand.

The basic functionality of a version control system are:

- Keeping track of software history over time
- Being able to revert changes to a previous point in time (in case you break something important)
- Being able to make, track and merge experimental functionality, while keeping a "good" version unpolluted
- Keeping a remote backup of the source code in case of a disaster
- Allow teams of people to work in this way on a codebase without treading on each others toes (too much)

One very useful tool is Git. We will be using Git both via command line and via Visual Studio Code. Git interfaces with online platforms such as GitHub and GitLab. You may use GUI based applications for GitHub, such as GitHub Desktop.

For a review of commands see this cheat sheet

1.1.4 VS Code IDE

An IDE is an acronym for an "Integrated Development Environment". The most basic development environment is simply a text file editor (such as vim) and somewhere to run the code (such as a Julia REPL). Software developers work with wide variety of such tools (with extensions to make coding a simpler task).

Features of an IDE that will make your life easier:

- Text Editor
- Source Code Navigation
- Syntax Highlighting
- Debugger
- Code Completion
- Version Control Integration
- Search Tools
- Language Support
- Code Linting

In this course we recommend using VSCode, an open source code editing tool made by Microsoft.

1.1.5 A Rough Field Guide to Software Licencing

There are a vast number of licencing arrangements for dealing with the use of software and the availability of source code for the software. Two broad classes to be aware of:

- Proprietary Software
- Free and Open Source Software

The source code for Proprietary Software is typically a closely guarded secret and unauthorised use can have severe legal consequences.

Free and Open source does not mean that source code can always be used in whatever fashion is desired. Many different philosophies around why a particular developer or organisation wishes to distribute source code need to be respected.

Open source code falls into three broad classes:

- Permissive Licence (e.g. BSD, MIT, Apache)
- Copyleft (e.g. GPL)
- Public Domain (e.g. CC0)

A rough summary is that Permissive Licences tend to be favorable for commercial use of derived software, Copyleft makes derived works free to redistribute, but they must also be Copyleft works, and Public Domain are the most free and can be used as desired. Again there can be severe legal consequences for misuse of derived works.

Examples of Software released under different licences:

- Proprietary - Microsoft Windows, MacOS, MATLAB
- Permissive - Julia (MIT), Python (PSFL), Android (Apache 2.0 for userspace, GPLv2 for kernel)
- Copyleft - Linux (GPLv2), R (GPLv2)
- Public Domain - BLAS, SQLite

1.2 Mathematical analysis of algorithms

As an illustration take sorting. Say we have a list of numbers and wish to sort it:

```
data = [65, 51, 32, 12, 23, 84, 68, 1]
sort!(data) #The '!' is part of the function name and indicates data is sorted in place
println("Sorted data: ", data)
```

```
Sorted data: [1, 12, 23, 32, 51, 65, 68, 84]
```

How would we implement such a sorting algorithm ourselves? One basic algorithm is called bubble sort. Here is an implementation:

```
function bubble_sort!(a)
   n = length(a)
   for i in 1:n-1
```

```
      for j in 1:n-i
         if a[j] > a[j+1]
            (a[j], a[j+1]) = (a[j+1], a[j])
         end
      end
   end
   return a
end

bubble_sort!(data)
println("Sorted data (with buble sort): ", data)
```

```
Sorted data (with buble sort): [1, 12, 23, 32, 51, 65, 68, 84]
```

A key question when considering such an algorithm is **how fast does it run?** Other key questions include **how much memory does it use?**, as well as other questions. Such questions are often critical during the phase of algorithm design, and are later critical with algorithm or datastructure choice.

Later in the course you'll use quite robust benchmarking tool such as BenchmarkTools.jl to empirically measure performance. But for now, lets empirically measure the performance of bubble sort. We also compare it to the in-built sorting algorithm.

```
using Random

function do_timing()
   data = [65, 51, 32, 12, 23, 84, 68, 1]
   println("Bubble sort on small data ($(length(data))):")
   @time bubble_sort!(data);

   data = [65, 51, 32, 12, 23, 84, 68, 1]
   println("In-built sort on small data ($(length(data))):")
   @time sort!(data);

   Random.seed!(0)
   data = rand(Int,10^4)
   println("Bubble sort on larger data ($(length(data))):")
   @time bubble_sort!(data);

   Random.seed!(0)
   data = rand(Int,10^4)
   println("In-built sort on larger data ($(length(data))):")
   @time sort!(data);
end
do_timing();
```

```
Bubble sort on small data (8):
  0.000000 seconds
In-built sort on small data (8):
  0.000001 seconds
Bubble sort on larger data (10000):
  0.055691 seconds
In-built sort on larger data (10000):
  0.000053 seconds (6 allocations: 86.328 KiB)
```

We can see that the in-built sort is much faster than our bubble sort.

Now obviously we can optimize our bubble sort a little bit, for example here is a slightly better version that has type specifications and avoids bounds checking:

```julia
using Random
Random.seed!(0)

function slightly_faster_bubble_sort!(a::Vector{Int})::Vector{Int}
   n = length(a)
   for i in 1:n-1
      for j in 1:n-i
         @inbounds begin
            if a[j] > a[j+1]
               (a[j], a[j+1]) = (a[j+1], a[j])
            end
         end
      end
   end
   return a
end

#Let it run once to remove precompilation overhead
slightly_faster_bubble_sort!(rand(Int,100))

Random.seed!(0)
data = rand(Int,1000)

println("bubble sort ($(length(data))):")
@time bubble_sort!(data);

Random.seed!(0)
data = rand(Int,1000)

println("Slightly faster bubble sort ($(length(data))):")
@time slightly_faster_bubble_sort!(data);
```

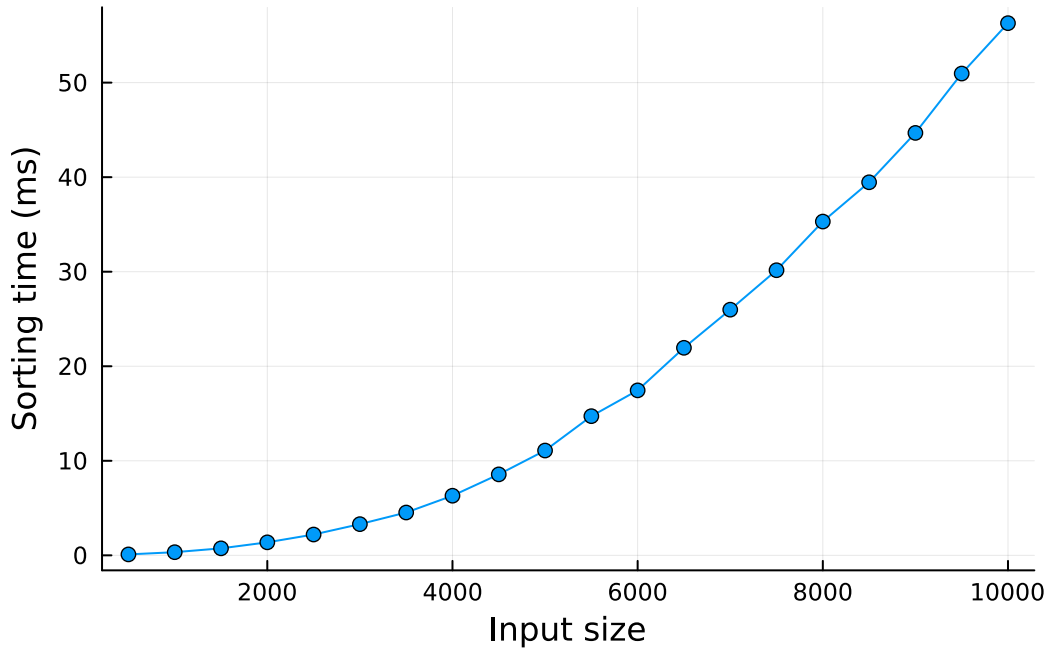
```
bubble sort (1000):
  0.000353 seconds
Slightly faster bubble sort (1000):
  0.000170 seconds
```

You can probably see that this shaved off a few milliseconds of compute time, but still the time is much longer than the in-built sort.

Let's investigate (empirically) a bit further:

```julia
using Random, Plots
Random.seed!(0)

function do_timing()
   times = Vector{Float64}(undef, 0)
   n_range = 500 : 500 : 10^4
   for n in n_range
      data = rand(Int64, n)
      start_time = time_ns()
      bubble_sort!(data)
      end_time = time_ns()
      push!(times, Int(end_time - start_time) / 10^6) # time in milliseconds
   end
   return times, n_range
end

times, n_range = do_timing()
```

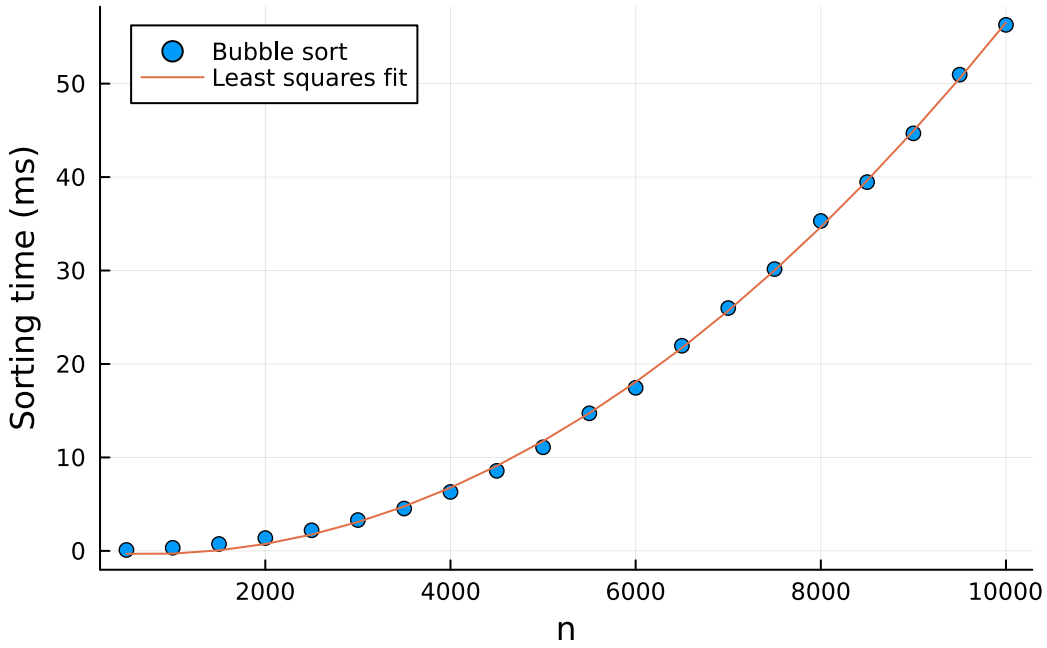
```
plot(n_range,times, xlabel="Input size", ylabel="Sorting time (ms)", legend = false, shape =
:circle)
```



It appears that the running time grows quadratically with the input size! We can event try to fit a quadratic model using least squares for it:

```
using DataFrames, GLM

data = DataFrame(X = Float64.(n_range), Y = times) #More on DataFrames later in the course
model = lm(@formula(Y ~ 1 + X + X^2), data) #This fits a linear model to the features 1, X, X^2
println(model)
predict_bubble_time(n) = coef(model)'*[1,n,n^2]
scatter(n_range, times,
   xlabel="n", label = "Bubble sort")
plot!(n_range, predict_bubble_time.(n_range),
   xlabel="n", ylabel="Sorting time (ms)",
   label = "Least squares fit", legend = :topleft)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, Li

Coefficients:

           Coef.    Std. Error       t  Pr(>|t|)     Lower 95%      Upper 95%

x1   0.0          NaN           NaN       NaN     NaN           NaN
x2  -0.000952859    6.98254e-5  -13.65    <1e-10   -0.00109956   -0.000806162
x3   6.60634e-7     8.80067e-9   75.07    <1e-23    6.42144e-7    6.79123e-7

, ModelFrame{@NamedTuple{Y::Vector{Float64}, X::Vector{Float64}}, LinearModel}(Y ~ 1 + X + :(X ^ 2), Sta
  Y => Y
  X => X
, (Y = [0.098574, 0.33258, 0.740081, 1.374916, 2.20165, 3.299739, 4.529614, 6.305498, 8.562692, 11.0857
```



Let's see how this measured performance works for 30 thousand observations and 50 thousand observations:

```
Random.seed!(0)

function do_timing()
   data = rand(Int,3*10^4)
   prediction_seconds =  predict_bubble_time(3*10^4)/10^3
   print("Predicted time: $prediction_seconds \n Actual time:")
   @time bubble_sort!(data);

   data = rand(Int,5*10^4)
   prediction_seconds =  predict_bubble_time(5*10^4)/10^3
   print("\nPredicted time: $prediction_seconds \n Actual time:")
   @time bubble_sort!(data);
end
do_timing();
```

```
Predicted time: 0.5659843848620368
 Actual time:  0.574127 seconds

Predicted time: 1.6039408284645944
 Actual time:  1.773046 seconds
```

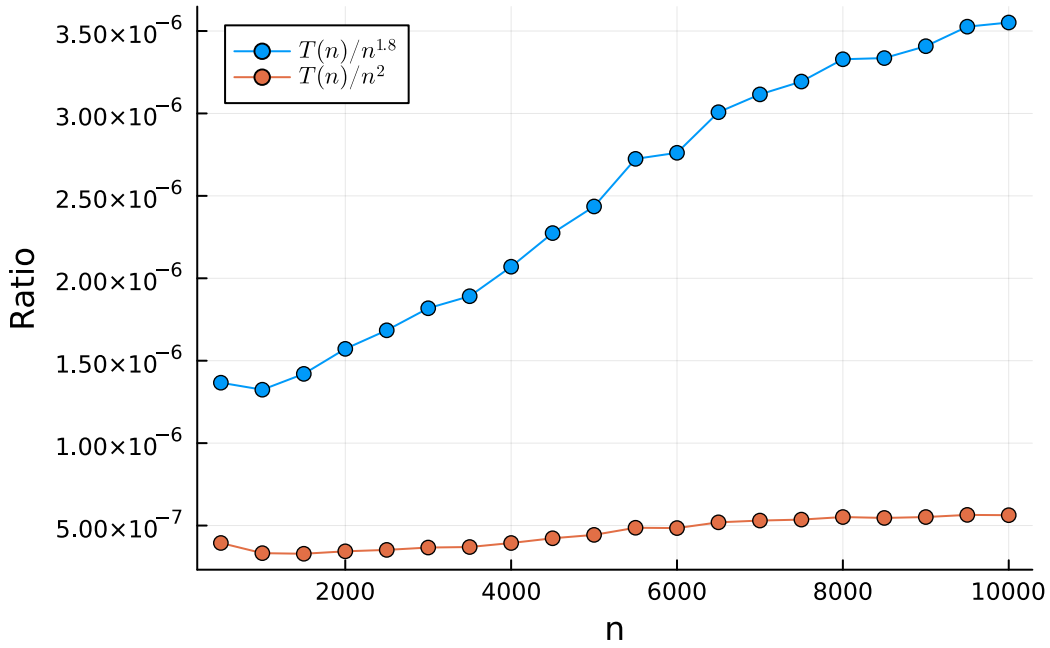
It isn't a bad prediction. So with the coefficients of the regression being β_0, β_1, and β_2 we have an empirical model for the computation time, $T(n)$, for input of size n:

$$T(n) = \beta_0 + \beta_1 n + \beta_2 n^2.$$

If you run these experiments on your computer you'll almost certainly get different β values from the ones obtained here (the data is the same with a fixed seed but the hardware and instance of runs is different). Even on the same machine, different runs of the experiment will yield different β values and hence a different $T(\cdot)$.

What if instead about worrying about the exact $T(\cdot)$ function we will focus on the most dominant term? Here we plot the running time first divided by $n^{1.8}$ and then by n^2.

```
using LaTeXStrings

plot(n_range, times ./ n_range .^ (1.8),shape = :circle,label = L"T(n) / n^{1.8}")
plot!(n_range, times ./ n_range .^ 2,label = L"T(n) / n^{2}",
   xlabel = "n", ylabel = "Ratio",legend = :topleft,shape = :circle,)
```



The fact that it roughly goes to a constant when dividing by n^2 gives us an indication that,

$$T(n) \sim n^2.$$

The symbol '$\sim$' means here that for some constant,

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = C.$$

You can also interpret it as,

$$T(n) = Cn^2 + o(n^2)$$

where $o(n^2)$ is some function such that when dividing by n^2 and taking $n \to \infty$, the ratio vanishes.

Now in computer science and the study of algorithms, you would often roughly say that,

$$T(n) = O(n^2).$$

Where the $O(\cdot)$ here follow Big O notation. However to be more precise, some would interpret $T(n) = O(n^2)$ to mean that $T(n)$ `grows` not faster than n^2. Then it is also $O(n^3)$ and $O(n^4)$ and even $O(2^n)$ (exponential growth. However in this case we (so far empirically) see that the computation is at $O(n^2)$ and hence sometimes $\Theta(n^2)$ is used for this. However, as this is not a full algorithms course, we'll just loosely write $O(n^2)$.

But why is bubble sort an "$O(n^2)$ time complexity algorithm"?

Well in this case we can inspect the algorithm and consider the number of comparisons taking place. To do so look at this "fragment of code":

```
1: for i in 1:n-1                          #Outer loop
2:     for j in 1:n-i                      #Inner loop
3:         if a[j] > a[j+1]                #Comparison
4:             a[j], a[j+1] = a[j+1], a[j] #Switch
```

It is essentially the heart of the bubble sort algorithm. The outer loop runs $n-1$ times in these runs, the number of iterations of the inner loop are $n-1$, $n-2$, $n-3$,... 1. Hence line 3, the comparison line runs,

$$1+2+...+n-1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{n}{2} = O(n^2).$$

It never hurts to check:

```
function profiling_bubble_sort!(a)
   k = 0
   n = length(a)
   for i in 1:n-1
      for j in 1:n-i
         k += 1 #profiling step
         if a[j] > a[j+1]
            a[j], a[j+1] = a[j+1], a[j]
         end
      end
   end
   return k #returns the number of comparisons made
end

data = [65, 51, 32, 12, 23, 84, 68, 1]
n = length(data)

@show profiling_bubble_sort!(data)
@show n*(n-1) ÷ 2;
```

```
profiling_bubble_sort!(data) = 28
(n * (n - 1)) ÷ 2 = 28
```

As an aside, lets do this profiling by making a different version of our bubble sort that accepts a comparison function. We can use the > function as a default argument and later replace it with something else. Lets first test it works:

```
function bubble_sort!(a; comp = >)
   n = length(a)
   for i in 1:n-1
      for j in 1:n-i
         if comp(a[j],a[j+1])
            a[j], a[j+1] = a[j+1], a[j]
         end
      end
   end
   return a
end

data = [65, 51, 32, 12, 23, 84, 68, 1]
bubble_sort!(data) #uses the default argument for comp
println("Sorted data (with bubble sort): ", data)
```

```
Sorted data (with bubble sort): [1, 12, 23, 32, 51, 65, 68, 84]
```

Now lets sort based on sum of digits (instead of decimal value of digits).

```
data = [65, 51, 32, 12, 23, 84, 68, 1]
my_comp(num1, num2) = sum(digits(num1)) > sum(digits(num2))
bubble_sort!(data, comp = my_comp)
println("Sorted by sum of digits: ", data)
```

```
Sorted by sum of digits: [1, 12, 32, 23, 51, 65, 84, 68]
```

Note that we could have also use an anonymous function:

```
data = [65, 51, 32, 12, 23, 84, 68, 1]
bubble_sort!(data, comp = (n1,n2) -> sum(digits(n1)) > sum(digits(n2)) )
println("Sorted by sum of digits: ", data)
```

```
Sorted by sum of digits: [1, 12, 32, 23, 51, 65, 84, 68]
```

Finally lets make a comparison function that increments a global counter of the number of times it is called.

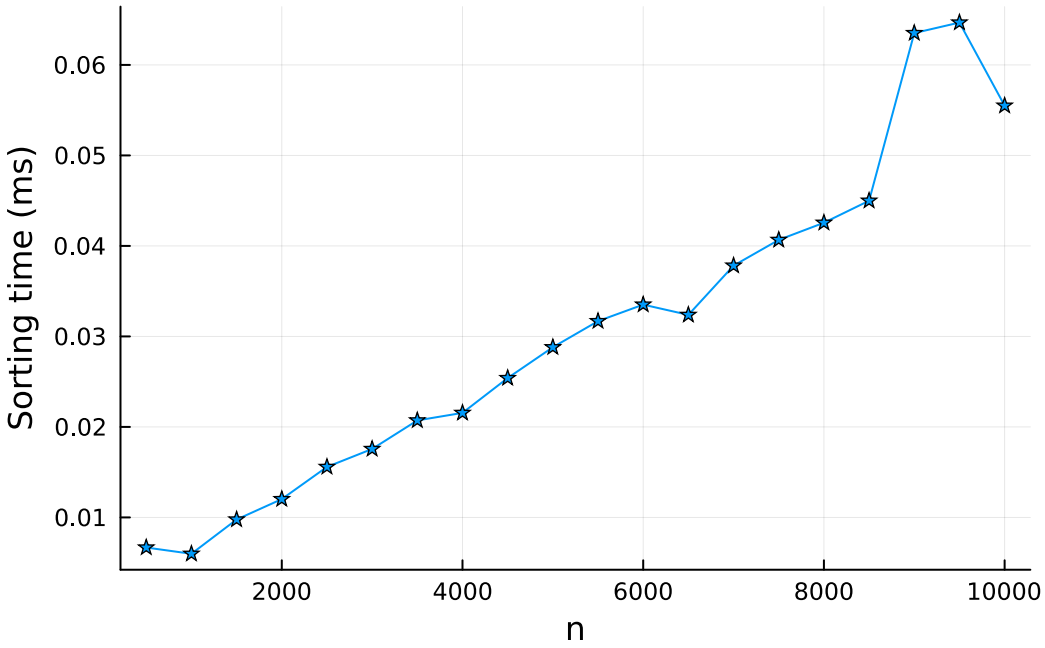
```
k = 0 #global variable for number of comparisons
bubble_sort!(data, comp = (n1,n2) -> begin        #we can use a begin-end block to make a
multi-line anonymous function
   global k+=1
   sum(digits(n1)) > sum(digits(n2))
   end )
println("Sorted data: ", data)
println("Number of comparisons: ", k)
```

```
Sorted data: [1, 12, 32, 23, 51, 65, 84, 68]
Number of comparisons: 28
```

1.2.0.1 Now lets compare bubble sort with the in-built sort:

As you can see from the plot, the in-built sort is about 200x faster! This is mostly **not** because of "in-built performance improvements" (remember that in Julia almost all the in-built functions are themselves written in Julia). It is because a different algorithm is used. In fact, sorting is one of the basic computer science problems that has been exhaustively studied. See for example the Wikipedia sorting algorithms entry with a table listing the complexity of many sorting methods.

```
using Random, Plots
Random.seed!(0)

function do_timing()
   times = Vector{Float64}(undef,0)
   n_range = 500:500:10^4
   for n in n_range
      Random.seed!(0)
      data = rand(Int64,n)
      start_time = time_ns()
      sort!(data[1:n]) #Use the in-built sort! instead
      end_time = time_ns()
      push!(times, Int(end_time - start_time)/10^6) #time in milliseconds
   end
   return times, n_range
end

times, n_range = do_timing()
plot(n_range,times, xlabel="n", ylabel="Sorting time (ms)", shape = :star, legend = false)
```



Note that the running time appears to grow linearly but actually we will theoretically see soon that it is of order $n \log n$. In fact, there are multiple sorting algorithms that have $O(n \log n)$ complexity and it can even be proved that for arbitrary data you cannot do better than $O(n \log n)$ in terms of the number of comparisons.

We'll explore one such "efficient" sorting algorithm, heap-sort, and this is because the heap data structure on which heap-sort is based will be with us for most of the course. heap-sort is not necessarily the best algorithm, and in fact when you call `sort!` in Julia, by default the quick-sort algorithm is used. See the actual source code in https://github.com/JuliaLang/julia/blob/master/base/sort.jl.

But first we discuss a few datastructures.

1.2.1 A Whirlwind tour of a Few Datastructures

Note: Do not confuse "Datastructures" with "Databases", these are related terms that mean different things.

When we approach programming without thinking about it too much there is one simple datastructure, an **array**. An array is a list of a given length with random access to any point. For example,

```
data = [65, 51, 32, 12, 23, 84, 68, 1]
@show length(data)
@show data[4]; #the fourth element
```

```
length(data) = 8
data[4] = 12
```

Lookups in an array are $O(1)$ constant time. But other operations can be more costly. This is because an array spans adjacent memory cells and if we want to `insert!`, `push!`, or `prepend!` to it, then more memory often needs to be managed and the array often needs to be copied. Just to get an understanding of arrays, lets use `pointer` to get the address of an array and certain elements in it.

```
data = [65, 51, 32, 12, 23, 84, 68, 1]
@show eltype(data)
@show sizeof(eltype(data))
@show pointer(data)
@show pointer(data,1)
```

```
@show pointer(data,2)
@show pointer(data,length(data));

#This is true for any array
always_true = (pointer(data,length(data)) - pointer(data))/sizeof(eltype(data)) + 1 == length(data)
@show always_true;
```

```
eltype(data) = Int64
sizeof(eltype(data)) = 8
pointer(data) = Ptr{Int64}(0x00007e6b09812a20)
pointer(data, 1) = Ptr{Int64}(0x00007e6b09812a20)
pointer(data, 2) = Ptr{Int64}(0x00007e6b09812a28)
pointer(data, length(data)) = Ptr{Int64}(0x00007e6b09812a58)
always_true = true
```

In a classic datastructures course, you would often study the following data structures just after speaking about arrays:

- Linked lists
- Queues
- Stacks

We won't do this thoroughly here due to lack of time (some of this material is taught in COMP3506 at UQ). You can see implementations of these datasturctures and others in DataStructures.jl.

We do consider a few of the key other Collections and Datastructures that Julia supplies. The key ones include:

- Multidimensional arrays (tensors)
- Dictionaries
- Sets

1.2.2 Multidimensional arrays

Multidimensional arrays are basically arrays where the indexing into the array is carried out in smarter way. The simplest example is a matrix.

```
m, n = 3, 5
data = Array{Int8,2}(undef,m,n)
N = n*m #Total size of data
for i in 1:N
   @show pointer(data,i)
   data[i] = i #even though data is a matrix, we can just go over the times sequentially.
end
data
```

```
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7530)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7531)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7532)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7533)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7534)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7535)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7536)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7537)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7538)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a7539)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a753a)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a753b)
```

```
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a753c)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a753d)
pointer(data, i) = Ptr{Int8}(0x00007e6aca9a753e)
```

```
3×5 Matrix{Int8}:
 1  4  7  10  13
 2  5  8  11  14
 3  6  9  12  15
```

As you can see, `data[i,j]` is just like `data[(j-1)*m + i]`:

```
for j in 1:n, i in 1:m
   @show data[i,j], data[(j-1)*m + i]
end
```

```
(data[i, j], data[(j - 1) * m + i]) = (1, 1)
(data[i, j], data[(j - 1) * m + i]) = (2, 2)
(data[i, j], data[(j - 1) * m + i]) = (3, 3)
(data[i, j], data[(j - 1) * m + i]) = (4, 4)
(data[i, j], data[(j - 1) * m + i]) = (5, 5)
(data[i, j], data[(j - 1) * m + i]) = (6, 6)
(data[i, j], data[(j - 1) * m + i]) = (7, 7)
(data[i, j], data[(j - 1) * m + i]) = (8, 8)
(data[i, j], data[(j - 1) * m + i]) = (9, 9)
(data[i, j], data[(j - 1) * m + i]) = (10, 10)
(data[i, j], data[(j - 1) * m + i]) = (11, 11)
(data[i, j], data[(j - 1) * m + i]) = (12, 12)
(data[i, j], data[(j - 1) * m + i]) = (13, 13)
(data[i, j], data[(j - 1) * m + i]) = (14, 14)
(data[i, j], data[(j - 1) * m + i]) = (15, 15)
```

Access of matrix elements is $O(1)$, however this doesn't mean that in practice (certainly in big numerical computations), different ways of accesing the matrix vary:

```
function sum_rows(data, n, m)
   s = zero(eltype(data))
   for i in 1:n
      for j in 1:m
         s += data[i,j]
      end
   end
   return s
end

function sum_columns(data, n, m)
   s = zero(eltype(data))
   for j in 1:m
      for i in 1:n
         s += data[i,j]
      end
   end
   return s
end

using Random
Random.seed!(0)
n, m = 500, 10^6
data = rand(n,m)
#dry run for precompilation
```

```
sum_columns(data, n, m)
sum_rows(data, n, m)
println("Summing with the column iteration moving fastest")
@time sum_columns(data, n, m)
println("Summing with the row iteration moving fastest")
@time sum_rows(data, n, m);
```

```
Summing with the column iteration moving fastest
  0.185325 seconds (1 allocation: 16 bytes)
Summing with the row iteration moving fastest
  1.102687 seconds (1 allocation: 16 bytes)
```

As you can see, going by columns is much faster than going by rows. This is because there are many **cache misses** if we go by rows. The key takeaway here is that while the speed complexity of the array algorithm $O(1)$ does not capture actual clock speed because of computer architecture issues such as the cache.

1.2.2.1 Dictionaries

As the name implies, dictionaries refer to collections that keep **keys** and **values** and allow for efficient lookup. For example:

```
#A dictionary mapping suburb to postcode
dict = Dict{String,Int16}()
dict["Saint Lucia"] = 4067
dict["South Brisbane"] = 4101
dict["Rochdale"] = 4123

@show dict["Rochdale"]
@show keys(dict)
@show values(dict)
dict
```

```
dict["Rochdale"] = 4123
keys(dict) = ["South Brisbane", "Rochdale", "Saint Lucia"]
values(dict) = Int16[4101, 4123, 4067]
```

```
Dict{String, Int16} with 3 entries:
  "South Brisbane" => 4101
  "Rochdale"       => 4123
  "Saint Lucia"    => 4067
```

```
dict["Taringa"] #Will not work
```

```
KeyError: KeyError("Taringa")
KeyError: key "Taringa" not found
Stacktrace:
 [1] getindex(h::Dict{String, Int16}, key::String)
   @ Base ./dict.jl:477
 [2] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-3.qmd:612
```

A dictionary where the keys are $\{1, \ldots, n\}$ or some subset can just be implemented via an array (via index lookup), but in general this is more complicated. A naive (and highly inefficient) way to implement a dictionary such as from the example above is:

```
dict_keys = Array{String}(undef,0)
dict_values = Array{Int16}(undef,0)
```

```
function add_key_value_pair(keys,values,new_key,new_value)
   #Note that keys and values are `passed by reference` and hence can be modified
   push!(keys, new_key)
   push!(values, new_value)
end

add_key_value_pair(dict_keys,dict_values,"Saint Lucia", 4067)
add_key_value_pair(dict_keys,dict_values,"South Brisbane", 4101)
add_key_value_pair(dict_keys,dict_values,"Rochdale", 4123)

function value_of_key(keys,values,key_in_question)
   for (i,k) in enumerate(keys)
      k == key_in_question && return values[i]
   end
   @error "didn't find the key $key_in_question"
end

@show value_of_key(dict_keys,dict_values,"Rochdale")

value_of_key(dict_keys,dict_values,"Taringa") #Will throw an error
```

```
value_of_key(dict_keys, dict_values, "Rochdale") = 4123
  Error: didn't find the key Taringa
  @ Main.Notebook ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture
```

```
  Error: didn't find the key Taringa
```

However you can probably see that the time complexity of `value_of_key()` is of the order $O(n)$ when n is the number of keys. This is a very time-consuming lookup. An (on average) efficient way to implement dictionaries is via a Hash table data structure. We won't get into the details now, but (solely for those interested) refer this (more advanced talk) related to alternative implementations of dictionaries (this is a recent talk by one of the course lecturers, Andy Ferris).

1.2.3 Sets

Another type of collection that is in-built in Julia and often very useful is a `Set`. Unlike arrays that maintain order, sets (like in mathematics) are unordered and do not have repetitions.

```
my_set = Set([1,2,3,4,1,2,3])
```

```
Set{Int64} with 4 elements:
  4
  2
  3
  1
```

You can do a variety of set operations on sets. See for example this code from SWJ

```
A = Set([2,7,2,3])
B = Set(1:6)
omega = Set(1:10)

AunionB = union(A, B)
AintersectionB = intersect(A, B)
BdifferenceA = setdiff(B,A)
Bcomplement = setdiff(omega,B)
AsymDifferenceB = union(setdiff(A,B),setdiff(B,A))
println("A = $A, B = $B")
println("A union B = $AunionB")
```

```
println("A intersection B = $AintersectionB")
println("B diff A = $BdifferenceA")
println("B complement = $Bcomplement")
println("A symDifference B = $AsymDifferenceB")
println("The element '6' is an element of A: $(in(6,A))")
println("Symmetric difference and intersection are subsets of the union: ",
   issubset(AsymDifferenceB,AunionB),", ", issubset(AintersectionB,AunionB))
```

```
A = Set([7, 2, 3]), B = Set([5, 4, 6, 2, 3, 1])
A union B = Set([5, 4, 6, 7, 2, 3, 1])
A intersection B = Set([2, 3])
B diff A = Set([5, 4, 6, 1])
B complement = Set([7, 10, 9, 8])
A symDifference B = Set([5, 4, 6, 7, 1])
The element '6' is an element of A: false
Symmetric difference and intersection are subsets of the union: true, true
```

Why use sets and what would you expect them to be efficient with? Let's take data that has 10^5 elements selected randomly out of the possible `Int32` (most chances are that all elements are unique - how would you compute this chance?):

```
Random.seed!(0)
array_data = rand(Int32,10^5)
my_set = Set(array_data)
@show length(my_set); #Since this shows 10^5 as output it indicates all values were unique.
```

```
length(my_set) = 100000
```

Say now we want to lookup if an arbitrary number, `n` is in the set. If we were just using an array there is no good way to look for it better than $O(n)$. But in the case of a set, seeing set membership is guaranteed to be much quicker.

```
function do_timing()
   @time begin
      x = 10^5 #smallest number will be 213796.
      while true
         x  array_data && break
         x += 1
      end
      @show x
   end

   @time begin
      x = 10^5
      while true
         x  my_set && break
         x += 1
      end
      @show x
   end
end

do_timing();
```

```
x = 149944
  0.988835 seconds (49.99 k allocations: 966.641 KiB)
x = 149944
  0.001208 seconds (49.97 k allocations: 1.005 MiB)
```

1.2.4 Priority queues, heaps, and back to sorting

We can't cover all of the fundamental algorithms and data structures in this course, so we chose one non-trivial data structure to focus on. This is the heap data structure. It will be used here for sorting, used in Unit 5 for algebraic manipulations, and used in Unit 6 as an aid for discrete event simulation.

Note: do not confuse the heap data structure with the other computer science usage of "heap", also known as "free store" which refers to the place in memory where allocations happen. In Julia the heap is used for allocations of immutable types and has garbage collection. More on that in Unit 4.

Before starting to see what a heap does (we'll continue in more depth in future units on how it works), we'll discuss what we expect it to do - namely implement an efficient **priority queue**.

Consider for example a hypothetical situation where thousands of individuals attend a mass vaccination centre. As they arrive, each individual is tagged by their name (a `String`), and recieves a priority for vaccination which is a number (say `Float32`). Assume (hypothetically) that the priority is a function of their age, their occupation (essential or not), and their medical condition. Hence there are many possible different priorities.

Continuing with this example, assume the individuals queue (e.g. wait in a big arena or showgrounds area) and are only called when their priority is the highest (clearly high priority individuals won't queue much while others wait for a long while). The management system will then keep track of the priorities of the individuals and do the following:

- Add individuals (this will happen at every arbitrary time in which someone arrives).
- Remove the individual with the highest priority (they then get vaccinated and leave).
- Modify an individual's priority if they complain about waiting too long or in case of other reasons.

There are different alternatives for implementing such a data structure and in all cases the associated "collection" will be called a **priority queue**. It turns out that the heap data structure is the most efficient way to implement a priority (certainly if the number of items is large). But for starters lets consider some naive implemintations.

A small digression is to see the Julia type of **named tuples**; which present one way of keeping individuals:

```
person_A = (priority = 23.2, name = "Zack") #This is a named tuple,
person_B = (priority = 12.6, name = "Lenna")#if there wasn't "priority =" and "name =" it would
just be a tuple

@show typeof(person_A)
@show person_A
@show person_B;

@show person_A.name
@show person_B.priority;
```

```
typeof(person_A) = @NamedTuple{priority::Float64, name::String}
person_A = (priority = 23.2, name = "Zack")
person_B = (priority = 12.6, name = "Lenna")
person_A.name = "Zack"
person_B.priority = 12.6
```

So the type of these objects is `NamedTuple{(:priority, :name), Tuple{Float64, String}}` and they are just convient to work with.

Now back to our naive priority queue. We will just keep an array of the named tuples (the naiveness is not because of the named tuples but because of the fact we are using an array).

```
#Create an empty priorty queue as an array of named tuples
naive_priority_queue = NamedTuple{(:priority, :name), Tuple{Float64, String}}[];
```

Now in the most naive implementation, every time we add an individual we'll just `push!` their named tuple to the array. For example,

```
new_person = (priority = 123.2, name = "Yarden")
push!(naive_priority_queue,new_person)

another_new_person = (priority = 12.5, name = "Sharon")
push!(naive_priority_queue,another_new_person)

yet_another_new_person = (priority = 4.4, name = "Miriam")
push!(naive_priority_queue,yet_another_new_person)

even_another_new_person = (priority = 54.2, name = "Moshe")
push!(naive_priority_queue,even_another_new_person)


naive_priority_queue
```

```
4-element Vector{@NamedTuple{priority::Float64, name::String}}:
 (priority = 123.2, name = "Yarden")
 (priority = 12.5, name = "Sharon")
 (priority = 4.4, name = "Miriam")
 (priority = 54.2, name = "Moshe")
```

Note that (in principle) such a `push!` is a costly operation because from time to time new memory needs to be allocated and the whole array copied.

Then each time we wish to remove an individual there will be the the costly (why?) operation of `deleteat!`. This is $O(n)$ every time.

```
index_of_person_we_want_to_delte = 3 #Say we know this index
deleteat!(naive_priority_queue, index_of_person_we_want_to_delte)
naive_priority_queue
```

```
3-element Vector{@NamedTuple{priority::Float64, name::String}}:
 (priority = 123.2, name = "Yarden")
 (priority = 12.5, name = "Sharon")
 (priority = 54.2, name = "Moshe")
```

Finally searching for the individual with the highest priority is also $O(n)$.

```
# You can ignore the type annotations of the inputs and outputs for now
function get_next_person(priority_queue::Vector{NamedTuple{(:priority, :name), Tuple{Float64,
String}}})::NamedTuple{(:priority, :name), Tuple{Float64, String}}
   @assert !isempty(priority_queue)
   max_p = first(priority_queue)
   for p in priority_queue
      (p.priority > max_p.priority) && (max_p = p)
   end
   return max_p
end

get_next_person(naive_priority_queue)
```

```
(priority = 123.2, name = "Yarden")
```

Now you may think of simple ways to improve this, for example by enforcing the **invariant** that `naive_priority_queue` is always sorted. But this then implies that adding individuals is an $O(n)$ worst case operation.

Now in a scenario like a mass vaccination centre it isn't really a problem to use such a priority queue with

$O(n)$ costs for most operations. This is because the frequency of events is not high and neither is n. However, in other scenarios where events (additions, deletions, finding the maximum priority) occur very frequently then $O(n)$ for each operation may be very painful. For example, what if you were simulating that vaccination centre on a computer (similar simulations will be considered in Unit 6).

1.2.4.0.1 The Heap data structure

The heap data structure is a smarter tree based way of keeping the elements of a priority queue. With this data structure, finding the maximum is $O(1)$, deleting the maximum is $O(\log n)$, insertions are $O(\log n)$, and updating of a priority is $O(\log n)$. We'll see the way it is implemented in Unit 6, but for now we'll use it as a "black box".

One implementation we can use is from DataStructures.jl.

Now observe that since the operations of insertions and deletion of the maximum are efficient, we also get an efficient sorting algorithm:

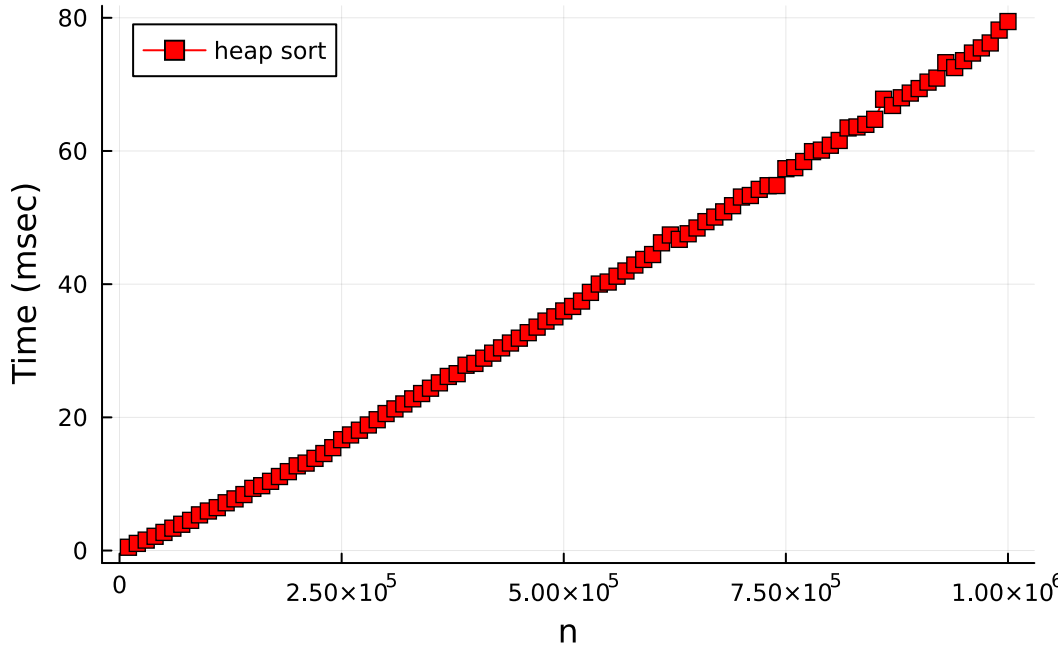
The heap sort algorithm just pushes the values to be sorted into a heap and then pops out the maximal value one after the other. Since there are n values and putting in a value takes $O(\log n)$ time, putting all n values in requires $O(n \log n)$ time. Then popping the values another $O(n \log n)$ time so the total time is twice of that which is still $O(n \log n)$. This is much better than the $O(n^2)$ of bubble sort and is equivalent to the best (asymptotic) sorting performance possible.

```
using Random, DataStructures
Random.seed!(0)


function heap_sort!(a::AbstractArray)
   h = BinaryMinHeap{eltype(a)}()
   for e in a
      push!(h, e) #This is an O(log n) operation
   end

   #Write back onto the original array
   for i in 1:length(a)
      a[i] = pop!(h) #This is an O(log n) operation
   end
   return a
end

data = [65, 51, 32, 12, 23, 84, 68, 1]
heap_sort!(data)
data
```

```
8-element Vector{Int64}:
  1
 12
 23
 32
 51
 65
 68
 84
```

Lets carry out rough benchmarks. As we can see (even though this is a very sloppy timing mechanism), there appears to be linear growth. Note that the difference between $O(n)$ and $O(n \log n)$ is hard to pick up with sloppy timings such as this.

```
Random.seed!(0)

function do_timing()
   data = rand(Int64,10^8)

   n_range = 10^4:10^4:10^6
   times_heap_sort = Vector{Float64}(undef,0)


   for n in n_range
      GC.enable(false) #This turns off garbage collection
      start_time = time_ns()
      heap_sort!(data[1:n])
      end_time = time_ns()
      GC.enable(true) #GC back on
      push!(times_heap_sort, Int(end_time - start_time)/10^6) #time in milliseconds
   end
   return times_heap_sort, n_range
end

times_heap_sort, n_range = do_timing()

plot(n_range,times_heap_sort,
   shape = :square, c = :red, legend = :topleft,
   label="heap sort",xlabel = "n", ylabel = "Time (msec)")
```



Lets also consider a single timing of 10^6 data points:

```
Random.seed!(0)
data = rand(Int64,10^6)
@time heap_sort!(data);
```

```
  0.085952 seconds (31 allocations: 28.299 MiB)
```

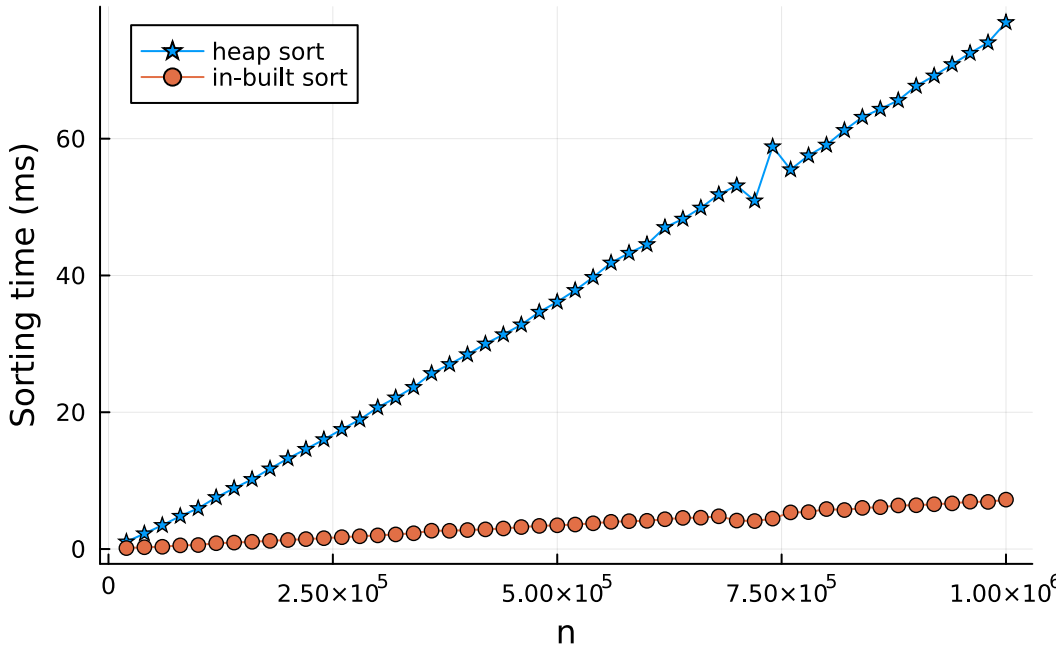
And compare it with the in-built sort:

```
Random.seed!(0)
data = rand(Int64,10^6)
@time sort!(data);
```

```
  0.006131 seconds (15 allocations: 7.638 MiB)
```

We still see that the in-built sort is about four times as fast and does not allocate memory. In contrast the heap sort algorithm used memory. Comparing the two together we see they appear to (empirically) be of similar order, but with the in-built sort about 3-4 times faster.

```
using Random, Plots
Random.seed!(0)

function do_timing()
   data = rand(Int64,10^6)

   n_range = 2*10^4:2*10^4:10^6
   times_heap_sort = Vector{Float64}(undef,0)
   times_in_built_sort = Vector{Float64}(undef,0)

   for n in n_range
      GC.enable(false)
      start_time = time_ns()
      heap_sort!(data[1:n])
      end_time = time_ns()
      GC.enable(true)
      push!(times_heap_sort, Int(end_time - start_time)/10^6) #time in milliseconds

      GC.enable(false)
      start_time = time_ns()
      sort!(data[1:n])
      end_time = time_ns()
      GC.enable(true)
      push!(times_in_built_sort, Int(end_time - start_time)/10^6) #time in milliseconds
   end
   return times_heap_sort, times_in_built_sort, n_range
end

times_heap_sort, times_in_built_sort, n_range = do_timing()

plot(n_range,times_heap_sort, label="heap sort",shape = :star)
plot!(n_range,times_in_built_sort, label = "in-built sort", shape = :circle,
   xlabel="n", ylabel="Sorting time (ms)", legend = :topleft)
```



So what is the in-built sort? Lets look at "quick sort".

1.2.4.1 Quicksort (and Merge Sort on the way)

On our path to understanding quick sort we'll start with a different sorting algorithm Merge sort. It uses a divide and conquer strategy somewhat similar to our target algorithm, quick-sort.

Here is an implementation of merge sort that is very wasteful in terms of memory copies (we won't present a more efficient implementation here). Nevertheless the key principle is that when presented with sorting an array of length n, merge-sort splits it into two arrays of size $n/2$ (or close to that if n is odd). Merge sort then sorts each of the sub-arrays by calling itself again (a recursive call), and then when presented with two sorted sub arrays merge sort merges them in order.

The code below implements this from first principles. It can be improved in terms of style and efficiency but is good enough to get the main idea:

```
#Note that there are much better implementations of this which avoid all this memory copying
function my_merge_sort(data)
   n = length(data)
   (n == 1) && return data
   p = n ÷ 2 #\div + [TAB]
   data_bottom = my_merge_sort(data[1:p]) #Get a sorted version of bottom half
   data_top = my_merge_sort(data[p+1:n]) #get a sorted version of top half
   data_new = Vector{eltype(data)}(undef,length(data))

   #Now merge, b and t are running indexes for the bottom and top
   b, t = 1, 1 #will
   for i in 1:n #loop on indexes of destination array
      if b > length(data_bottom) #if data_bottom array is finished
         data_new[i] = data_top[t]
         t += 1
         continue #returns to top of for loop
      end
      if t > length(data_top) #if data_top is finished
         data_new[i] = data_bottom[b]
         b += 1
```

```
            continue #returns to top of for loop
        end

        #if here then bottom and top are not finished - put the smaller element in the destination
        if data_bottom[b] <= data_top[t]
            data_new[i] = data_bottom[b]
            b += 1
        else
            data_new[i] = data_top[t]
            t+=1
        end
    end
    return data_new
end
```

```
my_merge_sort (generic function with 1 method)
```

Let's compare merge sort with the in-built sort on 10^6 integers. You can see from the output a check that both functions create the same sub-array. You can also see that the timing of merge-sort is significantly higher (by a factor of about 5) however it is still competitive timing (e.g. when comparing to bubble sort). Again keep in mind that our implementation was very wasteful due to multiple copying of the data.

```
Random.seed!(0)
data = rand(Int64,10^6)
@show my_merge_sort(data) == sort(data)

println("Time for Merge sort:")
@time my_merge_sort(data)

println("Time for in-built sort:")
@time sort(data);
```

```
my_merge_sort(data) == sort(data) = true
Time for Merge sort:
  0.126751 seconds (6.01 M allocations: 478.806 MiB, 17.35% gc time)
Time for in-built sort:
  0.045634 seconds (9 allocations: 15.267 MiB, 86.62% gc time)
```

We will now analyze the computational complexity time of merge sort and see it is $O(n \log n)$. The key is to consider the three main actions carried out by merge sort:

1. Sort the bottom sub list.
2. Sort the top sub list.
3. Merge the data.

If $T(n)$ is the time for sorting data of size n then $T(n/2)$ is the time for sorting each sub-list. Further, you can see that the time to merge the two sub-lists is of order $O(n)$ because it requires running on the whole length of the dataset once. Hence we get,

$$T(n) = 2T(\frac{n}{2}) + O(n).$$

This type of recursion is popular in several divide and conquer algorithms and we can now use the so called Master theorem for analysis of algorithms to see that this recursion is "solved by" $T(n) = O(n \log n)$. We won't get into all the mathematical details, but as an illustration that this is sensible assume that the $O(n)$ term in the recursion is $C(\log 2)n$ (that is, assume a specific constant and ignore lower order $o(n)$ terms). Then by plugging in $T(n) = Cn \log n$, for some constant C, we see it is a solution:

$$Cn\log n = 2C\frac{n}{2}\log\frac{n}{2} + C(\log 2)n = Cn\log n.$$

The Master theorem allows to formalize this taking into considerations all possible $O(n)$ terms.

1.2.5 A quick glimpse at quick-sort

We now explore quicksort. Like merge sort it is a divide and conquer algorithm. Implementing it is part of BigHW, so we won't provide an implementation. However here is pseudocode adapted from Wikipedia broken up into two functions `quicksort` which operates on an array `A` indexed by `lo` and `hi` (in an initial call you would use `lo = 1` and `hi = length(A)`:

```
algorithm quicksort(A, lo, hi):
    if lo > 0 && hi > 0 then
         if lo < hi then
              p := partition(A, lo, hi)
              quicksort(A, lo, p - 1)
              quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi):
    pivot := A[hi]
    i := lo - 1
    for j := lo to hi do
        if A[j] <= pivot then
            i := i + 1
            swap A[i] with A[j]
    return i
```

There are many resources on the web dealing with quicksort. Here is one very nice recommended video. Note that it uses Python where array indexing starts at `0` and not `1` like Julia. Also note that in Python `def` is how you define a function.

We note that quick-sort's worst case complexity is actually $O(n^2)$ however on average it is $O(n\log n)$. Still, even though it doesn't have guaranteed performance like merge sort or heap sort, it is a very fast algorithm and generally beats merge sort and heap sort in terms of the constants. It is also by default "in place", just like it's slow cousin, bubble sort.

Quicksort recently celebrated 60 years. Here is an useful article about it and here is the follow-up about it.