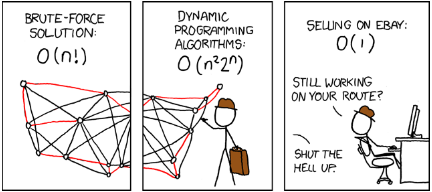
CS319: Scientific Computing (with MATLAB)

Sorting, Complexity, and `structs`

Niall Madden

Week 7: **9am and 4pm**, 22 Feb 2023



http://xkcd.com/399

Important: you should read:

- Learning MATLAB, Section 6.7 (Structures).
  https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9780898717662
- The MATLAB Guide, Chapters 18 and 19:
  https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669

In-Class test (updated)

We'll have an in-class test **10am** on Wednesday of next week (Week 8). There will be no 4pm lecture that day.

- Sample question at `https://www.niallmadden.ie/2223-CS319/CS319-SampleTest.pdf`
- Sample solutions are available on Blackboard.

This week, CS319 will be concerned with...

1. 1: A note on complexity
2. 2: Merge Sort
   - Why is Merge Sort is fast
3. 3: Comparing in practice
4. 4: The Password Problem
   - Algorithm (high-level)
   - Implementation
5. 5: Structures
   - Example: `pchip`

1: A note on complexity

In Lab 4 (right after this class) you'll be provided with code for a sorting algorithm, Bubble Sort. You'll then be challenged to write a "better" sorting function, using **Merge Sort**.

But what does "better" mean?

There are many ways that one algorithm could be considered superior to another, for example:

- takes less time to run;
- takes less memory to run;
- takes less time to program;
- is more accurate;
- is more reliable;
- ...?

1: A note on complexity

Focusing on efficiency, we now need a way of discussing how the time taken by an algorithm depends on the problem size.

The usual way to discuss this is in terms of "Big $\mathcal{O}$" notation, which classifies how, e.g., algorithms' run-times grow as the input size grows.

For example, if we say an algorithm for a problem of size n has complexity $\mathcal{O}(n^2)$, then we mean there is some constant, C such that the run-time is at most Cn^2.

Often, we don't really care too much about what C is. For example, if Algorithm 1 had complexity $0.1n^2$, and Algorithm 2 had complexity $100n$, then...

1: A note on complexity

The best to worst, some common complexities are

- $\mathcal{O}(1)$
- $\mathcal{O}(\log n)$
- $\mathcal{O}(n)$
- $\mathcal{O}(n \log n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^3)$
- $\mathcal{O}(2^n)$
- $\mathcal{O}(n!)$

The **Bubble Sort algorithm** too slow for project we will undertake: its worse-case complexity is $\mathcal{O}(N^2)$ for a list of length N.

Instead we'll implement the **Merge Sort** algorithm. It has complexity $\mathcal{O}(N \log N)$.

Merge Sort

- Split the list into two smaller lists,
- Split each of those into 2 smaller lists.
- Keep doing this until each list is of length 1.
- A list of length 1 is already sorted, so...
- Reassemble each of your sub-lists by merging these sorted list.

It is useful to write this as a **recursive algorithm**:

Recursive Merge Sort Algorithm

procedure $mergesort(L = a_1, a_2, \ldots, a_n)$
if $n > 1$ *then*
  $m := \text{floor}(n/2)$
  $L_1 := (a_1, a_2, \ldots, a_m)$
  $L_2 := (a_{m+1}, a_{m+2}, \ldots, a_n)$
  $L := merge\big(mergesort(L_1), mergesort(L_2)\big)$.
end if

So we need two functions:

(i) A `Merge()` function to merge two sorted list
(ii) A `MergeSort()` function that
  - splits the list in two,
  - calls `MergeSort()` for each half
  - calls the `Merge()` function

Example (Merge Sort)

Show how **Merge Sort** would sort the list

9 5 1 2 6 3 4 9 4

3: Comparing in practice

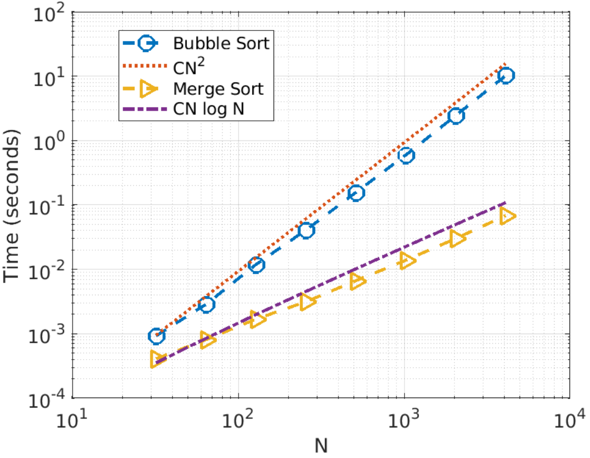
In Lab 3 you should find that...

- **Bubble Sort** has a worst-case complexity of $\mathcal{O}(N^2)$ for a list of length N.
- **Merge Sort** has a worst-case complexity of $\mathcal{O}(N \log N)$ for a list of length N.

This means that if we have a list of length N, then the expected time taken for the methods are $C_B N^2$ and $C_M N \log N$, for some constants C_B and C_M.

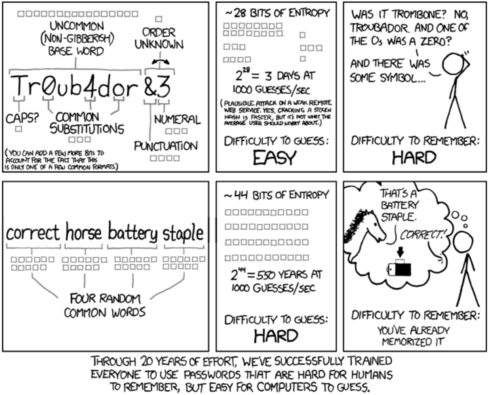
How to estimate these?

3: Comparing in practice



The data for this figure was collected using the `tic` and `toc` functions, which we saw before in Week 4.

4: The Password Problem



https://xkcd.com/936/

4: The Password Problem

In Lab 3, we considered the problem of sorting a long list of "stolen" passwords. Actually, our goal is the to determine the most common.

The source of the data is the infamous **RockYou** password file, a list of over 30,000,000 unencrypted passwords stolen from RockYou in 2009, and now widely available online.

The file contains one password per line, in no particular order. The first few are

```
password
mekster11
mekster11
progr4sm
khas8950
emilio1
holiday2
caitlin1
```

Given a list of 30,000,000 passwords, how shall we work out which 10 (say) occur most frequently?

Idea:

1. Load the list of passwords from a file.
2. Sort the list alphabetically, using MATLAB's `sort` function.
3. Create a list of words that contains no repetitions, using `unique`.
4. `[U, ai] = unique(P)` also returns a vector, `ai`, such that $U = P(ai)$. Since P is sorted, this can tell us the frequency. Example:

5. Sort the word frequency list, in descending order, storing the “key”.
6. Use the key to output the top 10.

The first step is to load array Passwords, from the file, count the number of entries, and sort it.

PasswordFrequency.m

```
%% Load the Passwords array
load UserAccount-1e6;
NumberOfPWDs = length(Passwords);
fprintf("Array Passwords has %d entries\n", NumberOfPWDs);
%% Sort the passwords
Passwords = sort(Passwords);
```

Make a list of the unique words, and their frequency.

PasswordFrequency.m

```
%% Find the most frequently occuring word.
%    - create a new list of unique words
%    - a corresponding count of the number on instances.
[UniqueWords, ai] = unique(Passwords);
WordFreq    = diff([ai; NumberOfPWDs+1]);
```

Sort the frequency count, in descending order, and use the key to to order the *UniqueWords* array.

PasswordFrequency.m

```
%% Sort by Frequency
% Again use the "sort" function, but keep the "key"
[WordFreq,key]=sort(WordFreq,'descend');
UniqueWords = UniqueWords(key);
```

Output the top 10:

PasswordFrequency.m

```
%% Output top 10
fprintf("The 10 most common words (and freqs) are:\n");
for i=1:10
   fprintf("\t%10s (%3d)\n", UniqueWords(i), WordFreq(i));
end
```

```
The 10 most common words (and freqs) are:
          password (215)
          iloveyou (175)
            123456 (126)
         password1 (117)
            abc123 (114)
         iloveyou1 ( 93)
          princess ( 93)
              love ( 77)
         princess1 ( 73)
```

5: Structures

So far, for all the arrays we have studied, we access specific elements using index/number. E.g, `x=[3 1 4 1 5 9]`, and access the 3rd element as `x(3)`.

A *`struct`* ure is a type of array where the entries have names. But more than that is true:

- Elements of the structure can be of different types;
- Elements can scalars, arrays, or even other structures.

5: Structures

In this simple example, we'll create a structure for a module.

ModuleStructure.m

```
%% Using a simple struture:
Module.code='2223-CS319';
Module.name='Scientific Computing';
Module.Students=[20123456, 19876543, 21212121];
Module.Graded=["A", "C", "B"];
disp(Module)
```

The variable *Module* is now of type struct.

```
>> Module
Module =
  struct with fields:
        code: 'CS319'
        name: 'Scientific Computing'
    Students: [20123456 19876543 21212121]
      Graded: ["A"    "C"    "B"]
```

5: Structures

We can access or set a struct's entries using the DOT operator:

```
>> Module.code='2223-CS319'
Module =
  struct with fields:
        code: '2122-CS319'
        name: 'Scientific Computing'
    Students: [20123456 19876543 21212121]
      Graded: ["A"    "C"    "B"]
```

There is lots more one can do with structs, such as creating arrays of structures, or stuctures with structures... but the main point today is as an introduction to user-defined **composite data types.**

However, it is also worth noting that some MATLAB functions return `stuct`s, including piecewise interpolation functions.

One can use the `pchip()` function to compute the piecewise cubic Hermite interpolant to a data set. E.g.,

```
    x=[0 .1 .5 1]
    y=[1, 0, 0.2, .3]
    Y = pchip(x,y, X); % Some big vector X
```

The *Y(i)* is the PCHIP interpolant to (x, y) evaluated at *X(i)*. But we cal also compute the interpolant itself.

```
>> p = pchip(x,y)
p =
  struct with fields:
        form: 'pp'
      breaks: [0 0.1000 0.5000 1]
       coefs: [3x4 double]
      pieces: 3
       order: 4
         dim: 1
```