CS319: Scientific Computing (with MATLAB)

CS319 Lab 4: Recursion and Complexity

Week 7, 2023

Niall Madden (Niall.Madden@UniversityOfGalway.ie)

- **What will I do?** Write some code to implement a Merge Sort algorithm and use it to sort a long list of strings.
- **Is that all?** No. You'll also time how long your code takes to run, and use that information to estimate the complexity of the algorithm.
- **What has this got to do with Scientific Computing?** Very frequently in Sci Comp, we need to estimate how long our programs will take to run before we run them.
- **But doesn't MATLAB have its own sort function?** Yes, and we should use it. But writing our own is instructive.

1. Bubble Sort

We wish to compare two sorting algorithms, verifying their complexity, and estimating how long they would take to sort a particular list.
The first method is **Bubble Sort**, which you can read about at https://en.wikipedia.org/wiki/Bubble_sort. We've also discussed it a little in class.
It can be implemented as

BubbleSort.m

```
function x = BubbleSort(x)
% Use the *Bubble Sort* algorithm to sort elements of a vector.
% For more on the algorithm, see
% <https://en.wikipedia.org/wiki/Bubble_sort>
n = length(x);
for i=n:-1:2
    for j=1:i-1
        if (x(j)>x(j+1))
            tmp = x(j+1);
            x(j+1) = x(j);
            x(j) = tmp;
        end
    end
end
```

1. Bubble Sort

Bubble Sort is famous for being easy to understand and code, and slow to run: the time taken to sort a list of N items is proportional to N^2.
This verify this, see the `TimeBubbleSort.m` script.

TimeBubbleSort.m

```
%% TimeBubbleSort.m : For 2122-CS319 Lab 3
clear;
k=0;
Ns = 2.^(8:15);
for N=Ns
   k=k+1;
   q = randi(N, 1, N); % list of N integers in [1,N]
   tic;  % start the stop-watch
   y=BubbleSort(q);
   TimeBubble(k)=toc;
   fprintf("N=%5d, Bubble Time=%8.5f\n", N, TimeBubble(k));
end
loglog(Ns, TimeBubble, '--o')
```

Modify the *TimeBubbleSort.m* program in the following ways.

1. We believe that the time taken is, approximately, CN^2, for some constant C. (We'll be more precise about what this means in lectures). Modify the `fprintf` line to output the value of C for each N.
2. Use a log-log plot to display the timing data, along with values of CN^2. You should be able to verify that the lines are parallel.
3. Fit a quadratic polynomial to the data, using `polyfit` (see lecture notes from Week 6). Using the result, and `polyval`, estimate how long it would take *BubbleSort()* to sort lists of numbers of length 100,000 and of length 1,000,000.

2. Sorting Strings

The file *UserAccount-1e6.mat* contains 100,000 strings, which are taken from a "stolen" password file (see lectures more information on this).

Warning

This is a real-world data set. Some of the words are of a crude and/or sexual nature. If you prefer, I can provide a sanitised version.

Load the file. The passwords are in a string array, called, *Passwords*.
As will be discussed in class, we wish to sort these, alphabetically. The `BubbleSort` function can do that (in theory). However, comparing strings is much slower that comparing doubles.
We want to know how long it would take Bubble Sort to sort this list, without actually doing it, because it would take too long.

2. Sorting Strings

1. Adapt the *TimeBubbleSort.m* code to time how long it would take to sort the first 8, 16, 32, ..., 1024 entries in the list.
2. Use this information to estimate how long it would take to sort the entire list.
3. This data set is actually part of a larger one of 30 million passwords. How long would that take?

3. Merge Sort

The **Merge Sort** algorithm is usually faster than Bubble Sort. It has complexity $\mathcal{O}(N \log N)$.

Merge Sort Algorithm

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

3. Merge Sort

Your next task is to:

1. Write MATLAB functions for *Merge()* and *MergeSort()*.
2. Test *MergeSort()* for double arrays. Can you verify its complexity?
3. Test *MergeSort()* for some subset of the Passswords array. Can you estimate how long it would take to sort the entire array?

You don't have to submit your work this week, but we will return to the idea of timing algorithms (and Live scripts) in Lab 4.