Exercises for 20 May 2020

1. Given the following pseudocode for matrix multiplication:

```
for i in [0, l):
    for j in [0, m):
        sum = 0

        for k in [0, n):
            sum += a[i,k] * b[k,j]

        c[i,j] = sum
```

   Count the number of arithmetic operations that will be performed in terms of l, m and n. What is the run-time complexity if l=m=n? Find values of c and n_0 to show that the algorithm's run-time is both $O(n^3)$ and $\Omega(n^3)$.

A: Run-time complexity is $l * m * (1 + n * (1 + 1) + 1) = 2n^3 + 2n^2$, l*m loops, 1 assignment, n loops, 1 add&equal, 1 multiply, 1 assignment.

For $O(n^3)$, c=3 and $n_0 = 2$, or $c \geq 4$ and $n_0 = 1$.

For $\Omega(n^3)$, c=2 or c=1, and $n_0 = 1$.

2. **Factorial**

   a. Write a naïve recursive algorithm for calculating the factorial of an integer. Analyze its time complexity. Write an iterative version of the same algorithm and analyze *its* time complexity. How do they compare?

   The time complexity is linear, which is $O(n)$. The time complexity of recursive version should be the same with the iterative version.

b. Implement both version of your algorithm in C++ and time their executions. How do they compare?

It turns out they're pretty close, as the following figure shows.

```
Factorial Recursive: Calculated 10! = 3628800 in 120 ns
Factorial Iterative: Calculated 10! = 3628800 in 135 ns
Factorial Recursive: Calculated 11! = 39916800 in 115 ns
Factorial Iterative: Calculated 11! = 39916800 in 117 ns
Factorial Recursive: Calculated 12! = 479001600 in 120 ns
Factorial Iterative: Calculated 12! = 479001600 in 135 ns
Factorial Recursive: Calculated 13! = 6227020800 in 118 ns
Factorial Iterative: Calculated 13! = 6227020800 in 124 ns
Factorial Recursive: Calculated 14! = 87178291200 in 133 ns
Factorial Iterative: Calculated 14! = 87178291200 in 135 ns
Factorial Recursive: Calculated 15! = 1307674368000 in 128 ns
Factorial Iterative: Calculated 15! = 1307674368000 in 127 ns
Factorial Recursive: Calculated 16! = 20922789888000 in 133 ns
Factorial Iterative: Calculated 16! = 20922789888000 in 127 ns
Factorial Recursive: Calculated 17! = 355687428096000 in 130 ns
Factorial Iterative: Calculated 17! = 355687428096000 in 130 ns
Factorial Recursive: Calculated 18! = 6402373705728000 in 138 ns
Factorial Iterative: Calculated 18! = 6402373705728000 in 128 ns
Factorial Recursive: Calculated 19! = 121645100408832000 in 133 ns
Factorial Iterative: Calculated 19! = 121645100408832000 in 135 ns
Factorial Recursive: Calculated 20! = 2432902008176640000 in 138 ns
Factorial Iterative: Calculated 20! = 2432902008176640000 in 138 ns
```

3. **Optional: Fibonacci sequence**
    1. As in the previous example, write both a naïve and an iterative algorithm for calculating a value, this time the i'th element of the Fibonacci sequence.

       Please see the code in "factorial.cpp"
    2. What is the time complexity of the iterative method?

       There is only one for loop, so the time complexity is simply $O(n)$.
    3. What is the time complexity of the recursive method? (**hint:** the Fibonacci numbers themselves grow with a bound of ≈Θ(1.6n).

       On the one hand, Fibonacci sequence can be expressed as:

       $F(n) = F(n-1) + F(n-2)$

       which is a linear homogeneous recurrence.

       Its characteristic equation is:

       $x^2 = x + 1$

       Solving characteristic equation:

       $x_1 = \frac{1+\sqrt{5}}{2}, x_2 = \frac{1-\sqrt{5}}{2}$

       So the solution of the linear homogeneous recurrence is:

       $F(n) = (x_1)^n + (x_2)^n$

       On the other hand, let $T(n)$ be the running time for the function call $F(n)$. Apparently,

       $$T(n) = \begin{cases} 1, n = 0, 1 \\ T(n-1) + T(n-2) + 4, N >= 2 \end{cases}$$

       Clearly, $T(N)$ and $F(n)$ are asymptotically the same, since both formulas have same form.

Hence, it can be said that:

$T(n) = (x_1)^n + (x_2)^n$

Since $x_2 < 0$, we can drop the second term.

$T(n) = O((\frac{1+\sqrt{5}}{2})^n)$

which is the tight upper bound of recursive Fibonacci algorithm.

Exercises for 22 May 2020

1. **Bubble sort complexity**
    1. Sketch the algorithm for the *bubble sort* of an array in pseudocode.

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        for i in rang(n-1) do
            for j in range(n-i-1) do
                if A[j] > A[j+1] then
                    swap(A[j], A[j+1])
                end if
            end for
        end for
end procedure
```

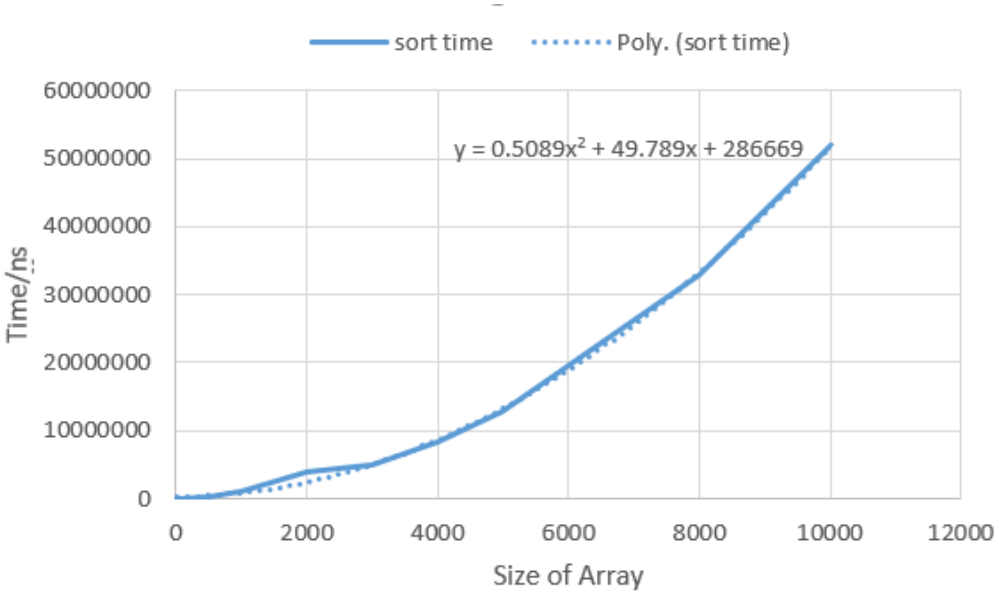
    2. Analyze your algorithm's run-time complexity in terms of n, the number of items in the array.

    $(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1 = \frac{n(n-1)}{2}$

    so the time complexity should be $O(n^2)$.
2. **Bubble sort implementation**
    1. Implement your bubble-sort algorithm as a C++ **function template**.

    Please refer to "bubble_sort.cpp"
    2. Measure and plot its run time for varying data sizes. You may find the C standard library's `rand()` function helpful when generating data[2].

    As the following figure shows, the time quadratically increase with the size of array.



3. **Optional: partitioning**
    - Write a C++ function that *partitions* an array or vector into two parts: the above-average values and the below-average values.
    - You may store the results in two separate vectors or, for slightly more challenge, rearrange values within the input vector (or array).

      Please refer to function split() in "bubble_sort.cpp".

      I don't understand what is "rearrange values within the input vector (or array)". Doesn't the bubble sort do the same thing? As long as we know the average, we can easily tell which part of the sorted array has all above-average values.