Algorithms and Data Structures: Homework #3

Due on February 24, 2020 at 23:00 PM

Henri Sota

Problem 3.1

Considering the following pairs of functions f and g, show for each pair whether or not it belongs to each of the relations $f \in \theta(g)$, $f \in O(g)$, $f \in o(g)$, $f \in \Omega(g)$, $f \in \omega(g)$, $g \in \theta(f)$, $g \in O(f)$, $g \in o(f)$, $g \in \Omega(f)$, or $g \in \omega(f)$.

a) $f(n) = 9n$ and $g(n) = 5n^3$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{9n}{5n^3} = \lim_{n\to\infty} \frac{\frac{9}{n^2}}{5} = \lim_{n\to\infty} \frac{9}{5n^2} = 0$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{5n^3}{9n} = \lim_{n\to\infty} \frac{5}{\frac{9}{n^2}} = \lim_{n\to\infty} \frac{5n^2}{9} = \infty$$

As a result, $f(n) \in o(g)$, $f(n) \in O(g)$, $f(n) \notin \Theta(g)$, $f(n) \notin \omega(g)$, $f(n) \notin \Omega(g)$ and $g(n) \in \omega(f)$, $g(n) \in \Omega(f)$, $g(n) \notin \Theta(f)$, $g(n) \notin o(f)$, $g(n) \notin O(f)$.

b) $f(n) = 9n^{0.8} + 2n^{0.3} + 14 \log n$ and $g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{9n^{0.8} + 2n^{0.3} + 14 \log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{9n^{0.8} + 2n^{0.3} + 14 \log n}{n^{0.5}} = \infty$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{\sqrt{n}}{9n^{0.8} + 2n^{0.3} + 14 \log n} = \lim_{n\to\infty} \frac{n^{0.5}}{9n^{0.8} + 2n^{0.3} + 14 \log n} = 0$$

As a result, $f(n) \in \omega(g)$, $f(n) \in \Omega(g)$, $f(n) \notin \Theta(g)$, $f(n) \notin o(g)$, $f(n) \notin O(g)$ and $g(n) \in o(f)$, $g(n) \in O(f)$, $g(n) \notin \Theta(f)$, $g(n) \notin \omega(f)$, $g(n) \notin \Omega(f)$.

c) $f(n) = n^2/\log n$ and $g(n) = n \log n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{n^2}{\log n}}{n \log n} = \lim_{n\to\infty} \frac{n^2}{n \log^2 n} = \infty$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n \log n}{\frac{n^2}{\log n}} = \lim_{n\to\infty} \frac{n \log^2 n}{n^2} = 0$$

As a result, $f(n) \in \omega(g)$, $f(n) \in \Omega(g)$, $f(n) \notin \Theta(g)$, $f(n) \notin o(g)$, $f(n) \notin O(g)$ and $g(n) \in o(f)$, $g(n) \in O(f)$, $g(n) \notin \Theta(f)$, $g(n) \notin \omega(f)$, $g(n) \notin \Omega(f)$.

d) $f(n) = (\log(3n))^3$ and $g(n) = 9 \log n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(\log(3n))^3}{9 \log n} = \lim_{n\to\infty} \frac{\log^3(3n)}{9 \log n} = \infty$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{9 \log n}{(\log(3n))^3} = \lim_{n\to\infty} \frac{9 \log n}{\log^3(3n)} = 0$$

As a result, $f(n) \in \omega(g)$, $f(n) \in \Omega(g)$, $f(n) \notin \Theta(g)$, $f(n) \notin o(g)$, $f(n) \notin O(g)$ and $g(n) \in o(f)$, $g(n) \in O(f)$, $g(n) \notin \Theta(f)$, $g(n) \notin \omega(f)$, $g(n) \notin \Omega(f)$.

Problem 3.2

a) Implement Selection Sort

Algorithm 1 Selection Sort[A,n]

```
for $i = 1$ to $n - 1$ do
  $min = i$
  for $j = i + 1$ to $n$ do
    if $A[j] < A[min]$ then
      $min = j$
    end if
  end for
  if $min$ != $i$ then
    $temp = A[min]$
    $A[min] = A[i]$
    $A[i] = temp$
  end if
end for
```

```cpp
void selectionSort(std::vector<int>& seq, int n) {
    for (int i = 0; i < n - 1; i++) {
        int temp, min = i;
        for (int j = i + 1; j < n; j++)
            if (seq[j] < seq[min])
                min = j;
        if (min != i) {
            temp = seq[min];
            seq[min] = seq[i];
            seq[i] = temp;
        }
    }
}
```

Listing 1: Implementation of Selection Sort in C++

b) Show that Selection Sort is correct

To prove the correctness of our algorithm, the loop invariant chosen must hold during initialization, iteration and termination of the loop. There are 2 for loops in the implementation of selection sort.

Outer Loop Invariant - For each iteration of the for loop, the subarray $A[1 \dots i-1]$ contains the $i-1$ smallest elements of A in increasing order.

- **Initialization:** At the beginning of the loop, there are no elements in the subarray $A[1 \dots i-1]$, so the subarray is empty and sorted.
- **Maintenance:** At the beginning of each iteration, the loop invariant must hold true. Elements of the subarray $A[1 \dots i-1]$ are placed in their correct place and all of them are smaller than the elements in the rest of the array, $A[i \dots n]$. The inner for loop will find the smallest element of the unsorted subarray, $A[i \dots n]$, and it will swap it with the element at position $A[i]$. The element at position i is therefore bigger than the elements of the subarray $A[1 \dots i-1]$ and smaller than the elements in the subarray $A[i+1 \dots n]$. On the next iteration, the sorted subarray will be $A[1 \dots i]$ and the invariant will be preserved due to the deductions given above.
- **Termination:** During the last iteration of the for loop, the last 2 elements of A could have been swapped or not, depending on their values. When i will equal n, the loop does not continue

to iterate through its body. Assuming the loop invariant held true during iteration, subarray $A[1 \ldots i-1]$, which is $A[1 \ldots n-1]$, shows that all the elements to the left of the last element are smaller than the last element. Since there is only one element that is unsorted and it is bigger than all the others, the element is in its correct spot. Therefore $A[1 \ldots n]$, which consists of all the elements of A is sorted, and that means A is sorted.

Inner Loop Invariant - For each iteration of the for loop, min holds the position of the smallest value of the elements of the subarray $A[i \ldots j-1]$.

- **Initialization:** At the beginning of the loop, min holds the position i, which is the position of the only element in the subarray, and therefore the smallest.
- **Maintenance:** At the beginning of each iteration, the loop invariant must hold true. min holds the position of the smallest element in the subarray $A[i \ldots j-1]$. If $A[j] < A[min]$, min is assigned value of j. Else min still holds the position of the smallest element found in subarray $A[i \ldots j-1]$. On the next iteration, min will still hold the position of the smallest value of the elements of the subarray $A[i \ldots (j+1)-1]$.
- **Termination:** During the last iteration of the loop, min would have been set the position of the last element, if $A[n]$ had a smaller value than min. When j will hold the value $n+1$, the loop does not continue to iterate through its body. Assuming the loop invariant held true during iteration, min will hold the position of the smallest element of the subarray $A[i \ldots j-1]$, which is $A[i \ldots n]$. After the last iteration, all elements of the subarray $A[i \ldots n]$ will have been checked to find the smallest element. Therefore min holds the position of the smallest value in the subarray $A[i \ldots n]$.

c) Generate random input sequences of length n as well as sequences of length n that represent **Case A** and **Case B** for the Selection Sort algorithm. **Case A:** the case which involves the most swaps (Hint: it is not a decreasingly ordered array). **Case B:** the case with the least swaps. Briefly describe how you generated the sequences (e.g., with a random sequence generator using your chosen language).

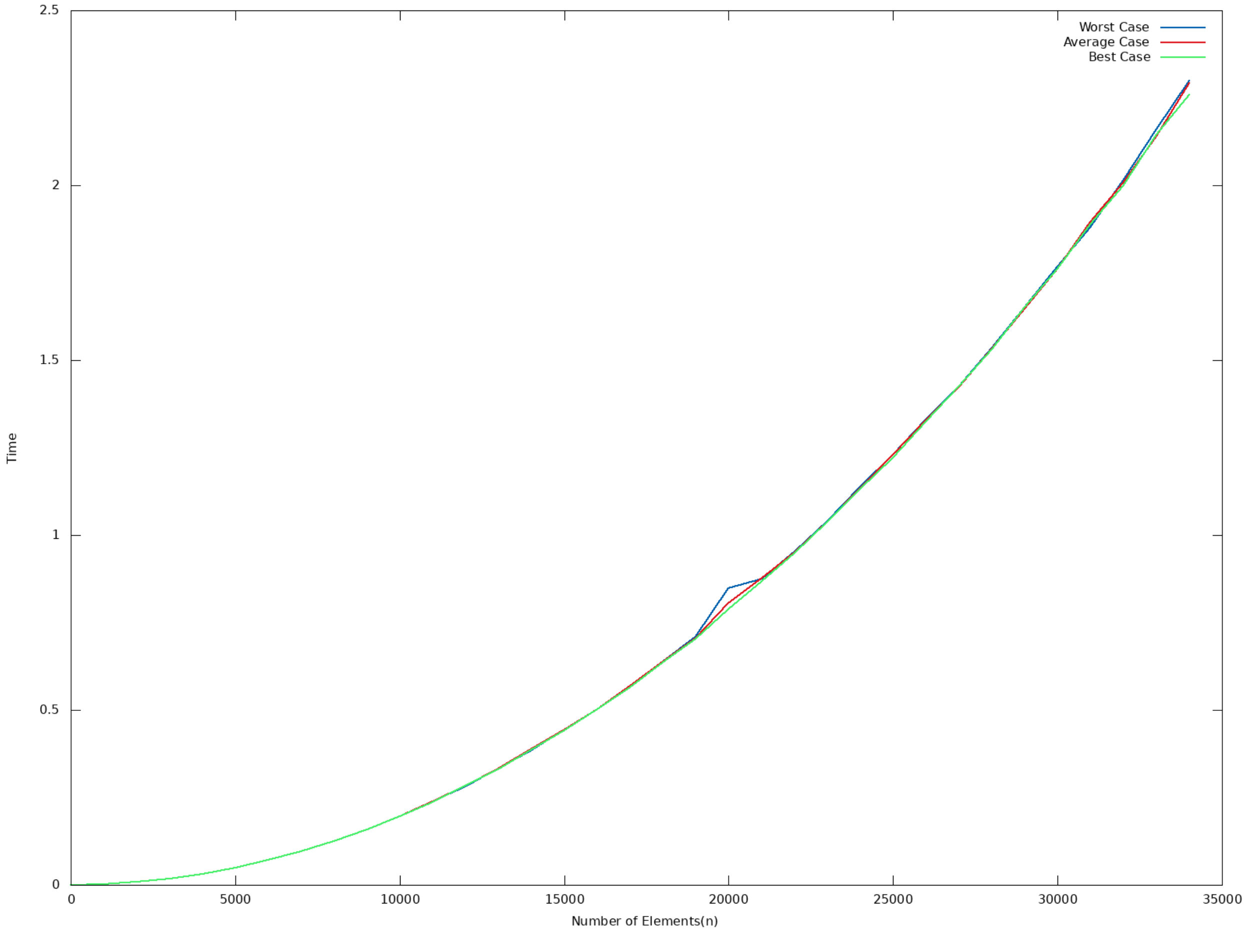
- Generating random sequences as input to the algorithm has been done using rand() function, part of cstdlib library, which produces a pseudo-random integral number. Output of rand() function is used to produce the remainder of division with *interval*, number from which the range of input values will come from.
- **Case A:** Case with the most amount of swaps happens on certain types of occasions. The highest number of swaps is $n-1$, one swap for each iteration of the outer for loop. This would happen only in cases when all the numbers in the sequence are unique. Theoretically, if the numbers form a single cycle with each other, it would take $n-1$ swaps to place them in their correct position. A single cycle of 3 elements would be: element at position X in the unsorted sequence is in the sorted position of Y, Y is in the place of Z, Z is in the place of X.
  Examples:
  - $[4, 5, 13, 18, 21, 27, 30, 32] \rightarrow [5, 21, 30, 13, 27, 32, 4, 18]$
    (Positions: $[A, B, C, D, E, F, G, H]$ would rearrange to $[B, E, G, C, F, H, A, D]$
  - $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \rightarrow [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
  - $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \rightarrow [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]$

  In case that the numbers are produced randomly, meaning there is a chance for duplicates, the number of swaps would be $n-k$, where k is the number of duplicates of the biggest number in the sequence (sorting increasingly).
  This case has been generated using rand() function as above to create a random sequence of numbers. A random index is computed using rand() to choose the first index to start the cycle

from. The first index would then be removed from the list in order to choose a new one in the next iteration of the for loop. The value at the first index will be placed in the position $swapIndex$, a random chosen index using rand() and the number in position $swapIndex$ will be saved in $swappingNumber$. The loop would iterate $n-2$ times to continue the cycle. In the last iteration, when $i = n-1$, $swappingNumber$ would be placed in the position $firstIndex$ which is empty.

- **Case B:** Case with the least amount of swaps theoretically happens when the input sequence has already been sorted. This case has been generated by choosing a random sequence of numbers using rand() function and sorting them increasingly using sort() function, part of the algorithm library.

d) Run the algorithm on the sequences from (c) with length n for increasing values of n and measure the computation times. Plot curves that show the computation time of the algorithm in **Case A**, **Case B**, and **average case** for an increasing input length n. Note that in order to compute reliable measurements for the **average case**, you have to run the algorithm multiple times for each entry in your plot. You can use a plotting tool/software of your choice (Gnuplot, R, Matlab, Excel, etc.).



The plot above has been created using gnuplot. It contains data from worstCaseData.txt, averageCaseData.txt and bestCaseData.txt files. Each file contains lines with 2 numbers in each one, first number is the input size, n, and second number is the time taken to sort the input size. Each entry of the plot has been calculated as the average of 5 different times taken from 5 iterations at each input size. Worst Case is represented by the blue line, average case is represented by the red line and best case is represented by the green line. Gnuplot code is found in the plot.gnu file.

e) Interpret the plots from (d) with respect to asymptotic behavior and constants.
The asymptotic behavior is determined by the number of comparisons made inside the inner loop. Algorithm has to go through the loop $n-1$ times, which is linear, and for each of them, it has to go through $n-i$ comparisons in order to find the smallest number in the remaining subarray, $A[i+1\dots n]$.

$$\sum_{i=1}^{n-1} n-i = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n^2-n}{2} = \frac{n^2-n}{2} = \frac{n(n-1)}{2} \in \theta(n^2)$$

The growth rate of the algorithm is quadratic and this is reflected in the graph, visible from its parabolic shape. All 3 lines have small differences between them, because assignment of *min* and number of swap processes take constant time and linear time respectively. Therefore as $n \to \infty$, time taken to sort will grow quadratically and still the differences will be small, as n^2 will dominate n of the swaps.