CSCI-GA.1170-001/002/003 Fundamental Algorithms February 9, 2021

Solutions to Problem 1 of Homework 1 (10 Points)

Name: Ben Wolfson (bw916) *Due: 8 pm on Thursday, February 11*

Collaborators: NetID1, NetID2

For each of the following pairs of functions $f(n)$ and $g(n)$, state whether f is $O(g)$; whether f is $o(g)$; whether f is $\Theta(g)$; whether f is $\Omega(g)$; and whether f is $\omega(g)$. (More than one of these can be true for a single pair!)

(a) $f(n) = n^{\log_2(\log_2 n)}$; $\quad g(n) = 2^{((\log_2 n)^2)}$.

Solution: □

Useful fact (1): For the next few problems we use the fact that $n \geq \log_n, \forall n > 0$. This can be shown by showing that $f(x) = x - \log x > 0, \forall x$. Taking the derivative we get that the maximum point is at $1 - 1/x = 0$ which occurs at x = 1. Therefore, $f(x)$ is always positive. We further note, that if we have constants a, b such that $f(x) = ax - b\log x > 0, \forall x$, the minimum of the function will be at b/a, and in order to make sure $f(x)$ is always greater than 0, it suffices to choose b (or a) such that $b - b/\log b/a > 0$.

The solution

We show that f is $O(g)$. $O(g(n)) = \{f(n) : \exists c \geq 0, n_0 \geq 0, 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$. We find such a n_0,c as follows (assume log, is $\log_2$):

- $n^{\log(\log n)} \leq c \cdot 2^{((\log n)^2)}$, taking the log
- $\log n \cdot \log(\log n) \leq \log c + (\log n)^2$, rearranging
- $\log n(\log(\log n) - \log n) \leq \log c$
- Now we note that $(\log(\log n) - \log n)$ is negative when $\log(\log n) \leq \log n$, but this is the same as solving $n \leq 2^n$, which is in fact true for any n. Therefore, the equation is true for $0 \leq c$, which is true for all c's.

By the above it can be seen that there is no c_2 that can be multiplied by $g(n)$ to make $\Theta(n)$ or $\Omega(n)$ (since for all $c \geq 0$, $f(n) \leq cg(n)$. Therefore f is $O(g(n))$ and $o(g(n))$.

(b) $f(n) = (\log_2(\log_2 n))^{\log_2 n}$; $\quad g(n) = n^{\log_2(\log_2(\log_2 n))}$.

Solution: Taking the log, one can see that the functions are equivalent. Therefore f is $O(g(n))$, $\Theta(g(n))$ and $\Omega(g(n))$. □

(c) $f(n) = (\log_2 n)^{10000}$; $\quad g(n) = \sqrt{n}$.

Solution: Showing O as above, we take the log of both functions and get that:

- $10000 \log\log n \leq \log c + 1/2 \log n$
- $10000 \log\log n - 1/2 \log n \leq \log c$

When $n_0 \geq 10.04$ the above is true for all c. Therefore, it can be seen that there is no c_2 that can be multiplied by $g(n)$ to make $\Theta(n)$ or $\Omega(n)$ (since for all $c \geq 0$, $f(n) \leq cg(n)$. Therefore f is $O(g(n))$ and $o(g(n))$. □

(d) $f(n) = (1.0001)^{0.00001n}$; $\quad g(n) = n^{1000001} + \log_2 n$.

Solution: To make the proof simpler we substitute as follows: $a = 1.0001, b = .00001, d = 1000001$ Showing O as above, we take the log of both functions and get that:

- $bn \log a \leq \log c + (d+1) \log n$, (where $n^{d+1} > n^d + \log n, \forall n > 2$)
- $bn \log a - (d+1) \log n \leq \log c$

but as we have shown above, the asymptotic behavior of $c \cdot n$ is greater than the asymptotic behavior of $\log n$, so there is no c that satisfies this equation. However, the "flipped equation" is satisfied ($bn \log a - (d+1) \log n \leq \log c$). Therefore, f is $\Omega(g(n))$, and $\omega(n)$, since there is a constant c, such that for all values $n \geq n_0$, $cg(n) \leq f(n)$. Namely we can multiply a function of the form bn by a constant c, such that it is strictly greater than a function of the form $d \log n$, the constant is d/b. See Useful Fact (1).

□

(e) $f(n) = n^{1/\log_2 n}$; $\quad g(n) = 10000$.

Solution: We show that f is $O(g)$.

- $n^{1/(\log n)} \leq c10000$
- $\log n \cdot 1/\log n \leq \log c + \log 10000$
- $1 \leq \log c \cdot 10000$ Here it is easy to see that $c = 2$. Since we are dealing with $f(n)$ being a constant, we can get exact bounds (using c_1 and c_2) as well as lower and upper bounds, both of which can be tight. So $g(n)$ is $O(g(n)), \Theta(g(n)), \Omega(g(n)), \omega(g(n))$.

□

CSCI-GA.1170-001/002/003 Fundamental Algorithms February 9, 2021

Solutions to Problem 2 of Homework 1 (12 points)

Name: Ben Wolfson (bw916) *Due: 8 pm on Thursday, February 11*

Collaborators: NetID1, NetID2

(a) (8 points) Write the following functions in the Θ-notation in the following simple form $c^n \cdot n^d \cdot (\log n)^r$, where c, d, r are constants. E.g., $20n^2 + 5n + 7 \sim n^2$, you can omit "Θ" in front of n^2. Briefly justify your answers

$$n^{0.99} \log n + 10, \ \log_2 \left(\frac{50n^4 + 10}{25n^4 - 7} \right), \ (1.0001)^n, \ 4^{\log_{16} n^2},$$
$$n^{\frac{n}{\log_5 n} + \frac{n}{\log_6 n} + \frac{n}{\log_7 n}}, \ n^{0.99} + 15(\log_2 n)^{100}, \ 216^{(4n+8)/12}, \ \log_5(n!) + 3n$$

Solution:

- $n^{0.99} \log n + 10 \sim n^{0.99} \log n$
- $\log_2 \left(\frac{50n^4+10}{25n^4-7} \right) \sim n^0$, since in the limit it tends to $\log_2 50/25 = 2$
- $(1.0001)^n \sim (1.0001)^n$
- $4^{\log_{16} n^2} = n \sim n$
- $n^{\frac{n}{\log_5 n} + \frac{n}{6 \log_6 n} + \frac{n}{\log_7 n}}$, dominated by the $\log_5 n$, so becomes $n^{\frac{n}{\log_5 n}} \sim n^n$, as n grows the bottom factor becomes unnecessary so $\sim n^n$.
- $n^{0.99} + 15(\log_2 n)^{100} \sim n$, since the log is dominated by n, $\sim n^{0.99}$
- $\log_5(n!) + 3n \sim n \log n$, as shown in the handout

□

(b) (4 points) List the simple functions you derived in part (a) in the order of asymptotic growth, from the smallest to the largest. (E.g., if one of the "complicated" functions in part (a) was $20n^2 + 5n + 7$, use the equivalent "simple" function n^2 in your ordering for this problem).

Solution:

- 2 or a constant
- $n^{.99}$
- n
- $n^{.99} \log n$
- $n \log n$
- 1.0001^n
- n^n

□

CSCI-GA.1170-001/002/003 Fundamental Algorithms February 9, 2021

Solutions to Problem 3 of Homework 1 (25 points)

Name: Ben Wolfson (bw916) *Due: 8 pm on Thursday, February 11*

Collaborators: NetID1, NetID2

Consider sorting n numbers stored in array A by first finding the largest element of A and exchanging it with the element in $A[n]$. Then find the second largest element of A and exchange it with $A[n-1]$. Continue in this manner for the first $n-1$ elements of A.

(a) (4 points) Write a (non-recursive) pseudocode for this algorithm, which is known as *selection sort*. Give the best-case and worst-case running times of selection sort in Θ-notation.

Solution:

```
SELECTION SORT(A, n)
    for j = n to 1
        max, max_i = A[j], j
        for i=j-1 to 1
            if A[i] > max
                max, max_i = A[i], i
        A[j], A[max_i] = A[max], A[j]
```

The best case solution is that the array is sorted, but the algorithm still searches through the array at every time step making $n-1$ compares on the first time step, $n-2$ etc. which is $\Theta(n^2)$. The worst case solution is similarly $\Theta(n^2)$, since the algorithm doesn't depend on the input array.

□

(b) (3 points) In this question, you will need to prove the correctness of your algorithm from part (a). In order to do this, you will first state a suitable loop invariant. You will then prove correctness using this loop invariant and induction.

Solution: Loop Invariant: At the start of the for loop, the elements $A[j...n]$ are in sorted order.
Initialization: True for $j = n$, since the elements of the sub array of 1 are in sorted order.
Maintenance: Assume the invariant is true at $j = k+1$, show that it is true for $j = k$. Since selection sort selects the maximum element of the "unsorted array" and puts in at the beginning of the sorted array. Let's call this element i. i is the largest of all of the elements of $A[1..n]$, by the max algorithm and it is less than or equal to all of the elements in $A[k+1...n]$, since it was not selected by the max algorithm to be in the $A[k+1...n]$ array (this can be show by a proof by contradiction (if i is greater than some element l in $A[k+1...n]$ than in the previous step where l was selected, it would not have been the max in the array (which contradicts the max function).

Termination: When the outer **for** loop ends, the array is in sorted order, since every element is greater than the one before it.

□

Let $A[1, \ldots, n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an *inversion* of A.

(c) (2 points) List all inversions of the array $[8, 5, 2, 7, 9]$.

Solution: $(8, 5), (8, 2), (8, 7), (5, 2)$

□

(d) (3 points) In the lecture, we discussed Insertion-Sort algorithm. This is a pseudocode of the same:

```
Insertion-Sort(A, n)
    for j = 2 to n
        key = A[j]
        i = j − 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i − 1
        A[i + 1] = key
```

Let $I(n)$ be the function representing the number of inversions of an input array $A[1, \ldots, n]$. Then, formulate an expression for:

(i) $S(n)$ where $S(n)$ is the function representing the number of swaps that the algorithm performs, in terms of n and $I(n)$.

(ii) $T(n)$ where $T(n)$ is the function representing the running time of the algorithm, in terms of $I(n)$ and n.

Justify your answers. (**Hint**: You may find it useful to first express $T(n)$ as a function of $S(n)$.)

Solution:

(i) Consider one iteration of insertion sort, where element $A[j]$ is being sorted. Insertion sort precisely un-does the inversions of $A[j]$, since it is swapped back before the elements that it is less than and ahead of the elements that is it greater than. (We have shown the loop invariant in class, so $A[j]$ is never swapped with elements less than it.) So the number of swaps is equivalent to the number of iterations, i.e. $S(n) = I(n)$.

(ii) Insertion sort has to look at every member of the array and then perform the number of swaps necessary to put it in proper order. If each of these is an inversion than the running time is $T(n) = I$, but if only one is, then it can be as high as $T(n) = I + n$

□

(e) (3 points) We will repeat part (d) but now for the SELECTION-SORT algorithm from part (a). Let $I(n)$ be the function representing the number of inversions of an input array $A[1, \ldots, n]$. Then, formulate an expression for:

(i) $S(n)$ where $S(n)$ is the function representing the number of swaps that the algorithm performs, in terms of n and $I(n)$.

(ii) $T(n)$ where $T(n)$ is the function representing the running time of the algorithm, in terms of $I(n)$ and n.

Justify your answers. (**Hint**: You may find it useful to first express $T(n)$ as a function of $S(n)$.)

Solution:

(i) Selection sort will always be swapping the element on the left with a the max of the array, so it will take at most $n - 1$ swaps.

(ii) Since Selection Sort is always comparing to every element it will take $n^2/2$ to run in the worst case.

□

(f) (4 points) Note that the INSERTION-SORT algorithm discussed in lecture works by comparing the key element to the largest element in the sorted portion of the array, and moving that element to the right, if it was larger than the key, and then comparing the key to successively smaller elements until the right position is found.

In this question, you will write a (non-recursive) pseudocode implementation of insertion sort, which works by comparing the key to the smallest element in the sorted portion of the array and then iterate by comparing to successively larger elements.

Solution:

```
INSERTION-SORT(A, n)
    for j = 2 to n
        key = A[j]
        i = 1
        while i > 0 and A[i] < key #here we find the place to insert
            i = i + 1
        for k = i + 1 to j #shift elements ahead by 1
            A[k] = A[k − 1]
        A[i] = key #insert key
```

□

(g) (3 points) We will repeat part (d) but now for the modified INSERTION-SORT algorithm from part (f). Let $I(n)$ be the function representing the number of inversions of an input array $A[1, \ldots, n]$. Then, formulate an expression for:

(i) $S(n)$ where $S(n)$ is the function representing the number of swaps that the algorithm performs, in terms of n and $I(n)$.

(ii) $T(n)$ where $T(n)$ is the function representing the running time of the algorithm, in terms of $I(n)$ and n.

Justify your answers. (**Hint**: You may find it useful to first express $T(n)$ as a function of $S(n)$.)

Solution:

(i) At every step, the algorithm is comparing the $A[j]$ element with all of the elements smaller than it and then swapping all of the elements above it into the locations greater than it i.e. the algorithm finds makes i compares, finds the $i+1$ spot for the $A[j]$ and then has to move all of the $i+1..j-1$ into the slots ahead of them, before inserting $A[j]$ into slot i. The numbers greater than j are precisely the inversions, so the algorithm makes $S(n) = I(n)$ swaps.

(ii) The algorithm (as described above) makes i compares and then $j - i$ swaps, leading to a total run time of j at every turn. Therefore, the algorithm $T(n) = (n)(n-1)/2$

□

(h) (3 point) Use parts (d), (e), and (g) to determine how each of the three algorithm performs for the following cases:

(i) $I(n) = \Theta(n^2)$

(ii) $I(n) = o(n^2)$

Justify your answer. Use this to conclude which algorithm is the best choice?

Solution: INSERT YOUR SOLUTION HERE □

We substitute in for the equations derived above for $T(n)$ and find that:

Algorithm	$I(n) = \Theta(n^2)$	$I(n) = o(n^2)$
Insertion Sort	$T(n) = n + I(n) = \Theta(n^2)$	$T(n) = n + I(n) = o(n^2)$
Selection Sort (inversion independent)	$T(n) = n^2 = \Theta(n^2)$	$T(n) = n^2 = \Theta(n^2)$
Modified Insertion Sort	$T(n) = n^2 = \Theta(n^2)$	$T(n) = n^2/2 = \Theta(n^2)$

Based on the above it would seem that the best algorithm is the insertion sort. This is because both the selection and the modified insertion sort always have a run time of $\Theta(n^2)$, while the Insertion Sort (in the case of $\emptyset(n^2)$ (a strict upper bound) has a faster run time.