ESO207A: Data Structures and Algorithms
Assignment 2 Solution

Ayush Bansal
Roll No. 160177

January 27, 2018

I Problem 1

We are given the **head** of a Linked List and I have to find out if it has a cycle or not.
I will use the following algorithm for accomplishing the task.

Algorithm 1.1 Cycle or not

```
function isCyclic(head)
    if head && (head → next) then                                  ▷ Checking if pointer is NULL
        return false
    ptr1 ← head                                                    ▷ ptr1 moves at Speed "1"
    ptr2 ← (head → next)                                           ▷ ptr2 moves at Speed "2"
    while (ptr1 ≠ ptr2) && (ptr1 → next) && (ptr2 → next) && (ptr2 → next → next) do
        ptr1 ← (ptr1 → next)
        ptr2 ← (ptr2 → next → next)
        if ptr1 = ptr2 then                                        ▷ Pointers equal→we have cycle
            return true
    return false
```

The above function will return **True** if there is a cycle else it will return **False**.

Proof of Correctness

Proof. We will take 2 pointers which traverse the list at different speeds (co-prime to each other). If there is a loop, the pointers will keep on traversing until they meet each other and return **true**. Since they are coprime to each other and the difference of their speed is 1, the **fast pointer** will keep gaining 1 node at a time on the **slower pointer** and finally meet it at some point because they are just moving in a cycle.
If there is no loop, the **fast pointer** will reach **NULL** at some point and will stop the loop and return **false**. □

Now, I will calculate the time complexity of the above algorithm.

Solution. According to the algorithm, the **fast pointer** will travel through the complete list when the **slow pointer** reaches the middle of the list and at this point, the **fast pointer** will begin its second traversal and will finally meet the first pointer in its traversal.
Thus, the time complexity will be $O(n)$, $\Omega(n)$, $\Theta(n)$. □

II Problem 2

2.1 Part i

Answer: $O(n^2)$.
The outer loop runs for N iterations and values of i are $\{0, 1, 2, \ldots, N-1\}$.
So the number of iterations $T(N)$ of inner loop can be calculated as follows:

$$T(N) = 0 + 1 + 2 + \cdots + N - 1$$
$$T(N) = \frac{N(N-1)}{2}$$

So the time complexity of the procedure will be $O(N^2)$.

2.2 Part ii

Answer: $O(n)$.
The outer loop runs for $\lfloor log(N-1) \rfloor + 1$ iterations and values of i are $\{1, 2, 4, 8, \ldots, 2^{\lfloor log(N-1) \rfloor}\}$.
So the number of iterations $T(N)$ of inner loop can be calculated as follows:

$$T(N) = 1 + 2 + 4 + 8 + \cdots + 2^{\lfloor log(N-1) \rfloor}$$
$$T(N) = 2^{\lfloor log(N-1) \rfloor + 1} - 1$$

So, the complexity of the procedure will be O(N).

2.3 Part iii

Answer: $O(log^2(n))$.
The outer loop runs for $\lfloor log(N-1) \rfloor + 1$ iterations and values of i are $\{1, 2, 4, 8, \ldots, 2^{\lfloor log(N-1) \rfloor}\}$.
So the number of iterations $T(N)$ of inner loop can be calculated as follows:

$$T(N) = 1 + 2 + 3 + \cdots + \lfloor log(N-1) \rfloor$$
$$T(N) = \frac{\lfloor log(N-1) \rfloor (\lfloor log(N-1) \rfloor - 1)}{2}$$

So, the complexity of the procedure will be $O(log^2(N))$.

2.4 Part iv

Answer: Infinite Loop (Never Terminates).

Solution. The loop starts by assigning N to i and it divides it by 2 each time until it is positive. Since, i is a real number, there will never be a time when $i \leq 0$, so the loop will never terminate. □

III Problem 3

According to definition of **Big Oh**, we will have to prove in each of the following cases:

$$f(x) \leq c \cdot g(x), c > 0$$

Putting the values of $f(x)$ and $g(x)$ in the above equation and assuming $n > 1$.

3.1 Part i

We are given $f(x) = alogn$ and $g(x) = log_a n$.
We have to show if $f(x) = O(g(x))$.

Proof.

$$\begin{aligned} alogn &\leq c \cdot log_a n \\ alogn &\leq \frac{clogn}{loga} \\ aloga &\leq c \\ \therefore c &> aloga \end{aligned}$$

Here, we can keep the value of constant $c > 0$ to be greater than $aloga$ and thus the function $c \cdot g(x)$ will always be greater than $f(x)$ $\forall n > 1$.
So, $f(x) = O(g(x))$. □

3.2 Part ii

We are given $f(x) = 7^{4n}$ and $g(x) = 2^{n/11}$.
We have to show if $f(x) = O(g(x))$ or $f(x) \neq O(g(x))$.

Proof.

$$\begin{aligned} 7^{4n} &\leq c \cdot 2^{n/11} \\ 4nlog7 &\leq logc \cdot \frac{nlog2}{11} \\ logc &\geq n\left(4log7 - \frac{log2}{11}\right) \end{aligned}$$

Since $4log7 > \frac{log2}{11}$ and n is positive, **RHS** is positive and and since it is dependent on n, **LHS** can't be a constant.
Thus, the function $f(x) \neq O(g(x))$. □

3.3 Part iii

We are given $f(x) = 2^{\sqrt{logn}}$ and $g(x) = \sqrt{n}$.
We have to show if $f(x) = O(g(x))$ or $f(x) \neq O(g(x))$.

Proof.

$$
\begin{aligned}
2^{\sqrt{logn}} &\leq c \cdot \sqrt{n} \\
log2 \cdot \sqrt{logn} &\leq logc \cdot \frac{logn}{2} \\
logc &\geq \sqrt{logn}\left(log2 - \frac{\sqrt{logn}}{2}\right)
\end{aligned}
$$

Putting $c = 1$ and taking $n \geq 2^4$.
Thus, $f(x) = O(g(x))$ and $c = 1, n \geq 2^4$. □

3.4 Part iv

We are given $f(x) = \sum_{i=1}^{n} \frac{1}{i}$ and $g(x) = logn$.
We have to show if $f(x) = O(g(x))$ or $f(x) \neq O(g(x))$.

Proof. We know that $logn = \int_1^n \frac{1}{x}dx$ and thus

$$
\begin{aligned}
\sum_{i=1}^{n} \frac{1}{i} &< 1 + logn \\
\sum_{i=1}^{n} \frac{1}{i} &< loge + logn
\end{aligned}
$$

Putting $c = 2$ and taking $n \geq e$.
Thus, $f(x) = O(g(x))$ and $c = 2, n \geq e$. □

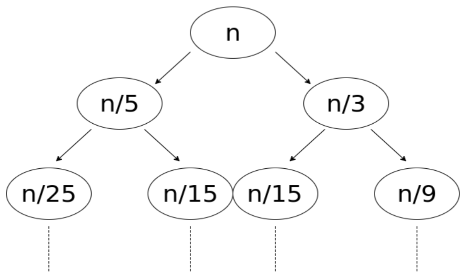
IV Problem 4

4.1 Part i

We are given the following recursion pattern:

$$T(n) = c * n + T(n/5) + T(n/3)$$

Walking through the recursion.



$$T(n) = n\left(1 + \frac{8}{15} + \frac{64}{225}\cdots\right)$$
$$T(n) = \frac{15n}{7}$$

So, the function $T(n)$ will be $O(n)$ i.e. tight upper bound will be Cn.

As observed above, a **G.P.** is formed in the recursion and the common ratio r must be less than 1. The common ratio r will be $a+b$ for the G.P. formed by the expression $T(n) = c*n+T(a*n)+T(b*n)$.
Conditions: $(a+b) < 1$ and $a, b > 0$.

4.2 Part ii

Answer: O(logn)
We are given the following recursion pattern:

$$T(n) = c + T(n/k), k > 1$$

Walking through the recursion.

$$T(n) = c + T(n/k)$$
$$T(n) = 2c + T(n/k^2)$$
$$T(n) = clog_k(n)$$

So, the function $T(n)$ will be $O(logn)$ i.e. tight upper bound will be $Clogn$.

4.3 Part iii

Answer: O(loglogn)
We are given the following recursion pattern:

$$T(n) = c + T(\sqrt[\alpha]{n}), \alpha > 1$$

Walking through the recursion.

$$\begin{aligned}
T(n) &= c + T(n^{1/\alpha}) \\
T(n) &= 2c + T(n^{1/\alpha^2}) \\
T(n) &= clog_\alpha(log_2(n))
\end{aligned}$$

So, the function $T(n)$ will be $O(loglogn)$ i.e. tight upper bound will be $Cloglogn$.

4.4 Part iv

Answer: O(n)
We are given the following recursion pattern:

$$T(n) = c * n + T(n/k), k > 1$$

Walking through the recursion.

$$\begin{aligned}
T(n) &= c * n + T(n/k) \\
T(n) &= c * (n + n/k) + T(n/k^2) \\
T(n) &= c * (n + n/k + n/k^2) + T(n/k^3) \\
T(n) &= \frac{cn}{k-1}
\end{aligned}$$

So, the function $T(n)$ will be $O(n)$ i.e. tight upper bound will be Cn.

V Problem 5

A function $f(n)$ is said to be $o(g(n))$ if

$$lim_{n\to\infty}\left|\frac{f(n)}{g(n)}\right| = 0 \tag{5.1}$$

A function $f(n)$ is said to be $\omega(g(n))$ if

$$lim_{n\to\infty}\left|\frac{f(n)}{g(n)}\right| = \infty \tag{5.2}$$

5.1 Part i

Statement: $4n+7$ is $o(n)$.
The above statement is **False**.

Proof. Putting values of $f(n)$ and $g(n)$ in (5.1) and using **L'Hôpital's rule**.

$$lim_{n\to\infty}\left|\frac{4n+7}{n}\right| \neq 0$$
$$lim_{n\to\infty}\left|\frac{4}{1}\right| = 4$$

Comparing above equation with (5.1) proves the result. □

5.2 Part ii

Statement: $4n+7$ is $o(n^2)$.
The above statement is **True**.

Proof. Putting values of $f(n)$ and $g(n)$ in (5.1) and using **L'Hôpital's rule**.

$$lim_{n\to\infty}\left|\frac{4n+7}{n^2}\right| = 0$$
$$lim_{n\to\infty}\left|\frac{4}{2n}\right| = 0$$

Comparing above equation with (5.1) proves the result. □

5.3 Part iii

Statement: $4n+7$ is $\omega(n)$.
The above statement is **False**.

Proof. Putting values of $f(n)$ and $g(n)$ in (5.1) and using **L'Hôpital's rule**.

$$lim_{n\to\infty}\left|\frac{4n+7}{n}\right| \neq \infty$$
$$lim_{n\to\infty}\left|\frac{4}{1}\right| = 4$$

Comparing above equation with (5.2) proves the result. □

5.4 Part iv

Statement: $4n+7$ is $\omega(log(n))$.
The above statement is **True**.

Proof. Putting values of $f(n)$ and $g(n)$ in (5.1) and using **L'Hôpital's rule**.

$$lim_{n\to\infty}\left|\frac{4n+7}{log(n)}\right| = \infty$$
$$lim_{n\to\infty}\left|\frac{4*n}{1}\right| = \infty$$

Comparing above equation with (5.2) proves the result. $\square$

VI Problem 6

6.1 Part i

There will be 2 cases for this question ($c > 1$ and $c \leq 1$).
Case (c>1) Answer: (b)

We are given $f(n) = c^n$ and $g(n) = n^k$, here c, k are constants.
I have to find the relation between the 2 given functions.

Solution. Consider the following limit

$$lim_{n\to\infty}\left|\frac{g(n)}{f(n)}\right|$$

If the value of the above limit is finite and 0 then $g(n) = o(f(n))$ but $g(n) \neq \omega(g(n))$.
Putting values of $f(n)$ and $g(n)$ in (5.1) and using **L'Hôpital's rule**.

$$lim_{n\to\infty}\left|\frac{n^k}{c^n}\right| = 0$$
$$lim_{n\to\infty}\left|\frac{k!}{c^n(logc)^k}\right| = 0$$

now, since $g(n) = O(f(n))$, $f(n) = \Omega(g(n))$.
Also, since $g(n) \neq \Omega(f(n))$, $f(n) \neq O(g(n))$ and thus $f(n) \neq \Theta(g(n))$. □

Case (c≤1) Answer: (a)

We are given $f(n) = c^n$ and $g(n) = n^k$, here c, k are constants.
I have to find the relation between the 2 given functions.

Solution. Since $c < 1$, the function $f(n) = c^n$ will keep on decreasing with increasing n and function $g(n) = n^k$ will keep on increasing with increasing n.
Thus, after certain $n = n_o$, $f(n) \leq g(n)$ and thus $f(n) = O(g(n))$. □

6.2 Part ii

Answer: (a),(b),(c)
We are given $f(n) = log_2(n)$, $g(n) = ln(n)$.
I have to find the relation between the 2 given functions.

Solution. We can also write $g(n)$ in the following manner

$$g(n) = \frac{log_2(n)}{log_2(e)}$$

So, we have $f(n) = c \cdot g(n)\ \forall n \in N$, here $c = \frac{1}{log_2(e)}$.
Thus, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, which shows that $f(n) = \Theta(g(n))$. □

6.3 Part iii

Answer: (b)
We are given $f(n) = n^2 log_2(n)$ and $g(n) = n log_2(n^3)$.
I have to find the relation between the 2 given functions.

Solution. Consider the following limit

$$lim_{n\to\infty} \left| \frac{g(n)}{f(n)} \right|$$

If the value of the above limit is finite and 0 then $g(n) = o(f(n))$ but $g(n) \neq \omega(g(n))$.
Putting values of $f(n)$ and $g(n)$ in (5.1) and using **L'Hôpital's rule**.

$$lim_{n\to\infty} \left| \frac{n log_2(n^3)}{n^2 log_2(n)} \right| = 0$$
$$lim_{n\to\infty} \left| \frac{3}{n} \right| = 0$$

Now, since $g(n) = O(f(n))$, $f(n) = \Omega(g(n))$.
Also, since $g(n) \neq \Omega(f(n))$, $f(n) \neq O(g(n))$ and thus $f(n) \neq \Theta(g(n))$.
So, the final answer is **(b)**. □

VII Problem 7

We are given an array A and I have to find the number of **inversions**.
An inversion for an **ordered pair** (A_i, A_j) is defined as below

$$A_i > A_j \text{ and } j > i$$

I have to devise an algorithm to count the **number** of **inversions** in the given array, assuming it's **length** to be n.
For this problem, I will use a **Modified Merge Sort** for better efficiency.
Below is a brief explanation before the actual pseudo-code.

Explanation

I will devise a procedure **MergeSort** which will take an array as argument and if it has more than one element, it will divide the array in 2 halves and call **itself** on both of the subarrays.
After that it will call another procedure **Merge**(explanation below) which will merge these 2 sub-arrays back.
Lastly, this procedure will take the **#inversions** from sub-arrays and merge procedure and **return** their sum.
The **Merge** procedure will take the 2 arrays as arguments and combine them comparing 1 element at a time from each array and will return the **#inversions**.
At the time of merging, if element being picked up is **smaller in 2nd** sorted sub-array, then all **remaining** elements of **1st** sorted sub-array are bigger than it and thus must be inverted.
Finally, we will have a **sorted array** and **#inversions**.

First one is the pseudo-code for the **MergeSort** Procedure.

Algorithm 7.1 MergeSort

```
procedure MERGESORT(array, start, end)
    inversions ← 0
    if start < end then                                                ▷ More than 1 element in array
        mid ← (start + end)/2
        inversions ← inversions + MergeSort(array, start, mid)         ▷ First sub-array
        inversions ← inversions + MergeSort(array, mid + 1, end)       ▷ Second sub-array
        inversions ← inversions + Merge(array, start, mid, end)        ▷ Merging sub-arrays
    return inversions
```

Second one is the pseudo-code for the **Merge** Procedure.

Algorithm 7.2 Merge

```
procedure MERGE(array, start, mid, end)
    inver ← 0
    itr ← 0                                            ▷ Interator for net element
    itr1 ← start                                       ▷ Interators for sub-arrays
    itr2 ← mid + 1
    while (itr1 ≤ mid)&&(itr2 ≤ end) do                ▷ Till both sub-arrays are non-empty
        if array[itr1] ≤ array[itr2] then              ▷ Second sub-array has bigger element
            temp[itr] ← array[itr1]                    ▷ Temp array to store for replacement
            itr1 ← itr1 + 1
            itr ← itr + 1
        else
            temp[itr] ← array[itr2]
            itr2 ← itr2 + 1
            itr ← itr + 1
            inver ← inver + (1 + mid − itr1)           ▷ 1st array's remaining are inversions
    while itr1 ≤ mid do                                ▷ Placing remaining elements of 1st if any
        temp[itr] ← array[itr1]
        itr1 ← itr1 + 1
        itr ← itr + 1
    while itr2 ≤ end do                                ▷ Placing remaining elements of 2nd if any
        temp[itr] ← array[itr2]
        itr2 ← itr2 + 1
        itr ← itr + 1
    itr ← 0
    itr1 ← start
    while itr1 ≤ end do                                ▷ Now placing all elements from temp to array
        array[itr1] ← temp[itr]
        itr1 ← itr1 + 1
        itr ← itr + 1
    return inver
```

Procedure Outline

The **MergeSort** procedure is focusing on breaking the problem in smaller parts and getting **#inversions** for each part, so main focus will be on **Merge** procedure.
The **Merge** procedure combines the sorted arrays into 1 sorted array and the **#inversions** are counted in the way that if element encountered in second sub-array is smaller than element of first sub-array then all sub-sequent elements of first sub-array will be bigger and thus must be **inverted**.
Since we are breaking the array till a sub-arrays have single element and recombining from there, the merge procedure will count the inversions and mergesort will add them up and thus give out total number of inversions.

Time Complexity

The time complexity of **Merge** procedure is **O(n)**.
For **MergeSort**

$$T(n) = 2T(n/2) + cn$$

By **Master's Theorem**, the final complexity will be **O(nlogn)**.