# Analysis of Algorithms - Home Work 1

CSE 548 Fall '19

*Prof. Jie Gao*

Submission By:

Narayan Acharya

112734365

## Contents

**1 Question 1** **2**

**2 Question 2** **4**

**3 Question 3** **5**

**4 Question 4** **7**

**5 Question 5** **8**

**References** **10**

Narayan Acharya
112734365

# 1 Question 1

**Big-O notation (10pts)** Prove or disprove (i.e., give counter examples) for the following claims. $f(n), g(n)$ are non-negative functions.

1. $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.
   **Solution:** Given $f(n)$ and $g(n)$ are non-negative functions, either function is less than the sum of the two functions:

$$f(n) \leq f(n) + g(n)$$
$$g(n) \leq f(n) + g(n)$$
$$\therefore max(f(n), g(n)) \leq f(n) + g(n)$$
$$\boxed{max(f(n), g(n)) = O(f(n) + g(n))} \tag{1}$$

   Also the max will be greater than the average of the two functions,

$$max(f(n), g(n)) >= \frac{(f(n) + g(n))}{2}$$
$$\boxed{max(f(n), g(n)) = \Omega(f(n) + g(n))} \tag{2}$$

   From results 1 and 2,

$$\boxed{max(f(n), g(n)) = \Theta(f(n) + g(n))}$$

2. $o(f(n)) \cap \omega(f(n)) = \emptyset$.
   **Solution:** $o(f(n))$, by definition, is a function that *grows strictly slower* than $f(n)$ while $\omega(f(n))$, by definition, is a function that *grows strictly faster* than $f(n)$.

$$o(f(n)) < f(n) : \forall n \tag{3}$$
$$\omega(f(n)) > f(n) : \forall n \tag{4}$$

   Equations 3 and 4 imply that these functions are on the opposite sides of the tightest possible growth rate function for $f(n)$ and can never be same or overlap. Therefore, the intersection of these functions is rightly an empty set, $\emptyset$ and the statement $o(f(n)) \cap \omega(f(n)) = \emptyset$ **is true**.

3. $(n + a)^b = \Theta(n^b)$, $a, b$ are positive integers.
   **Solution:** $(n + a)^b$ is a polynomial of order b. The expression expanded using Binomial Theorem[1] with all its terms will look like this:

$$(n + a)^b = \binom{b}{0}.n^b.a^0 + \binom{b}{1}.n^{(b-1)}.a^1 + \ldots + \binom{b}{b}.n^0.a^b \tag{5}$$

   Given $n \geq 1$, for terms in equation 5 where order of n is $\leq b$ we have,

$$\binom{b}{1}.n^{(b-1)}.a^1 \leq n \times \binom{b}{1}.n^{(b-1)}.a^1 : \forall n \geq 1 \tag{6}$$
$$\binom{b}{1}.n^{(b-2)}.a^1 \leq n^2 \times \binom{b}{1}.n^{(b-2)}.a^2 : \forall n \geq 1 \ldots \tag{7}$$

Narayan Acharya
112734365

Therefore from equations 5, 6 and 7we can deduce that, $\forall n \geq 1$,

$$\binom{b}{0}.n^b.a^0+\binom{b}{1}.n^{(b-1)}.a^1+\ldots+\binom{b}{b}.n^0.a^b \leq \binom{b}{0}.n^b.a^0+n\times\binom{b}{1}.n^{(b-1)}.a^1+n^2\times\binom{b}{1}.n^{(b-2)}.a^2\ldots$$

$$\begin{aligned}(n+a)^b &\leq n^b \times \left[\binom{b}{1}.a^1+\binom{b}{1}.a^2\ldots\binom{b}{b}.a^b\right]\\ &\leq c\times n^b\end{aligned}$$

where $c = \binom{b}{1}.a^1+\binom{b}{1}.a^2\ldots\binom{b}{b}.a^b$

$$\boxed{\therefore (n+a)^b = O(n^b)} \tag{8}$$

On similar lines, we can also prove $(n+a)^b = \Omega(n^b)$ using the fact that all terms other than that of order b for n contribute positively in equation 5,

$$\therefore (n+a)^b \geq \binom{b}{0}.a^0.n^b$$

$$\boxed{\therefore (n+a)^b = \Omega(n^b)} \tag{9}$$

From equations 8 and 9 we have,

$$\boxed{(n+a)^b = \Theta(n^b)}$$

4. $f(n) = O(f(n)^2)$.
**Solution:** Given that $f(n)$ is a non-negative function,

$$f(n) \leq c \times f(n) : \forall n \geq n_0, c > 0$$

$$\therefore f(n)^2 \leq c^2 \times f(n)^2$$

$$\boxed{\therefore f(n)^2 = O(f(n)^2)}$$

$f(n)^2$ may or may not be the tightest bound for $f(n)$ but it is asymptotically upper bound for a non-negative function $f(n)$.

5. $f(n) = O(g(n))$ implies that $2^{f(n)} = O(2^{g(n)})$.
**Solution:** Say $f(n) = k \times n$, given we have $f(n) = O(g(n))$,

$$\therefore g(n) = n$$

$$\therefore 2^{f(n)} = 2^{(k\times n)}, 2^{g(n)} = 2^n$$

$$\therefore O(2^{f(n)}) = O(2^{(k\times n)})$$

But, $O(2^{g(n)}) = O(2^n) \neq O(2^{(k\times n))} : \forall k \neq 1$

$$\boxed{\therefore 2^{f(n)} \neq O(2^{g(n)})}$$

Narayan Acharya
112734365

## 2 Question 2

Sort the following functions from asymptotically smallest to asymptotically largest. That is, the function $f(n)$ and the next function $g(n)$ must always follow that $f(n) \in O(g(n))$. If the two functions have asymptotic the same order, i.e., $f(n) = \Theta(g(n))$, then also indicate that. No need to write down proofs. Remember $\lg n = \log_2 n$.

$n$, $\lg n$, $\sqrt{n}$, $\sqrt{\lg n}$, $\lg\sqrt{n}$, $2^n$, $2^{\sqrt{n}}$, $\sqrt{2^n}$, $2^{\lg n}$, $\lg(2^n)$, $2^{\lg\sqrt{n}}$, $2^{\sqrt{\lg n}}$, $\sqrt{2^{\lg n}}$, $\lg(\sqrt{2^n})$, $\sqrt{\lg(2^n)}$, $3^n$, $3^{\sqrt{n}}$, $\sqrt{3^n}$, $3^{\lg n}$, $\lg(3^n)$, $3^{\lg\sqrt{n}}$, $3^{\sqrt{\lg n}}$, $\sqrt{3^{\lg n}}$, $\lg(\sqrt{3^n})$, $\sqrt{\lg(3^n)}$.

**Solution:** Below is table of the above functions sorted from asymptotically smallest to largest. *Note:* Even though Row 14 onwards all functions are exponential in nature they are asymptotically different and therefore have different values of $g(n) = O(f(n))$.

| Row | Function | Simplified Representation | $g(n) = O(f(n))$ |
|---|---|---|---|
| 1 | $\sqrt{\lg n}$ | $\sqrt{\lg n}$ | $\sqrt{\lg n}$ |
| 2 | $\lg\sqrt{n}$ | $\lg(n)/2$ | $\lg n, \Theta(\lg n)$ |
| 3 | $\lg n$ | $\lg n$ | |
| 4 | $2^{\sqrt{\lg n}}$ | $2^{\sqrt{\lg n}}$ | $2^{\sqrt{\lg n}}$ |
| 5 | $3^{\sqrt{\lg n}}$ | $3^{\sqrt{\lg n}}$ | $3^{\sqrt{\lg n}}$ |
| 6 | $\sqrt{n}$ , $\sqrt{2^{\lg n}}$ , $2^{\lg\sqrt{n}}$ | $\sqrt{n}$ | $\sqrt{n}, \Theta(\sqrt{n})$ |
| 7 | $\sqrt{\lg(2^n)}$ | $\sqrt{\lg(2)} \times \sqrt{n}$ | |
| 8 | $\sqrt{\lg(3^n)}$ | $\sqrt{\lg(3)} \times \sqrt{n}$ | |
| 9 | $3^{\lg\sqrt{n}}$ , $\sqrt{3^{\lg n}}$ | $n^{\lg\sqrt{3}}$ | $n^{\lg\sqrt{3}}$ |
| 10 | $\lg(\sqrt{2^n})$ | $n \times \lg(2)/2$ | $n, \Theta(n)$ |
| 11 | $\lg(\sqrt{3^n})$ | $n \times \lg(3)/2$ | |
| 12 | $n$ , $\lg(2^n)$ , $2^{\lg n}$ | $n$ | |
| 13 | $\lg(3^n)$ | $n \times lg(3)$ | |
| 14 | $3^{\lg n}$ | $n^{\lg 3}$ | $n^{\lg 3}$ |
| 15 | $2^{\sqrt{n}}$ | $2^{\sqrt{n}}$ | $2^{\sqrt{n}}$ |
| 16 | $3^{\sqrt{n}}$ | $3^{\sqrt{n}}$ | $3^{\sqrt{n}}$ |
| 17 | $\sqrt{2^n}$ | $\sqrt{2^n}$ | $\sqrt{2^n}$ |
| 18 | $\sqrt{3^n}$ | $\sqrt{2^n}$ | $\sqrt{2^n}$ |
| 19 | $2^n$ | $2^n$ | $2^n$ |
| 20 | $3^n$ | $3^n$ | $3^n$ |

Narayan Acharya
112734365

# 3 Question 3

Textbook [Kleinberg & Tardos] Chapter 2, page 67, problem #6.

**Solution:**

(a) The algorithm given in the question has an upper bound of the order $n^3$. This can be proved as follows:

The algorithm loops over i n-times. During each iteration over i it loops over j (n-i)-times. For adding all the elements from A[i] through A[j], there needs to be an additional loop, say with index k, (j-i+1)-times.

The above can be represented as a function of n as:

$$f(n) = \sum_{i=1}^{i=n} \sum_{j=i+1}^{j=n} \sum_{k=i}^{k=j} Add!$$

Expanding and simplifying the series:
*Note: The 'Add!' operation is considered constant time and hence omitted from the series below as it contributes only a factor of 1 to each term.*

$$\begin{aligned} f(n) &= (n-1).(1) + (n-2).(2) + (n-3).(3) + \ldots + (n-n).(n) \\ &= [(n + 2n + 3n + \ldots + n.n) - (1 + 4 + 9 + \ldots + n^2)] \end{aligned}$$

Simplifying this equation we get (Using formulas for adding a series n natural numbers and their squares):

$$f(n) = \frac{(n^3 - n)}{6}$$

The function is a polynomial of the order 3.

$$\boxed{\therefore f(n) = O(n^3)}$$

(b) To show that $\Omega(f(n)) = n^3$ we need $\exists c > 0$ for $n \geq n_0$ satisfying $f(n) \geq c \times n^3$. We will find a valid $c$ assuming that the condition holds.

$$\begin{aligned} \therefore \frac{(n^3 - n)}{6} &\geq c \times n^3 \\ \therefore \frac{(n^3 - n)}{6 \times n^3} &\geq c \\ \therefore c &\leq \frac{(n^2 - 1)}{6 \times n^2} \end{aligned}$$

A valid $c$ exists for all $n > 1$.

$$\boxed{\therefore f(n) = \Omega(n^3)}$$

(c) The given algorithm is inefficient as the innermost loop is not needed. Instead of iterating over all elements A[i] through A[j] while adding them, we can keep track of the sum after each iteration over i and use that as our seed for adding as we loop over j as shown.

Narayan Acharya
112734365

**Algorithm 1** Improved algorithm with $O(n^2)$

```
1: for i ← 1, 2, 3 . . . n do
2:     sumSoFar ← A[i]
3:     for j ← i + 1, i + 2, i + 3 . . . n do
4:         sumSoFar ← sumSoFar + A[j]
5:         B[i][j] ← sumSoFar
6:     end for
7: end for
```

The running time of this algorithm can be represented as a function of n as:

$$g(n) = \sum_{i=1}^{i=n} \sum_{j=i+1}^{j=n} Add!$$

Expanding the series:

$$\begin{aligned} g(n) &= (n-1) + (n-2) + (n-3) + \ldots + (n-n) \\ &= (n + n + n + \ldots + n) - (1 + 2 + 3 + \ldots + n) \end{aligned}$$

Simplifying this equation we get:

$$g(n) = \frac{(n^2 - n)}{2}$$

To prove that this algorithm is asymptotically better, we need to prove $lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

$$\begin{aligned} lim_{n\to\infty} \frac{g(n)}{f(n)} &= lim_{n\to\infty} \frac{\frac{(n^2-n)}{2}}{\frac{(n^3-n)}{6}} \\ &= lim_{n\to\infty} \frac{(n-1) \times 3}{n^2 - 1} \\ &= lim_{n\to\infty} \frac{(n-1) \times 3}{(n-1) \times (n+1)} \\ &= lim_{n\to\infty} \frac{3}{n+1} \\ &= 0 \end{aligned}$$

$$\therefore lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Hence, proposed algorithm's running time is asymptotically better.

Narayan Acharya
112734365

# 4 Question 4

Recall that in a heap we keep the elements such that the parent is smaller than children. Thus the root (stored as the first element in the array) is the smallest item in the array. Insertion and of a new element can be done in $O(\lg n)$ time. Deletion of an element can be done in $O(\lg n)$ time. Thus one start from an empty heap and insert elements one by one to build a heap of $n$ elements. Further, we can use it to sort $n$ elements. Once the heap is built, we remove the root (the smallest element), which is the smallest of the $n$ elements. Iterate and we will get all $n$ elements output in the increasing sorted order. This algorithm is called HeapSort.

1. What is the running time for HeapSort? Assume elements come in an arbitrary order and represent the running time in big-O notation. (2pts)
   **Solution:**

   In order to construct a heap with stable heap-order property for $n$ elements when they arrive in **arbitrary** order, we need $O(\log(n))$ operations for each new element that we insert i.e we need $O(n\log(n))$ time to construct the heap.

   To sort the elements we need to extract the root element and then *heapify* the heap to enforce heap-order property again for $n$ elements. The operation will take $O(\log(n))$ for each of the $n$ elements. Thus, to sort the $n$ elements we need $O(n\log(n))$ time.

   Thus, we need $O(n\log(n))$ to construct the heap and sort it when the elements arrive in arbitrary order.

2. What is the running time of HeapSort on $n$ elements in increasing order? Represent the running time in big-$\Theta$ notation. (4pts)
   **Solution:**

   In order to construct a heap with stable heap-order property for $n$ elements when they arrive in **ascending** order, we need $O(1)$ operation for each new element as they simply will be added to the end of the heap and heap-order property will be intact. Thus, we need $O(n)$ time to construct the heap.

   To sort the elements we need to extract the root element and then *heapify* the heap to enforce heap-order property again for $n$ elements. The operation will take $O(\log(n))$ for each of the $n$ elements. Thus, to sort the $n$ elements we need $O(n\log(n))$ time.

   Thus, we need $O(n\log(n))$ to construct the heap and sort it when the elements arrive in ascending order.

3. What is the running time of HeapSort on $n$ elements in decreasing order? Represent the running time in big-$\Theta$ notation. (4pts)
   **Solution:**

   The case when numbers are in **descending** order is the same as they arrive in arbitrary order, asymptotically speaking. We will need similar operations to construct the heap and extract the elements for sorting.

   Thus, we need $O(n\log(n))$ to construct the heap and sort it when the elements arrive in descending order.

Narayan Acharya
112734365

# 5 Question 5

**Young Tableaus** An $m \times n$ Young tableau is an $m \times n$ matrix such that the entries of each row are in increasing order from left to right and the entries of each column are in increasing order from top to bottom. Some of the entries of each column may be $\infty$, which are treated as nonexistent elements. A Young tableau with some elements as $\infty$ is not full.

Give an algorithm that extracts MIN (i.e., delete the minimum element and restore the matrix to be a Young tableau) on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time.

**Solution:**

For a Young Tableau, with $m$ rows and $n$ columns, we extract the smallest number from the $(0,0)$ index and put $\infty$ in its place. We then try to use recursion and move this $\infty$, now at $(0,0)$, to its appropriate index in order to restore the Young Tableau property[2].

Consider the following algorithm that extracts minimum from Young Tableau matrix and then restores the Young Tableau property. Notice that each recursive call moves the largest number (the one that we moved to $(0,0)$) one step closer to its intended position by comparing it with the next row below and/or column to the right. Worst case, we need to move the largest number from $(0,0)$ to end of the matrix at index $(m-1, n-1)$. In this case we end up moving making $m+n$ moves (or recursive calls) to get this done. Therefore, this algorithm is bound by $O(m+n)$ to extract the minimum value out of it.

**Algorithm 2** Algorithm to extract minimum from Young Tableau

```
1:  procedure ExtractMin(YT)                    ▷ Extracts minimum from Young Tableau YT
2:      extractedMin = YT[0][0]
3:      YT[0][0] = ∞
4:      MakeYT(YT, 0, 0)
5:      return extractedMin
6:  end procedure
7:  procedure MakeYT(T, i, j)                   ▷ Restores Young Tableau property of matrix T
8:      m ← len(T)                                                    ▷ m rows of T
9:      n ← len(T[0])                                                 ▷ n columns of T
10:     if i < m and j < n then
11:         current ← T[i][j]                   ▷ Check if next row or column exist and assign
12:         nextRow ← T[i+1][j] if i+1 ≠ m else None
13:         nextCol ← T[i][j+1] if j+1 ≠ n else None
14:         if nextRow = None and nextCol = None then
15:             return                          ▷ No number to the right or below to compare. Done!
16:         end if
17:                                                   ▷ We have 2 numbers to choose from
18:         if nextRow ≠ None and nextCol ≠ None then
19:             if nextRow ≥ nextCol and nextRow > current then          ▷ Swap with next row
20:                 T[i][j] ← nextRow
21:                 T[i+1][j] ← current
22:                 MakeYT(T, i+1, j)
23:             else                                                  ▷ Swap with next column
24:                 T[i][j] ← nextCol
```

```
25:                 T[i][j + 1] ← current
26:                 MakeYT(T, i, j + 1)
27:             end if
28:         end if
29:                                          ▷ Only one of next row or column available
30:         if nextCol = None and nextRow > current then
31:             T[i][j] ← nextRow
32:             T[i + 1][j] ← current
33:             MakeYT(T, i + 1, j)
34:         else
35:             if nextRow = None and nextCol > current then
36:                 T[i][j] ← nextCol
37:                 T[i][j + 1] ← current
38:                 MakeYT(T, i, j + 1)
39:             end if
40:         end if
41:     end if
42: end procedure
```

## References

[1] Binomial Theorem,
https://en.wikipedia.org/wiki/Binomial_theorem

[2] Young Tableau,
https://en.wikipedia.org/wiki/Young_tableau