Advanced Algorithms Homework#1

1. Find and prove (e.g. using the Master Theorem or other method) tight bounds for the following recurrences (where h is a positive constant). To get full grade, you must correctly justify your answer (i.e. lucky guesses will not help):

a) T(n) = 8T $(\frac{n}{16})$ + $\frac{n}{\sqrt[4]{n}}$ + $\frac{1}{\sqrt{n}}$ $\quad n \geq 16$

Solution: The given problem can be solved by using Master Theorem. According to Master Theorem the recurrence equation should be of form T(n) = a T $(\frac{n}{b}) + f(n)$ for n ≥ b where a ≥ 1 and b > 1 .

Theorem: Let c = $\log_b a$. The recurrence T(n) has the following bound (for $\epsilon > 0$):
1. If f(n) ∈ $O(n^{c-\epsilon})$ $then\ T(n) \in \Theta(n^c)$
2. If f(n) ∈ $\Theta(n^c \log k\, n)$ $for\ some\ k \geq 0\ then\ T(n) \in \Theta(n^c \log k + 1\, n)$.
3. If f(n) ∈ $\Omega(n^{c+\epsilon})$ $and\ af(\frac{n}{b}) \leq df(n)\ for\ 0 < d < 1$ $(this\ is\ called\ regularity\ condition)\ then\ T(n) \in \Theta(f(n))$.

Now in the above Problem a = 8, b = 16, hence c = $\log_b a$ = $\log_{16} 8$ = 0.75
F(n) = $\frac{n}{\sqrt[4]{n}} + \frac{1}{\sqrt{n}} = n^{\frac{3}{4}} + n^{\frac{-1}{2}} = n^{\frac{3}{4}}$ $(By\ taking\ the\ max\ powers\ and\ ignoring\ the\ least)$
F(n) = $n^{\frac{3}{4}} \in \Theta(n^c \log k\, n)\ for\ k = 0$.
From the **2nd case of Master Theorem, $T(n) \in \Theta(n^c \log k + 1\, n) = \Theta(n^{\frac{3}{4}} \log n)$**

b) T(n) = 8T $(\frac{n}{8})$ + $n^{ln\, n}$ $\quad n \geq 8$

Solution: The given problem can be solved by using Master Theorem. According to Master Theorem the recurrence equation should be of form T(n) = a T $(\frac{n}{b}) + f(n)$ for n ≥ b where a ≥ 1 and b > 1 .

Theorem: Let c = $\log_b a$. The recurrence T(n) has the following bound (for $\epsilon > 0$):
1. If f(n) ∈ $O(n^{c-\epsilon})$ $then\ T(n) \in \Theta(n^c)$
2. If f(n) ∈ $\Theta(n^c \log k\, n)$ $for\ some\ k \geq 0\ then\ T(n) \in \Theta(n^c \log k + 1\, n)$.
3. If f(n) ∈ $\Omega(n^{c+\epsilon})$ $and\ af(\frac{n}{b}) \leq df(n)\ for\ 0 < d < 1$ $(this\ is\ called\ regularity\ condition)\ then\ T(n) \in \Theta(f(n))$.

Now in the given Problem a = 8, b = 8, hence c = $\log_b a$ = $\log_8 8$ = 1
F(n) = $n^{ln\, n} = n^{ln\, 8} = n^{2.02.079441541779}$ $(for\ n \geq 8)$
F(n) = $n^{2.079} \in \Omega(n^{c+\epsilon})\ for\ \epsilon =$ 2.0794415417 – 1 = 1.079 ≅ and af $(\frac{n}{b}) = 8(\frac{n}{8})^{ln\, \frac{n}{8}} = 8\, \frac{n^{ln\, \frac{n}{8}}}{8^{ln\, \frac{n}{8}}} = 8\, \frac{n^{ln\, n - ln\, 8}}{8^{ln\, n - ln\, 8}} = 8\, \frac{\frac{n^{ln\, n}}{n^{ln\, 8}}}{\frac{8^{ln\, n}}{8^{ln\, 8}}} =$
$8\left[\frac{n^{ln\, n}}{n^{ln\, 8}} * \frac{8^{ln\, 8}}{8^{ln\, n}}\right] = \frac{8 * 8^{ln\, 8}}{n^{ln\, 8} * 8^{ln\, n}} = \frac{75.4958}{n^{log\, 64}}\ n^{ln\, n} = \frac{75.4958}{n^{log\, 64}}\ f(n) \leq df(n)\ for\ d = \frac{75.4958}{n^{log\, 64}} < 1$
From the **3rd case of Master Theorem, $T(n) \in \Theta(n^{ln\, n})$**.

c) T(n) = 19T $(\frac{n}{11})$ + $n\sqrt[5]{n}$ $\quad n \geq 11$

Solution: The given problem can be solved by using Master Theorem. According to Master Theorem the recurrence equation should be of form T(n) = a T $(\frac{n}{b}) + f(n)$ for n ≥ b where a ≥ 1 and b > 1 .

Theorem: Let c = $\log_b a$. The recurrence T(n) has the following bound (for $\epsilon > 0$):
1. If f(n) ∈ $O(n^{c-\epsilon})$ $then\ T(n) \in \Theta(n^c)$
2. If f(n) ∈ $\Theta(n^c \log k\, n)$ $for\ some\ k \geq 0\ then\ T(n) \in \Theta(n^c \log k + 1\, n)$.
3. If f(n) ∈ $\Omega(n^{c+\epsilon})$ $and\ af(\frac{n}{b}) \leq df(n)\ for\ 0 < d < 1$ $(this\ is\ called\ regularity\ condition)\ then\ T(n) \in \Theta(f(n))$.

Now in the given Problem a = 19, b = 16, hence c = $\log_b a = \log_{11} 19 = 1.2279264289$

F(n) = $n\sqrt[5]{n} = \boldsymbol{n^{\frac{6}{5}}} = \boldsymbol{n^{1.2}} \in O(\, n^{c-\epsilon}\,)\ for\ \epsilon = 1.2279264289 - 1.2 = 0.027 \cong$

From the **1st case of Master Theorem, $\boldsymbol{T(n) \in \Theta(n^{1.227})}$.**

d) T(n) = $\frac{6}{5}\boldsymbol{T}(\frac{6n}{7}) + \boldsymbol{n\sqrt[5]{n}}$ $\qquad\qquad$ $\boldsymbol{n > 1}$

Solution: The given problem can be solved by using Master Theorem. According to Master Theorem the recurrence equation should be of form T(n) = a T $(\frac{n}{b}) + f(n)$ for n ≥ b where a ≥ 1 and b > 1 .

Theorem: Let c = $\log_b a$. The recurrence T(n) has the following bound (for $\epsilon > 0$):

1. If f(n) ∈ $O(\, n^{c-\epsilon}\,)\ then\ T(n) \in \Theta(n^c\,)$
2. If f(n) ∈ $\Theta(n^c \log k\, n)\ for\ some\ k \geq 0\ then\ T(n) \in \Theta(n^c\ \log k + 1\, n)$.
3. If f(n) ∈ $\Omega(n^{c+\epsilon})\ and\ af(\frac{n}{b}) \leq df(n)\ for\ 0 < d < 1\ (this\ is\ called\ regularity\ condition)\ then\ T(n) \in \Theta(f(n))$.

Now in the given Problem a = $\frac{6}{5} = \boldsymbol{1.2}, \boldsymbol{b = \frac{7}{6} = 1.1666666667, hence}$ c $= \log_b a = \log_{\mathbf{1.1666666667}} 1.2 = 1.1827489633$

F(n) = $n\sqrt[5]{n} = \boldsymbol{n^{\frac{6}{5}}} = \boldsymbol{n^{1.2}} \in \Omega(n^{c+\epsilon})\ for\ \epsilon = 1.2 - 1.1827489633 = 0.01725 \cong$ and af$(\frac{n}{b}) = 1.2f(\frac{n}{1.667}) =$

$1.2\left(\frac{n}{1.667}\right)\left((\frac{n}{1.667})^{\frac{1}{5}}\right)$

$= \frac{1.2}{1.1667^2}\left(n\sqrt[5]{n}\right) = \frac{1.2}{1.1667^2}\ f(n) \leq df(n)\ for\ d = \frac{1.2}{1.1667^2} < 1$

From the **3rd case of Master Theorem, $\boldsymbol{T(n) \in \Theta(n\sqrt[5]{n})}$**

e) T(n) = $\frac{3}{5}\ \boldsymbol{T}(\frac{3n}{5}) + \frac{h}{n}$ $\qquad$ $\boldsymbol{n > 1}$

Solution: The given problem cannot be solved by using Master Theorem. According to Master Theorem the recurrence equation should be of form T(n) = a T $(\frac{n}{b}) + f(n)$ for n ≥ b where a ≥ 1 and b > 1 . But in this case the value of 'a' is not satisfied. Therefore, it cannot be solved with the help of Master Theorem. For this we have another approach or method. That is **Prove and Guess by induction.**

Let's find a close formula for T(n) above. This can be done by algebraic iteration as follows-

Given **T(n) = $\frac{3}{5}\ \boldsymbol{T}(\frac{3n}{5}) + \frac{h}{n}$**

T $(\frac{3n}{5}) = \frac{3}{5}T(\frac{3}{5}\ \frac{3n}{5}) + \frac{5h}{3n} \implies T(n) = \frac{3}{5}\left[\frac{3}{5}T(\frac{9n}{5}) + \frac{5h}{3n}\right] + \frac{h}{n} = \frac{9}{25}T(\frac{9n}{25}) + \frac{2h}{n}$

$T(\frac{9n}{25}) = \frac{3}{5}T(\frac{3}{5}\ \frac{9n}{25}) + \frac{25h}{9n} \implies T(n) = \frac{9}{25}\left[\frac{3}{5}T(\frac{27n}{125}) + \frac{25h}{9n}\right] + \frac{2h}{n} = \frac{27}{125}T(\frac{27n}{125}) + \frac{3h}{n}$

After so 'k' number of iterations we get---$\boldsymbol{T(n) = (\frac{3}{5})^k + T[(\frac{3}{5})^k\, n] + \frac{kh}{n}}$

We know that T (1) = 1 and this is a way to end the derivation above. For that lets assume n = $(\frac{5}{3})^k \implies \log_{\frac{5}{3}} n = k$

Substituting the value of k in the generalized equation-

T(n)= $(\frac{3}{5})^{\log_{\frac{5}{3}} n} T\left[(\frac{3}{5})^{\log_{\frac{5}{3}} n}\, n\right] + \frac{\log_{\frac{5}{3}} n}{n} h = (\frac{3}{5})^{-\log_{\frac{3}{5}} n} T\left[\frac{n}{(\frac{3}{5})^{\log_{\frac{3}{5}} n}}\right] + \frac{\log_{\frac{5}{3}} n}{n} h = \frac{1}{n}T(1) + \frac{\log_{\frac{5}{3}} n}{n} h = O(\frac{log\ n}{n})$

We have guessed the bound that is T(n) = $\frac{log\ n}{n}$

Now we need to prove it by using induction –

We know **T(n) = $\frac{3}{5}\ \boldsymbol{T}(\frac{3n}{5}) + \frac{h}{n} = \frac{3}{5}\left(\frac{log\frac{3n}{5}}{\frac{3n}{5}}\right) + \frac{h}{\frac{3n}{5}} = \frac{log\frac{3n}{5}}{n} + \frac{5h}{3n} = \frac{log\ n}{n} + \frac{1}{n} = \frac{1}{n}[\mathbf{1} + \boldsymbol{log\ n}] = \frac{\boldsymbol{log\ n}}{n}$ $(\boldsymbol{As\ we\ expected})$**

(I have ignored the constants in the above simplification)

Therefore, tight bound for the following recurrence is T(n) = $\boldsymbol{\Theta}(\frac{\boldsymbol{log\ n}}{\boldsymbol{n}})$

2. Consider the following basic problem. You're given an array A consisting of n integers A [1], A [2], . . . , A[n]. You'd like to output a two-dimensional n-by-n array B in which B[i, j] (for i < j) contains the sum of array entries A[i] through A[j]—that is, the sum A[i]+ A[i + 1]+ ... + A[j]. (The value of array entry B[i, j] is left unspecified whenever i ≥ j, so it doesn't matter what is output for these values.) Here's a simple algorithm to solve this problem

```
For i = 1, 2, ……., n
    For j = i + 1, i + 2, ……, n
        Add up array entries A[i] through A[j]
        Store the result in B [i, j]
    End for
End for
```

(a) For some function f that you should choose, give a bound of the form O(f(n)) on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

Solution: Here is a small Python Program for the given Algorithm.

```
For i in range (len(A)): ----------O (n)
    For j in range (i+1, len(A)): ----------O (n)
        Sum = 0 ----------O (1)
        For k in range (i, j+1): ----------O (n)
            Sum = Sum + A[k] ----------O (1)
        B[i][j] = Sum ----------O (1)
```

The time complexities for each line in the above program can be explained as follows-

- Since the first loop iterates through all the elements in the array, the upper bound for that loop is O(n).
- Likewise, the second loop iterates from i+1 to the length of the array and when i = 0 the loop iterates for (n-1) times which also has an upper bound of O(n).
- Similarly, when i = 0 and j = n the third loop iterates for (n-1) times which also has an upper bound of O(n)

Therefore, the time complexity for the given algorithm is T(n) = O(n) * O(n) * O(1) * O(n) = O(n^3)

T(n) = O (n^3) is the running time of this algorithm on an input size n

(b) For this same function f, show that the running time of the algorithm on an input of size n is also Ω(f(n)). (This shows an asymptotically tight bound of (Θf(n)) on the running time.)

Solution: The $\boldsymbol{\Omega}\ notation\ can\ be\ defined\ as\ \Omega(g(n)) = \{f(n): such\ that\ 0 \leq cg(n) \leq\ f(n)$. g(n) is asymptotic lower bound for f(n). Our aim is to give rate of growth g(n) which is less than or equal to the given algorithm's rate of growth f(n).

We have f(n) = n^3 and we also know that $\boldsymbol{0 \leq cg(n) \leq f(n)}\ \Longrightarrow cg(n) \leq f(n)$

$$\Longrightarrow cn^3 \leq n^3$$

$$\Longrightarrow (1)n^3 \leq n^3$$

$$\Longrightarrow n^3 \leq n^3 \quad \text{(For } 0 < c \leq 1\text{)}$$

Thus, the running time of the algorithm on an input size n is also Ω(f(n)).

(c) Although the algorithm we analyzed in parts (a) and (b) is the most natural way to solve the problem—after all, it just iterates through the relevant entries of the array B, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time O(g(n)), where limn→∞ g(n)/f(n) = 0.

Solution: The given algorithm can be optimized as follows-

```
For i = 1, 2, ……, n
    Initialize sum = A[i]
    For j = i+1, i+2, ……., n
        sum = sum + A[j]
        Store the value of sum in B [i, j]
    End For
End For
```

The Program for the above algorithm in Python-

```
For i in range(len(A)): ----------O (n)
    Sum=A[i] ----------O (1)
    For j in range (i+1, len(A)): -------O (n)
        Sum=Sum + A[j] ----------O (1)
        B[i][j] = Sum ----------O (1)
```

The time complexities for each line in the above program can be explained as follows-

- Since the first loop iterates through all the elements in the array, the upper bound for that loop is O(n).
- Likewise, the second loop iterates from i+1 to the length of the array and when i = 0 the loop iterates for (n-1) times which also has an upper bound of O(n).

Therefore, the time complexity for the given algorithm is T(n) = O(n) * O(n) = O (n^2)

That is $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$

T(n) = O (n^2) is the running time of this algorithm on an input size n

3. Expressing the asymptotic running time (Big-Theta) of the following algorithm as a function of the input length n:

Solution: The given algorithm is as follows-

```
Void boem (int [ ] a){
    int n = a. length; ----------O (1)
    for (int k = 2; k <= n; k<<=1) {----------O (log n)
        for (int b = 0; b < n; b + = k) {----------O (n/k)
            for (int d = k >>1; d > 0; d >>= 1) {----------O (log k)
                for (int j = (d = = k >> 1)? 0: d; j + d < k; j + = d << 1) {----------O (2k/d)
                    for (int i = 0; i < d; i++) {----------O (d)
                        int u = b + j + i; ----------O (1)
                        int v = u + d; ----------O (1)
                        if (a[v] < a[u]) {----------O (1)
                            int swap = a[u]; a[u] = a[v]; a[v] = swap; ----------O (1)
                        }
                    }
                }
            }
        }
    }
}
```

The time complexities for each line in the above program can be explained as follows-

- In the first 'for' loop the value of 'k' is getting incremented by powers of 2 i.e. for the first iteration the k value will be k = 2, for the second iteration it will be k = 4 and so on. The loop iterates until k <= n. Therefore, the complexity for the first loop is O (log n)
- In the second 'for' loop the value of 'b' is added to the value of 'k' and the loop iterates until b > = n. Therefore, the complexity for the second loop is O ($\frac{n}{k}$)
- In the third 'for' loop the initial value of 'd' is set to '$\frac{k}{2}$' and 'd' is reduced by half due to the right shift operator and the loop terminates when d<=0. We know that 'k' is a power of 2.

  Let k = 2^x and d = $\frac{k}{2}$ which implies d = 2^{x-1}. Since 'd' is reduced to half on every iteration until d = 0. We can say that loop runs for (x-1) times.

  We have d = 2^{x-1}

  ➔ log d = x-1

  ➔ log d + 1 = x

  ➔ log d + log 2 = x

  ➔ log 2d = x

  ➔ x = log k

  Therefore, the upper bound for the third loop is O (log k).
- In the fourth loop excluding the case when d = k >> 1, the loop runs in the sequence of 1,3,7,15,31,63 ……. Which is equivalent to $2^n - 1$ where n = $\frac{k}{d}$ .So this loop runs exponentially i.e. O ($2^{\frac{k}{d}}$).
- In the fifth loop the value of 'i' gets incremented linearly until 'd'. Therefore, the complexity of the loop is O(d)
- Remaining statements run for at constant time. Therefore, it is O (1).

The Total running time of the given algorithm is as follows-

T(n) = log n * $\frac{n}{k}$ * log k * $2^{\frac{k}{d}}$ * d = log n * $\frac{n}{log\ n}$ * log n * $2^{\frac{\frac{k}{k}}{2^c}}$ $(where\ c\ is\ a\ constant)$ * $\frac{log\ n}{2}$ = $\boldsymbol{O(nlog^2n)}$

We can say that $\boldsymbol{1\ nlog^2n \leq nlog^2n \leq 100\ nlog^2n}$ (for some random numbers)

From the above inequality we can say that the it is equal to $\boldsymbol{O(nlog^2n)}$ **,** $\boldsymbol{\Omega(nlog^2n)}$ **and** $\boldsymbol{\Theta(nlog^2n)}$

4. Keith and Randy, who work at the UW Tacoma Copy Center, have a series of n tasks of copying and then binding documents. They only have one copy machine and one binding machine, and at most one document can be processed by each of these machines at any time, but they can process distinct tasks simultaneously, as long as every document only enters the binding machine after it has been copied.

Based on the number of pages of each document, Keith and Randy know that the times needed to copy and bind them are, respectively, A_1,…….A_n and B_1,……B_n.Our friends need to know the shortest time they can get all tasks done and which order they should be performed, and they ask you to help them.

To that end, prove the following property:

(a) The sequence of documents processed by either machine can be made the same as that of the other machine without loss of time.

Solution: For the given problem we need to prove that the sequence of documents processed by either machined can be made the same as that of the other machine without loss of time. This can be proved as follows:

Let us consider that the two machines tasks that is $A_1, A_2, A_3, \ldots\ldots A_n$ and $B_1, B_2, B_3, \ldots\ldots\ldots B_n$ are arranged not in a same sequential order. For example, if we consider the copying tasks which are in processed in $A_1, A_2, A_3, \ldots\ldots A_n$ and the binding tasks are processed in some random pattern ($B_3, B_{10}, B_6 \ldots\ldots$) then the waiting times on machine B will be more as the copying tasks are not yet complete.

The waiting times can be optimized by processing the tasks $A_1, A_2, A_3, \ldots\ldots A_n$ and $B_1, B_2, B_3, \ldots\ldots\ldots B_n$ in the same order. That is when the task A_1 is completed B_1 automatically starts its work without any delay and thus reducing the delay time.

(b) Prove that an optimal sequence is done by following rule: document S[j] preceded S[j+1] if and only if max (K_j, k_{j+1}) < max (k_j', k_{j+1}').

Solution: We know that $K_u = \sum_{j=1}^{u} A_{s[j]} - \sum_{j=1}^{u-1} B_{s[j]}$

We need to prove $\boldsymbol{max\ (K_j, K_{j+1}) < max(K'_j, K'_{j+1})}$

➔ $\max(\sum_{j=1}^{u} A_{s[j]} - \sum_{j=1}^{u-1} B_{s[j]}\ ,\ \sum_{j=1}^{u} A_{s[j+1]} - \sum_{j=1}^{u-1} B_{s[j+1]}) \ < \ \max(\sum_{j=1}^{u} A_{s'[j]} - \sum_{j=1}^{u-1} B_{s'[j]}\ , \sum_{j=1}^{u} A_{s'[j+1]} - \sum_{j=1}^{u-1} B_{s'[j+1]}$

If suppose we have six copying tasks - $A_1, A_2, A_3, A_4, A_5, A_6$ and six binding tasks - $B_1, B_2, B_3, B_4, B_5, B_6$

I have considered j as "A_5 " $and\ j+1\ as$ "A_6 "

The above equation can be solved as follows-

$\max[(A_1 + A_2 + A_3 + A_4 + A_5) - (B_1 + B_2 + B_3 + B_4)\ ;\ (A_1 + A_2 + A_3 + A_4 + A_5 + A_6) - (B_1 + B_2 + B_3 + B_4 + B_5)]$

$<$

$\max[(A_1 + A_2 + A_3 + A_4 + A_6) - (B_1 + B_2 + B_3 + B_4)\ ;\ (A_1 + A_2 + A_3 + A_4 + A_6 + A_5) - (B_1 + B_2 + B_3 + B_4 + B_6)]$

(As the terms are interchanged)

The above equation can be solved as follows-

$\max[(A_1 + A_2 + A_3 + A_4 + A_5 - B_1 - B_2 - B_3 - B_4)\ ;\ (A_1 + A_2 + A_3 + A_4 + A_5 + A_6 - B_1 - B_2 - B_3 - B_4 - B_5)]$

$<$

$\max[(A_1 + A_2 + A_3 + A_4 + A_6 - B_1 - B_2 - B_3 - B_4)\ ;\ (A_1 + A_2 + A_3 + A_4 + A_6 + A_5 - B_1 - B_2 - B_3 - B_4 - B_6)]$

If we subtract $(A_1 + A_2 + A_3 + A_4 + A_5 - B_1 - B_2 - B_3 - B_4)$ on both sides, the equation is reduced as follows-

➔ $\max(0;\ A_6 - B_5) \ < \ \max(A_6 - A_5\ ;\ A_6 - B_6)$

If we subtract "$-A_6$ " on both sides the equation, we be as follows-

➔ $\max(-A_6\ ;\ -B_5) \ < \ \max(-A_5\ ;\ -B_6)$

Or

➔ $\min(A_5\ ;\ B_6) \ < \ \min(A_6\ ;\ B_5)$

If we re-write the final equation in terms of j and j+1 we get-

➔ min (A_j ; B_{j+1}) < min (A_{j+1} ; B_j)

Thus, optimal sequence is done if document s[j] precedes s[j+1].

(c) Design an algorithm that will find the shortest time that Keith and Randy can complete their copying and binding jobs, and also suggests an ordering of tasks that attains that shortest time.

Solution: The shortest time that Keith and Randy can complete their copying and binding jobs can be accomplished by Johnson's Algorithm for the two-machine flow shop problem.

Algorithm:

- Firstly, search for all the times that are taken for copying and binding. Get the smallest time from the times taken for copying and binding.
- If it is in copy machine place that in the first position.
- If it is in the binding machine place it in the last position.
- Remove that task from the list i.e. copying time and binding time.
- Repeat the above steps for the remaining tasks until there are no tasks left in the list.
- If there is a tie between copying and binding times, then order the task according to copying.

Reference for c) https://www.youtube.com/watch?v=O3Zuq9-9Hu0&t=1s