COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 1 Solutions

Announced on: July 25 (Thurs)

Note: For proving asymptotic complexity of functions, some students have used limits based definition, which has not been discussed in the Lectures yet. This being the first tut-sheet, they wont be penalised. But it is encouraged to use the constant-based definition instead.

1. Prove that $\log_2(n!) = \Theta(n \log_2 n)$.

To prove that $\log_2(n!) = \Theta(n \log_2 n)$ we need to show that there exists n_0, k_1, k_2 such that $\forall n \geqslant n_0.\ k_1 n \log_2 n \leqslant \log_2(n!) \leqslant k_2 n \log_2 n$.

Let us take $n_0 = 10$, $k_1 = \frac{1}{2}$ and $k_2 = 2$. Now we can prove the above inequality via **induction**.

Base case: For $n = 10$, we can check that $\frac{1}{2} \times 10 \log_2(10) \leqslant \log_2(10!) \leqslant 2 \times 10 \log_2(10)$

Induction step: We can divide the induction step into two parts, first checking for the lower bound and then for the upper bound:

a) Given that $\frac{n}{2} \log_2(n) \leqslant \log_2(n!)$ we want $\frac{n+1}{2} \log_2(n+1) \leqslant \log_2((n+1)!)$.

Claim 1: For all $n > 0$, $(1 + \frac{1}{n})^n \leqslant e$. From this we can show:

$$\forall n > 0.\ n \log_2(n+1) - n \log_2(n) \leqslant \log_2(e)$$

$$\implies \forall n \geqslant 10.\ n \log_2(n+1) - n \log_2(n) \leqslant \frac{1}{2} \log_2(n+1)$$

$$(\text{as } (n+1)^{\frac{1}{2}} > e \text{ for } n \geqslant 10)$$

Now we can write our proof:

$$\begin{aligned}
\frac{n}{2}\log_2(n) &\leqslant \log_2(n!) \\
\frac{n}{2}\log_2(n) + \log_2(n+1) &\leqslant \log_2(n!) + \log_2(n+1) \\
\frac{n}{2}\log_2(n) + \frac{1}{2}\log_2(n+1) + \frac{1}{2}\log_2(n+1) &\leqslant \log_2(n! \times (n+1)) \\
\frac{n}{2}\log_2(n) + n\log_2(n+1) - n\log_2(n) + \frac{1}{2}\log_2(n+1) &\leqslant \log_2((n+1)!) \\
n\log_2(n+1) - \frac{n}{2}\log_2(n) + \frac{1}{2}\log_2(n+1) &\leqslant \log_2((n+1)!) \\
\frac{n+1}{2}\log_2(n+1) &\leqslant \log_2((n+1)!)
\end{aligned}$$

b) Given that $\log_2(n!) \leqslant 2n\log_2(n)$ we want $\log_2((n+1)!) \leqslant 2(n+1)\log_2(n+1)$

$$\begin{aligned}
\log_2(n!) &\leqslant 2n\log_2(n) \\
\log_2(n!) + \log_2(n+1) &\leqslant 2n\log_2(n) + \log_2(n+1) \\
&\textit{For } n \geqslant 10,\ \log_2(n+1) \leqslant 2\log_2(n). \textit{ So, we get:} \\
\log_2(n! \times (n+1)) &\leqslant 2n\log_2(n) + 2\log_2(n) \\
\log_2((n+1)!) &\leqslant 2(n+1)\log_2(n) \\
\log_2((n+1)!) &\leqslant 2(n+1)\log_2(n+1)
\end{aligned}$$

Hence we proved that $\log_2(n!) = \Theta(n\log_2 n)$.
Alternate Proof Idea: To show that $\log_2(n!) = \mathcal{O}(n\log n)$, express the left hand side as $\sum_{i=1}^{n}\log_2 i$ and observe that each term is at most $\log_2 n$. To prove that $\log_2(n!) = \Omega(n\log n)$, observe that $\sum_{i=1}^{n}\log_2 i \geqslant \frac{n}{2}\log_2\frac{n}{2}$ (by considering only the terms with $i \geqslant n/2$).

- Written "I donot know how to approach this problem" - 0.6 points.
- Approach 1:
  - Mentioning that one needs to show $\log_2(n!) = O(n\log_2 n)$ and $\log_2(n!) = \Omega(n\log_2 n)$ - 0.5 points
  - Mentioning proof by induction - 0.5 points
  - Correct proof for $\log_2(n!) = O(n\log_2 n)$ (Base case as well as inductive step) - 1 point
  - Correct proof for $\log_2(n!) = \Omega(n\log_2 n)$ (Base case as well as inductive step) - 1 point

- Approach 2:
  - Mentioning that one needs to show $\log_2(n!) = O(n \log_2 n)$ and $\log_2(n!) = \Omega(n \log_2 n)$ - 0.5 points
  - Proving $\log_2(n!) = \mathcal{O}(n \log n)$ - 1 point
  - Proving $\sum_{i=1}^{n} \log_2 i \geqslant \frac{n}{2} \log_2 \frac{n}{2}$ - 1 point
  - Proving why $\sum_{i=1}^{n} \log_2 i \geqslant \frac{n}{2} \log_2 \frac{n}{2}$ implies $\log_2(n!) = \Omega(n \log_2 n)$ - 0.5 points

2. Prove or disprove:
   a) $\log n = \mathcal{O}(n^\varepsilon)$ for any constant $\varepsilon > 0$.
   b) $4^n = \mathcal{O}(2^n)$.
   c) $n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$.
   d) $\lceil n \rceil = \Theta(n)$.

a) $\log n = \mathcal{O}(n^\varepsilon)$ for any constant $\varepsilon > 0$: This claim is true.

Proof: To prove this we need to show that for any $\varepsilon > 0$ there exists n_0, k_1 such that for any $n > n_0$, $\log n \leqslant k_1 n^\varepsilon$.

Let us take $k_1 = \frac{1}{\varepsilon}$ and $n_0 = 1$. So now we have to prove:

$$\begin{aligned} \log n &\leqslant \frac{1}{\varepsilon} n^\varepsilon \\ \Leftrightarrow \varepsilon \log n &\leqslant n^\varepsilon \\ \Leftrightarrow \log(n^\varepsilon) &\leqslant n^\varepsilon \end{aligned}$$

This follows from the fact that $n^\varepsilon > 1$ because $n > 1, \varepsilon > 0$ and $log(x) \leqslant x$ for any positive x.

b) $4^n = \mathcal{O}(2^n)$: This claim is false.

Proof: We need to show that there does not exist any n_0, k_1 such that $4^n \leqslant k_1 2^n$ for all $n > n_0$.

Consider any arbitrary choice of n_0 and k_1. Let us consider the n value of $\max(n_0, \log_2(k_1)) + 1$. It is true that $n > n_0$. Furthermore, we can show that for any $n > \log_2(k_1)$, $4^n > k_1 2^n$, as $2^{log_2(k_1)} = k_1$, so for any $x > \log_2(k_1)$ we get $2^x > k_1$.

c) This claim is true. Consider a function $f(n) = O(n)$, we know that there exist fixed constants k, n_0 such that $f(n) \leqslant k \cdot n \; \forall n > n_0$. Now, consider the function $g(n) = n^2 + f(n)$. We can say that:

$$\begin{aligned} & n^2 + f(n) \leqslant n^2 + k \cdot n && \forall n > n_0 \\ \implies & n^2 + f(n) \leqslant n^2 + n^2 && \forall n > \max(n_0, k) \\ \implies & n^2 + f(n) \leqslant 2n^2 && \forall n > \max(n_0, k) \\ \implies & n^2 + f(n) = O(n^2) \end{aligned}$$

Hence, proved.

d) This claim is true. We know that $n \leqslant \lceil n \rceil < n + 1$. So, for all $n \geqslant 1$, we know that $n \leqslant \lceil n \rceil \leqslant 2n$. Hence, $\lceil n \rceil = \Theta(n)$.

a)
- Written "I donot know how to approach this problem" - 0.3 points
- – Claiming for any $\varepsilon > 0$, there exist n_0, k_1 such that for any $n > n_0$, $\log n \leqslant k_1 n^{\varepsilon}$ - 0.5 points
- – Choosing the correct n_0, ε and proving the correctness - 1 point

b)
- Written "I donot know how to approach this problem" - 0.3 points
- – Claiming there does not exist any n_0, k_1 such that $4^n \leqslant k_1 2^n$ for all $n > n_0$ - 0.5 points
- – Proving the above claim - 1 point

3. Identify and discuss the error(s) in the following false statement and its false proof. Mention the erroneous step(s) clearly.

Claim: Let $f(n) = \sum_{i=1}^{n} i$. Then, $f(n) = \mathcal{O}(n)$.

Proof: By induction on n.

Let the induction hypothesis be $P(n) : f(n) = \mathcal{O}(n)$.

Base case: $f(1)$ equals 1, which is $\mathcal{O}(1)$. Thus, $P(1)$ is TRUE.

Induction step: Observe that $f(n+1) = f(n) + (n+1)$. Since $f(n) = \mathcal{O}(n)$ by induction assumption, and since $n + 1 = \mathcal{O}(n)$, we have that $f(n+1) = \mathcal{O}(n)$, which, in turn, is $\mathcal{O}(n+1)$. This proves the claim. □

The erroneous step way we are using induction for this problem, as n is not a **fixed** constant. $f(n) = O(n)$ implies that there exist fixed constants n_0, k such that for all $n > n_0$, $f(n) \leqslant kn$. However, while using the induction in the given way, the constant k grows monotonically with n and hence, is not a fixed constant anymore.
The correct way to do induction on n would be to fix some n_0 and k and have the induction hypothesis be $f(n) \leqslant k \times n$, with the base case being on n_0.

- Written "I donot know how to approach this problem" - 0.6 points.
-
  - Claiming that $f(n) = O(n)$ implies that there exist **fixed** constants n_0, k such that for all $n > n_0$, $f(n) \leqslant kn$ - 1.5 points
  - Identifying that the k in the induction used is not a fixed constant - 1.5 points

4. Consider the following modification to the `MergeSort` algorithm: Divide the input array into *thirds* (rather than halves), recursively sort each third, and combine the results using a three-way `Merge` subroutine. What is the number of pairwise comparisons made by this algorithm as a function of the length n of the input array, ignoring constant factors and lower-order terms?

   (For the purpose of the tutorial quiz, you don't need to write the algorithm or its correctness analysis. Just highlight the key steps in your argument.)

Similar to the `Merge` discussed in the lecture, if the three subarrays (say A_1, A_2, A_3) have size $\leqslant l$, then one can perform a three-way merge by merging A_1, A_2 first, and then merging this merged array with A_3. Since a two-way merge is invoked 2 times and it uses $\leqslant cl$ operations on each invocation (for some fixed constant $c > 0$), so this three-way merge also takes $< c^{'} l$ time (for some fixed constant $c^{'} > 0$). So, a three-way merge routine also takes $O(n)$ time at each level of the recursion tree.
Since size of the array becomes one third at each level, there are about $O(\log_3(n))$ levels in the recursion tree.
Hence, the overall number of pairwise comparisons is $O(n \log n)$ in this algorithm.

- Written "I donot know how to approach this problem" - 0.6 points
-
  - Claiming three-merge takes $O(n)$ time at each level - 1.5 points
  - Claiming recursion tree has $O(\log n)$ levels - 1 point
  - Combining the above two claims to obtain $O(nlogn)$ number of comparisons. - 0.5 points

5. Suppose the running time $T(n)$ of an algorithm is bounded by the recurrence with $T(1) = 1$

and

$$T(n) \leqslant T(\lfloor \sqrt{n} \rfloor) + 1 \text{ for } n > 1,$$

where $\lfloor x \rfloor$ denotes the *floor* function that rounds its argument down to the nearest integer. Provide the smallest correct upper bound on the asymptotic running time of the algorithm and justify your answer. Note that the master theorem does not apply.

Let us first observe some values for $T(\cdot)$:

$$\begin{aligned} T(1) &= 1 \\ T(2), T(3) &= 2 \\ T(4), T(5), \ldots, T(15) &= 3 \\ T(16), T(17), \ldots, T(255) &= 4 \\ &\vdots \end{aligned}$$

Now, we claim the following:
Claim 1. *For any integer any $n > 1$, $T(n) = k + 2$, where k is the non-negative integer satisfying $2^{2^k} \leqslant n < 2^{2^{k+1}}$.*

Proof. We will prove this by **strong induction** on n. Our induction hypothesis $P(n)$ is that for any integer any $n > 1$, $T(n) = k + 2$, where k is the non-negative integer satisfying $2^{2^k} \leqslant n < 2^{2^{k+1}}$. Note that for any $n > 1$, a unique such k will always exist.

Base case(s): $n = 2, 3$ - Since $2^{2^0} \leqslant 2, 3 < 2^{2^1}$, so $T(2), T(3) = 0 + 2 = 2$. As noted earlier, $T(2), T(3)$ are indeed equal to 2. Hence, the base case holds.

Induction Step: Suppose $P(i)$ holds for all $1 < i \leqslant n$. So, we'll show that $P(n+1)$ holds. We know that there exists a k such that:

$$\begin{aligned} & 2^{2^k} \leqslant n + 1 < 2^{2^{k+1}} \\ \implies & 2^{2^{k-1}} \leqslant \lfloor \sqrt{n+1} \rfloor < 2^{2^k} \end{aligned}$$

So, $\lfloor \sqrt{n+1} \rfloor < n$, which means that $P(\lfloor \sqrt{n+1} \rfloor)$ holds. Hence, $T(\lfloor \sqrt{n+1} \rfloor) = (k-1) + 2 = k + 1$. □

Now,

$$\begin{aligned} & T(n) = T(\lfloor \sqrt{n+1} \rfloor) + 1 \\ \implies & T(n) = (k+1) + 1 \\ \implies & T(n) = k + 2 \end{aligned}$$

Hence, our induction step holds true. Since base case(s) and induction step are true, our induction hypothesis $P(n)$ is correct.

So, $T(n) = k + 2$, where:

$$\begin{aligned} & 2^{2^k} \leqslant n < 2^{2^{k+1}} \\ \implies & k \leqslant \log_2 \log_2 n < k + 1 \\ \implies & k = \lfloor \leqslant \log_2 \log_2 n \rfloor \\ \implies & T(n) = \lfloor \leqslant \log_2 \log_2 n \rfloor + 2 \end{aligned}$$

So, $T(n) = O(\log \log n)$.

6. Imagine you are a trader in a market in which a particular commodity can be bought only once and sold only once during a season. Naturally, the purchase much precede the sale. The price of the commodity fluctuates every day. The data from the last season is now available, and you want to know the maximum profit you could have made. Design an $\mathcal{O}(n)$ algorithm for this problem, where the input is the list of prices $\{p_1, p_2, \ldots, p_n\}$ for the n days in the last season and the output is the maximum possible profit. Assume that addition, subtraction, pairwise comparison, etc. takes unit time.

   Sample input prices: 70, 100, 140, 40, 60, 90, 120, 30, 60.
   Output: Max profit: 80.

ALGORITHM 1: Find Maximum Profit

Data: $p[1 \ldots n]$: Array of values
Result: profit: Maximum profit

```
1 Function FindMaxProfit(p):
      Input: p: Array of values
      Output: max_profit: Maximum profit
2     max_profit ← 0;
3     min_price ← ∞;
4     for i ← 1 to n do
5         min_price ← min(min_price, p[i]);
6         max_profit ← max(max_profit, p[i] − min_price);
7     return profit;
```

The maximum profit that can be made is:

$$\max(p_j - p_i) \quad \forall\, i < j$$

The maximum profit on day i is obtained by buying the stock at the lowest price seen before day i and selling it on day i, and the overall maximum profit is the highest profit achievable across all days.

Proof Of Correctness using loop invariant:

Loop Invariant: max_profit contains maximum acheivable profit in first i days.

- **Intialisation:** max_profit is initialized to 0, which represents no profit.
- **Iteration:** Let us assume max_profit contains the maximum acheivable profit in first i-1 days. In the ith iteration we update min_price to be the minimum of the current min_price and p_i. This ensures that min_price always holds the lowest price encountered so far. We update max_profit to be the maximum of the current max_profit and the maximum profit by selling stock on day i i.e p_i-min_price. This ensures that max_profit always holds the maximum acheivable profit in first i days.
- **Termination:** After iterating through the list, max_profit will hold the maximum possible profit that can be achieved by buying and then selling once during the season.

Alternate approach: We can also solve the above problem using divide and conquer. Divide the array into two halves A_1 and A_2. We can either (i) buy and sell stock in A_1, or (ii) buy and sell stock in A_2 or (iii) we can buy stock in A_1 and sell it in A_2. The maximum profit will be the maximum of these 3 cases. The maximum profit in the first 2 cases can be calculated using recursion. For the third case, the maximum profit will be when we buy on the lowest price of A_1 and sell on the highest price of A_2. The minimum of A_1 and maximum of A_2 can be found in $O(n)$ time and so, maximum profit by buying stock in A_1 and selling it in A_2 can be calculated in $O(n)$.
Time Complexity:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$
$$T(n) = O(n \log n)$$

7. ($\star$) You are given as input an unsorted array of n distinct numbers, where n is a power of 2. Give an algorithm that identifies the second-largest number in the array while using at most $n + \log_2 n - 2$ pairwise comparisons.

We first find the maximum element in the list using divide and conquer. Along with maximum element we keep track of elements that were compared to maximum element. Size of this list would be $\log_2 n$ - 1. Second maximum element should belong to the elements that were compared to maximum element. Thus finding the maximum of these elements will give us the second maximum element.

Claim: Second maximum element is in the list L of elements that were compared to maximum element.
Proof by contradiction: Suppose x is the second maximum element and is not present in the L. x is unique as all elements are distinct. As x is not in L this means that there must be an element t larger than x which is not equal to maximum element else it would be in the list L. But this contradicts the fact that x is the second maximum

ALGORITHM 2: Get Second Largest Element

Input: nums_lst: List of numbers
Output: The second largest number in the list

```
Function get_greaters_lst(lst):
    if size(lst) = 1 then
        return lst
    lst1 ← get_greaters_lst(lst[0 : size(lst) / 2]);
    lst2 ← get_greaters_lst(lst[size(lst) / 2 : ]);
    if lst1[0] > lst2[0] then
        lst1.add(lst2[0]);
        return lst1
    else
        lst2.add(lst1[0]);
        return lst2
greaters_lst ← get_greaters_lst(nums_lst)[1:];
max_num ← greaters_lst[0];
foreach candidate in greaters_lst do
    if candidate > max_num then
        max_num ← candidate;
return max_num;
```

element.

Comparisons required to find maximum element:

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + 1 \\
T(n) &= 2\left(2 \cdot T\left(\frac{n}{4}\right) + 1\right) + 1 \\
T(n) &= 2^k \cdot T(1) + 2^{k-1} + \cdots + 1 \quad \text{where} \quad k = \log_2 n, \quad T(1) = 0 \\
T(n) &= n - 1
\end{aligned}
$$

Comparisons required to find the second maximum element: $\log_2 n$ - 1
Total Number of comparisons: n + $\log_2 n - 2$.
Alternate approach: Imagine a "tournament" between the elements of the array in the form of a binary tree. The leaves are the original elements, and each parent is the "winner" of the game (i.e., the larger number) between its children. The root of this tree is the maximum element. It takes at most $n - 1$ comparisons to determine the maximum element. Further, the maximum element is compared with $\#levels = \log_2 n$ elements during the tree's construction. One of these comparisons *must* involve the second-maximum element. To determine the maximum among these $\log_2 n$ elements, one needs at most $\log_2 n - 1$ comparisons.

8. ($\star$) Consider a tournament with n teams in which each team plays every other team exactly once, and each game has one winner (i.e., no ties). Unfortunately, the tournament is being held behind closed doors and you are unable to attend the games. You can, however, make a call to the organizers and, in each call, ask for the result of exactly one game. Say you want to find out if there is an *undisputed* champion, which is a team that wins all its games. Show that you can determine the existence of such a team with at most $2n - \lfloor \log_2 n \rfloor - 2$ calls.

   This is similar to the previous question. On a high level, the idea is as follows: First, we'll find a good guess for the *undisputed* champion team. Then, we will verify if that team is indeed the *undisputed* champion. For simplicity, we will assume that n is a power of 2 here, i.e., $n = 2^k$ for some integer $k > 0$.

   a) To find a good guess for the *undisputed* champion team, imagine a "tournament" between the elements of the array in the form of a binary tree. The leaves are the original elements, and each parent is the "winner" of the game between its children. The root will be our guess for the *undisputed* champion. It can be observed that it takes $n - 1$ calls to determine this team (lets call this team T_i). Also, $\log_2 n$ calls must have been made to matches involving T_i.

   b) Now, to check if T_i is indeed the *undisputed* champion, we find out results of all $n - 1$ matches of T_i, and check if T_i won all of them. Note that we already know the $\log_2 n$ matches of T_i which it won. So we need to make further $n - 1 - \lg_2 n$ calls only.

   So, Total calls made = $n - 1 + (n - 1 - \log_2 n) = 2n - 2 - \log_2 n$.
   Note: To handle the general case when $n \neq 2^k$, in the first step, one can form a similar binary tree tournament, but will have to 'balance" the number of games played by each team at different levels, so that the teams at the same level should have played approximately the same number of games. By this, one can ensure that the team at the root plays at least $\lfloor \log_2 n \rfloor$ games in the tournament. Then, the second step remains the same, and we get a bound of $2n - 2 - \lfloor \log_2 n \rfloor$ on the total number of calls made.