CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Homework #1 Solution

February 11

Problem 1. This question reviews asymptotic notation. You may assume the following inequalities for this question (and throughout the course): For any constant $c \geq 1$,

$$(\log n)^c = o(n) \quad , \quad n^c = o(n^{c+1}) \quad , \quad c^n = o((c+1)^n) \quad , \quad (n/2)^{(n/2)} = o(n!) \quad , \quad n! = o(n^n).$$

(a) Rank the following functions based on their asymptotic value in the increasing order, i.e., list them as functions $f_1, f_2, f_3, \ldots, f_9$ such that $f_1 = O(f_2)$, $f_2 = O(f_3), \ldots, f_8 = O(f_9)$. Remember to write down your proof for each equation $f_i = O(f_{i+1})$ in the sequence above. **(15 points)**

$\sqrt{\log n}$	$\log \log n$	$2^{\log n}$
$100n$	10^n	$2^{2^{2^{2^2}}}$
2^n	$n!$	$\frac{n}{\log n}$

Hint: For some of the proofs, you can simply show that $f_i(n) \leq f_{i+1}(n)$ for all sufficiently large n which immediately implies $f_i = O(f_{i+1})$.

Solution. The correct ordering is as follows:

$f_1 = 2^{2^{2^{2^2}}}$	$f_2 = \log \log n$	$f_3 = \sqrt{\log n}$
$f_4 = \frac{n}{\log n}$	$f_5 = 2^{\log n}$	$f_6 = 100n$
$f_7 = 2^n$	$f_8 = 10^n$	$f_9 = n!$

We now prove each part.

- $2^{2^{2^{2^2}}} = O(\log \log n)$: $2^{2^{2^{2^2}}}$ is a constant after all (even though a gigantic one), while $\log \log n$ grows with n (even though very slowly) so $\lim_{n\to\infty} \frac{2^{2^{2^{2^2}}}}{\log \log n} = 0$.
- $\log \log n = O(\sqrt{\log n})$: Change of variable $m = \sqrt{\log n}$ so $m^2 = \log n$ and $2^{m^2} = n$. This also means that $\log \log n = \log \log 2^{m^2} = \log m^2 = 2 \log m$. So we only need to show $2 \log m = O(m)$ which holds by the first item in the equation of the problem statement.
- $\sqrt{\log n} = O(\frac{n}{\log n})$: We have $\lim_{n\to\infty} \frac{\sqrt{\log n}}{\frac{n}{\log n}} = \lim_{n\to\infty} \frac{\log^{(3/2)}(n)}{n} = 0$; the last part is again by the first item in the equation of the problem statement (by taking $c = 3/2$).
- $\frac{n}{\log n} = O(2^{\log n})$: $2^{\log n}$ is simply n for us[1] so we need to show $\frac{n}{\log n} = O(n)$. For this, we have $\lim_{n\to\infty} \frac{\frac{n}{\log n}}{n} = \lim_{n\to\infty} \frac{1}{\log n} = 0$.
- $2^{\log n} = O(100n)$: In this case, $2^{\log n} = n$ and we have $n = \Theta(100n)$ so this is true. Note that we could have even changed the order and prove $100n = O(2^{\log n})$ instead.

[1] Throughout the course, the base of log is always 2 unless specified otherwise. That being said, since this may have been ambigious, if you used base 10 for the log and thus instead have $2^{\log_{10} n} = O(\frac{n}{\log n})$, which is the correct answer for $\log_{10}(n)$ but not $\log n$, you will receive the full credit.

- $100n = O(2^n)$: $100n = \Theta(n)$ and $n = O(2^n)$ by the first part of the equation in the problem statement (after doing a change of variable $m = 2^n$, we have $\log m = O(m)$).
- $2^n = O(10^n)$: Follows immediately from third item of the equation in the problem statement and transitivity of O-notation.
- $10^n = O(n!)$: $10^n = 100^{n/2}$ and for $n \geq 100$, $100^{n/2} \leq (n/2)^{n/2}$. As such, $10^n = O((n/2)^{n/2})$. By the fourth item of the equation in the problem statement, $(n/2)^{n/2} = O(n!)$ and thus by transitivity, $10^n = O(n!)$ as well.

(b) Consider the following four different functions $f(n)$:

$$1 \qquad \log n \qquad n^2 \qquad 4^{4^n}.$$

For each of these functions, determine which of the following statements is true and which one is false. Remember to write down your proof for each choice. **(10 points)**

- $f(n) = \Theta(f(n-1))$;
- $f(n) = \Theta(f(\frac{n}{2}))$;
- $f(n) = \Theta(f(\sqrt{n}))$;

Solution. Solution to part (b) goes here.

- $f(n) = 1$: All of the equations are true because both sides will be the constant 1.

- $f(n) = \log n$:
  - Statement 1 is true: $\lim_{n\to\infty} \frac{\log(n-1)}{\log n} = \lim_{n\to\infty} \frac{\log(n\cdot(1-1/n))}{\log n} = \lim_{n\to\infty} \frac{\log n + \log(1-1/n)}{\log n} = \lim_{n\to\infty} 1 + 0 = 1$, which is a constant other than 0.
  - Statement 2 is true: $\lim_{n\to\infty} \frac{\log(n/2)}{\log n} = \lim_{n\to\infty} \frac{\log n - 1}{\log n} = 1 - 0 = 1$, which is a constant other than 0.
  - Statement 3 is true: $\lim_{n\to\infty} \frac{\log(\sqrt{n})}{\log n} = \lim_{n\to\infty} \frac{\frac{1}{2}\cdot\log n}{\log n} = \frac{1}{2}$, which is a constant other than 0.

- $f(n) = n^2$:
  - Statement 1 is true: $(n-1)^2 = n^2 - 2n + 1$ and so $n^2/2 \leq (n-1)^2 \leq n^2$ for all $n > 4$, implying that $(n-1)^2 = \Theta(n^2)$.
  - Statement 2 is true: $(n/2)^2 = n^2/4 = \Theta(n^2)$.
  - Statement 3 is false: $(\sqrt{n})^2 = n$ and $n = o(n^2)$ so $(\sqrt{n})^2 \neq \Omega(n^2)$ and so $(\sqrt{n})^2 \neq \Theta(n^2)$.

- $f(n) = 4^{4^n}$:
  - Statement 1 is false: $4^{4^{n-1}} = 4^{4^n \cdot 4^{-1}}$ and so $4^{4^{n-1}} = (4^{4^n})^{\frac{1}{4}}$. A change of variable $m = 4^{4^n}$ let us compare m and $m^{1/4}$. But $m^{1/4} = o(m)$ and so $4^{4^{n-1}} \neq \Omega(4^{4^n})$, hence $4^{4^{n-1}} \neq \Theta(4^{4^n})$.
  - Statement 2 is false: $4^{4^{n/2}} \leq 4^{4^{n-1}}$ for $n > 2$ and so $4^{4^{n/2}} = o(4^{4^n})$ as well by the first part, hence $4^{4^{n/2}} \neq \Theta(4^{4^n})$.
  - Statement 3 is false: $4^{4^{\sqrt{n}}} \leq 4^{4^{n-1}}$ for $n > 2$ and so $4^{4^{\sqrt{n}}} = o(4^{4^n})$ as well by the first part, hence $4^{4^{n/2}} \neq \Theta(4^{4^n})$

Problem 2. Your goal in this problem is to analyze the *runtime* of the following (imaginary) recursive algorithms for some (even more imaginary) problem:

(A) Algorithm A divides an instance of size n into 4 subproblems of size $n/4$ each, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

(B) Algorithm B divides an instance of size n into 2 subproblems, one with size $n/4$ and one with size $n/5$, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

(C) Algorithm C divides an instance of size n into 4 subproblems of size $n/3$ each, recursively solves each one, and then takes $O(n^2)$ time to combine the solutions and output the answer.

(D) Algorithm D divides an instance of size n in to 2 subproblems of size $n-1$ each, recursively solves each one, and then takes $O(1)$ time to combine the solutions and output the answer.

For each algorithm, write a recurrence for its runtime and *use the recursion tree method* of Lecture 3 to solve this recurrence and find the *tightest asymptotic* upper bound on runtime of the algorithm. **(25 points)**

Solution. For each part, let $T(n)$ be the worst case time each of the algorithms on an input of size n.

(A) *Recurrence for algorithm A:* $T(n) \leq 4 \cdot T(n/4) + O(n)$. We prove $T(n) = O(n \log n)$ in this case. We will replace $O(n)$ with $C \cdot n$ for some integer $C > 0$. The recursion tree is as follows:

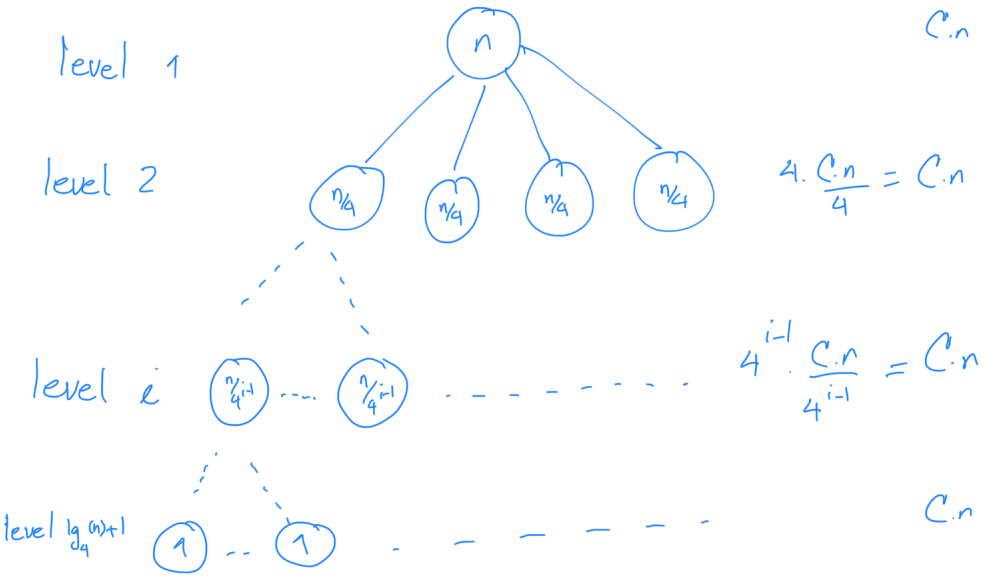


Figure 1: Recursion tree for algorithm A.

We have that the problem instances at each level are getting divided by four in size, as we go down the tree. This can happen at most $\log_4 n$ times until we reach a problem of constant size, which takes constant time to solve. Thus, there are $\log_4 n + 1$ levels in the tree (the depth of the tree is $\log_4 n$).

At level i, there are 4^{i-1} sub-problems, each of size $n/4^{i-1}$, and each such problem incurs a cost of $C \cdot n/4^{i-1}$. Thus, each level of the tree incurs a cost of $C \cdot n$, and since we have $\log_4 n + 1$ levels, the total cost is at most $C \cdot n \cdot (\log_4 n + 1)$, which means $T(n) = O(n \log n)$.

(B) *Recurrence for algorithm B:* $T(n) \leq T(n/4) + T(n/5) + O(n)$. We prove $T(n) = O(n)$ in this case. We replace $O(n)$ with $C \cdot n$ for some integer $C > 0$. The recursion tree is as follows.

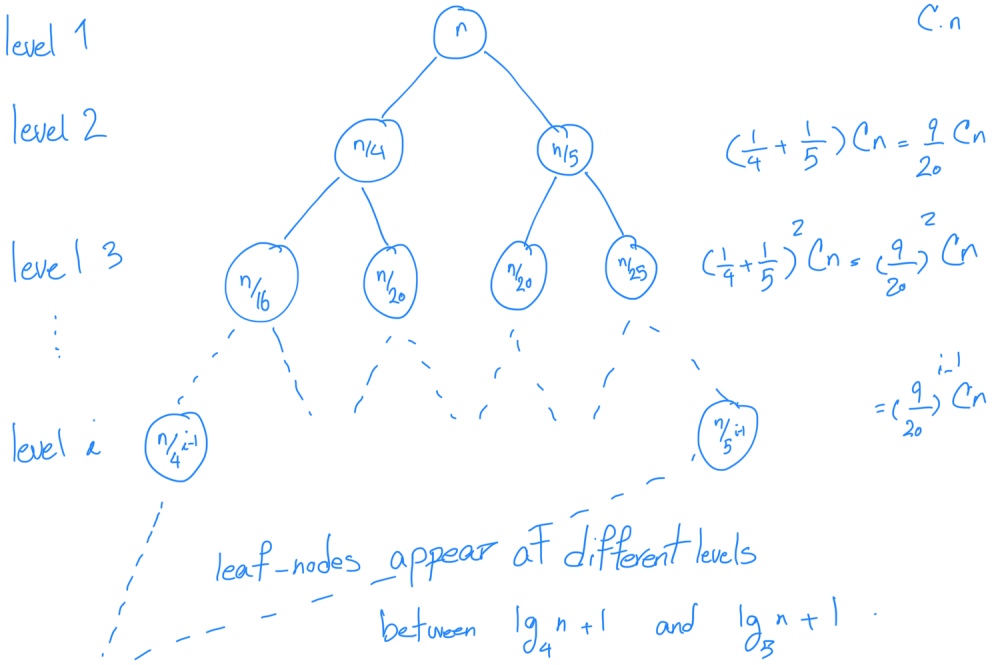


Figure 2: Recursion tree for algorithm B.

We have that the problem instances at each level are divided by either four or five in size, as we go down the tree. This is going to be an unbalanced tree, which means that the length of the shortest path from different leaves to the root node can be different.

At level i, there are 2^{i-1} problems, each of different size, and each such problem incurs a cost proportional to its size while combining solutions to its subproblems. The longest distance between a leaf and the root is $\log_4 n$ and the shortest one is $\log_5 n$.

Observe that the sum of costs at each level i of the tree is $(1/4 + 1/5)^{i-1} \cdot C \cdot n = (9/20)^{i-1} \cdot C \cdot n$. This is true for all level where the tree is still full, i.e. the first $\log_5 n$ levels. For the remaining levels, the sum is upper bounded by this number. Since this will be a geometric series with common ratio less than one, we can upper bound by summing to infinity and have,

$$T(n) \leq \sum_{i=0}^{\infty} C \cdot n \cdot (\frac{9}{20})^i = C \cdot n \cdot \frac{1}{1 - \frac{9}{20}} = O(n).$$

(C) *Recurrence for algorithm C:* $T(n) \leq 4 \cdot T(n/3) + O(n^2)$. We prove $T(n) = O(n^2)$ in this case. We will replace $O(n^2)$ with $C \cdot n^2$ for some integer $C > 0$. The recursion tree is drawn in Figure 3 next page.

The problem instances at each level are divided by three in size, as we go down the tree. Thus, there are $\log_3 n + 1$ levels in the recursion tree (the depth of the tree is $\log_3 n$).

At depth i, there are 4^{i-1} problems, each of size $n/3^{i-1}$, and each such problem incurs a cost of $C \cdot (n/3^{i-1})^2$ while combining solutions to its subproblems. Hence, by summing up all the values and since the resulting series is converging:

$$T(n) \leq \sum_{i=0}^{\log_3 n} (4^i \cdot c \cdot (\frac{n}{3^i})^2) = c \cdot n^2 \cdot \sum_{i=0}^{\log_3 n} (\frac{4}{9})^i \leq c \cdot n^2 \cdot \sum_{i=0}^{\infty} (\frac{4}{9})^i = O(n^2).$$

(D) *Recurrence for algorithm D:* $T(n) \leq 2 \cdot T(n-1) + O(1)$. We prove $T(n) = O(2^n)$. We will replace $O(1)$ with some integer $C > 0$. The recursion tree is drawn in Figure 4 next page.

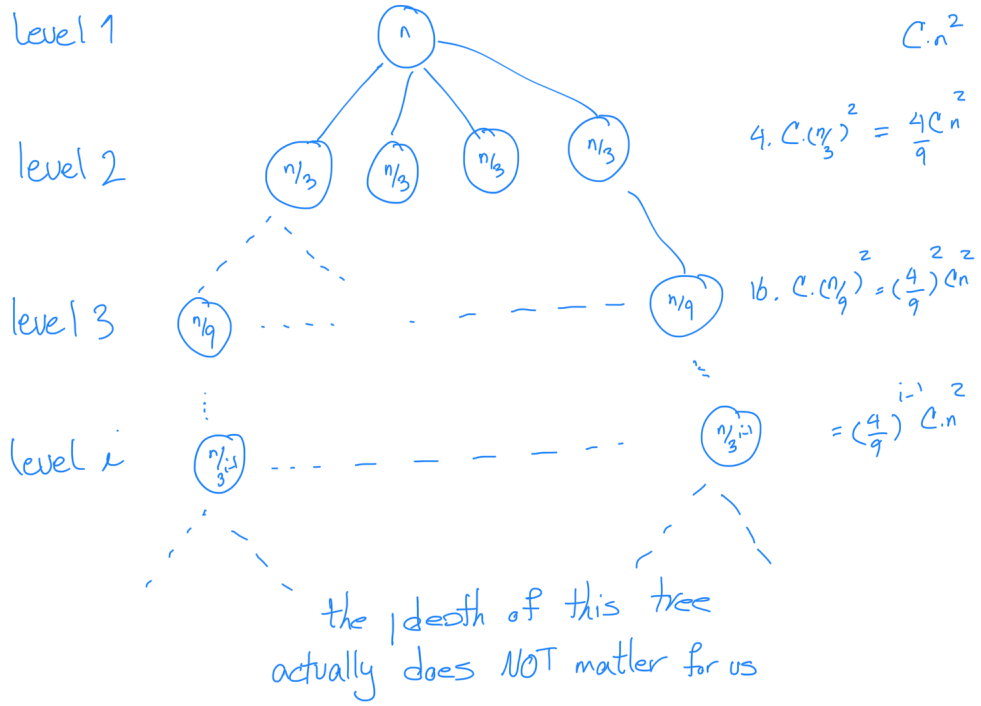


Figure 3: Recursion tree for algorithm C.

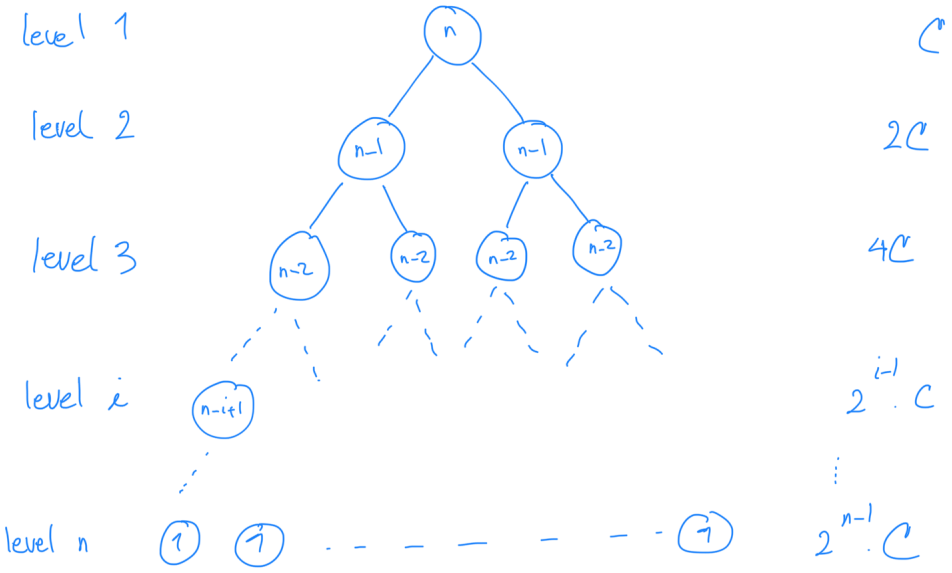


Figure 4: Recursion tree for algorithm D.

We have that the problem instances at each level are getting reduced by 1 in size, as we go down the tree. This can happen at most $n-1$ times until we reach a problem of constant size, which takes constant time to solve. Thus, there are n levels in the recursion tree (the depth of the tree is $n-1$).

At level i, there are 2^{i-1} problems, each of size $n-i+1$, and each such problem incurs a cost of C while combining solutions to its subproblems. Thus, the total cost for a problem of size n, using the

recursion tree is

$$T(n) \leq \sum_{i=0}^{n-1}(2^i \cdot c) = c \cdot \sum_{i=0}^{n-1}(2^i) = c \cdot \frac{2^n - 1}{1} = O(2^n). \quad \text{(by the formula for sum of geometric series)}$$

Problem 3. In this problem, we consider a non-standard sorting algorithm called the *Silly Sort*. Given an array $A[1:n]$ of n integers, the algorithm is as follows:

- **Silly-Sort**$(A[1:n])$:

1. If $n < 5$, run merge sort (or selection sort or insertion sort) on A.
2. Otherwise, run **Silly-Sort**$(A[1:3n/4])$, **Silly-Sort**$(A[n/4:n])$, and **Silly-Sort**$(A[1:3n/4])$ again.

We now analyze this algorithm.

(a) Prove the correctness of **Silly-Sort**. **(10 points)**

Solution. We will prove this via induction. Our induction hypothesis is that **Silly-Sort** sorts the input correctly for all arrays (this is a recursive algorithm so coming up with the induction hypothesis is straightforward). We note that in this question, we assume when dividing the numbers in **Silly-Sort**$(A[1:3n/4])$ and **Silly-Sort**$(A[n/4:n])$, we round *up* the numbers, so these subproblems are strictly smaller in size.

Our base case is $n \leq 4$. The induction hypothesis is true in this case because we know that merge sort is correct from lecture, Silly-Sort must be correct.

We know prove the induction step, meaning that we assume **Silly-Sort** is correct for all arrays of length $n \leq k$ and we prove it for $n = k + 1$ length arrays.

An illustration. Before getting to the proof, let us show an example of running just the first level of **Silly-Sort** on the array $A = [8, 7, 6, 5, 4, 3, 2, 1]$.

- Running **Silly-Sort**$(A[1:3n/4])$ = **Silly-Sort**$(A[1:6])$ results in

$$A = [8, 7, 6, 5, 4, 3, 2, 1] \implies A = [3, 4, 5, 6, 7, 8, 2, 1]$$

  as we assume recursive call sort that part of the array correctly.

- Then, running **Silly-Sort**$(A[n/4:n])$ = **Silly-Sort**$(A[2:8])$ results in

$$A = [3, 4, 5, 6, 7, 8, 2, 1] \implies A = [3, 1, 2, 4, 5, 6, 7, 8]$$

  as we assume recursive call sort that part correctly. The array is still not sorted.

- Finally, running **Silly-Sort**$(A[1:3n/4])$ = **Silly-Sort**$(A[1:6])$ again results in

$$A = [3, 1, 2, 4, 5, 6, 7, 8] \implies A = [1, 2, 3, 4, 5, 6, 7, 8]$$

  as we assume recursive call sort that part of the array correctly. The array is now sorted.

We emphasize that this step is **neither a proof nor at all necessary** but it is a good way for building intuition about the proof. When writing your own solutions, you are strongly *discouraged* to include an example as part of your solution, but it is generally a good idea to think about some examples *before* developing your proof.

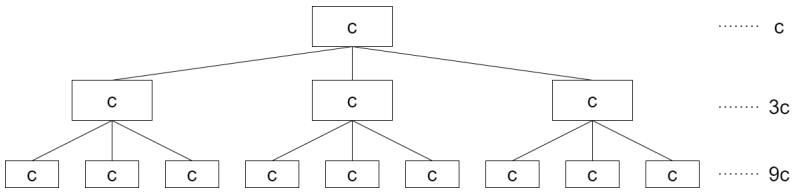
We now prove the induction step (following the illustration above). Note that since size of inner instances passed to **Silly-Sort** is strictly less than n, we can assume by induction hypothesis that they are being sorted correctly (locally). To continue, we need a definition. The *rank* of an element is the correct index of the element in the *sorted* array A. We sometimes also refer to the rank of element as its *correct* position in the array A. We refer to the set of elements with rank in $[3n/4+1 : n]$ as the set L (stands for "large" numbers).

- After the first run of **Silly-Sort**$(A[1 : 3n/4])$, all elements in L will belong to $[n/4 : n]$. This is because at least $n/2$ other elements outside L will be part of $A[1 : 3n/4]$ originally and since this part is sorted correctly, all elements of L appear after them, placing them (well) into $[n/4 : n]$.
- After the next run of **Silly-Sort**$(A[n/4 : n])$, all elements in L will be placed in their *correct* position, i.e., according to their rank. This is because by part one, all elements in L now belong to $A[n/4 : n]$ and since we are sorting this part, and elements in L are the largest ones, they will be placed correctly at $A[3n/4+1 : n]$.
- After the final run of **Silly-Sort**$(A[1 : 3n/4])$, all elements of A outside L will be placed in their *correct* position, i.e., according to their rank. This is simply because by the previous part, all the elements in L are outside of $A[1 : 3n/4]$ now and hence sorting $A[1 : 3n/4]$ will place all elements with rank $[1 : 3n/4]$ in their correct position.

The above proves that A is now sorted, hence proving the induction step. This also concludes the proof of the correctness of the algorithm by induction.

(b) Write a recurrence for **Silly-Sort** and use the recursion tree method of Lecture 3 to solve this recurrence and find the *tightest asymptotic* upper bound on the runtime of **Silly-Sort**. **(10 points)**

Solution. The recurrence for **Silly-Sort** is $T(n) \leq 3 \cdot T(\frac{3n}{4}) + O(1)$, as it involves running 3 recursive calls on arrays of length $3n/4$ and then just $O(1)$ time for making these calls. We now replace the $O(1)$ term with an integer $C > 0$ and write the recursion tree below:



As we can see, there are $C \cdot 3^{i-1}$ operations done at every level i of the tree. Moreover, as we are dividing the instance size by $4/3$ at each level, the total number of levels is $\log_{4/3}(n) + 1$. This means that the total running time of $T(n)$ is

$$\begin{aligned} T(n) &\leq C \cdot \sum_{i=1}^{\log_{4/3}(n)+1} 3^{i-1} = C \cdot \frac{3^{\log_{\frac{4}{3}} n+1} - 1}{3-1} \\ &\leq 3C \cdot 3^{\log_{\frac{4}{3}} n} = 3C \cdot n^{\log_{\frac{4}{3}} 3} && (\text{as } a^{\log_b n} = n^{\log_b a}) \\ &= O(n^{3.8188\ldots}). \end{aligned}$$

Notice that **Silly-Sort** is truly silly and despite all its seemingly complicated work, obtains a very bad running time (much worse than even selection sort or insertion sort).

(c) Suppose we like to change the second line of the algorithm to

$$\textbf{Silly-Sort}(A[1:m]),\ \textbf{Silly-Sort}(A[n-m:n]),\ \text{and}\ \textbf{Silly-Sort}(A[1:m])$$

for some other value of m instead (in the original algorithm, $m = 3n/4$).

What is the smallest number we could pick while still maintaining the correctness of the algorithm? What would be the runtime of the resulting algorithm? For this part of the question, you can simply write a few lines for proof of correctness and runtime analysis by pointing out how your proofs and calculations in parts (a) and (b) should be changed. **(5 points)**

Solution. The smallest number that we could pick would be $m = \frac{2n}{3}$ (and we should round up the division). By modifying our analysis from part (a), we can see that the first **Silly-Sort** recurrence will ensure that the highest ranked $\frac{n}{3}$ elements are placed in $A[\frac{n}{3} : n]$, meaning that our second recurrence will place the highest-ranked $\frac{n}{3}$ elements into their true position. We can then see that our final recurrence will correctly sort the lowest-ranked $\frac{2n}{3}$ elements into their true position, again correctly sorting the entirety of A. We know that this is the smallest number we could pick because this is the smallest number that still ensures that the top $n - m$ elements are placed in $A[n - m : n]$, which must be true the correctness of the algorithm in part (a).

The recurrence relation in this case would be changed to $T(n) \leq 3 \cdot T(\frac{2n}{3}) + O(1)$. We can see that although the general shape of our recurrence tree wont change, the depth will instead become $\log_{\frac{3}{2}} n$. This means, by the same reasoning as before, the overall runtime is $\Theta(3^{\log_{1.5} n}) \approx \Theta(n^{2.7095\ldots})$.

Finally, let us mention that this algorithm with $m = 2n/3$ is actually known and has its own name: the **Stooge Sort** algorithm. And according to Wikipedia, "[Stooge Sort] is notable for its exceptionally bad time complexity".

Problem 4. You are given an array $A[1 : n]$ which includes the scores of n players in a game. They are ranked in the following way: Rank of a player is an integer r if there are exactly $r - 1$ *distinct* scores strictly smaller than the score of this player (irrespective of the number of players).

(a) Design and analyze an algorithm that given the array A, can find the rank of all players in the array in $O(n \log n)$ time. **(15 points)**

Example. Suppose the input array is $A = [1, 7, 6, 5, 2, 4, 5, 2]$ for 8 players; then the rank of players is:

- Player $A[1]$ has rank 1 (as $A[1] = 1$ is the smallest number);
- Player $A[2]$ has rank 6 (as $A[2] = 7$ has 5 distinct smaller numbers: $\{1, 6, 5, 4, 2\}$);
- Player $A[3]$ has rank 5 (as $A[3] = 6$ has 4 distinct smaller numbers: $\{1, 5, 4, 2\}$);
- Player $A[4]$ has rank 4 (as $A[4] = 5$ has 3 distinct smaller numbers: $\{1, 4, 2\}$);
- Player $A[5]$ has rank 2 (as $A[5] = 2$ has 1 distinct smaller number: $\{1\}$);
- Player $A[6]$ has rank 3 (as $A[6] = 4$ has 2 distinct smaller numbers: $\{1, 2\}$);
- Player $A[7]$ has rank 4 (as $A[7] = 5$ has 3 distinct smaller numbers: $\{1, 4, 2\}$);
- Player $A[8]$ has rank 2 (as $A[8] = 2$ has 1 distinct smaller number: $\{1\}$);

Solution. A complete proof consists of three steps, the algorithm, the proof of correctness, and the runtime analysis.

Algorithm. The algorithm is simple: we first sort A using merge sort. Then, we go over the elements one by one and make sure to give the same rank to the equal scores and only increase the rank once the score we iterate over is larger than the previous one. Formally,

(a) Turn every element of $A[i]$ into a pair $(A[i], i)$ so we have the index of each player in the original array alongside it. In the following, we use $A[i].first$ and $A[i]$.second to refer to the first and second part of the pair in $A[i]$ respectively.

(b) Sort the pairs in A with merge sort in increasing order of $A[i]$.first. This way, after the sorting, each entry $A[i]$ of the array will be a pair $(A[i], j)$ where j is the initial index of $A[i]$ and we will have $A[1].first \leq A[2].first \leq \cdots \leq A[n].first$.

(c) Let $R = 1$ originally and output rank of $A[1].second$ as 1. For $i = 2$ to n:

- if $A[i].first > A[i-1].first$, increase R by one;
- Output rank of $A[i].second$ as R.

Proof of Correctness: By the correctness of merge sort, we can assume that A will be sorted correctly. We now prove that Line (c) computes rank of each player correctly. The proof is by induction on the value of $i \in \{1, \ldots, n\}$, and the statement is that after iteration i of the for-loop, rank of players $A[1:i].second$ is computed correctly.

For the base case, we have that when $i = 1$, the algorithm output rank of player $A[1].second$ as 1. Considering $A[1].first$ is the smallest score in the array, this is certainly correct.

For the induction step, let us assume the statement is true for $i = j$ and we prove it for $i = j + 1$. By induction hypothesis, rank of players $A[1:j].second$ is computed correctly; now if $A[j+1].first = A[j].first$, we give the same rank to $A[j+1].second$ as well which will be correct as the rank of players with the same score should be equal. On the other hand, if $A[j+1].first > A[j].first$, then since A is sorted, we know that $A[j+1].first$ is the smallest score larger than $A[j].first$ and thus rank of player $A[j+1].second$ should be one larger than $A[j].second$, exactly as done by the algorithm.

This proves the induction step and thus the entire induction hypothesis. By applying the hypothesis to $i = n$, we get that after the last iteration, rank of every player is computed correctly.

Runtime Analysis: The first line of the algorithm for creating pairing takes $O(n)$ time, the second line for merge sort takes $O(n \log n)$ time, and the last line for computing the ranks takes another $O(n)$ time. So the total runtime is $O(n \log n)$.

(b) Suppose you are additionally given an array $B[1:m]$ with the score of m new players. Design and analyze an algorithm that given both arrays A and B, can find the rank of each player B inside the array A, i.e., for each $B[i]$, determines what would be the rank of $B[i]$ in the array consisting of all elements of A plus $B[i]$. Your algorithm should run in $O((n+m) \cdot \log n)$ time. **(10 points)**

Example. Suppose the input array is $A = [1, 7, 6, 5, 2, 4, 5, 2]$ as before and $B = [3, 9, 4]$; then the correct answer for each player in B is:

- Player $B[1]$ will have rank 3 (as $B[1] = 3$ has 2 distinct smaller numbers in A: $\{1, 2\}$);
- Player $B[2]$ will have rank 7 (as $B[2] = 9$ has 6 distinct smaller numbers in A: $\{1, 7, 6, 5, 4, 2\}$);
- Player $B[3]$ will have rank 3 (as $B[3] = 4$ has 2 distinct smaller numbers in A: $\{1, 2\}$);

Solution. We first design a binary search type algorithm for the problem of finding the largest index i in an array A such that $A[i] < t$ for a given t. We then use this to design the whole algorithm. We partition the solution into these two parts separately. Let us note that if you did not explicitly write an algorithm for this first part and instead simply used "binary search" to mean an algorithm for this problem, you will receive the full credit for this part. However, we provide a complete proof of this modification for anyone who is interested.

First Part: *Algorithm:* We call the modified binary search algorithm **Lower-Bound**

- **Lower-Bound**$(A[1:n], t)$: Given a sorted array $A[a:b]$ and an integer t;
  (a) If $a > b$: return 'no such index'. If $a = b$: $A[a] < t$ return a, otherwise return 'no such index'. If $a + 1 = b$: if $A[b] < t$, return b, else if $A[a] < t$, return a, otherwise return 'no such index'.
  (b) Let $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$.
  (c) If $A[m] \geq t$, return **Lower-Bound**$(A[a : m-1], t)$.
  (d) If $A[m] < t$, return **Lower-Bound**$(A[m : b], t)$.

Proof of Correctness: The proof is by induction for the hypothesis that **Lower-Bound** outputs the correct answer for every array of length n (i.e., $n = \min\{b - a + 1, 0\}$).

The induction base when $n = 0$, $n = 1$, $n = 2$ is true by the logic of the algorithm exactly as in the binary search and we do not repeat the argument here.

Suppose the algorithm works for all arrays of length $n \leq k$ and we prove it for $n = k + 1$. Case one is when $A[m] \geq t$. Since the array A is sorted, the correct index, say j, in this case cannot possibly be in $A[m : b]$: this is because $A[j] < t$ and since A is sorted and $A[m] \geq t$, j cannot be part of $A[m : b]$. As such, we only need to search for the correct index in $A[a : m - 1]$. By induction, since size of $A[a : m - 1]$ is smaller than $n = k + 1$, the algorithm finds the correct index in this array. The second case is when $A[m] < t$: in this case the answer can be either $j = m$, or some other index in $[m : b]$. However, since $A[m] < t$ already and we are looking for the *largest* index smaller than t, the correct index j cannot be part of $A[a : m - 1]$; hence, in this case also, since size of $A[m : b]$ is smaller than $n = k + 1$ (here we have to use the fact that $n > 2$ and that is the reason we proved the base case for $n = 2$ also), by induction, we will find the correct index. This proves the correctness of the algorithm.

Runtime Analysis: The runtime is asymptotically the same as binary search and is hence $O(\log n)$.

Second part: We now solve the main problem.

Algorithm: The algorithm is as follows:

(a) Run the algorithm on part (a) to compute the rank of every player in A (we assume that A is sorted now).

(b) For every $i = 1$ to m, run **Lower-Bound**$(A[1 : n], B[i])$ to get the largest index j such that $A[j] \leq B[i]$. If $A[j] = B[i]$ return the rank of $B[i]$ as the rank of $A[j].second$, and otherwise return the rank of $B[i]$ as the rank of $A[j].second$ plus one.

Proof of Correctness: By the correctness of part (a), we know that rank of every player in A is computed correctly. By the correctness of **Lower-Bound**, we know that $A[j]$ has the highest score, smaller than or equal to $B[i]$. As such, if the score of $A[j]$ is the same as $B[i]$, then their rank should be equal, and if score of $A[j]$ is strictly less than $B[i]$, then rank of $B[i]$ should be one higher, exactly as is done in the algorithm. This proves the correctness.

Runtime Analysis: Line (a) of the algorithm takes $O(n \log n)$ time by the previous part. Each of the m iterations of the for-loop takes $O(\log n)$ time by the runtime of **Lower-Bound** so the total runtime of the for-loop is $O(m \log n)$. This means the total runtime is $O((m + n) \log n)$ as desired.
