COMP3121/9101: Algorithm Design and Analysis

Problem Set 1 – Preliminary Material

[K] – key questions **[H]** – harder questions **[E]** – extended questions

☆ – contains sample solutions

Contents

Section One: Data Structures and Algorithms

☆ [K] **Exercise 1.** Assume that you have an array of $2n$ distinct integers. Find the largest and smallest integers using at most $3n - 2$ comparisons.

Hint. *How many comparisons do you need to find the largest element?*

[K] **Exercise 2.** Assume that you have an array of 2^n distinct integers. Find the largest and second largest integers using at most $2^n + n - 2$ comparisons.

Hint. *Use the same idea as Exercise 1. What data structure should we use instead?*

[K] **Exercise 3.** You are given an array A of n integers and another integer x.

(a) Design an $O(n \log n)$ algorithm that decides whether there exists two integers in A that sum to x.

(b) Design an algorithm that runs in $O(n)$ *expected time* that solves the same problem.

☆ [K] **Exercise 4.** Let A be an array with n positive integers, and let k be a positive integer. Your goal is to determine if there exist two integers in A whose difference is exactly k. In other words, you want to determine if there exist distinct indices $1 \leq i, j \leq n$ such that $A[i] - A[j] = k$.

(a) Assume that A is sorted (in increasing order).

- If $A[i] - A[j] > k$, show that $A[i] - A[j'] > k$ whenever $j' \leq j$.
- Similarly, if $A[i] - A[j] < k$, show that $A[i'] - A[j] < k$ whenever $i' \leq i$.

(b) Hence, design a linear-time algorithm that determines if there exist distinct indices $1 \leq i, j \leq n$ such that $A[i] - A[j] = k$.

(c) We now remove the assumption that A is sorted. Design an $O(n \log n)$ algorithm to solve the same problem. Briefly justify the correctness of your algorithm.

[K] **Exercise 5.** We generalise Exercise 4 to the following problem: let A be an array with n positive integers and let k be a positive integer. Your goal is to determine if there exist two integers in A whose *absolute* difference is exactly k. In other words, you want to determine if there exist distinct indices $1 \leq i, j \leq n$ such that $|A[i] - A[j]| = k$.

(a) Extend the algorithm from Exercise 4 to solve the problem. Your algorithm should run in $O(n \log n)$ time. Briefly justify the correctness of your algorithm.

(b) Design an algorithm that runs in $O(n)$ *expected time* that solves the same problem.

☆ [H] **Exercise 6.** There are n teams in the local cricket competition and you happen to have n friends that keenly follow it. Each friend supports some subset (possibly all or none too) of the n teams. Not being the sporty type, but wanting to fit in nonetheless, you must decide for yourself which subset of teams (again, possibly all or none too) to support.

You don't want to be branded as a copy cat so your subset must not be identical to anyone else's. The trouble is, you don't know which friends support which teams but you can ask your friends some question of the form: "*does friend A support team B?*" (you choose A and B in advance).

Design a strategy that determines a suitable subset of teams for you to support and asks the fewest number of questions as possible.

[H] Exercise 7. You are at a party attended by n people (not including yourself!), and you suspect that there might be a celebrity present. A *celebrity* is someone who is known by everyone but doesn't know anyone present. Your task is to design a strategy to work out if there is a celebrity present and if so, who the celebrity is. However, you can only do so by asking a person X if they know person Y, where you can choose X and Y when asking the question.

(a) Show that there can be at most one celebrity; that is, if a celebrity is present, then the celebrity is unique.

Hint. *When asking a question, what information do you gain?*

(b) Show that you can always accomplish the task by asking at most $3n - 3$ questions.

Hint. *How many questions do you need to ask to solve part (a)? How many questions do you need to ask to verify that the celebrity really is a celebrity?*

(c) Show that you can always accomplish the task by asking at most $(3n - 3) - \lfloor \log_2 n \rfloor$ questions.

Hint. *In the verification step, what piece of information can you reuse? What data structure can we use?*

Section Two: Sorting and Searching

☆ **[K] Exercise 8.** Let A be an array with $n-1$ elements, containing the elements from $1, \dots, n$ except for one. Design an $O(n)$ algorithm that finds the missing integer.

There are many ways to do this.

[K] Exercise 9. An array A containing n elements is *palindromic* if it can be read the same forwards and backwards. For example, the array $A = [1, 2, 3, 2, 1]$ is palindromic while $B = [1, 2, 3, 1, 1]$ is not palindromic. Given an array A of n elements, design an $O(n)$ algorithm that decides whether A is palindromic.

[K] Exercise 10. You are given an array A of n distinct integers, sorted in increasing order. In other words, $A[1] < A[2] < \cdots < A[n]$.

(a) Design an $O(\log n)$ algorithm that either computes an index i such that $A[i] = i$, or return that no such index exists.

Hint. *Consider* $B[i] = A[i] - i$.

(b) Now suppose we know that $A[1] > 0$. Design an $O(1)$ algorithm to solve the same problem.

Hint. *Again, consider* $B[i] = A[i] - i$.

[K] Exercise 11. Your army consists of a line of n giants, each with a certain height. You must designate precisely $\ell \leq n$ of them to be leaders. Leaders must be spaced out across the line; specifically, every pair of leaders must have at least $k \geq 0$ giants standing in between them. Given n, ℓ, k and the heights, $H[1], H[2], \dots, H[n]$, of the giants in the order that they stand in the line as input, find the *maximum* height of the *shortest* leader among all valid choices of ℓ leaders. We call this the *optimisation* version of the problem.

For example, consider the following inputs: $n = 10$, $\ell = 3$, $K = 2$, and $H = [1, 10, 4, 2, 3, 7, 12, 8, 7, 2]$. Then, among the 10 giants, you must choose 3 leaders so that each pair of leaders has at least 2 giants standing in between them. The best choice of leaders has heights 10, 7 and 7, with the shortest leader having height 7. This is the best possible for this case.

(a) In the *decision* version of this problem, we are given an additional integer T as input. Our task is to decide if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T.

Give an algorithm that solves the decision version of this problem in $O(n)$ time.

(b) Hence, show that you can solve the optimisation version of this problem in $O(n \log n)$ time.

Hint. *How could you use the decision version to solve the optimisation problem?*

☆ **[H] Exercise 12.** You are given a sorted array A containing n distinct positive integers. You are also given a function $f(x)$ which is known to be a decreasing function whenever $x > 0$. Design an $O(\log n)$ algorithm that determines if A contains a value for x such that $f(x) = 0$.

[H] Exercise 13. You are given two arrays A and B, and each array contains n distinct positive integers. You are also given a two-dimensional function $f(x, y) = y^6 + x^4y^4 + x^2y^2 - x^8 + 10$.

(a) Let $x, y > 0$. For a fixed value of x, show that $f(x, y)$ is an increasing function in terms of y.

Hint. *Differentiate the function with respect to y and show that the derivative is positive as long as $x, y > 0$.*

(b) Hence, design an $O(n \log n)$ algorithm that determines if A contains a value for x and B contains a value for y such that $f(x, y) = 0$.

Hint. *Extend the algorithm from Exercise 12.*

☆ **[H] Exercise 14.** Let $f : \mathbb{N} \to \mathbb{Z}$ be a monotone and increasing function; that is, for all $i \in \mathbb{N}$, we have that $f(i) < f(i+1)$. Our task is to find the smallest integer $i \in \mathbb{N}$ such that $f(i) \geq 0$, or return that no such index exists.

(a) Let N be the maximum value of f. In other words, for all $i \in \mathbb{N}$, we have that $f(i) \leq N$. Design an $O(\log N)$ algorithm to find the smallest integer i for which $f(i) \geq 0$, or return that no such index exists.

(b) How could you generalise this to an unbounded function?

[H] Exercise 15. We now extend Exercise 14 to the following problem: you are given a monotone and decreasing function $f : \mathbb{N} \to \mathbb{Z}$ and an integer k. You want to find the smallest index i for which $f(i) \leq k$.

Extend the previous algorithm to solve this problem.

☆ **[H] Exercise 16.** Let M be an $n \times n$ matrix of distinct integers $M(i, j)$ with $1 \leq i, j \leq n$. Each row and each column of the matrix is sorted in increasing order so that for each row i, we have that

$$M(i,1) < M(i,2) < \cdots < M(i,n)$$

and for each column j, we have that

$$M(1,j) < M(2,j) < \cdots < M(n,j).$$

Given the matrix M and an integer x, design an $O(n)$ algorithm that decides whether M contains the integer x.

[H] Exercise 17. Let A be an array with n integers. You need to answer a series of n queries, each of which is of the form "*how many elements a of the array A satisfy $L_k \leq a \leq R_k$?*", where L_k, R_k (for some $1 \leq k \leq n$) are integers such that $L_k \leq R_k$. Design an $O(n \log n)$ algorithm that answers each of these n queries.

[H] Exercise 18. Suppose that you are taking care of n kids, who took their shoes off. You have to take the kids out and it is your task to make sure that each kid is wearing a pair of shoes of the right size (not necessarily their own, but one of the same size). All you can do is to try to put a pair of shoes on a kid, and see if they fit, or are too large or too small; you are *not* allowed to compare a shoe with another shoe or a foot with another foot. Describe an algorithm whose expected number of shoe trials is $O(n \log n)$ which properly fits shoes on every kid.

☆ **[E] Exercise 19.** You are given n numbers $x_1, \ldots, x_n$, where each x_i is a real number in the interval $[0, 1]$. Your task is to return a permutation $y_1, \ldots, y_n$ such that

$$\sum_{i=2}^{n} |y_i - y_{i-1}| < 2.$$

(a) Consider the set of intervals $b_0, \ldots, b_{n-1}$ where b_i is the interval

$$b_i = \left[\frac{i}{n}, \frac{i+1}{n}\right).$$

It is not hard to show that the set $\{b_0, \ldots, b_{n-1}\}$ partitions the interval $[0, 1)$. Consider the function $f : \mathbb{R} \to \{0, \ldots, n-1\}$ where $f(x) = k$ if and only if $x \in b_k$. Prove that if $x_i \in [0, 1]$, then $f(x_i) = \lfloor nx_i \rfloor$.

(b) Let $z_1, \ldots, z_m$ be m elements belonging in the same bucket. Show that

$$\sum_{i=2}^{m} |z_i - z_{i-1}| < 1.$$

Hint. *What is the size of each bucket? Can you find an upper bound for m?*

(c) Hence, describe an $O(n \log n)$ algorithm that outputs a permutation $y_1, \ldots, y_n$ of the n numbers such that

$$\sum_{i=1}^{n} |y_i - y_{i-1}| < 2.$$

☆ **[K] Exercise 20**. Show the following asymptotic relations by providing suitable constants for $c, N > 0$.

(a) $n^2 = O(n^3)$.

(b) $4n^3 + 8n^2 + 1 = O(n^3)$.

(c) $n^2 + \sin(n) = O(n^2)$.

(d) $\cos(n) + \sin(n) = O(1)$.

[K] Exercise 21. Show the following asymptotic relations by providing suitable constants for $c, N > 0$.

(a) $n^3 = \Omega(n^2)$.

(b) $2^n = \Theta(2^{n+1})$.

(c) $b^n = \Omega(a^n)$ whenever $a < b$.

(d) $\cos(n) + \sin(n) + 2 = \Theta(1)$.

[K] Exercise 22. Determine if $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, both (i.e. $f(n) = \Theta(g(n))$) or neither for the following pairs of functions. Justify your answer in each.

(a) $f(n) = (\log_2 n)^2$, $g(n) = \log_2\left(n^{\log_2 n}\right) + 2\log_2 n$.

(b) $f(n) = n^{100}$, $g(n) = 2^{n/100}$.

(c) $f(n) = \sqrt{n}$, $g(n) = 2^{\sqrt{\log_2 n}}$.

(d) $f(n) = n^{1.001}$, $g(n) = n\log_2 n$.

(e) $f(n) = n^{(1+\sin(\pi n/2))/2}$, $g(n) = \sqrt{n}$.

[K] Exercise 23. Let $f(n)$ and $g(n)$ be two positive functions on $\mathbb{N}$. Prove that $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

Hint. *First show that if $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$. Then show that if $g(n) = \Omega(f(n))$, then $f(n) = O(g(n))$.*

[K] Exercise 24. Prove the following Big-Oh properties.

(a) If $c > 0$, then $c \cdot f(n) = O(f(n))$. In other words, constants do not affect the growth rate.

(b) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. In other words, the Big-Oh operation is transitive.

(c) If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

(d) $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$.

Hint. *Show that any function in $O(f(n) + g(n))$ belongs in $O(\max\{f(n), g(n)\})$ and any function in $O(\max\{f(n), g(n)\})$ also belongs in $O(f(n) + g(n))$.*

☆ **[H] Exercise 25**. Let $f(n)$ and $g(n)$ be two positive functions for all $n \in \mathbb{N}$.

(a) Show that, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = O(g(n))$.

(b) Show that, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n) = \Omega(g(n))$.

[H] Exercise 26. This exercise shows that Big-Oh is not preserved under monotonic functions. Suppose that $f(n) = O(g(n))$, and let $h(n)$ be some monotonic function in n.

(a) Let $h(n) = 2^n$. By considering the derivative of $h(n)$, or otherwise, show that $h(n)$ is monotonic in n.

(b) Consider $f(n) = 2^{n+1}$ and $g(n) = 2^n$. Show that $f(n) = O(g(n))$.

(c) Show that $h(f(n)) \neq O(h(g(n)))$. This shows that Big-Oh is not necessarily preserved under monotonic functions.

☆ **[E] Exercise 27**. Classify the following pairs of functions by their asymptotic relation; that is, determine if $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, both (i.e. $f(n) = \Theta(g(n))$) or neither.

(a) $f(n) = n^{\log n}$, $g(n) = (\log n)^n$.

(b) $f(n) = (-1)^n$, $g(n) = \tan(n)$.

(c) $f(n) = n^{5/2}$, $g(n) = \left[\log\left(\sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!}\right)\right]^2$.

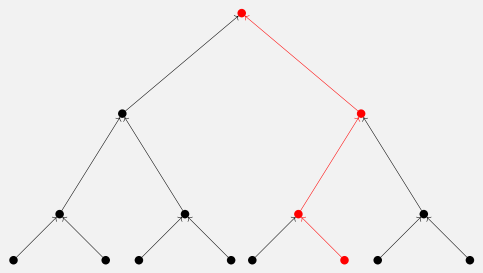
Solutions

Exercise 1. Finding the largest and smallest integers naively requires $(2n-1)+(2n-2) = 4n-3$ comparisons, which is too many for our purposes. Instead, notice that each comparison we make tells us that one integer is *definitely* not the largest and the other integer is *definitely* not the smallest. Therefore, to avoid making unnecessary comparisons, we shall store the smaller integers into an array and the larger integers into a separate array. In other words, our strategy is as follows:

- Form n pairs of integers.
- In each pair, store the larger integer into an array L and store the smaller integer into an array S. After we make all n comparisons, we now have two arrays, each of size n.
- Find the maximum in L and the minimum in S using only $(n-1)$ comparisons.

Therefore, we can find the largest and smallest integers using only $n + 2(n-1) = 3n-2$ comparisons.

Exercise 2. Consider the figure below.



We see a complete binary tree with 2^n leaves and $2^n - 1$ internal nodes and of depth n (the root has depth 0). We then place all the numbers at the leaves, compare each pair and "promote" the larger element (shown in red) to the next level and proceed in such a way till you reach the root of the tree, which will contain the largest element.

Clearly, each internal node is a result of one comparison and there are $2^n - 1$ many nodes thus also the same number of comparisons so far. Now just note that the second largest element must be among the black nodes which were compared with the largest element along the way - all elements underneath them must be smaller or equal to the elements shown in black. There are n many such elements so finding the largest among them will take $n-1$ comparisons by brute force.

In total, requires at most $2^n + n - 2$ many comparisons.

Exercise 3.

(a) Note that a brute force solution considers all possible pairs of $(A[i], A[j])$ does not suffice as it runs in $O(n^2)$ time. Instead, we start by sorting the array in ascending order, which can be done in $O(n \log n)$ in the worst case using an algorithm such as Merge Sort. Here we give 2 valid approaches.

Approach 1: For each element a in the array, we can check if there exists an element $x - a$ also in the array in $O(\log n)$ time using binary search. The only special case is if $a = x - a$ (i.e. $x = 2a$), where we just need to check the two elements adjacent to a in the sorted array to see if another a exists.

Hence, we take at most $O(\log n)$ time for each element, so this part is also $O(n \log n)$ time in the worst case, giving an $O(n \log n)$ algorithm.

Approach 2: Alternatively, we add the smallest and the largest elements of the array. If the sum exceeds x no solution can exist involving the largest element; if the sum is smaller than x then no solution can exist involving the smallest element. Thus, if this sum is not equal to x we can eliminate 1 element.

After at most $n-1$ many such steps you will either find a solution or will eliminate all elements except one, thus verifying no such elements exist. This takes $O(n)$ time in the worst case, so the overall running time is $O(n \log n)$ as the sorting dominates.

(b) We take a similar approach in part (a), except we use a hash map (or hash table, denoted H) to check if elements exist in the array (rather than sorting it): each insertion and lookup takes $O(1)$ **expected** time. We again provide 2 valid approaches.

Approach 1: At index $i \in \{1, 2, \ldots, n\}$, we assume the previous $i-1$ elements of A are already stored in H. Then we check if $x - A[i]$ is in H in expected $O(1)$, then insert $A[i]$ into H, also in expected $O(1)$.

As this process will take n insertions and n look-ups in the worst case, we conclude that the algorithm runs in $O(2n) = O(n)$ **expected** time.

Approach 2: Alternatively, we hash all elements of A and store its occurrence frequency. We then go through elements of A again, this time for each element a we check if $x - a$ is in H. However, if $2a = x$, we must also check if at least 2 copies of a appear in the corresponding slot H.

As this process will take again n insertions and n look-ups in the worst case, hence the algorithm runs in $O(n)$ **expected** time.

Exercise 4.

(a) Since A is sorted, we have that $A[1] \le A[2] \le \cdots \le A[n]$. Now, suppose that $A[i] - A[j] > k$. Since $j' \le j$, we have that $A[j'] \le A[j]$ which implies that

$$A[i] - A[j'] \ge A[i] - A[j] > k.$$

Therefore, $A[i] - A[j'] > k$.

Now, suppose that $A[i] - A[j] < k$. By a similar analysis, since $i' \le i$, we have that $A[i'] \le A[i]$ which implies that

$$A[i'] - A[j] \le A[i] - A[j] < k.$$

Therefore, $A[i'] - A[j] < k$.

(b) Our algorithm is as follows: construct two pointers i, j which point to the first two elements of the array. Here, j points to the first element and i points to the second element. Our algorithm will iterate through the array in the following way:

- If $A[i] - A[j] = k$, then we terminate and return that such a pair exists.
- If $A[i] - A[j] > k$, then we increment the j pointer to consider indices larger than j.
- If $A[i] - A[j] < k$, then we increment the i pointer to consider indices larger than i.

If we exhaust all possible indices without terminating early, then no such pair of indices exist. Since the i and j pointers never backtrack, the algorithm is equivalent to performing a linear scan of the array and so, the algorithm has running time $O(n)$ which is linear in the size of the array.

Correctness
The correctness follows from our analysis in part (a). We showed that if $A[i] - A[j] > k$, then $A[i] - A[j'] > k$ for each $j' \le j$. Thus, it is sufficient to look for pairs of indices (i, j') for which $j' > j$. Similarly, we showed that if $A[i] - A[j] < k$, then $A[i'] - A[j] < k$ for each $i' \le i$. Thus, it is also sufficient to look for pairs of indices (i', j) for which $i' > i$. This justifies the correctness of the last two cases.

Finally, if a pair of indices (i, j) such that $A[i] - A[j] = k$ exist, then the first case will terminate correctly. If no pair of indices exist, then the algorithm will have exhausted all pairs of indices without triggering the first case and again, this is correctly returned by our algorithm. This justifies the correctness of our algorithm.

(c) Using *merge sort*, we can sort A in increasing order and apply the algorithm from part (b). The merge sort process has running time $O(n \log n)$ and we showed that the subroutine from part (b) is linear. Therefore, the running time of our algorithm is $O(n \log n) + O(n) = O(n \log n)$. The correctness follows from the correctness of part (b).

Exercise 5.

(a) We extend the previous algorithm. Note that the previous algorithm only checks if there exist indices i, j such that $A[i] - A[j] = k$. To check whether $|A[i] - A[j]| = k$, we need to additionally check if $A[i] - A[j] = -k$. Therefore, we run the previous algorithm twice: once for k and once for $-k$. If either algorithm returns true, then we return true; otherwise, return false.

Correctness
The correctness of the algorithm comes from the correctness of Exercise 4. Note that

$$|A[i] - A[j]| = k \iff A[i] - A[j] = \begin{cases} k & \text{if } A[i] - A[j] > 0, \\ -k & \text{if } A[i] - A[j] < 0. \end{cases}$$

Therefore, if we check both cases and return false, then no pair of indices are possible for the absolute difference.

(b) We create a hash table H, which is initially empty. Now, for each element a in A, we first perform a check to see whether a is already hashed in H. If it hasn't been hashed before, then we hash $a + k$ and $a - k$ into H. Otherwise, we must have hashed a previous element from A into a. But by the way that we have hashed our values, this implies that $b + k = a$ or $b - k = a$ for some element b. But this implies that either $a - b = k$ or $b - a = k$, in which case we can terminate and return that there is such a pair of elements in A.

This operation is $O(n)$ in the *expected* time because each time we search for an element in the hash table, the operation is expected $O(1)$ time complexity. Doing this for all n elements in A gives us $O(n)$ expected time complexity.

Exercise 6. Our strategy is as follows:

- Number your friends arbitrarily in order from 1 to n, and number the teams arbitrarily in order from 1 to n.
- For each i from 1 to n, ask the ith friend if they support the ith team.
  - If the ith person supports the ith team, then we do not support the ith team.
  - If the ith person does not support the ith team, then we support the ith team.
- Return the final list of teams that we support.

Note that we use only n queries which is the smallest possible number of queries since we have to ask every friend at least once.

Correctness
We now claim that the list we return is not the same as any of the n friends. To see why, a list L is the same as list R if and only if L agrees with R on every index. For some $1 \leq k \leq n$, let the kth friend be the first friend that agrees with our list on every team. Since $k \leq n$, we must have asked the kth friend if they support the kth team. By our strategy, we must have mismatched on the kth team and therefore, our lists must not have been the same to begin with. We can

repeat the same argument for each subsequent k, which implies that no friend's list is the same as our list. Therefore, our algorithm returns a list that is unique.

Exercise 7. Assume that the people are assigned a number from 1 to n.

(a) Suppose that there are two celebrities, say A and B. Since A is a celebrity, B must know A since everyone knows the celebrity. But then this implies that A cannot be a celebrity since a celebrity does not know anyone. So there can only be *at most* one celebrity.

(b) Using the result from the previous part, we proceed as follows.

Arbitrarily pick any two people, say A and B. Ask if A knows B.

- If A knows B, then we know that A cannot be a celebrity.
- If A does not know B, then we know that B cannot be a celebrity.

In either case, we can eliminate one person from our *celebrity candidate* list. Since there are n people, we can ask $n - 1$ people to find the *celebrity candidate*. Let this candidate be C.

We now check if C is indeed the celebrity (it could very well be the case that C is not the celebrity and as such, no celebrity exists). For each person X in the party, we ask the following question:

Does X know C?

- If X does not know C, then we conclude that C cannot be the celebrity and thus, no celebrity is present.
- Otherwise, C may still be the celebrity.

We finally need to check if C knows anyone in the party. Again, choosing every person X in the party, we ask the following question:

Does C know X?

- If C knows X, then we conclude that C cannot be the celebrity and thus, no celebrity is present.
- If C does not know X, we continue until we have exhausted everyone in the party.

We only conclude that C must be the only celebrity once we have concluded that C does not know anyone. In the worst case, we consume $n-1$ questions to determine who the potential celebrity candidate is and then $2(n-1)$ questions to verify whether C is indeed the celebrity. Thus, in the worst case, we use $(n-1)+2(n-1) = 3n-3$ questions in total.

(c) We arrange n people present as leaves of a **balanced full tree**, i.e., a tree in which every node has either 2 or 0 children and the depth of the tree is as small as possible. To do that compute $m = \lfloor \log_2 n \rfloor$ and construct a perfect binary three with $2^m \leq n$ leaves. If $2^m < n$ add two children to each of the leftmost $n-2^m$ leaves of such a perfect binary tree. In this way you obtain $2(n-2^m)+(2^m-(n-2^m)) = 2n-2^{m+1}+2^m-n+2^m = n$ leaves exactly, but each leaf now has its pair, and the depth of each leaf is $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor + 1$. For each pair we ask if, say, the left child knows the right child and, depending on the answer as in (a) we promote the potential celebrity one level closer to the root. It will again take $n-1$ questions to determine a potential celebrity, but during the verification step we can save $\lfloor \log_2 n \rfloor$ questions (one on each level) because we can reuse answers obtained along the path that the potential celebrity traversed through the tree. Thus, $3n-3-\lfloor \log_2 n \rfloor$ questions suffice.

Exercise 8.
Solution 1.
Our algorithm is as follows: sum all of the elements in A, say $S = A[1] + \cdots + A[n-1]$. Then the missing element is

just the integer given by

$$\frac{n(n+1)}{2} - S.$$

Our algorithm does a single traversal through the array, and computes a constant-time operation to compute $\frac{n(n+1)}{2}$ and $\frac{n(n+1)}{2} - S$. Therefore, the overall running time of the algorithm is $O(n)$.

Correctness
Since A is an array with $n-1$ elements containing all but one integer in the set $\{1,\ldots,n\}$, we deduce that every element in A appears at most once. Let x be the missing integer. Then we see that

$$(1 + 2 + \cdots + (x-1)) + x + ((x+1) + \cdots + n) - (A[1] + \cdots + A[n-1]) = x. \tag{1}$$

By the arithmetic progression formula, we have that

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

Therefore, equation (1) can be written as

$$\frac{n(n+1)}{2} - (A[1] + \cdots + A[n-1]) = x,$$

which proves that our expression is correct.

Solution 2.
The following solution uses properties of the *exclusive-or* operation. For integers x, y, z where $x \neq y \neq z$, we have the following properties:

- **Self-inverse**: $x \oplus x = 0$.
- **Commutative**: $x \oplus y = y \oplus x$.
- **Associative**: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- **Identity**: $x \oplus 0 = x$.

The algorithm computes two big *exclusive-or* operations. We first consider the exclusive-or of every integer from $1,\ldots,n$. Let $S = 1 \oplus 2 \oplus \cdots \oplus n$. We then take the exclusive-or of every element in A: $T = A[1] \oplus A[2] \oplus \cdots \oplus A[n-1]$. The missing integer is then $S \oplus T$.

Each *exclusive-or* operation is constant-time and our algorithm does two traversals, each of which is a linear-time traversal. Therefore, the overall running time of the algorithm is $O(n)$.

Correctness
Let x be the missing integer. Every integer from $1,\ldots,n$ appears *exactly* twice except for x. By commutativity and associativity, we can re-express $S \oplus T$ as

$$(1 \oplus 1) \oplus \cdots \oplus ((x-1) \oplus (x-1)) \oplus x \oplus ((x+1) \oplus (x+1)) \oplus \cdots \oplus (n \oplus n).$$

By the self-inverse property, we have that $i \oplus i = 0$ for each $i \in \{1,\ldots,x-1,x+1,\ldots,n\}$. We can eventually reduce the expression to $0 \oplus x = x$, which is just the missing integer. Therefore, the expression $S \oplus T$ correctly returns the missing integer.

Exercise 9. To ensure that our array is *palindromic*, we need to ensure that $A[i] = A[n-i+1]$ for each $i = 1,\ldots,n-1$. Note that it is enough to check that the equality holds for $i = 1,\ldots,\lceil n/2 \rceil$ because any index $i > \lceil n/2 \rceil$ is equivalent to an index from 1 to $\lceil n/2 \rceil$ by symmetry, where $\lceil x \rceil$ denotes the *smallest integer bigger than or equal to* x.

Therefore, our strategy is as follows: we set up two pointers, one that keeps track of $A[i]$ and one that keeps track of $A[n-i+1]$. If $A[i] \neq A[n-i+1]$ for any i from 1 to $\lceil n/2 \rceil$, then we can terminate our algorithm and return that the array is not palindromic. Otherwise, if we have exhausted all possible indices and equality is satisfied, then we return that the array is palindromic.

The algorithm has a linear-time running time since we are looping through at most $O(n/2) = O(n)$ times.

Exercise 10.

(a) Since A is sorted, we claim that B is sorted where $B[i] = A[i] - i$. To see why, we note that

$$B[i] = A[i] - i \leq (A[i+1] - 1) - i = A[i+1] - (i+1) = B[i+1].$$

Therefore, B is non-decreasing. Secondly, $A[i] = i$ if and only if $B[i] = 0$. Therefore, we may apply binary search on B to find if such an index exists by checking whether $B[i] = 0$ for some index i. Note that we don't need to construct B, it is only necessary to compute $B[i]$ whenever we need.

For some index i to check, we perform the following:

- If $B[i] = 0$, we return that such an index exists.
- If $B[i] > 0$, then so is $B[j]$ for any $j \geq i$; therefore, we traverse on the bottom half of the array.
- If $B[i] < 0$, then so is $B[j]$ for any $j \leq i$; therefore, we traverse on the top half of the array.

If the binary search operation terminates without finding such an index, return that no such index is possible. This has running time $O(\log n)$.

(b) We claim that $A[i] = i$ for some i if and only if $A[1] = 1$. We prove both directions.

- ($\implies$) Suppose that $A[i] = i$ for some i. Let $B[i] = A[i] - i$. From the previous proof, we still see that

$$B[1] \leq B[2] \leq \cdots \leq B[n].$$

Now, since $A[1] > 0$, then $A[1] \geq 1$. If $A[1] > 1$, then $B[1] = A[1] - 1 > 0$ and so, $B[i] > 0$ for all i, which implies that $A[i] \neq i$ for all i. Therefore, $A[1] = 1$.

- ($\impliedby$) Suppose that $A[1] = 1$. Then clearly $A[i] = i$ for some i; just pick $i = 1$.

Such an algorithm has a constant-time running time since we are simply checking whether $A[1] = 1$.

Exercise 11.

(a) Notice that for the decision variant, we only care for each giant whether its height is at least T, or less than T: the actual value doesn't matter. Call a giant *eligible* if their height as at least T.

We sweep from left to right, taking the first eligible giant we can, then skipping the next K giants and repeating. We return `true` if the total number of giants we obtain from this process is at least L, or `false` otherwise. Each giant is processed in constant time, so the algorithm is clearly $O(n)$.

(b) Observe that the optimisation problem corresponds to finding the largest value of T for which the answer to the decision problem is `true`.

Suppose our decision algorithm returns `true` for some T. Then clearly it will return true for all smaller values of T as well: since every giant that is eligible for this T will also be eligible for smaller T. Hence, we can say that our decision problem is *monotonic* in T.

Thus, we can use binary search to work out the maximum value of T where our decision problem returns `true`. Note that it suffices to check only heights of giants as candidate answers: the answer won't change between them. Thus, we can sort our heights in $O(n \log n)$ and binary search over these values, deciding whether to go higher or lower based on a run of our decision problem. Since there are $O(\log n)$ iterations in the binary search, each taking $O(n)$ to resolve, our algorithm is $O(n \log n)$ overall.

Exercise 12. We run a binary search on A. For a value $x \in A$ to check, we compute $f(x)$.

- If $f(x) = 0$, then we terminate and return that A contains a value x for which $f(x) = 0$.
- If $f(x) > 0$, then we recurse on the top half (i.e. we look at all values $x_1 > x$ and recurse).
- If $f(x) < 0$, then we recurse on the bottom half (i.e. we look at all values $x_1 < x$ and recurse).

If the binary search terminates without finding a value, then we return that A does not have an element x for which $f(x) = 0$. The binary search operation takes $O(\log n)$ iterations and each check is constant-time. Therefore, the overall running time is $O(\log n)$.

Correctness
The correctness follows from the correctness of the binary search since A is sorted; all we need to show is that the halves that we recurse on are correct. For some fixed $x \in A$, suppose that $f(x) > 0$. Since f is a *decreasing* function, any element $x' < x$ in A has the property that $f(x') > f(x) > 0$. Therefore, it is enough to look at all values $x' > x$. Similarly, if $f(x) < 0$, any element $x' > x$ in A has the property that $f(x') < f(x) < 0$ and so, it is enough to look at all values $x' < x$. This proves that our recursive calls are correct and the rest of the algorithm follows from the correctness of binary search.

Exercise 13.

(a) Fix a value of x. Then $f_y(x, y) = 6y^5 + 4x^4y^3 + 2x^2y > 0$ whenever $y > 0$. Therefore, for positive inputs of y, we see that $f_y(x, y) > 0$ which shows that it is increasing in the input of y.

(b) From the above observation, we can perform a binary search by sorting B using merge sort and binary search for values of y. For each $x \in A$, binary search for a value of y such that $f(x, y) = 0$. If $f(x, y) > 0$, then we recurse on the top half of the array B; otherwise, we recurse on the bottom half of B. If we have exhausted all possible values of B, then there are no possible pairs (x, y) that satisfy the equation. This is $O(n \log n)$ since merge sort is $O(n \log n)$ in the worst case and we perform an $O(\log n)$ operation on n many values, giving us a final complexity of $O(n \log n)$.

Exercise 14.

(a) Construct the array $A = [1, \ldots, N]$ and run a binary search on A. For a value $x \in A$ to check, we compute $f(x)$.

- If $f(x) = 0$, then we return x because f is strictly increasing (i.e. $f(x) < f(x + 1)$ for each $x \in A$). If x is the only element left in the array, then we return x.
- If $f(x) < 0$, then we recurse on the top half (i.e. we look at all values $x_1 > x$ and recurse).
- If $f(x) > 0$, then we recurse on the bottom half (i.e. we look at all values $x_1 < x$ and recurse).

We can potentially speed up the process by first checking if $N < 0$. If $N < 0$, then we return that such an index does not exist. Otherwise, such an index must exist and our binary search will find the index.

To justify the time complexity of our algorithm, we run a binary search over an array of size N, which has $O(\log N)$ running time.

Correctness
The correctness follows from the correctness of the binary search. See the correctness of Exercise 12 and follow a similar line of reasoning; this time, you need to make an argument with the increasing nature of f.

(b) We follow a similar strategy even when we do not know N. However, there is a slight problem; there is no suitable range to binary search over. Instead, we need to artificially construct our range. We use the following strategy:

- Start by computing $f(1)$. If $f(1) \geq 0$, then we return 1.
- Otherwise, we compute $f(2)$. If $f(2) \geq 0$, then we return 2.
- Otherwise, we compute $f(4)$. If $f(4) \geq 0$, then we now have a suitable range $[2,4]$ since $f(2) < 0$ and $f(4) \geq 0$. Therefore, we can run a binary search over the array.
- In general, we keep computing $f(2^k)$ until we hit $f(2^k) \geq 0$ for the first time.
    - Note that we are guaranteed that such a k exists because f is *unbounded*.

The moment we hit such an integer 2^k for which $f(2^k) \geq 0$, we have a suitable range $A = [2^{k-1}, 2^{k-1}+1, \ldots, 2^k]$ to binary search over. The binary search will give us the smallest index i such that $f(i) \geq 0$.

Exercise 15. Let $g(i) = k - f(i)$. We claim that the smallest integer i for which $g(i) \geq 0$ is the smallest integer i for which $f(i) \leq k$. We prove both directions.

- ($\Longrightarrow$) Suppose that i is the smallest integer such that $g(i) \geq 0$. Then we have that
$$g(i-1) < 0 \leq g(i) \leq g(i+1).$$
By the definition of g, we have that
$$\begin{aligned} k - f(i-1) < 0 &\implies f(i-1) > k, \\ k - f(i) \geq 0 &\implies f(i) \leq k. \end{aligned}$$
Thus, i is the smallest integer such that $f(i) \leq k$.
- ($\Longleftarrow$) Suppose that i is the smallest integer such that $f(i) \leq k$. We can repeat the same argument to show that i is also the smallest integer such that $g(i) \geq 0$.

Therefore, we can run the previous algorithm on g where $g(i) = k - f(i)$.

Exercise 16. Consider $M(1,n)$ (i.e. top right cell).

- If $M(1,n) = x$, we are done.
- If $M(1,n) < x$, then we can safely ignore the top row and recurse on the board without the top row.
- If $M(1,n) > x$, then we can safely ignore the rightmost column and recurse on the board without the rightmost column.

We repeat this process until either there are no elements left to check, in which case x does not belong in A or we have found x. Since each iteration of the algorithm removes one row or column, the algorithm takes at most $2n$ iterations and each iteration performs a constant-time comparison. Therefore, the algorithm has $O(n)$ running time.

Correctness
We now prove the correctness of the algorithm. To do this, it suffices to argue that we can ignore the top row or the last column depending on the value of $M(1,n)$. The correctness of the rest of the algorithm follows.

- We first argue that if $M(1,n) < x$, then we can ignore the top row. Since each column is sorted in increasing order, we have that
$$M(1,1) < M(1,2) < \cdots < M(1,n) < x.$$
Therefore, none of the integers on the top row can contain x and so, it is safe to discard the top row.
- We now argue that if $M(1,n) > x$, then we can ignore the rightmost column. Since each row is sorted in increasing order, we have that
$$x < M(1,n) < M(2,n) < \cdots < M(n,n).$$
Therefore, none of the integers on the rightmost column can contain x and so, it is safe to discard the rightmost column.

We've shown that each time we iterate, we remove elements that are guaranteed to not contain x. Since we keep iterating until no elements are left to check, we virtually exhaust all possible elements and so, our algorithm will correct return whether M contains x.

Exercise 17. We start by sorting A in $O(n \log n)$ in ascending order, using MERGE SORT. Then, for each query, we can 2 binary searches to find the indexes (denote with (i,j)) of the:

- First element with value **no less** than L; and
- First element with value **strictly greater** than R.

The difference between these indices is the answer to the query, i.e., $j - i$. Note that if your binary search hits L you have to see if the preceding element is smaller than L; if it is also equal to L, you have to continue the binary search (going towards the smaller elements) until you find the first element equal to L. A similar observation applies if your binary search hits R.

Each binary search then takes $O(\log n)$ so for n queries in total, the algorithm runs in $O(n \log n)$ overall.

Exercise 18. This is done by a "double QUICKSORT" as follows. Pick a shoe and use it as a pivot to split the kids into three groups: those for whom the shoe was too large, those who fit the shoe and those for whom the shoe was too small. Then pick a kid for whom the shoe was a fit and let him try all the shoes, splitting them in three groups as well: shoes that are too small, shoes that fit him and the shoes which were too large for him. Continue this process with the first group of kids and first group of shoes and then also the third group of shoes with the third group of kids. If kids and shoes are picked randomly, the expected time complexity will be $O(n \log n)$.

Exercise 19.

(a) By definition, we have that
$$\begin{aligned} f(x_i) = k &\iff x_i \in b_k \\ &\iff \frac{k}{n} \le x_i < \frac{k+1}{n} \\ &\iff k \le nx_i < k+1 \\ &\iff k = \lfloor nx_i \rfloor. \end{aligned}$$
Therefore, $f(x_i) = \lfloor nx_i \rfloor$.

(b) The size of each bucket is $\frac{1}{n}$ since the interval $[0,1]$ is partitioned into n buckets of equal size. There are at most $n-1$ adjacent elements in a bucket, when all of the elements appear in the same bucket; therefore, we have that $m \le n-1$. This gives us

$$\begin{aligned}\sum_{i=2}^{m} |z_i - z_{i-1}| &\le \sum_{i=2}^{n-1} |z_i - z_{i-1}| \\ &\le \sum_{i=2}^{n-1} \frac{1}{n} \\ &= \frac{n-2}{n} \\ &< 1.\end{aligned}$$

(c) We start by splitting the interval $[0,1)$ into n equal buckets, namely

$$B = \{b_0, b_1, \dots, b_{n-1}\} = \left\{\left[0, \frac{1}{n}\right), \left[\frac{1}{n}, \frac{2}{n}\right), \dots, \left[\frac{n-1}{n}, 1\right)\right\}$$

Then we consider the function $b : \mathbb{R} \to \{0, \dots, n-1\}$ which computes $b(x) = k$ such that $x \in b_k$. Then we consider that the value x_i belongs to bucket number $b(x_i) = \lfloor nx_i \rfloor$. This was proven in the first part.

Therefore we can form n pairs $\langle x_i, b(x_i)\rangle$, each in constant time. Then we can now sort these pairs according to their bucket number $b(x_i)$; since all bucket numbers are less than n, CountingSort does that in linear time. One can show that this sequence already satisfies the condition of the problem, but to make things simpler we do another extra step. We go through the sequence and in each bucket we find the smallest and the largest element; this can clearly be done in linear time.

We now slightly change the ordering of each bucket: we always start with the smallest element in that bucket and finish with the largest element (leaving all other elements in the same order). Discard the bucket numbers, leaving only a sequence y_j of real numbers between 0 and 1, which are the original x_i rearranged.

We now prove that this sequence satisfies $\sum_{i=2}^{n} |y_i - y_{i-1}| < 2$. We start by splitting this sum into two parts:

$$\begin{aligned}\sum_{i=2}^{n} |y_i - y_{i-1}| = &\sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}\right) + \\ &\sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}\right).\end{aligned}$$

From part (b), we showed that

$$\sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}\right) < 1.$$

Also,

$$\sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}\right) \le 1$$

because each such pair joins the largest entry of one bucket to the smallest entry of the next non-empty bucket. Therefore, this sequence gives an ordering for which the sum of the absolute difference is at most 2, as required.

Exercise 20. There are many answers but generally, try to keep your constants as simple as possible!

(a) $n^2 \le n^3$ whenever $n \ge 1$, so we can choose $c = N = 1$.

(b) We see that $4n^3 + 8n^2 + 1 \le 4n^3 + 8n^3 + n^3 = 13n^3$ whenever $n \ge 1$, so we can choose $N = 1$ and $c = 13$.

(c) We see that $\sin(n) \leq 1$ for all $n \in \mathbb{N}$. Therefore, $n^2 + \sin(n) \leq n^2 + 1 \leq n^2 + n^2 = 2n^2$ whenever $n \geq 1$. Therefore, we can choose $N = 1$ and $c = 2$.

(d) Since $\cos(n), \sin(n) \leq 1$, we have that $\cos(n) + \sin(n) \leq 1 + 1 = 2$ for all $n \geq 1$. Therefore, we can choose $N = 1$ and $c = 2$.

Exercise 21.

(a) Note that $n^2 \leq n^3$ for all $n \geq 1$, so we can let $c = N = 1$.

(b) Note that $2^{n+1} = 2 \cdot 2^n$. So we can choose $c = 2$ and $N = 1$ to show that $2^n = O(2^{n+1})$. On the other hand, we can choose $c = 1/2$ and $N = 1$ to show that $2^n = \Omega(2^{n+1})$. Therefore, $2^n = \Theta(2^{n+1})$.

(c) Since $a < b$, we have that

$$a^n = \underbrace{a \cdot \ldots \cdot a}_{n \text{ times}} < \underbrace{b \cdot \ldots \cdot b}_{n \text{ times}} = b^n.$$

Therefore, we can choose $c = N = 1$.

(d) We first note that $|\cos(n)| \leq 1$ and $|\sin(n)| \leq 1$. Therefore,

$$\cos(n) + \sin(n) + 2 \leq 1 + 1 + 2 = 4.$$

So we can choose $N = 1$ and $c = 4$ to show that $\cos(n) + \sin(n) + 2 = O(1)$. On the other hand, we see that $\cos(n) + \sin(n) = \sqrt{2}\sin(n + \alpha)$ for some $\alpha \in \mathbb{R}$. Therefore, $|\cos(n) + \sin(n)| \leq \sqrt{2}$ and so, $\cos(n) + \sin(n) \geq -\sqrt{2}$, which implies that $\cos(n) + \sin(n) + 2 \geq 2 - \sqrt{2}$. Therefore, a lower bound is $2 - \sqrt{2}$ and so, we can choose $c = 2 - \sqrt{2}$ with $N = 1$. This proves that $\cos(n) + \sin(n) + 2 = \Theta(1)$.

Exercise 22.

(a) We see that

$$\begin{aligned} g(n) &= \log_2 n \cdot \log_2 n + 2\log_2 n \\ &= \Theta((\log_2 n)^2) = \Theta(f(n)), \end{aligned}$$

since $2\log_2 n < (\log_2 n)^2$ for sufficiently large n.

(b) We show that $f(n) = O(g(n))$. To do this, we want to find some constants $c, N > 0$ such that, for every $n > N$, $f(n) < c \cdot g(n)$. Since log is a monotonically increasing function, $\log f(n) < \log g(n)$ immediately implies that $f(n) < g(n)$. We see that

$$\begin{aligned} \log_2 f(n) &= \log_2\left(n^{100}\right) \\ &= 100\log_2 n, \\ \log_2 g(n) &= \log_2\left(2^{n/100}\right) \\ &= \frac{n}{100}. \end{aligned}$$

It therefore suffices to show that, for sufficiently large n, $10000\log_2 n < n$. We see that

$$\lim_{n\to\infty} \frac{10000\log_2 n}{n} = \lim_{n\to\infty} \frac{10000}{n\log 2} = 0,$$

by L'Hôpital's rule. In other words, there exist some $N > 1$ such that, for all $n > N$, $10000\log_2 n/n < 1$ and thus, $\log_2 f(n) < \log_2 g(n)$ which implies that $f(n) < g(n)$ and so, $f(n) = O(g(n))$.

(c) We show that $f(n) = \Omega(g(n))$. This amounts to showing that $\sqrt{n} > c \cdot 2^{\sqrt{\log_2 n}}$ for some $c > 0$ and for all sufficiently large n. Since log is monotonically increasing, this is equivalent to showing that

$$\log_2 \sqrt{n} = \frac{1}{2}\log_2 n > \log_2 c + \sqrt{\log_2 n} = \log_2\left(c \cdot 2^{\sqrt{\log_2 n}}\right).$$

Taking $c = 1$, it is clear that $\log_2 n$ grows asymptotically faster than $\sqrt{\log_2 n}$. Hence, $\log_2 \sqrt{n} > \log_2\left(2^{\sqrt{\log_2 n}}\right)$ which implies that $f(n) = \Omega(g(n))$.

(d) We again wish to show that $n^{1.001} = \Omega(n \log n)$, i.e., that $n^{1.001} > cn \log n$ for some c and all sufficiently large n. Since $n > 0$ we can divide both sides by n, so we have to show that $n^{0.001} > c \log n$. We again take $c = 1$ and show that $n^{0.001} > \log n$ for all sufficiently large n, which is equivalent to showing that $\log n / n^{0.001} < 1$ for sufficiently large n. To this end we use the L'Hôpital's to compute the limit

$$\lim_{n\to\infty} \frac{\log n}{n^{0.001}} = \lim_{n\to\infty} \frac{(\log n)'}{(n^{0.001})'} = \lim_{n\to\infty} \frac{\frac{1}{n}}{0.001 n^{0.001-1}}$$
$$= \lim_{n\to\infty} \frac{\frac{1}{n}}{0.001\frac{1}{n} \cdot n^{0.001}} = \lim_{n\to\infty} \frac{1}{0.001 \cdot n^{0.001}} = 0.$$

Since $\lim_{n\to\infty} \frac{\log n}{n^{0.001}} = 0$ then, for sufficiently large n we will have $\frac{\log n}{n^{0.001}} < 1$.

(e) Just note that $(1 + \sin \pi n/2)/2$ cycles, with one period equal to $\{1/2, 1, 1/2, 0\}$. Thus, for all $n = 4k + 1$ we have $(1 + \sin \pi n/2)/2 = 1$ and for all $n = 4k + 3$ we have $(1 + \sin \pi n/2)/2 = 0$. Thus for any fixed constant $c > 0$ for all $n = 4k + 1$ eventually $n^{(1+\sin \pi n/2)/2} = n > c\sqrt{n}$, and for all $n = 4k + 3$ we have $n^{(1+\sin \pi n/2)/2} = n^0 = 1$ and so $n^{(1+\sin \pi n/2)/2} = 1 < c\sqrt{n}$. Thus, neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$.

Exercise 23. Suppose that $f(n) = O(g(n))$. Then there exist constants $c, n_0 > 0$ such that

$$f(n) \le c \cdot g(n),$$

for all $n \ge n_0$. Then we have that

$$\frac{1}{c} f(n) \le g(n),$$

for all $n \ge n_0$. Therefore, choose $d = 1/c$, which implies that $g(n) = \Omega(f(n))$. Now, suppose that $g(n) = \Omega(f(n))$. Then there exist constants $c, n_0 > 0$ such that

$$c \cdot f(n) \le g(n).$$

But again, this implies that

$$f(n) \le \frac{1}{c} \cdot g(n).$$

Choosing $d = 1/c$ again implies that $f(n) = O(g(n))$. Therefore, $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

Exercise 24.

(a) This is equivalent to finding constants $c', n_0 > 0$ such that

$$c \cdot f(n) \le c' \cdot f(n),$$

for all $n > n_0$. Choose $c' = c$ and $n_0 = 1$. Then $c \cdot f(n)$ is identically $c' \cdot f(n)$ and the inequality also holds since equality is always true.

(b) Suppose that $f(n) = O(g(n))$ and $g(n) = O(h(n))$. Since $f(n) = O(g(n))$, there exist constants $c_0, n_0 > 0$ such that

$$f(n) \le c_0 \cdot g(n),$$

for all $n > n_0$. Similarly, since $g(n) = O(h(n))$, then there exist constants $c_1, n_1 > 0$ such that

$$g(n) \leq c_1 \cdot h(n),$$

for all $n > n_1$. But this implies that

$$f(n) \leq c_0 \cdot g(n) \leq c_0 \left(c_1 \cdot h(n)\right),$$

where this inequality holds whenever $n > \max\{n_0, n_1\}$. Choosing $c = c_0 \cdot c_1$ and $N = \max\{n_0, n_1\}$, this shows that $f(n) = O(h(n))$.

(c) Suppose that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Using a similar argument as above, we have constants $c_1, c_2, n_1, n_2 > 0$ such that

$$f_1(n) \leq c_1 \cdot g_1(n),\ f_2(n) \leq c_2 \cdot g_2(n),$$

for all $n > \max\{n_1, n_2\}$. Under this domain, notice that

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \leq \max\{c_1, c_2\} \left(g_1(n) + g_2(n)\right),$$

which gives us the constants and domain for which this inequality holds, proving that $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$.

(d) We first show that $O(f(n) + g(n)) \subseteq O(\max\{f(n), g(n)\})$ and then show the reverse subset inclusion.

Suppose that $h(n) = O(f(n) + g(n))$. Then there exist $c, n_0 > 0$ such that $h(n) \leq c(f(n) + g(n))$. But note that this implies that

$$h(n) \leq c \cdot f(n) + c \cdot g(n) \leq c \cdot \max\{f(n), g(n)\},$$

which implies that $h(n) = O(\max\{f(n), g(n)\})$. Therefore, $O(f(n) + g(n)) \subseteq O(\max\{f(n), g(n)\})$

Now suppose that $h(n) = O(\max\{f(n), g(n)\})$. Then there exist constants $c, n_0 > 0$ such that

$$h(n) \leq c \cdot \max\{f(n), g(n)\}.$$

But note that this implies that

$$h(n) \leq c \cdot \max\{f(n), g(n)\} + \min\{f(n), g(n)\} = c(f(n) + g(n)),$$

which implies that $h(n) = O(f(n) + g(n))$. This proves both results.

Exercise 25. We say that $\lim_{n\to\infty} f(n) = L$ if, for each $\epsilon > 0$, there exist some $M(\epsilon) > 0$ such that

$$n > M(\epsilon) \implies |f(n) - L| < \epsilon.$$

If $L = \infty$, then we can tweak this definition to the following; we say that $\lim_{n\to\infty} f(n) = \infty$ if, for each $\epsilon > 0$, there exist some $M(\epsilon) > 0$ such that

$$n > M(\epsilon) \implies |f(n)| > \epsilon.$$

We are ready to use these definitions to prove our results.

(a) Suppose that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Then, for each $\epsilon > 0$, we can always find some $M(\epsilon) > 0$ such that $\left|\frac{f(n)}{g(n)}\right| < \epsilon$. Since $f(n)$ and $g(n)$ are positive functions, we can remove the absolute values, so that

$$\frac{f(n)}{g(n)} < \epsilon \implies f(n) < \epsilon \cdot g(n),$$

whenever $n > M(\epsilon)$. This proves that $f(n) = O(g(n))$ by choosing $c = \epsilon$ and $N = M(\epsilon)$.

(b) Now suppose that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Then, for each $\epsilon > 0$, we can always find some $M(\epsilon) > 0$ such that $\left|\frac{f(n)}{g(n)}\right| > \epsilon$. Again, since $f(n)$ and $g(n)$ are positive functions, we can remove the absolute values, so that

$$\frac{f(n)}{g(n)} > \epsilon \implies f(n) > \epsilon \cdot g(n) \implies g(n) < \frac{1}{\epsilon} \cdot f(n),$$

whenever $n > M(\epsilon)$. This prove that $f(n) = \Omega(g(n))$ by choosing $c = 1/\epsilon$ and $N = M(\epsilon)$.

Exercise 26.

(a) We can rewrite $h(n)$ as

$$h(n) = e^{\ln(2^n)} = e^{n \ln 2}.$$

Computing the derivative, we have

$$h'(n) = \ln 2 \cdot e^{n \ln 2} = 2^n \cdot \ln 2.$$

Since $\ln 2 > \ln 1 > 0$ and the fact that $2^n > 0$ for all $n > 0$, we have that $h'(n) > 0$ for all n. Therefore, $h(n)$ is monotonic in n.

(b) We now prove that $f(n) = O(g(n))$. But this is clear since we have that

$$f(n) = 2^{n+1} = 2 \cdot 2^n = 2g(n),$$

for all n. Therefore, choosing $c = 2$ and $n = 1$ shows that $f(n) = O(g(n))$.

(c) Consider $h(f(n))$ and $h(g(n))$. We have

$$h(f(n)) = 2^{2^{n+1}}, \ h(g(n)) = 2^{2^n}.$$

However, it is easy to see that

$$\frac{h(f(n))}{h(g(n))} = \frac{2^{2^{n+1}}}{2^{2^n}} \to \infty,$$

as $n \to \infty$ which means that $h(f(n)) \neq O(h(g(n))$.

Exercise 27.

(a) We show that

$$n^{\log n} \ll 2^n \ll (\log n)^n,$$

where $f(n) \ll g(n)$ is denoted to mean that $f(n) = O(g(n))$.

We prove the first half of the inequality; that is, $n^{\log n} \ll 2^n$. To see this, we can rewrite both expressions as

$$n^{\log n} = e^{\log\left(n^{\log n}\right)} = e^{(\log n)^2}, \quad 2^n = e^{\log(2^n)} = e^{n \log 2}.$$

To determine the order of the growth, we compare $(\log n)^2$ and $n \log 2$. By L'Hôpital's Rule, we have that

$$\lim_{n\to\infty} \frac{(\log n)^2}{n \log 2} = \lim_{n\to\infty} \frac{2 \log n}{n \log 2}.$$

Since $\log n < n$ for all $n > 1$, the limit is precisely 0. Thus, there exist some $N > 1$ such that $(\log n)^2 < n \log 2$ for all $n > N$. In other words, we have that

$$e^{(\log n)^2} \ll e^{n \log 2} \implies n^{\log n} \ll 2^n.$$

Now, for sufficiently large N, $\log N > 2$. So, for all $n > N$, $2^n \ll (\log n)^n$. From these two results, it follows that

$$f(n) = O(g(n)).$$

(b) We observe that $g(n) = \tan(n)$ is π-periodic and passes through $g(n) = 0$ infinitely many times. Thus, there is no value of N such that, for *every* $n > N$, $g(n) > c \cdot f(n)$ or $g(n) < c \cdot f(n)$. In other words, it is neither $O(g(n))$ nor $\Omega(g(n))$.

(c) Recall that the Taylor series expansion of $h(x) = e^x$ is

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Thus, we have that

$$\sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!} = \sum_{k=0}^{\infty} \frac{4^n \left(4^k n^k\right)}{k!} = 4^n \sum_{k=0}^{\infty} \frac{(4n)^k}{k!} = 4^n e^{4n}.$$

Then

$$\begin{aligned} g(n) &= \left[\log\left(\sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!}\right)\right]^2 \\ &= \left(\log\left(4^n e^{4n}\right)\right)^2 \\ &= \left(\log(4^n) + \log(e^{4n})\right)^2 \\ &= (n \log 4 + 4n)^2 \\ &= c \cdot n^2 \\ &= O(n^{5/2}). \end{aligned}$$

In other words, $f(n) = \Omega(g(n))$.