COMP3121/9101
Algorithm Design

PRACTICE PROBLEM SET 1 – DATA STRUCTURES REVISION

[**K**] – key questions [**H**] – harder questions [**E**] – extended questions [**X**] – beyond the scope of this course

Contents

§ SECTION ONE: DATA STRUCTURES AND ALGORITHMS

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

Solution. To design an algorithm that finds the missing integer, note that the sum of all integers from 1 to n is given by

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

Moreover, the missing integer (say x) can be found by noting that

$$\begin{aligned} x &= 1+2+\cdots+(x-1)+x+(x+1)+\cdots+n \\ &\quad -(1+2+\cdots+(x-1)+(x+1)+\cdots+n) \\ &= \frac{n(n+1)}{2}-\sum_{i=1}^{n-1}A[i]. \end{aligned}$$

Therefore, our algorithm first computes $n(n+1)/2$ in constant time and performs a running sum to compute $\sum_{i=1}^{n-1}A[i]$ in linear time. The missing integer, therefore, is

$$x=\frac{n(n+1)}{2}-\sum_{i=1}^{n-1}A[i].$$

The overall algorithm runs in $O(n)$ time since computing the sum of the elements in A takes linear time. □

[K] Exercise 2. Let A be an array with n elements. We say that A 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 $B=[1,2,3,4,2,1]$ is not. Design an $O(n)$ algorithm that determines whether A is a palindromic array.

Solution. 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. □

[K] Exercise 3. Let A be an array with n distinct integers. You have to determine if there exist an integer (not necessarily in A) which can be written as a sum of squares of two distinct integers from A in two different ways. Note that $A[i]^2+A[j]^2$ is treated the same as $A[j]^2+A[i]^2$.

For example, if $A=[1,8,9,12]$, then the integer 145 can be written as 1^2+12^2 *and* also 8^2+9^2.

(a) Design an $O(n^2\log n)$ algorithm that determines if such an integer exists in the *worst case*.

(b) Design an algorithm that solves the same problem and runs in $O(n^2)$ in the *expected case*.

Solution.

(a) There are $\binom{n}{2} = n(n-1)/2$ pairs of indices $1 \leq i < j \leq n$. Now, for each such pairs, we compute the sum of the squares $A[i]^2 + A[j]^2$ and store it in an array B of size $n(n-1)/2$. Note that B now consists of all possible sums of squares of elements in A. The problem, therefore, reduces to the problem of determining whether there is a duplicate value in B. We can do this by first sorting B with merge sort, and then perform a linear scan for duplicate values; note that duplicate values must appear consecutively.

To argue the time complexity, we note that there are $n(n-1)/2 = O(n^2)$ many pairs of elements in A. Since B consists of $O(n^2)$ many elements, sorting B takes $O(n^2 \log(n^2))$. But, by our log properties, $n^2 \log(n^2) = 2n^2 \log n = O(n^2 \log n$. Therefore, sorting B takes $O(n^2 \log n)$ running time. Finally, the linear scan takes $O(n^2)$ running time which implies that the overall running time of our algorithm is $O(n^2 \log n)$.

(b) As above, compute each sum $A[k]^2 + A[m]^2$, but instead store the sums in a hash table. Before adding a sum to the hash table, we check whether the same sum already appears there, if so, report `yes`.

For each of $O(n^2)$ pairs of indices, we perform expected $O(1)$ work, so the overall time complexity is expected $O(n^2)$.

□

[K] Exercise 4. Let A be an array with n integers, and let k be a positive integer. You have 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 i, j such that $|A[i] - A[j]| = k$.

(a) Design an $O(n \log n)$ algorithm that determines if two such integers in A exist.

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

Solution.

(a) The idea is that we can check for a pair of elements whose difference of k first, and then independently check for a pair of elements whose difference is $-k$. To do this efficiently, start by sorting A using merge sort which gives $O(n \log n)$ time for sorting. Then, with two pointers, have one pointer for the bigger element and one pointer for the smaller element. Let i be the index of the larger value and j be the index of the smaller value.

Given the two indices, check for their difference and see if the difference is equal to k. If it is equal to k, then there is nothing to do. Otherwise, either $A[i] - A[j] > k$ or $A[i] - A[j] < k$. We consider each case separately.

- **Case 1**: $A[i] - A[j] > k$. In this case, we observe that any value larger than $A[j]$ will also not work. To see this, suppose that $A[j] < A[j']$. Then

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

Therefore, we only need to consider all indices $j' < j$.

- **Case 2**: $A[i] - A[j] < k$. In this case, we observe that any value smaller than $A[i]$ will also not work. To see this, suppose that $A[i] > A[i']$. Then

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

Therefore, we only need to consider all indices $i' > i$, which reduces the search space for i under this constraint.

With these observations in mind, we now design our general algorithm. From sorting A in $O(n \log n)$ with merge sort, we now consider two indices j, i that points to the first two elements in the array respectively.

- If $A[i] - A[j] = k$, then we terminate and return that there are such pairs in A.

- If $A[i] - A[j] > k$, then we are in case 1 and by our observation above, we decrease the pointer for j to consider indices smaller than j.
- If $A[i] - A[j] < k$, then we are in case 2 and by our observation above, we increase the pointer for i to consider indices larger than i.

If we exhaust all possible indices without terminating early, then there is no such pair whose difference is k. We repeat this same process except replace k with $-k$. If we exhaust all possible indices without terminating early with difference $-k$ as well, then there is no pair of points whose absolute difference is k, in which case we return without a match.

Note that both parts of the algorithm perform exactly the same operations. Since we are doing a linear scan through the entire array once, checking for a match takes running time $O(n)$ in the worst case, which is dominated by the running time of the running time of merge sort. Since we are doing this algorithm twice, the overall running time is $O(2n \log n) = O(n \log n)$.

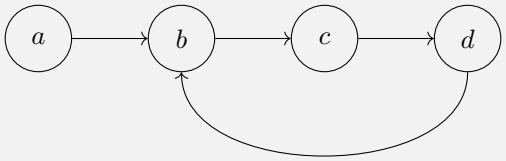
(b) We create a hash table H, which is initially empty. Now, for each element a in A, we first perform a check to see whether a is already hashed in H. If it hasn't been hashed before, then we hash $a + k$ and $a - k$ into H. Otherwise, we must have hashed a previous element from A into a. But by the 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.

□

[K] Exercise 5. Let L be a linked list with n nodes. We say that L is *cyclic* if there is a node that can be reached again by continuously following the next node.

For example, the linked list visualised as



is cyclic because node b can be reached again from d. Design an $O(n)$ algorithm that determines whether a linked list is cyclic.

Solution. The solution is useful to know since it introduces an algorithm called *Floyd's Cycle Detection algorithm*, also colloquially named the *tortoise and hare algorithm*, which can be used any time a problem asks to detect whether a cycle is present.

We first present the algorithm in its entirety and then prove why it works. The *tortoise and hare algorithm* is a prototypical two pointer algorithm, where one pointer moves in one step and the other pointer moves in two steps. Therefore, the algorithm is as follows: we initialise the tortoise and hare pointers to the head. Now, while the hare pointer isn't at the end of the linked list, we iterate through the entire list in two different speeds:

At every iteration,

- Move the tortoise pointer by one space (i.e. $i \mapsto i + 1$);
- Move the hare pointer by two spaces (i.e. $i \mapsto i + 2$).

If the two pointers ever meet in the same place, then there must have been a cycle in the linked list and our algorithm returns that the linked list is cyclic. Otherwise, if we reach the end of the linked list and the two pointers never met, then there must have been no cycle in the linked list.

To see why this algorithm is correct, we need to prove two things; the first is to prove that, *if* a linked list contains a cycle, then the hare and tortoise pointers will eventually meet. The second is to prove that, *if* a linked list does not contain a cycle, then the hare and tortoise pointers will never meet. We prove these separately.

- Suppose that the linked list does contain a cycle. Denote the length of the cycle as C and suppose that the number of nodes before arriving at the start of a cycle is T. Firstly, we can write $T = kC + r$ for some integer k and some $0 \leq r < C$. This allows us to track where the hare's pointer is when the tortoise pointer finally reaches the beginning of the cycle. When the tortoise is at the beginning of the cycle (at node T), the hare will be at node $2T = 2(kC + r)$. However, the first T steps will be in the nodes leading up to the cycle. Therefore, in the cycle, the hare will have moved T steps. This implies that the hare will have moved around the cycle k times, leaving the hare on node r inside the cycle.

  We now have a measure of where the tortoise and hare are in the cycle. We now observe that, after $C - r$ more steps, the tortoise will be on node $C - r$ in the cycle (think of entering the cycle as node 0 now). Since the hare was r moves ahead of the tortoise when the tortoise is inside the cycle, the hare must be on node $r + 2(C - r)$ since the hare was an r number of steps ahead of the tortoise and then we had $C - r$ many more iterations of the algorithm. But this implies that the hare is on node $r + 2C - 2r = 2C - r$. This is equivalent to stepping around the cycle once and ending back at node $C - r$. But this implies that the hare and the tortoise are both on node $C - r$ *in* the cycle, which implies that both pointers met at node $C - r$. This completes the proof that a cycle inside a linked list leads to both pointers meeting at some point inside the cycle.

- Now suppose that the linked list does not contain a cycle. Then the linked list consists of paths, but this implies that the index of the hare will never be any smaller than the index of the tortoise. This shows that the hare will never meet the tortoise, which completes the proof.

This shows that the hare and tortoise will meet at the same node if and only if the linked list contains a cycle. This proves the correctness of the algorithm. □

[H] Exercise 6. 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 known by everyone, but who does not know anyone else present. Your task is to work out if there is a celebrity present, and if so, which of the n people present is a celebrity. To do so, you can ask a person X if they know another person Y (where you choose X and Y when asking the question).

(a) Show that there can be at most one celebrity. In other words, *if* a celebrity exists, then the celebrity is unique.

(b) Use the previous part to show that your task can always be accomplished by asking no more than $3n - 3$ such questions.

(c) Show that your task can always be accomplished by asking no more than $3n - \lfloor \log_2 n \rfloor - 3$ such questions.

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

(a) We make the following observation.

Observation. *There can be at most one celebrity.*

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. With this observation, 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.

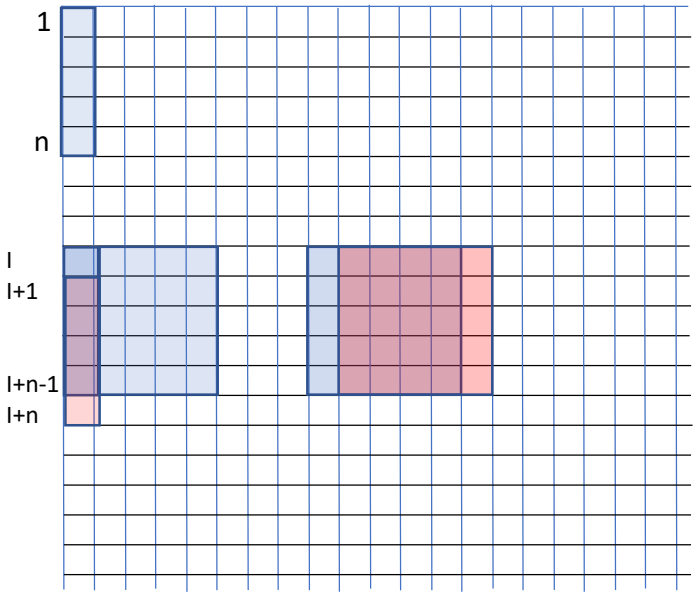
We only conclude that C must be the only celebrity once we have concluded that C does not know anyone. In the worst case, we consume $n-1$ questions to determine who the potential celebrity candidate is and then $2(n-1)$ questions to verify whether C is indeed the celebrity. Thus, in the worst case,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

(b) 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.

□

[H] Exercise 7. You are in a square orchard of $4n$ by $4n$ equally spaced trees. You want to purchase apples from precisely n^2 many of those trees, which also form a square. Fortunately, the owner is allowing you to choose such a square anywhere in the orchard and you have a map with the number of apples on each tree. Your task is to choose a square that contains the largest amount of apples and which runs in time $O(n^2)$.

Solution. We start by noting that there is a heavy overlap between such possible squares and we should use this fact to compute the number of apples in all of such squares in an efficient way. Consider to the figure below.



Setup: Let us visualize the orchard as a $4n \times 4n$ discrete grid and let $C(i, j)$ denote the cell at the coordinate (i, j). Also let $A[i, j]$ denote the amount of apples at $C[i, j]$.

Step 1: Now, we start by examining the first column by computing the sum $\alpha(1, 1) = \sum_{k=1}^{n} A[k, 1]$, (corresponding to cells $C[1, 1]$ to $C[n, 1]$ shown in blue in the top left corner of the orchard map) which takes $n - 1 = O(n)$ additions. We then compute the number of apples $\alpha(i, 1)$ in all rectangles $r(i, 1)$ consisting of cells $C[i, 1]$ to $C[i + n - 1, 1]$ for all i such that $2 \leq i \leq 3n + 1$, starting from $\alpha(1, 1)$ and using recurrence

$$\alpha(i + 1, 1) = \alpha(i, 1) - A[i, 1] + A[i + n, 1].$$

We know that the recurrence is valid as the rectangle (denoted with $r(i, 1)$) consisting of cells $C[i, 1]$ to $C[i + n - 1, 1]$ and rectangle $r(i + 1, 1)$ consisting of cells $C[i + 1, 1]$ to $C[i + n, 1]$ in the first column overlap and differ only in the first square of $r(i)$ and the last square of $r(i + 1)$(this idea is similar to Question 1 but embedded 2 dimensions). Since each recursion step involves only one addition and one subtraction, this can all be done in $O(n)$ steps.

Step 2: We can now apply the same procedure to each different column j, thus obtaining the number of apples $\alpha(i, j)$ in every rectangle consisting of cells $C(i, j)$ to $C(i + n - 1, j)$. As each column requires $O(n)$ many computations, it takes $O(n^2)$ many computations in total.

Step 3: Now for $1 \leq i \leq 3n + 1$, we compute $\beta(i, 1) = \sum_{p=1}^{n} \alpha(i, p)$. This gives the total number of apples acquired in the square with corner vertices at cells $C(i, 1), C(i, n), C(i + n - 1, 1)$ and $C(i + n - 1, n)$ (square with vertices along a single column). For n such computations it takes $O(n)$ additions each, thus the total number of additions then runs in $O(n^2)$.

Step 4: So for each fixed i such that$1 \leq i \leq 3n + 1$, with the value of $\beta(i, 1)$, we can now compute $\beta(i, j)$ for all $2 \leq j \leq 3n + 1$ using recursion

$$\beta(i, j + 1) = \beta(i, j) - \alpha(i, j) + \alpha(i, j + n).$$

because the two adjacent squares overlap, except for the first column of the first square and the last column of the second square (overlap between red and blue square), the rest of the considered squares overlap. Each step of recursion takes only one addition and one subtraction so the whole recursion takes $O(n)$ many steps. Thus, recursions for all $1 \leq i \leq 3n + 1$ takes $O(n^2)$ many operations. (idea is again similar to Question 1 but in 2 dimensions).

Step 5: Lastly,we only require to find the largest value of $\beta(i, j)$ among all $1 \leq i, j \leq 3n + 1$, giving a complexity of $O((3n)^2) = O(n^2)$ for finding the maximum.

As all required steps of the algorithm runs in a complexity of $O(n^2)$, the total complexity of the algorithm is then $O(n^2)$ as required. □

[E] Exercise 8. 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.

> Given your n friends, how many questions do you have to at least ask each friend? Can we generate a strategy just asking these questions?

Solution. Suppose your friends are numbered 1 to n and the teams are also numbered 1 to n. Then, for each i, ask friend i if they support team i. If they do, we choose not to support them and if they don't, we do support them. To see why this is different to all of our friends, if it were the same as *some* friend, then, in each index, our lists must agree. However, when we ask friend i whether they support team i, our algorithm forces us to choose a different answer to theirs, which means it cannot possibly agree for every team. Therefore, our subset of teams that we support cannot possibly align with all of the choices that any other friend makes. This also uses n queries which is minimal since we require some information about *every* possible friend. □

[E] Exercise 9. You are conducting an election among a class of n students. Each student casts precisely one vote by writing their name and that of their chosen classmate on a single piece of paper. However, the students have forgotten to specify the order of names on each piece of paper; for example, "Alice Bob" could mean that *Alice voted for Bob* or *Bob voted for Alice*.

(a) Show how you can still uniquely determine how many votes each student received.

(b) Hence, explain how you can determine which students did not receive any votes. Can you also determine who these people voted for?

(c) Suppose now that every student received at least one vote. Show that each student received *exactly* one vote.

(d) Using the previous two parts, design an $O(n)$ algorithm that constructs a list of votes of the form "X voted for Y" consistent with the pieces of paper. Specifically, each piece of paper should match up with precisely one of these votes. If multiple such lists exist, produce any.

> First, use **(c)** to consider how you would solve it in the case where every student received at least one vote. Then apply **(b)**.

Solution. We make the following observation.

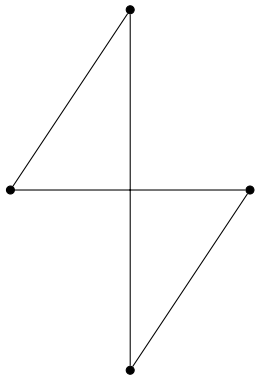
Observation. Every student has their name on *at least* one paper. To see this, note that every student casts a vote by writing two names. One name signifies that they are the voter and the other being who they voted for. Therefore, because every student votes, their name must appear on *at least* one paper.

(a) To see how we can uniquely determine how many votes each student receives, we make use of the assumption that every student casted precisely one vote. Therefore, if their name appeared on x many papers, precisely one of these papers must have been the paper that they used to vote on. In other words, they have received $x - 1$ many votes.

(b) Using the previous part, if they received 0 votes and the observation, we deduce that they must have appeared on precisely 1 paper (the paper that they voted on!). The person that they voted is precisely the other name on the single paper.

(c) If every student received at least one vote, then we require *at least* n distinct pieces of papers which correspond to the distinct votes. However, there are *at most* n papers because there are maximally n students, each of whom cast a vote. Therefore, every student can receive *at most* 1 vote. In other words, if every student received at least one vote, then they receive *exactly* 1 vote.

(d) We begin with the case explored in part (c). Suppose that every student received at least one vote. Then we deduced that they received *exactly* 1 vote. We can represent the following scenario as an undirected graph, where each of the vertices represent a student and each edge represents a vote. That is, x and y are connected if either x voted for y or y voted for x.



Above is an example of such a graph. One key property is that each student must appear on *exactly* two papers; these correspond to the edges of any particular vertex and correspondingly, it is easy to see that any such graph can be partitioned into a disjoint union of cycle subgraphs.

Pick any student s appearing on two pieces of paper, and arbitrarily choose one of their pieces of paper as their vote. Suppose they voted for t. We are now left with a single choice for t's vote. We can repeatedly follow these pieces of paper until we arrive back to s. We then repeat with another student appearing on two pieces of paper until all votes have been resolved. We can do this in $O(n)$ altogether, for instance, using a (simplified) Depth-First Search (DFS).

Now we combine this with part (b) to obtain an algorithm for the general case. We repeatedly check if a student has no votes (by counting votes) and resolve their vote. Once we reach a point where this is no longer possible, we know every student received at least one vote, and use the algorithm above.

This can be done in $O(n^2)$ by repeatedly taking $O(n)$ to identify a student who has no votes, or more cleverly in $O(n)$ as follows. We keep a count, for each student, how many pieces of paper they appear on and maintain a queue of students who appear on only one piece of paper. We can initially populate this queue in $O(n)$. Then, we repeatedly process the front student of the queue by removing their vote. Note that this *only changes the vote count of the person they voted for*, so we simply decrease their count. If their count reaches 1, we push them onto the queue. Hence, we process each student, updating counts and the queue in $O(1)$ so this step is $O(n)$ as well, giving an $O(n)$ algorithm.

□

§ SECTION TWO: SEARCHING AND SORTING ALGORITHMS

[K] Exercise 10.

(a) Assume you have an array of $2n$ distinct integers. Find the largest and the smallest number using at most $3n-2$ comparisons.

(b) Assume you have an array of 2^n distinct integers. Find the largest and second largest number using at most 2^n+n-2 comparisons.

Solution.

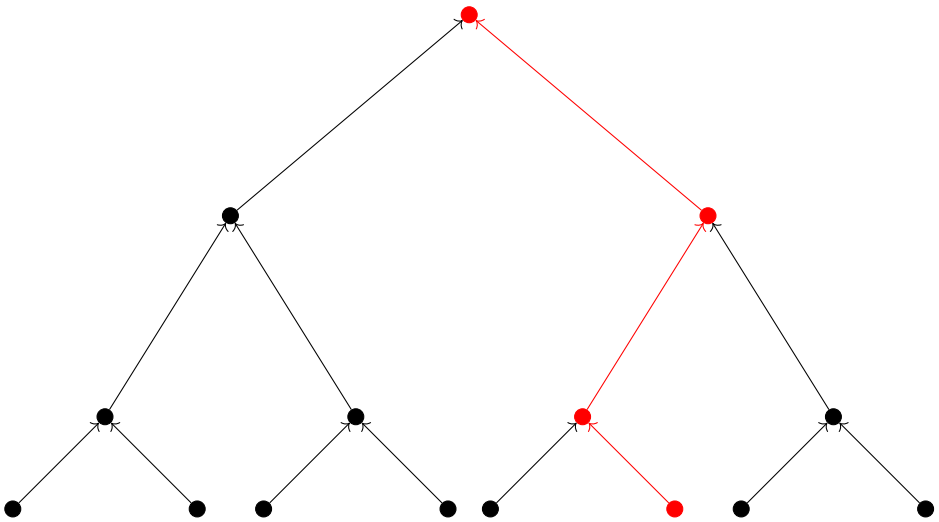
(a) Consider first the brute force algorithm:

- Compare the first two elements $A[1]$ and $A[2]$, and set m as the smaller of them and M as the larger.
- For each of the following $2n-2$ elements, compare it with both m and M, updating either if necessary.

In the best case, we will update m at each step, making it unnecessary to ever compare against M, and resulting in a total of $2n-1$ comparisons. However, this algorithm is not acceptable because in the worst case, each of $2n-2$ elements requires two comparisons, for a total of $2(2n-2)+1 = 4n-3$ comparisons.

Instead, we first form n pairs and compare the two elements of each pair, putting the smaller into a new array S and the larger into a new array L. Note that the smallest of all $2n$ elements must be in S and the largest in L, and we have made n comparisons. We now use two linear searches for the minimum of S and the maximum of L, each taking $n-1$ comparisons. In total this takes $n+n-1+n-1 = 3n-2$ comparisons.

(b) 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.

□

[K] Exercise 11. You are given an array A of n integers and an integer x.

(a) Design an $O(n \log n)$ algorithm that determines whether or not there exist two integers in A that sum to x.

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

Solution.

(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.

□

[K] Exercise 12. Let $f : \mathbb{N} \to \mathbb{Z}$ be a monotonically increasing function. That is, for all $i \in \mathbb{N}$, we have that $f(i) < f(i + 1)$. Our goal is to find the smallest value of $i \in \mathbb{N}$ so that $f(i) \geq 0$. Design an $O(\log n)$ algorithm to find the value of i so that $f(i) \geq 0$ for the first time.

How could you use binary search here?

Solution. This is similar to the problem 2 of tutorial 1.

If we wanted to apply binary search here, we need some sort of range of values to look through; however, we don't have a suitable range to binary search over. We can, instead, construct a more dynamic range of values. To see this in action, we can start by computing the first two values. Computing $f(1)$ and $f(2)$ gives us two values to work with. If $f(1) < 0 \leq f(2)$, then we are done. Similarly, if $0 < f(1) < f(2)$, then we are also done. Therefore, we may assume that $f(1) < f(2) < 0$. For any subsequent iteration, we compute the function at the next power of two, say 2^k. If $f(2^k) < 0$, then we compute $f(2^{k+1})$. However, if $f(2^k) \geq 0$, then we know that the first index in which the function is non-negative must lie in the range $[2^{k-1}, \ldots, 2^k]$. Since the function is monotonically increasing, this allows us to perform a binary search.

Since we're computing the function at powers of two, the number of times we are performing our computation is $\log_2 n$, where $n = 2^k$ since we are performing this over k iterations. Now, since we're binary searching over the interval $[2^{k-1}, \ldots, 2^k]$, the length of our array is $2^k - 2^{k-1} = 2^{k-1}$. Therefore, the running time of our binary search is $\log(2^{k-1}) = k - 1 < k = \log n$. Therefore, the overall running time of our algorithm is $O(\log n)$. □

[K] Exercise 13. Let M be an $n \times n$ matrix of distinct integers $M(i, j)$ where $1 \leq i, j \leq n$. Each row and each column of the matrix is sorted in increasing order, so that for each row i,

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

and for each column j,

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

Design an $O(n)$ algorithm that, given an integer x, determines whether M contains the integer x.

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

- If $M(1, n) = x$, we are done.
- If $M(1, n) < x$, the number x is not found in the top row because $M(1, 1) < M(1, 2) < \ldots < M(1, n) < x$. We can therefore ignore this row.
- If $M(1, n) > x$, then similarly x cannot be found in the rightmost column because all other elements there are larger than $M(1, n)$. Thus the last column can be ignored.

In the worst case, the sum of the width and height of the search table is reduced by one. We continue in this manner until either x is found or we reach an empty table and thus ascertain that x does not occur in the table. Since the initial sum of the height and the width of the table is $2n$ and at each step we make only one comparison, the algorithm takes $O(n)$ many steps. □

[H] Exercise 14. You are given two arrays, A and B, each containing n distinct positive integers each. Let $f(x, y) = y^6 + x^4y^4 + x^2y^2 - x^8 + 10$.

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

(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 $f(x, y) = 0$.

Solution.

(a) Here, we make a simplifying assumption that x, y are both positive integers. 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 $O(n \log n)$.

□

[H] Exercise 15. 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.

Solution. 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. □

[H] Exercise 16. 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.

Solution. 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 $O(n \log n)$. □

[E] Exercise 17. 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, $H[1], H[2], \ldots, H[n]$, 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.

Solution.

(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.

□

[E] Exercise 18. You are given n numbers $x_1, \ldots, x_n$, where each x_i is a real number in the interval $[0, 1]$.

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

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

In other words, the sum of the absolute difference between adjacent elements is strictly smaller than 2.

(b) Describe an $O(n)$ algorithm that solves the same problem as the previous question.

> Tweak the BucketSort algorithm. What size buckets should we use?

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

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

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

Proof. From the definition, we know that x_i is in the bucket k if and only if

$$\frac{k}{n} \le x_i < \frac{k+1}{n} \implies k \le n\,x_i < k+1 \quad \text{then} \quad k = \lfloor n\,x_i \rfloor.$$

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:

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

Note that there can be at most $n-1$ pairs of consecutive elements y_i, y_{i-1} which are in the same bucket (when all elements are in the same bucket). Whenever two elements y_i, y_{i-1} are in the same bucket, $|y_i - y_{i-1}|$ is at most equal to the size of the bucket, i.e, $< \frac{1}{n}$.

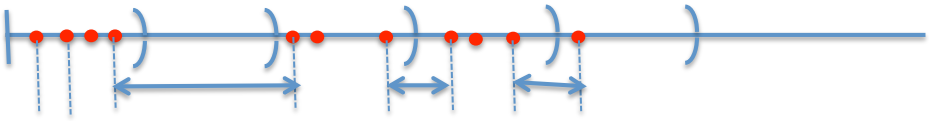
Thus,

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

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; as a result, the intervals in question are disjoint, as shown in the figure below.



Thus the total sum is smaller than $1 + 1 = 2$. □

§ SECTION THREE: TIME COMPLEXITY ANALYSIS

[K] Exercise 19. Show the following asymptotic relations by providing suitable constants for c and N.

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

(b) $4n^4 + 8n^2 + 1 = \Omega(n^3)$

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

(d) $\dfrac{2n^2 + 3n + 1}{3n + 1} = \Theta(n)$.

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

Solution.

(a) Note that, for all $n \geq 1$, $n^2 \leq n \cdot n^2 = n^3$. Therefore, choosing $c = 1$ and $N = 1$ shows that $n^2 = O(n^3)$.

(b) We want to find $c, N > 0$ such that

$$c \cdot n^3 \leq 4n^4 + 8n^2 + 1,$$

for all $n \geq N$. If $n > 8$, then $8n^2 < n^3 < n^4$ and $1 < n^4$. Therefore, we can bound the right hand side by $4n^4 + n^4 + n^4 = 6n^4$ for all $n > 8$. Choosing $c = 1$ and $N = 8$ proves the result.

(c) Note that $\sin(n) \leq 1$. Therefore, $n^2 + \sin(n) \leq n^2 + 1 \leq n^2 + n^2$ for all $n > 1$. Therefore, choose $c = 2$ and $N = 1$.

(d) Using long division, we can show that

$$\frac{2n^2 + 3n + 1}{3n + 1} = \frac{2n}{3} + \frac{2}{9(3n + 1)} + \frac{7}{9}.$$

Since $n > 0$, we note that

$$f(n) \leq \frac{2n}{3}.$$

Therefore, let $c = 2/3$ and $N = 1$.

(e) Note that $\cos(n) + \sin(n) \leq \sqrt{2}$. Therefore, let $c = \sqrt{2}$ and $N = 1$.

□

[K] Exercise 20. 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) $f(n) = n^{(1+\sin(\pi n/2))/2}$, $g(n) = \sqrt{n}$.

Solution.

(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

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

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 $f(n) = \Omega(g(n))$.

□

[K] Exercise 21. Let $f(n)$ and $g(n)$ be two positive functions on $\mathbb{N}$. Prove that $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

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

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

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

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

for all $n \geq n_0$. 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

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

But again, this implies that

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

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

[K] Exercise 22. Let $f(x) = x^2 - 12$, $g(x) = x^3$, and $h(x) = 7x - 3$. Show the following asymptotic relations.

(a) $f(x) = O(g(x))$.

(b) $f(x) \cdot h(x) = \Theta(g(x))$.

(c) $\dfrac{1}{f(x)} = O(1)$.

(d) $\dfrac{1}{g(x)} = \Omega(e^{-x})$.

Solution.

(a) Observe that $f(x) = x^2 - 12 < x^2 + 12 < x^2 + x$, for all $x > 12$. But since $x > 12$ is positive, this implies that

$$f(x) < x^2 + x^2 < \underbrace{x^2 + \cdots + x^2}_{x \text{ times}} = x \cdot x^2 = x^3 = g(x).$$

Therefore, for all $x > 12$, $f(x) < g(x)$, which shows that $f(x) = O(g(x))$.

(b) We now prove that $f(x) \cdot h(x) = \Theta(g(x))$. We first show that $f(x) \cdot h(x) = O(g(x))$. Consider

$$f(x) \cdot h(x) = 7x^3 - 3x^2 - 96x + 36.$$

For all $x > 0$, we note that

$$f(x) \cdot h(x) < 7x^3 + 3x^2 + 96x + 36.$$

For all $x > 96$, we see that $3x^2 < 96x^2 < x^3$, $96x < x^2 < x^3$, and $36 < x < x^3$. Therefore,

$$f(x) \cdot h(x) < 7x^3 + x^3 + x^3 + x^3 = 10x^3.$$

Choosing $c = 10$ and $n_0 = 96$, this shows that $f(x) \cdot h(x) = O(g(x))$. We now prove that $f(x) \cdot h(x) = \Omega(g(x))$. Clearly, we have that

$$f(x) \cdot h(x) > 7x^3 - 3x^2 - 96x.$$

Now, for all $x > 96$, we have that $-x < -96$. Therefore, $f(x) \cdot h(x) > 7x^3 - 3x^2 - x^2 = 7x^3 - 4x^2$. Similarly, since $x > 4$, we have that $-x < -4$ which implies that $-x^3 < -4x^2$. But then this implies that

$$f(x) \cdot h(x) > 7x^3 - x^3 = 6x^3.$$

Therefore, $f(x) \cdot h(x) > 6g(x)$ for all $x > 96$. Choosing $c = 6$ and $n_0 = 96$, this shows that $f(x) \cdot h(x) = \Omega(g(x))$ which shows that $f(x) \cdot h(x) = \Theta(g(x))$.

(c) Note that $1/f(x) \to 0$ as $x \to \infty$. Therefore, for large enough n_0, we can pick an arbitrary constant c such that $c > 1/f(x)$ for some n_0.

(d) From Exercise 21, we will prove that $e^{-x} = O(1/g(x))$. We will also use the result from Exercise 23, part (a) to show the Big-Oh result.

Observe that

$$\begin{aligned}
\lim_{x\to\infty} \frac{e^{-x}}{1/x^3} &= \lim_{x\to\infty} \frac{x^3}{e^x} \\
&= \lim_{x\to\infty} \frac{3x^2}{e^x} && \text{(by L'Hôpital's rule)} \\
&= \lim_{x\to\infty} \frac{6x}{e^x} && \text{(by L'Hôpital's rule)} \\
&= \lim_{x\to\infty} \frac{6}{e^x} = 0.
\end{aligned}$$

Therefore, $e^{-x} = O(1/x^3) = O(1/g(x))$. But this implies that $1/g(x) = \Omega(e^{-x})$, which completes the proof.

□

[K] Exercise 23. 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 $f(n) = \Omega(g(n))$.

Solution.

(a) Suppose that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Then, for every $\epsilon > 0$, there exist some $N_\epsilon > 0$ such that, for all $n > N_\epsilon$, we have that

$$\left|\frac{f(n)}{g(n)}\right| < \epsilon.$$

Since f, g are positive functions, this is equivalent to $f(n) < \epsilon \cdot g(n)$, which is precisely what it means to say that $f(n) = O(g(n))$.

(b) Suppose that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Then, for every $\epsilon > 0$, there exist some $N_\epsilon > 0$ such that, for all $n > N_\epsilon$, we have that

$$\left|\frac{f(n)}{g(n)}\right| > \epsilon.$$

Since f, g are positive functions, this is equivalent to $f(n) > \epsilon \cdot g(n)$, which is precisely what it means to say that $f(n) = \Omega(g(n))$.

□

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

(a) If $c > 0$ is a constant, 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, Big-Oh 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) $O(f(n) + g(n)) = O\left(\max\{f(n), g(n)\}\right)$.

Solution.

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

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

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

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

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.

□

[H] Exercise 25. 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.

Solution.

(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 $h(f(n)) \neq O(h(g(n)))$.

□

[E] Exercise 26. 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) $f(n) = n^{5/2}$, $g(n) = \left[\log\left(\sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!}\right)\right]^2$.

Solution.

(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, $f(n) = \Omega(g(n))$.

□

[E] Exercise 27. Let $f : \mathbb{N} \to \mathbb{R}$ be a positive and strictly increasing function; that is, for all $n \in \mathbb{N}$, $f(n) > 0$. Show that $1/f(n) = O(f(n))$.

Solution. Since f is strictly increasing, then $f'(n) > 0$ for all $n \in \mathbb{N}$. Leting $g(n) = 1/f(n)$, We see that

$$g'(n) = -\frac{f'(n)}{[f(n)]^2} < 0.$$

Therefore, $g(n)$ is a strictly decreasing function. Since g is decreasing and bounded below by 0, we know that g converges to a positive number, say c. However, f is strictly increasing and unbounded; therefore, for large enough n_0, $f(n) > 1/f(n) \to c$ for every $n > n_0$. □

[X] Exercise 28. Define $f : \mathbb{N} \to \mathbb{R}$ by

$$f(n) = \begin{cases} 1 & \text{if } n \leq 2, \\ f\left(\left\lceil \frac{n}{\log_2 n} \right\rceil\right) + n & \text{if } n \geq 3. \end{cases}$$

Show that $f(n) = \Theta(n)$.

- Show that $f(n) \leq 2n$ for $n \geq 512$.
- Show that $f(n) \geq n$ for $n > 2$.

Solution. Throughout the proof, we will omit the ceiling because it is negligible in asymptotic complexity analysis. We show that $f(n) = \Theta(n)$ by first showing that $f(n) = O(n)$ and then $f(n) = \Omega(n)$. We make the following claim.

Claim. For $n \geq 512$, $f(n) \leq 2n$.

Solution. We proceed with *strong induction*.

The base case is easy to check; we have

$$f(512) = 935 \leq 1024 = 2 \cdot 512.$$

Now assume that $f(k) \leq 2k$ for all $512 \leq k < n$. Now since $k \geq 512$, we have that $\log_2 k \geq \log_2 512 = 9$. Thus,

$$\frac{n}{\log_2 512} \leq \frac{n}{9}.$$

Then we have that

$$\begin{aligned} f(n) &= f\left(\frac{n}{\log_2 n}\right) + n \\ &\leq f\left(\frac{n}{9}\right) + n && (f \text{ is increasing}) \\ &\leq 2 \cdot \frac{n}{9} + n && (\text{inductive hypothesis}) \\ &= n\left(\frac{2}{9} + 1\right) \\ &\leq 2n. \end{aligned}$$

□

This shows that $f(n) = O(n)$. We now show that $f(n) = \Omega(n)$. To do this, it is enough to show that $f(n) \geq n$ for $n > 2$. Note that $n/\log_2 n \geq 1$ for $n > 2$. Hence, we have that

$$f(n) = f\left(\frac{n}{\log_2 n}\right) + n \geq f(1) + n = n + 1 > n.$$

Thus, $f(n) = \Omega(n)$ and combining this result with the previous result, we have that $f(n) = \Theta(n)$. □