Algorithms Tutorial 1
Solutions

1. You are given an array S of n integers and another integer x.

   (a) Describe an $O(n \log n)$ algorithm (in the sense of the worst case performance) that determines whether or not there exist two elements in S whose sum is exactly x.

   (b) Describe an algorithm that accomplishes the same task, but runs in $O(n)$ **expected** (i.e., average) time.

   Solution: Note that brute force does not work here, because it runs in $O(n^2)$ time.

   (a) First, we sort the array – we can do this in $O(n \log n)$ in the worst case, for example, using Merge Sort. A couple of approaches from here:

   - 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.
   - Alternatively again, you 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 you can eliminate one 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.

(b) We take a similar approach as in (a), except using a hash map (hash table) to check if elements exist in the array: each insertion and lookup takes $O(1)$ expected time.

The following approaches correspond to the approaches in (a):

- At index i, we assume $A[1..i-1]$ is already stored in the hash map. Then we check if $x - A[i]$ is in the hash map in $O(1)$, then insert $A[i]$ into the hash map, also in $O(1)$.
- Alternatively, we hash all elements of A and then go through elements of A again, this time for each element a checking in $O(1)$ time if $x - a$ is in the hash table. If $2a = x$, we also check if at least 2 copies of a appear in the corresponding slot of the hash table.

2. Given two arrays of n integers, design an algorithm that finds out in $O(n \log n)$ steps if the two arrays have an element in common. (Microsoft interview question)

   Solution: Option 1: Sort one of the arrays and then go through all of the elements of the other array and for each element do a binary search in the sorted array to see if this element also appears there. Option 2: Sort both arrays and then remove elements from both arrays in the same manner as you would if you were merging the arrays (but you do not need to store the elements) and see if you ever encounter the same (smallest remaining) element in both arrays.

3. Given an array $A[1..100]$ which contains all natural numbers between 1 and 99, design an algorithm that runs in $O(n)$ and returns the duplicate value. (Microsoft interview question)

   Solution: A simple solution: Initialise an array B of length 99; go through A placing each element i into the slot $B[i]$ and see which slot would get 2 elements. A fancier solution: just add up all elements of A and subtract from such a sum the sum of all numbers from 1 to 99, i.e., subtract $100 \times 99/2 = 50 \times 99 = 4950$ to get which number appears twice.

4. Assume you are given two arrays A and B, each containing n distinct positive numbers and the equation $x^8 - x^4y^4 = y^6 + x^2y^2 + 10$. Design an algorithm which runs in time $O(n \log n)$ which finds if A contains a value for x and B contains a value for y that satisfy the equation.

Solution: Solution: write the equation in the form

$$x^8 = x^4y^4 + y^6 + x^2y^2 + 10;$$

Now note that the right hand side is monotonic in y (actually, in both x and y, but monotonicity in y is important here.) Sort array B in time $n \log n$; go through all elements of A; for each element $x \in A$ do a binary search to see if there is a y satisfying the equation; this is possible because the right hand side is monotonic in y and B is sorted. Thus, if one element y produced a value of the right hand side bigger than the value of x^8 then all values larger than such y will also produce a value of the right hand side exceeding x^8.

5. You're given an array of n integers, and must answer a series of n queries, each of the form: "how many elements of the array have value between L and R?", where L and R are integers. Design an $O(n \log n)$ algorithm that answers all of these queries.

   Solution: We first sort the array in $O(n \log n)$, using Merge Sort. For each query, we can binary search to find the index 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. Each binary search takes $O(\log n)$ so the algorithm runs in $O(n \log n)$ overall. 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. Similar observation applies if your binary search hits R.

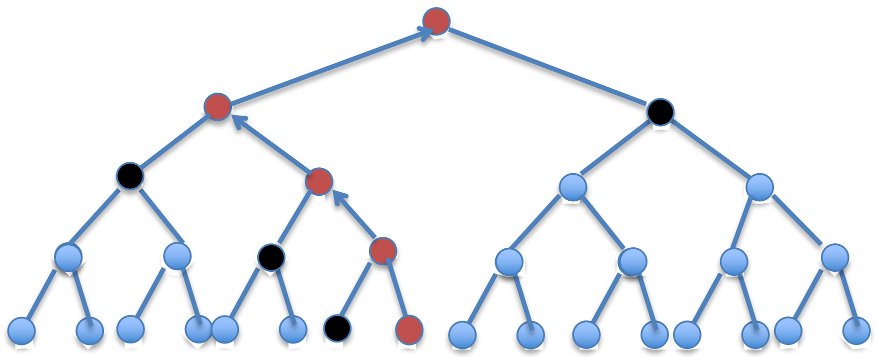
6. Assume you have an array of $2n$ distinct integers. Find the largest and the smallest number using $3n - 2$ comparisons only.

   Solution: Note that the brute force does not work - if you take the first two elements $A[1]$ and $A[2]$, compare them and set $m = \min\{A[1], A[2]\}$ and $M = \max\{A[1], A[2]\}$, you made 1 comparison and are left with $2n-2$ elements each of which has to be compared both with m and M to see if they need to be revised, which in total gives you $2(2n-2)+1 = 4n-3$ 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 element must be in S and the largest in L and we have made n

comparisons. We now use the brute force on the two arrays which takes $n-1$ comparisons in each array. In total this takes $n+n-1+n-1=3n-2$ comparisons.

7. Assume that you have an array of 2^n distinct integers. Find the largest and the second largest number using only 2^n+n-2 comparisons.

   Solution: See the figure below. On the figure 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). 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 this is exactly 2^n+n-2 many comparisons.

8. 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 your task can always be accomplished by asking no more than $3n - 3$ such questions, even in the worst case.

(b) Show that your task can always be accomplished by asking no more than $3n - \lfloor \log_2 n \rfloor - 3$ such questions, even in the worst case.

Solution: Assume the people are numbered 1 to n.

(a) We observe that our conditions imply that there can be **at most** one person who is a celebrity. This is because if there were two celebrities A and B, then A would have to know B because B is a celebrity. But then A cannot be a celebrity because he knows someone else at the party. We now proceed as follows. We pick two arbitrary people A and B. We ask A if she knows B. If she does, then A cannot be a celebrity. If she does not then B cannot be a celebrity. Thus, it takes 1 question to eliminate one person as a potential celebrity. So after $n - 1$ questions we arrive at a single possible candidate c who could be a celebrity.

It is possible that c isn't a celebrity either, so we must verify that they are. To do this, we need to ask $n - 1$ questions of the form "does j know c?"; if the answer is always yes, c still could be a celebrity (otherwise he is not and we conclude there is no celebrity at the party); then we ask $n-1$ questions of the form "does c know j?" and if the answer is always "no" we have found a celebrity, otherwise no celebrity is present. Hence, our algorithm uses $n-1+2(n-1) = 3n-3$ questions, even in the worst case. Important for the (b) part of the problem, note also that $3n-4$ questions suffice, because we can reuse the answer to at least one question which was asked during the initial search for the potential celebrity c, which was either in of the form "does X knows c" (and the answer was yes) or of the form "does c knows X" (and the answer was no, because c was the final candidate for a celebrity).

(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.

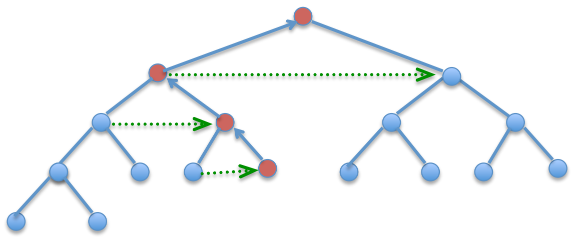


Figure 1: Here $n = 9$; thus, $m = \lfloor \log_2 n \rfloor = 3$ and we add two children to $n - 2^m = 9 - 2^3 = 1$ leaf of a perfect binary tree with 8 leaves. Thus obtained tree has 9 leaves. The potential celebrity is in red. Note that we do not have to repeat 3 questions represented by the green arrows.

9. Given n numbers $x_1, \dots, x_n$ where each x_i is a real number in the interval $(0, 1)$, devise an algorithm that runs in linear time that outputs a permutation of the n numbers, say $y_1, \dots, y_n$, such that $\sum_{i=2}^{n} |y_i - y_{i-1}| < 2$.

   Hint: this is easy to do in $O(n \log n)$ time: just sort the sequence in ascending order. In this case, $\sum_{i=2}^{n} |y_i - y_{i-1}| = \sum_{i=2}^{n} (y_i - y_{i-1}) = y_n - y_1 \leq 1 - 0 = 1$. Here $|y_i - y_{i-1}| = y_i - y_{i-1}$ because all the differences are non-negative, and all the terms in the sum except the first and the last one cancel out. To solve this problem, one might think about tweaking the BucketSort *algorithm, by carefully avoiding sorting numbers in the same bucket.*

   Solution: Split the interval $[0, 1]$ into n equal buckets. Assign to each x_i its bucket number $b(x_i) = \lfloor n\, x_i \rfloor$. Why is this the bucket where x_i belongs? x_i is in the bucket k if and only if $\frac{k}{n} \leq x_i < \frac{k+1}{n}$. This holds if and only if $k \leq n\, x_i < k + 1$ i.e., if $k = \lfloor n\, x_i \rfloor$. So in linear time we can form pairs $\langle x_i, b(x_i) \rangle$, $(1 \leq i \leq n)$. We can now sort these pairs according to their bucket number $b(x_i)$; since all bucket numbers are $< n$, e.g., CountingSort does that in linear time. Denote the sequence produced by CountingSort by y_i, $1 \leq i \leq n$.

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). We now prove that the sequence produced satisfies that $\sum_{i=2}^{n}|y_i - y_{i-1}| < 2$.

We split this sum into two sums:

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

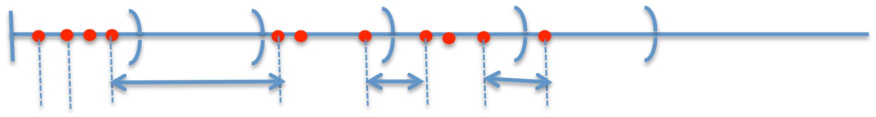
Note that there can be at most $n-1$ pairs of elements y_i, y_{i-1} which are in the same bucket (this happens if all elements end up in the same bucket!); whenever two elements y_i, y_{i-1} are in the same bucket $|y_i - y_{i-1}|$ is at most equal to the size of the bucket, i.e, $\leq \frac{1}{n}$. Thus,

$$\sum_{j}\big(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}\big) \leq \frac{n-1}{n} < 1;$$

also

$$\sum_{j}\big(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}\big) \leq 1$$

because it is a sum of lengths of disjoint intervals in $[0,1]$, see the figure. Thus the total sum is smaller than $1 + 1 = 2$.



10. Given n real numbers $x_1, ..., x_n$ where each x_i is a real number in the interval $[0,1]$, devise an algorithm that runs in linear time and that will output a

permutation of the n numbers, say $y_1, ..., y_n$, such that $\sum_{i=2}^{n} |y_i - y_{i-1}| < 1.01$.

Solution: Exactly the same as the previous problem, just instead of using n buckets, use $100n$ many buckets.

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

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

and for each column j, $1 \leq j \leq n$,

$$M(1, j) < M(2, j) < \ldots < M(n, j)$$

You need to determine whether M contains an integer x in $O(n)$ time.

Solution: Consider $M(1, n)$ (i.e., top right cell); if $M(1, n) = x$ you are done; else if $M(1, n) < x$ number x certainly is not found in the top row because $M(1, 1) < M(1, 2) < ... < M(1, n) < x$ so this row can be ignored; else if $M(1, n) > x$ then similarly x cannot be found in the rightmost column because all element there are larger than $M(1, n)$, thus the last column can be ignored. In both cases the sum of the width and height of the search table is reduced by 1. 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.

12. 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.

Hint: try "double" QuickSort*; one which uses a foot as a pivot for shoes and another one which uses a shoe as a pivot for feet.*

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 $O(nlogn)$.

13. 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. We assume that students are not allowed to vote for themselves. However, the students have forgotten to specify the order of names on each piece of paper – for instance, "Alice Bob" could mean that Alice voted for Bob, or that 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 determine who these students voted for?

    (c) Suppose every student received at least one vote. What is the maximum possible number of votes received by any student? Justify your answer.

    (d) Using parts (b) and (c), or otherwise, design an 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. An $O(n^2)$ algorithm earns partial credit, but you should aim for an $O(n)$ algorithm.

    Hint: first, use part (c) to consider how you would solve it in the case where every student received at least one vote. Then, apply part (b).

    Solution:

    (a) If a student's name appears on x pieces of paper, then the student received $x - 1$ votes since each student voted precisely once.

    (b) If a student did not receive any votes, their name only appears on precisely one piece of paper. The name of the other student is who they voted for.

    (c) If every student received at least one vote, then at least n distinct pieces of paper are required to correspond to these votes. There are no more pieces of paper to be distributed, so every student received exactly one vote. Hence, each student also received a maximum of **one vote**.

(d) Suppose every student received at least one vote. Then, by (c), every student received exactly one vote. By considering the votes as an undirected graph (where each student is a vertex and every vote is an edge between two students), or otherwise, we can see that every student appears on precisely two pieces of paper. This corresponds to a set of disjoint cycle graphs where students are vertices and pieces of paper are edges between students.

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.

14. 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) of the N teams. Not being the sporty type – but wanting to fit in nonetheless – you must decide for yourself some subset of teams (possibly all, or none) to support. You don't want to be branded a copycat, so your subset must not be identical to anyone else's. The trouble is, you don't know which friends support which teams, so you can ask your friends some questions of the

form "Does friend A support team B?" (you choose A and B before asking each question). Design an algorithm that determines a suitable subset of teams for you to support and asks as few questions as possible in doing so.

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. Clearly, this subset of teams is different to all of our friends', and it uses N queries, which is the minimal possible for any deterministic solution (we must have some information about each friend). Note: this method is important and is called "diagonalisation".

15. You are given an array A consisting of $2n-1$ integers. Design an algorithm which finds all of the n possible sums of n consecutive elements of A and **which runs in time $\mathbf{O(n)}$**. Thus, you have to find the values of all of the sums

$$\begin{aligned}
S[1] &= A[1] + A[2] + \ldots + A[n-1] + A[n];\\
S[2] &= A[2] + A[3] + \ldots + A[n] + A[n+1];\\
\vdots \quad & \quad\quad \vdots \quad\quad\quad\quad \vdots\\
S[n] &= A[n] + A[n+1] + \ldots + A[2n-2] + A[2n-1],
\end{aligned}$$

and your algorithm **should run in time $\mathbf{O(n)}$**.

Solution: Here are at least two possible solutions:

- We can compute $S[1]$ in $O(n)$ by simply adding up $A[1], \ldots, A[n]$. Then, for each $i \geq 2$, we have that $S[i] = S[i-1] - A[i-1] + A[n+i-1]$, so we can compute each subsequent $S[i]$ in $O(1)$ time each, giving an $O(n)$ algorithm.
- We can first compute the array $C[0..2n-1]$ where $C[i] = A[1] + A[2] + \cdots + A[2n-1]$. We have that $C[0] = 0$, and for $i \geq 1$, $C[i] = C[i-1] + A[i]$. Hence, we compute C in $O(n)$. Then $S[i] = C[i+n-1] - C[i-1]$, so we can compute all the S values in $O(n)$.

16. You are a fisherman, trying to catch fish with a net that is W meters wide. Using your advanced technology, you know that the positions of all N fish in the sea can be represented as integers on a number line. There may be more than one fish at the same location.
To catch the fish, you will cast your net at position x, and will catch all fish with positions between x and $x + W$, **inclusive**. Given N, W and an array

$X[1..N]$ denoting the positions of fish in the sea, give an $\mathbf{O(N \log N)}$ algorithm to find the maximum number of fish you can catch by casting your net once. For example, if $N = 7, W = 3$ and $X = [1, 11, 4, 10, 6, 7, 7]$, then the most fish you can catch is 4: by placing your net at $x = 4$, you will catch one fish at position 4, one fish at position 6 and two fish at position 7.

Solution: First, sort the array (e.g. using MergeSort) in $O(N \log N)$ time. We know that it optimal to cast our net starting from the same position as a fish. We can use a "two pointers" approach: we keep variables L and R, so the fish in positions $[L..R]$ of the array are those in our net.

Initially $L = 1$, and in $O(N)$ time we find the largest R so that the R^{th} fish in sorted order is still in our net. We then repeatedly increment L, and respectively repeatedly increment R to include all fish that fit within the net starting at the L^{th} fish. At each stage, we know we can catch $R - L + 1$ fish, so we take the maximum among these.

Since L and R are only ever incremented, and they are each incremented at most N times, the algorithm is $O(N)$.

17. Your army consists of a line of N giants, each with a certain height. You must designate precisely $L \leq N$ 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 $H[1..N]$ 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 instance, suppose $N = 10, L = 3, K = 2$ 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.

This 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.

18. You are given an array A of n distinct integers.

(a) You have to determine if there exists a number (not necessarily in A) which can be written as a sum of squares of two distinct numbers from A in two different ways (note: $A[m]^2 + A[k]^2$ and $A[k]^2 + A[m]^2$ counts as a single way) and which runs in time $n^2 \log n$ in the **worst case** performance. Note that the brute force algorithm would examine all quadruples of elements in A and there are $\binom{n}{4} = O(n^4)$ such quadruples. (10 points)

(b) Solve the same problem but with an algorithm which runs in the **expected time** of $O(n^2)$.

Solution: (a) Go through all of $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs $(A[k], A[m])$, $k < m$, of distinct integers in A; compute the sums $A[k]^2 + A[m]^2$ for all $1 \leq k < m \leq n$ and put them in an array of size $n(n-1)/2$. Sort the array in time $O(n^2 \log_2 n^2) = O(n^2 \log_2 n)$. Go through the sorted array and determine if a

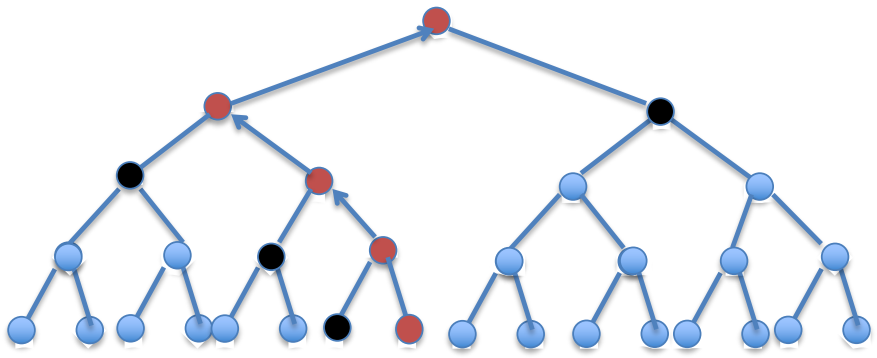
number appears in it at least twice (such occurrences would be consecutive).
(b) Use a hash table of size n^2 to hash all the sums $A[i]^2 + A[j]^2$ for all $1 \leq i < j \leq n$ and go through all of the n^2 slots and check by brute force if the same number appears twice.

19. Suppose that you bought a bag of n bolts with nuts screwed on them. Your 5 year old nephew unscrewed all the nuts from the bolts and put both the nuts and the bolts back into the bag. The bolts are all of similar quite large size but are actually of many different diameters, differing only by at most a few millimetres, so the only way to see if a nut fits a bolt is to try to screw it on and determine if the nut is too small, if it fits or if it is too large. Design an algorithm for matching each bolt with a nut of a fitting size which runs in the **expected** time $O(n \log n)$.

    Solution: Pick a bolt and try all nuts, placing them on three piles: too small, fitting, too large. Now pick a nut from the fitting pile and try all bolts, placing them on three piles: too small, fitting, too large. Screw fitting nuts on each of the fitting bolts. Continue such a process with the piles of too small bolts and too small nuts, as well as with piles of too large bolts and too large nuts. The algorithm runs in time $O(n \log n)$ for exactly the same reason as the QuickSort algorithm.

20. You are given 1024 apples, all of similar but different sizes and a small pan balance which can accommodate only one apple on each side. Your task is to find the heaviest and the second heaviest apple while making at most 1032 weighings.

    Solution: We prove a more general statement: you are given a list of 2^n numbers and you can only compare them pairwise, determining which one is larger or if both are equal. We will show that you can find the largest and the second largest number using only $2^n + n - 2$ comparisons. We now put all of these numbers at the leaves of a complete binary tree, which is a tree in which every node is either a leaf or has exactly two children, see the figure. A complete binary tree with 2^n leaves has $2^n - 1$ internal nodes and is of depth n. 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



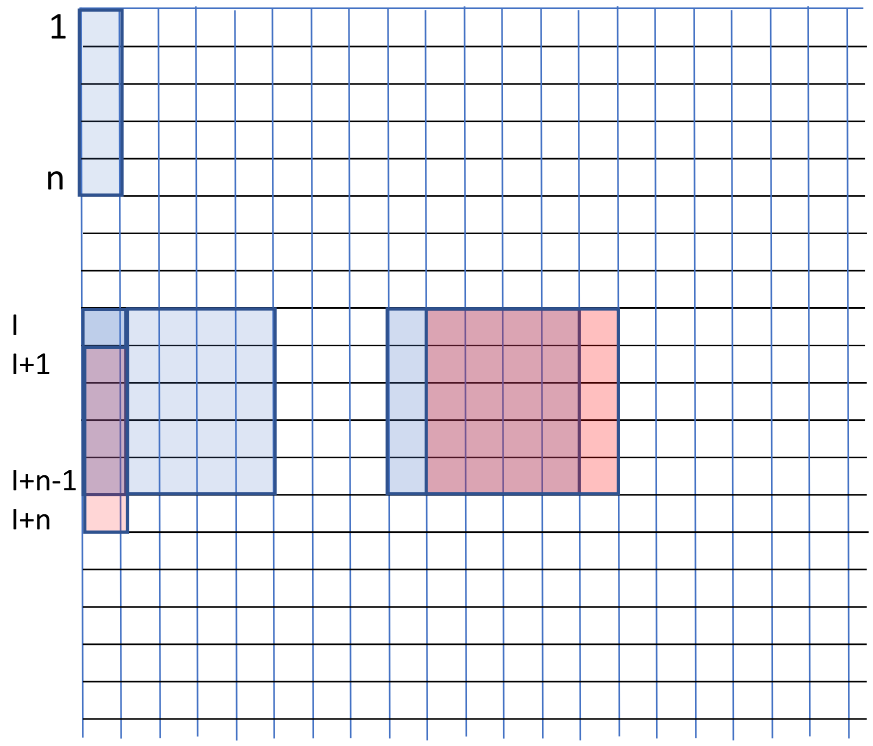
largest element along the way - all elements underneath them must be smaller or equal to the elements shown in black. There are n many such elements so finding the largest among them will take $n-1$ comparisons by brute force. In total this is exactly $2^n + n - 2$ many comparisons.

21. You are in an orchard which has the shape of a square of size $4n$ by $4n$ with equally spaced trees. You purchased apples from n^2 trees which also form a square, but the owner is allowing to choose such a square anywhere in the orchard. You have a map with the number of apples on each tree. Your task is to choose such a square which contains the largest total number of apples and which runs in time $O(n^2)$. Note that the brute force algorithm would run in time $\Theta(n^4)$.

    Solution: The idea here is to note that there is a heavy overlap between such possible squares and to use this fact to compute the number of apples in all of such squares in an efficient way. Referring to the figure below we first consider the first column. Let the number of apples in cell $C[i,j]$ be $A[i,j]$. We first compute 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). This 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 recursion

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

    This is correct because the rectangle $r(i,1)$ 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 $r(i+1)$, see the picture. Since each recursion step involves only one addition and one subtraction this can all be done in O(n) many steps in total.

We now perform the same procedure in each column j, thus obtaining the number of apples $\alpha(i,j)$ in every rectangle consisting of cells $C(i,j)$ to $C(i+n-1,j)$, which will take in total $O(n)\times O(n)=O(n^2)$ many steps.

We now for each i such that $1\leq i\leq 3n+1$ compute $\beta(i,1)=\sum_{p=1}^{n}\alpha(i,p)$; this clearly gives the total number of apples in the square with vertices at cells $C(i,1), C(i,n), C(i+n-1,1)$ and $C(i+n-1,n)$. Each such computation takes $O(n)$ additions so in total doing this for all i it takes $O(n^2)$ many additions.

For each fixed i such that1 $\leq i\leq 3n+1$, starting with $\beta(i,1)$ 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. Each step of recursion takes only one addition and one subtraction so the whole recursion takes $O(n)$ many steps. Thus, all recursions for all $1 \leq i \leq 3n+1$ takes $O(n^2)$ many operations. We finally find the largest $\beta(i,j)$.

22. Determine if $f(n) = O(g(n))$ or $g(n) = O(f(n))$ or both (i.e., $f(n) = \Theta(g(n))$) or neither of the two, for the following pairs of functions

    (a) $f(n) = (\log_2(n))^2$; $g(n) = \log_2(n^{\log_2 n})^2$; (6 points)

    (b) $f(n) = n^{10}$; $g(n) = 2^{\sqrt[10]{n}}$; (6 points)

    (c) $f(n) = n^{1+(-1)^n}$; $g(n) = n$. (8 points)

    Solution:

    (a) Note that $\log_2(n^{\log_2 n})^2 = \log_2(n^{2\log_2 n}) = 2\log_2 n \log_2 n = 2(\log_2 n)^2 = \Theta((\log_2(n))^2)$

    (b) We show that eventually $n^{10} < 2^{\sqrt[10]{n}}$. Taking the logs of both sides, since $\log_2 n$ is a monotonically increasing function, it is enough to prove that $\log n^{10} = 10\log n < \log_2 2^{\sqrt[10]{n}} = \sqrt[10]{n}$. To show that $10\log n = O(\sqrt[10]{n})$ we show that $\lim_{n\to\infty} \log n/\sqrt[10]{n} = 0$. Since both the numerator and the denominator go to infinity, we can apply the L'Hôpital rule and get

$$\lim_{n\to\infty}\frac{\log n}{\sqrt[10]{n}} = \lim_{n\to\infty}\frac{(\log n)'}{(\sqrt[10]{n})'} = \lim_{n\to\infty}\frac{\frac{1}{n}}{\frac{1}{10}n^{-9/10}} = \lim_{n\to\infty} 10\frac{1}{n^{1/10}} = 0$$

    (c) Just note that for all even n, $f(n) = n^2$ which grows much faster than $g(n) = n$. However for all odd n we have $f(n) = n^0 = 1$ so for odd n $g(n) = n$ grows much faster than $f(n)$. Thus, neither $f(n) = O(g(n))$ nor $g(n) = O(f(n))$.

23. Read the review material from the class website on asymptotic notation and basic properties of logarithms, pages 38-44 and then determine if $f(n) = \Omega(g(n))$, $f(n) = O(g(n))$ or $f(n) = \Theta(g(n))$ for the following pairs. Justify your answers.

$f(n)$	$g(n)$
$(\log_2 n)^2$	$\log_2(n^{\log_2 n}) + 2\log_2 n$
n^{100}	$2^{n/100}$
$\sqrt{n}$	$2^{\sqrt{\log_2 n}}$
$n^{1.001}$	$n \log_2 n$
$n^{(1+\sin(\pi n/2))/2}$	$\sqrt{n}$

You might find the following inequality useful: if $f(n), g(n), c > 0$ then $f(n) < c\ g(n)$ if and only if $\log f(n) < \log c + \log g(n)$. Also remember that $O(f(n))$ does not define a linear ordering; for some f, g neither $f = O(g)$ nor $g = O(f)$.

Solution:

(a) Using $\log(a^b) = b \log a$ we obtain $\log_2(n^{\log_2 n}) + 2\log_2 n = \log_2 n \cdot \log_2 n + 2\log_2 n = (\log_2(n))^2 + 2\log_2 n = \Theta((\log_2(n))^2)$ because $2\log_2 n$ grows much slower than $(\log_2(n))^2$.

(b) We want to show that $n^{100} = O(2^{n/100})$, which means that we have to show that $n^{100} < c\,2^{n/100}$ for some positive c and all sufficiently large n. But, since the log function is monotonically increasing, by taking the log of both sides, this will hold just in case

$$\log n^{100} < \log c + \log(2^{n/100})$$

which holds just in case

$$100 \log n < \log c + n/100$$

We now see that if we take $c = 1$ then it is enough to show that

$$100 \log n < n/100$$

for all sufficiently large n which holds because

$$10000 \log n < n$$

for all sufficiently large n.

(c) We want to show that $\sqrt{n} = \Omega(2^{\sqrt{\log_2 n}})$, i.e., that $\sqrt{n} > c\ 2^{\sqrt{\log_2 n}}$ for some c and all sufficiently large n. By the same argument as in the previous case it is enough to show that $\log \sqrt{n} > \log_2(c \cdot 2^{\sqrt{\log_2 n}})$. But $\log \sqrt{n} = \log n^{1/2} = 1/2 \log n$ and $\log_2(c \cdot 2^{\sqrt{\log_2 n}}) = \log c + \log_2 2^{\sqrt{\log_2 n}} = \log c + \sqrt{\log_2 n}$. Again taking $c = 1$, it is enough to show that $1/2 \log n > \sqrt{\log_2 n}$ which clearly holds for all sufficiently large 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 $f(n) = \Omega(g(n))$.

24. Determine the asymptotic growth rate of the solutions to the following recurrences. If possible, you can use the Master Theorem, if not, find another way of solving it.

(a) $T(n) = 2T(n/2) + n(2 + \sin n)$

(b) $T(n) = 2T(n/2) + \sqrt{n} + \log n$

(c) $T(n) = 8T(n/2) + n^{\log n}$

(d) $T(n) = T(n - 1) + n$

Solution:

(a) Note that in this case $a = 2$ and $b = 2$ so $n^{\log_b a} = n^{\log_2 2} = n$. On the other hand, $f(n) = n(2 + \sin n) = \Theta(n)$ because $1 \leq 2 + \sin n \leq 3$. Thus, the second case of the Master Theorem applies and we get $T(n) = \Theta(n \log n)$.

(b) Again, $n^{\log_b a} = n$. On the other hand, we have $\log n = O(\sqrt{n})$ and so $\sqrt{n} + \log n = \Theta(\sqrt{n})$. This implies $\sqrt{n} = n^{.5} = O(n^{0.9}) = O(n^{\log_2 2 - .1})$ so the first case of the Master Theorem applies and we obtain $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

(c) We have $n^{\log_b a} = n^{\log_2 8} = n^3$. Thus $f(n) = n^{\log n} = \Omega(n^4)$. Consequently, $f(n) = \Omega(n^{\log_b a + 1})$. To be able to use the third case of the Master Theorem, we have to show that for some $0 < c < 1$ the following holds for sufficiently large n: $a\,f(n/b) = 8f(n/2) < cf(n)$ which in our case translates to

$$8\left(\frac{n}{2}\right)^{\log(n/2)} < c\,n^{\log n}$$

However, we have

$$8\left(\frac{n}{2}\right)^{\log(n/2)} = 8\left(\frac{n}{2}\right)^{\log n-\log_2 2} = 8\left(\frac{n}{2}\right)^{\log n-1} < 8\left(\frac{n}{2}\right)^{\log n}$$
$$= \frac{8\,n^{\log n}}{2^{\log n}} = \frac{8}{n}n^{\log n}$$

Thus, if $n > 16$ then $8\left(\frac{n}{2}\right)^{\log(n/2)} < \frac{1}{2}n^{\log n}$ and the condition is satisfied with $c = 1/2$ and all $n > 16$.

(d) Note that for every k we have $T(k) = T(k-1) + k$. So just unwind the recurrence to get

$$\begin{aligned} T(n) &= T(n-1)+n = T(n-2)+(n-1)+n = T(n-3)+(n-2)+(n-1)+n = \dots \\ &= T(1)+(n-(n-2))+(n-(n-3))+\dots+(n-1)+n \\ &= T(1)+(2+3+4+\dots+n) = T(1)+\frac{n(n+1)}{2}-1 \\ &= \Theta(n^2); \end{aligned}$$

25. Assume that you are given an array A containing $2n$ numbers. The only operation that you can perform is make a query if element $A[i]$ is equal to element $A[j]$, $1 \leq i, j \leq 2n$. Your task is to determine if there is a number which appears in A at least n times using an algorithm which runs in linear time.

Warning and a Hint: a tricky one. The reasoning resembles a little bit the reasoning used in the celebrity problem: try comparing them in pairs and first find one or at most two possible candidates and then count how many times they appear.

Solution: Let A be any array with $2n$ elements for which we want to determine if there exists a number appearing in the array at least n times. Repeat the following procedure: pick any two elements and compare them. If the numbers are different throw both away. Note that in this way we can throw away at most one copy of the number x appearing n times (if there is such) so in whatever is left x will also have number of copies equal to at least half of the total leftover elements. If the numbers in the picked pair are equal set them into two separate piles X and Y. Now note that X has the same elements as Y you can throw away Y and keep X which will still have the same property that at least half of its elements are equal just in case this was true of the original pile. Now continue the algorithm on pile X, until you are left with at most 2 elements. Then directly count for each of them how many times they appear in the original pile.