# Sample solutions to assignment 1

1. (a) Describe an $O(n \log n)$ algorithm (in the sense of the worst case performance) that, given an array S of $n$ integers and another integer $x$, 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 (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 $i$ from 1 to $n$, we can use binary search to check, in $O(\log n)$ time, if $x - A[i]$ exists in the sorted prefix $A[1..i-1]$. This is sufficient since each pair is considered exactly once and we also exclude $A[i]$, important in case $x = 2A[i]$ but the value of $A[i]$ appears only once in $A$. Hence, this gives an $O(n \log n)$ algorithm.
   - Alternatively, 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 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 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. 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 $R$.

3. 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).

4. Given $n$ numbers $x_1, \ldots, x_n$ 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 $y_1, \ldots, y_n$, 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.*

**Solution:** We will bucket the elements into $n$ disjoint buckets, each of width $\frac{1}{n}$. Specifically, our buckets contain elements from $\left[\frac{\delta}{n}, \frac{\delta+1}{n}\right)$ for $\delta$ ranging over integers between 0 and $n-1$ inclusively. We note that we can handle elements precisely equal to 1 in any number of ways, for instance, moving them all to the end in $O(n)$, considering them as a part of the final bucket, or making them a bucket unto themselves. Then, for each element in our array, we place it into its correct bucket. We can represent each bucket as a linked list, so we use $O(n)$ memory, and insertion into a bucket is $O(1)$, so this step is $O(1)$. Note: we DO NOT sort the buckets (this cannot be done in linear time!); however, to simplify the proof of the correctness of our algorithm we slightly process the buckets by finding the smallest and the largest element of the bucket in time linear in the number of elements in that bucket.

Then, in order from the first bucket $[0, 1/n)$ to the last bucket $[(n-1)/b, 1]$, we place all elements of each bucket into an output array, always starting with the smallest and finishing with the largest element in that particular bucket. This is all done in $O(n)$, so the entire algorithm runs in $O(n)$ time.

To show that the resulting sequence $y_1, y_2, \ldots, y_n$ satisfies $\sum_{i=2}^{n}|y_i - y_{i-1}| < 2$ we split this sum into two sums

$$\sum_{i=2}^{n}|y_i - y_{i-1}| = \sum\{|y_i - y_{i-1}| \ : \ y_{i-1}, y_i \text{ belong to the same bucket}\}$$
$$+ \sum\{|y_i - y_{i-1}| \ : \ y_{i-1} \text{ belongs to a bucket preceding the bucket containing } y_i\}$$

Since $|y_i - y_{i-1}| < 1/n$ whenever $y_{i-1}$ and $y_i$ belong to the same bucket, and since there can be at most $n-1$ such pairs with consecutive indices, the first sum is smaller than $(n-1)/n$. Since we always start with the smallest element of a bucket and finish with the largest one, one can easily see that pairs $(y_{i-1}, y_i)$ from the second sum form a set of disjoint intervals $[y_{i-1}, y_i]$, all contained in the interval $[0, 1]$ and thus, the second sum, being the total length of these intervals, is at most 1. Thus sum of these two sums is less than $(n-1)/n \quad +1 < 2$.

5. 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 does not know anyone except herself/himself. (Of course everyone knows herself/himself). 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).

**Solution:** Assume the people are numbered 1 to $n$.

(a) We observe that **at most** one person can be a celebrity. We proceed as follows. First, we find a single candidate, i.e., the only person person who **could** be a celebrity. Initially our candidate $c$ is person 1. Then, for each person $i$ from 2 to $n$, we ask if $c$ knows $i$. If $c$ does, then $c$ cannot be a celebrity (for they know someone else) and $i$ is our new candidate. If $c$ doesn't, then $i$ can't be a celebrity and $c$ remains our candidate. Thus, after asking $n-1$ questions we can establish that only the final $c$ is possibly a celebrity.

It is possible that $c$ also isn't a celebrity, 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.

**Solution:** 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 at least $\lfloor \log_2 n \rfloor$. 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. *Note this is less than the required bound, $3n - \lfloor \log_2 n \rfloor - 2$!*

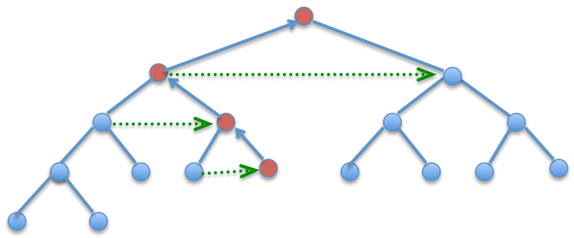

Figure 1: Here $n = 9$; thus, $m = \lfloor \log_2 n \rfloor = 3$ and we add two children to $n - 2^m = 9 - 2^3 = 1$ leaf of a perfect binary tree with 8 leaves. Thus obtained tree has 9 leaves. Potential celebrity is in red. Note that we do not have to repeat 3 questions represented by the green arrows.

6. 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 instance, "Alice Bob" could mean 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 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 7 marks, and an $O(n)$ algorithm earns an additional 3 marks.

   **Solution:** Unfortunately, we forgot to specify if students can vote for themselves. If this is allowed, then we can immediately resolve any pieces of paper where a student's name appears twice: they must have voted for themselves.

   (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 vote. 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.