Sample solutions to assignment 1

1. **[20 marks]** You're given an array A of n integers, and must answer a series of n queries, each of the form: "How many elements a of the array A satisfy $L_k \leq a \leq R_k$?", where L_k and R_k $(1 \leq k \leq n)$ are some integers such that $L_k \leq R_k$. 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_k; and
   - First element with value **strictly greater** than R_k.

   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_k you have to see if the preceding element is smaller than L_k; if it is also equal to L_k, you have to continue the binary search (going towards the smaller elements) until you find the first element equal to L_k. Similar observation applies if your binary search hits R_k.

2. **[20 marks, both (a) and (b) 10 marks each]** 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.

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

   Solution:

   (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.

3. **[20 marks, both (a) and (b) 10 marks each; if you solve (b) you do not have to solve (a)]** 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 themselves. You may assume everyone knows themselves.

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 $3n - \lfloor \log_2 n \rfloor - 2$ such questions, even in the worst case.

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. Note also that $3n-4$ questions suffice, because we can reuse the answer to at least one question the potential celebrity was asked during the initial search for a potential 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 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,* $3n - \lfloor \log_2 n \rfloor - 2$!

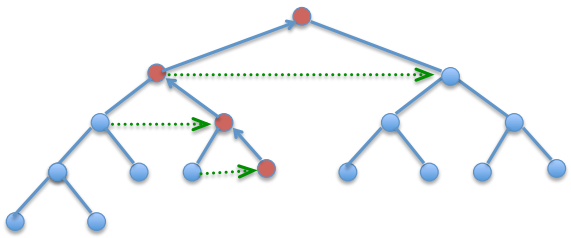

Figure 1: Here $n = 9$; thus, $m = \lfloor \log_2 n \rfloor = 3$ and we add two children to $n - 2^m = 9 - 2^3 = 1$ leaf of a perfect binary tree with 8 leaves. Thus obtained tree has 9 leaves. Potential celebrity is in red. Note that we do not have to repeat 3 questions represented by the green arrows.

4. **[20 marks, each pair 4 marks]** 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}	$2^{n/100}$
$\sqrt{n}$	$2^{\sqrt{\log_2 n}}$
$n^{1.001}$	$n \log_2 n$
$n^{(1+\sin(\pi n/2))/2}$	$\sqrt{n}$

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 $n^{100} < c2^{n/100}$ for some positive c and all sufficiently large n. But, since the log function is monotonically increasing, 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 $\log \sqrt{n} = \log n^{1/2} > \log c + \log_2 2^{\sqrt{\log_2 n}} = \log c + \sqrt{\log_2 n}$. 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

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

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

5. **[20 marks, each recurrence 5 marks)** 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 $1 \le n(2+\sin n) \le 3$. 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 $\sqrt{n} = n^{.5} = O(n^{0.9}) = O(n^{\log_b a - .1})$ 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: $a\, f(n/b) = 8f(n/2) < cf(n)$ 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 $8\left(\frac{n}{2}\right)^{\log(n/2)} < 1/2 n^{\log n}$ 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

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