This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Question 1

Your friend attempted to send you an array of n bits, starting with a 0, and alternating between 0s and 1s. However, due to the network being unreliable, one of the bits was not sent. You received the array A of length $n-1$, and you would like to determine which bit was not sent.

1.1 **[2 marks]** Suppose $n = 8$, and you received the array $A = [0, 1, 0, 1, 1, 0, 1]$. Identify the bit that was not sent by its 1-based index in the array your friend tried to send you.

Bit 5 was not sent.

1.2 **[5 marks]** Suppose $n = 20$, and you received an array in which $A[10] = 0$. Which of the twenty bits could have been omitted from the original array? Provide reasoning to justify your answer.

If the missing bit occurred at index 11 or later, then the first ten bits would be unaffected, i.e. $[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]$. However, $A[10] = 0$, so the missing bit must have instead occurred at index 10 or earlier.

1.3 **[13 marks]** Design an algorithm which runs in $O(\log n)$ time and finds the index of the missing bit.

Before the missing bit, the bit in position i will be 0 if i is even, and 1 if i is odd. After the missing bit, the bit in position i will be 1 if i is even, and 0 if i is odd. Thus the answer is the first index i such that the value at position i is not $i \bmod 2$, which we can binary search for. At each stage, we test the middle bit; if it is 'correct' then we recurse right, if not then we recurse left. This recursion ends when our range narrows to a single element, from which the missing index will either be the first element, the last element, or at the element where the search concludes.

This is a single application of binary search, and hence runs in $O(\log n)$ time.

Question 2

You are given an array A of n integers. You are required to find indices i, j, k (not necessarily distinct) such that $A[i] + A[j] = A[k]$, or return that no such indices exist.

Design algorithms which solve this problem and run in:

2.1 **[6 marks]** worst case $\Theta(n^2 \log n)$ time.

Construct pairs $(A[i], i)$ and mergesort by $A[i]$ in $O(n \log n)$ time. Now iterate over pairs of indices (i, j), and for each pair perform a binary search to find whether $A[i] + A[j]$ is a value in the original array, and if so output i, j and the index corresponding to this value. The binary search inside of the two nested loops runs in $O(\log n)$, giving an overall worst case complexity of $O(n^2 \log n)$.

Alternatively, put all entries of A in a self-balancing binary search tree, then iterate over all pairs (i, j) and each time query the tree for $A[i] + A[j]$. Again, the indices must be stored alongside the values in order to output the answer.

2.2 **[6 marks]** *expected* $\Theta(n^2)$ time.

Create a hash table in $O(n^2)$, where each key is an integer k and its value is an index i such that $A[i] = k$. This allows us to check whether an integer is in the array, and if so, an index at which it appears, in expected $O(1)$ time. Now iterate over all possible pairs i, j with two nested loops, and check if $A[i] + A[j]$ is a key in the hash table. If we ever find a key in the hash table, we return i, j and the value for that key, otherwise no such indices exist. The inside of the two nested loops runs $O(n^2)$ time, each in expected constant time, and so the overall complexity is expected $O(n^2)$.

2.3 **[8 marks]** worst case $\Theta(n^2)$ time.

We first use mergesort to sort the array in increasing order, keeping track of the original indices so we can convert them back at the end (as in 2.1). Now, suppose we fix k, and are looking for indices i, j such that $A[i] + A[j] = A[k]$. We start with $i = 1$ and $j = n$, and perform the following step while $i \leq j$:

- if $A[i] + A[j] = A[k]$, report i, j and k as the answer, and exit;
- if $A[i] + A[j] < A[k]$, increment i;
- if $A[i] + A[j] > A[k]$, decrement j.

This is correct because if we do not find an answer, for each index i we will have found $j \in [0, n]$ where $A[i] + A[j] < A[k]$ and $A[i] + A[j + 1] > A[k]$ (where by convention the inequality holds if an index is invalid), meaning that indeed no answer exists. This step terminates after $O(n)$ steps, since we can only increment i and decrement j a total of $O(n)$ times before they fall outside the range of array indices. So checking each value of k from 1 to n takes $O(n)$, resulting in an overall worst case complexity of $O(n^2)$.

Question 3

You are given an array A containing each integer from 1 to n exactly once. Your task is to compute $f(A)$, the sum of $\max(A[\ell..r])$ over all pairs of indices (ℓ, r) such that $\ell \leq r$.

3.1 **[2 marks]** Suppose $n = 3$ and the array is $A = [2, 1, 3]$. Determine the value of $f(A)$.

In this case,

$$\begin{aligned} f(A) &= \max([2]) + \max([1]) + \max([3]) + \max([2,1]) + \max([1,3]) + \max([2,1,3]) \\ &= 2 + 1 + 3 + 2 + 3 + 3 \\ &= 14. \end{aligned}$$

For 3.2 and 3.3, suppose i and j are indices such that $1 \leq i \leq j \leq n$, and let $g(i, j)$ be the number of subarrays $A[\ell..r]$ where $r > j$ and the maximum value is $A[i]$.

3.2 **[4 marks]** For a given pair of indices (i, j), under what conditions is $g(i, j)$ nonzero? In other words, what is the criterion for $A[i]$ to be the maximum of some subarray with its right endpoint at an index greater than j?

For $A[i]$ to be the maximum of some subarray with its right endpoint at an index greater than j, this array must include the index i and the index $j + 1$, so $A[i]$ must at least be the maximum of the subarray starting at index i, and ending at index $j+1$, that is, the subarray $A[i..(j + 1)]$. So the criterion is that $A[i] = \max(A[i..(j + 1)])$. Equivalently, the criterion is

$$A[i] > A[i + 1] \text{ and } A[i] > A[i + 2] \text{ and } ... \text{ and } A[i] > A[j + 1].$$

3.3 **[6 marks]** Given an index j, design an algorithm which runs in $O(n)$ time and determines the values of $g(i, j)$ for all $i < j$.

We start by determining $g(i, j)$ for $i = j$, then calculate the values of $g(i, j)$ with decreasing values of i until we reach $i = 1$. From 3.2, $g(i, j)$ will be 0 unless $A[i]$ is larger that $A[i + 1], \ldots, A[j + 1]$, so we maintain a variable M (the "maximum seen so far"), which starts as $A[j + 1]$, and should be updated to $\max(M, A[i])$ whenever we finish processing any value of i and are about to decrement it. Now we only need to consider the cases where $A[i] > M$, as otherwise $g(i, j) = 0$.

The subarrays $A[\ell..r]$ for which $A[i]$ is the maximum are precisely those for which $A[i] = \max(A[\ell..i])$ and $\max(A[j..r]) < A[i]$. These two conditions are independent, so for each i, if we can determine the number of values of ℓ satisfying the first condition, and the number of values of r satisfying the second, we can multiply them to get $g(i, j)$.

The valid indices for ℓ will be $i, i - 1, ...$ up to and excluding the first index $k < i$ such that $A[k] > A[i]$. This can be precomputed for all relevant values of i in $O(n)$ by iterating over the array from right to left and keeping track of the greatest element seen so far and its index. Since we only care about whenever a new maximum is seen, this means for each maximum we can keep track of when it was overtaken, allowing us to determine the number of valid values of ℓ for each i in $O(n)$ overall.

To determine the number of valid values for r, that is, the number of values for r such that $\max(A[j..r]) < A[i]$, we will simply keep track of the first index $k \geq j$ such that $A[k] > A[i]$, which we also allow to be $n + 1$ if no actual such index exists. Since the relevant values of i are processed in increasing order of $A[i]$, k can only increase, and so to keep track of k

whenever i changes, while $A[k] < A[i]$ (or until $k = n+1$) we increment k, which can happen at most $O(n)$ times overall. The valid values of r are exactly those between $j+1$ and $k-1$ inclusive, and so there are $k-j+1$ valid values of r.

3.4 **[8 marks]** Design an algorithm which runs in $O(n \log n)$ time and determines the value of $f(A)$.

We use a divide and conquer approach. In the definition of $f(A)$, let n be the length of A, let $m = \lfloor n/2 \rfloor$, and partition the pairs of indices into three sets: those where $r \leq m$, those where $l > m$, and those where $l \leq m < r$. The first two cases are instances of the problem with half the array size, since in these cases the left half is independent of the right half.

For the final case, the subarrays can be calculated using the function g. In particular, $A[i] \cdot g(i, m)$ gives the sum of maximums of subarrays ending strictly past m for which $A[i]$ is the maximum, and so $A[1] \cdot g(1, m) + A[2] \cdot g(2, m) + ... + A[m] \cdot g(m, m)$ will calculate the the sum of $\max(A[\ell..r])$ where $l \leq m < r$ and the index of the maximum value is less than or equal to m. For the cases where the maximum value lies to the right of m, we use a similar function g', where $g'(i, j)$ is the number of subarrays $A[\ell..r]$ where $\ell < j$ and the maximum value is $A[i]$, which is computed very similarly to $g(i, j)$. This contributes $A[n] \cdot g'(n, m+1) + A[n-1] \cdot g'(n-1, m+1) + ... + A[m+1] \cdot g'(m+1, m+1)$ more to this case. Note that from 3.3, the values of $g(i, m)$ and $g'(i, m+1)$ over all i can be determined in $O(n)$, allowing the contribution to the answer for this case to be calculated in linear time.

The base case is that if $|A| = 1$, $f(A) = A[1]$.

The recurrence relation is $T(n) = 2T(n/2) + \Theta(n)$. The critical exponent is $c^* = \log_2 2$, so case 2 of the Master Theorem applies, and therefore the asymptotic solution is $T(n) = \Theta(n \log n)$ as required.

Question 4

Your friend has constructed an array A of n distinct integers, where $n \geq 2$. However, you cannot access the elements of the array directly; instead, they instead only allow you to ask questions of the form "What is the maximum value amongst $A[l], A[l+1], \ldots, A[r-1], A[r]$?", where you may choose any valid indices l and r such that $l \leq r$. You may assume any questions you ask are answered in constant time.

Your goal is to determine the value of the **second largest** element in the array.

4.1 **[2 marks]** How can you find the value of the largest element in the array using only one question?

Query the range $[1, n]$.

4.2 **[2 marks]** If you know that the largest element occurs at index i, how could you then find the value of the second largest element using only two questions?

Query the ranges $[1, i-1]$ and $[i+1, n]$ and take the maximum.

4.3 **[11 marks]** Design an algorithm which runs in $O(\log n)$ time and determines the value of the second largest element in the array.

Let $Q(l, r)$ be the maximum value amongst $A[l], A[l+1], \ldots, A[r-1], A[r]$, which can be found in constant time by asking a question.

Method 1: First find M, the value of the largest element in the array, which is equal to $Q(1, n)$. Because the elements are distinct, there will be exactly one index k such that $A[k] = M$. We have that k is the least index i such that $Q(1, i) = M$, since ranges ending before index k will have maximum below M, while ranges including index k will have a maximum of at least $A[k] = M$. We can thus perform a binary search to determine k in $O(\log n)$. Our final answer is $\max(Q(1, k-1), Q(k+1, n))$ (ignoring the first query if $k = 1$, and ignoring the second query if $k = n$), which is the maximum of the array excluding $A[k]$.

Method 2: This is the same as Method 1, except in the details of finding k. We initially know that $k \in [1, n]$. Suppose at some point we know that $k \in [l, r]$, where $l \neq r$, then we let $m = \lfloor (l + r)/2 \rfloor$. Then if $Q(l, m) = M$, we know that k is in the left half so $k \in [l, m]$, and otherwise we know that $k \in [m+1, r]$. Since the range size halves each time, this runs in $O(\log n)$.

4.4 **[5 marks]** Now your friend imposes an extra restriction: for each question you ask except the first, the value of $r - l$ should be no larger than the value of $r - l$ in the previous question. Subject to this restriction, design an algorithm which runs in $O(\log n)$ time and determines the value of the second largest element in the array.

You may choose to skip 4.3, in which case we will mark your submission for 4.4 as if it was submitted for 4.3 also.

First find M, the value of the largest element in the array, which is equal to $Q(1, n)$. Maintain a range $[l, r]$, initially $[1, n]$, that contains this largest element, and also keep track of the maximum value seen so far except for M, initially $-\infty$.

While $l \neq r$, let $m = \lfloor (l + r)/2 \rfloor$, and find $Q(l, m)$. If $Q(l, m) = M$, the maximum lies in the left half, so set r to m (updating the range to $[l, m]$), and set the answer so far to the

maximum of itself and $Q(m+1, r)$ so that we can discard the right half, even if it includes the second largest element. If $Q(l, m) \neq M$, instead set l to $m+1$ (updating the range to $[m+1, r]$) and set the answer so far to the maximum of itself and $Q(l, m)$.

For each range size, we ask at most two questions: one where the $r - l$ value is half the current range size rounded up, then possibly another one where the $r - l$ value is half the current range size rounded down, meaning within these two questions, the $r - l$ value cannot increase. Since the range size always decreases, this also means the value of $r - l$ never increases between questions, as required. Finally, since the range size halves each time, this runs in $O(\log n)$.