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

Question 1 *The Cheapest Fridge*

[30 marks] Song wants to buy a fridge with volume at least V cubic centimetres. The shop sells a large variety of fridges. More precisely, for each positive integer x, the shop sells a fridge for x dollars with the following dimensions:

- width $3x$ centimetres,
- depth $2x + 1$ centimetres and
- height 2^x centimetres.

1.1 **[12 marks]** Design an algorithm which runs in $O(\log V)$ time and finds the minimum amount that Song must spend to buy a suitable fridge.

We first observe that the volume of any box can be expressed as $V_{\text{box}} = 3x(2x+1)2^x$. Thus, we can define $f(x) = 3x(2x+1)2^x$ so that $f(x)$ represents the volume of some box for a particular x. Using this reformulation, we then want to find the *smallest* x such that $f(x) \geq V$.Note that $f(x)$ is strictly increasing because it is composed of strictly increasing functions, and hence we can binary search with it.

Our algorithm works as follows. Define the interval $[0, V]$ observing that $f(0) = 0$ and $f(V) > V$. Binary search on this interval, computing $f(m)$ where m is the middle value of the interval. If $f(m) \geq V$, then we discard any value $k > m$, which refines our new interval. Similarly, if $f(m) < V$, then we discard any value $k < m$, which again refines our new interval. We repeat this process until we have one value left, in which case we terminate.

In terms of time complexity, since we can compute $f(x)$ in $O(1)$ time, we are simply performing binary search over an interval of size V which runs in $O(\log V)$ time. More formally, we can use the master theorem with the recurrence:

$$T(V) = T\left(\left\lfloor \frac{V}{2} \right\rfloor\right) + O(1).$$

The critical exponent is $c^* = \log_2 1 = 0$ which satisfies the second case of Master Theorem. Hence, and our time complexity is $T(V) = \Theta(V^0 \cdot \log V) = \Theta(\log V)$.

1.2 **[18 marks]** Design an algorithm which runs in $O(\log(\log V))$ time and finds the minimum amount that Song must spend to buy a suitable fridge.

You may choose to skip 1.1, in which case your answer to 1.2 will be marked as your answer to 1.1 also.

This is very similar to the solution for part 1. Let $\nu = \log_2 V$ and therefore we have $V = 2^\nu$. We just observe that we can reduce the size of the initial interval from $[0, V]$ down to $[0, \nu]$ as $f(\nu) = 3\nu(2\nu+1)2^\nu > V$ and $V = 2^\nu$. Binary searching on $[0, \nu]$ interval will give the same result, while meeting the new complexity requirement. We can use the master theorem with the recurrence:

$$T(\nu) = T\left(\left\lfloor \frac{\nu}{2} \right\rfloor\right) + O(1).$$

The critical exponent is $c^* = \log_2 1 = 0$ which satisfies the second case of Master Theorem. Hence, and our time complexity is $T(\nu) = \Theta(\nu^0 \cdot \log \nu) = \Theta(\log \nu) = \Theta(\log(\log V))$.

Question 2 *Counting Occurrences*

[30 marks] Answer the following:

Be very careful with the requested time bounds.

2.1 [6 marks] Let $X = [x_1, \ldots, x_n]$ be a **sorted** array of n positive integers. Design an algorithm to count the number of occurrences of each *distinct* integer in X in worst case $O(n)$ time.

We firstly note that the array is *sorted*. This tells us that all of the multiple occurrences of x_i will be grouped together. Therefore, our algorithm works as follows.

Iterate through X. We can keep track of precisely how many times an element appears by keeping track of where a new distinct integer appears. If a new distinct integer starts at index i and the next new integer begins at index j, then we know there are precisely $j - i$ many occurrences of integer x_i. The number of occurrences for each distinct integer computed with a single linear scan of X; since there are n many comparisons to make, the algorithm runs in $O(n)$ time.

2.2 [12 marks] Let $Y = [y_1, \ldots, y_n]$ be an **unsorted** array of n positive integers. Design an algorithm to count the number of occurrences of each *distinct* integer in Y in *expected* $O(n)$ time.

Denote our hash table as H and begin by initialising it to be empty. For each element y_i in Y, we check if exists in H. If not, we add it to H as a key, and give it a count of 1. Otherwise we get and increment its count by one. Once we finish performing the linear scan of Y, each key of H will be the distinct integer of Y and the count is the number of occurrences of the distinct integer.

This runs in $O(n)$ *expected* time because, for each element in Y, we perform an expected $O(1)$ insertion. Since there are n elements, we perform n $O(1)$ expected time operations and so, our final time complexity is $O(n)$ in the expected time.

2.3 [12 marks] Let $Z = [z_1, \ldots, z_n]$ be a **sorted** array of n positive integers. You are also given an integer $k \leq n$, the number of *distinct* integers in this array. Design an algorithm to count the number of occurrences of each distinct integer in Z in worst case $O(k \log n)$ time.

Note that Z is sorted, so our approach here will be to binary search for the last occurrences of each distinct integer. The first distinct integer occurs at index 1. To find the last occurrence of the integer z_1, we perform a binary search on Z. If $z_i > z_1$, then we strictly consider the subarray $[z_1, \ldots, z_{i-1}]$. If $z_i = z_1$, then we consider the subarray $[z_{i+1}, \ldots, z_n]$. We continue this process until we find the last occurrence of z_1 – say, at index k. This will tell us precisely how many occurrences of z_1 are present in the array Z and that the next distinct integer occurs on index $k + 1$. We repeat this process for each distinct integer, discarding all of the integers previously found.

To argue the time complexity, we observe that we are performing a binary search on Z for *each* distinct integer. The time complexity for each binary search is $O(\log n)$ and since there

are k distinct integers, we perform k $O(\log n)$ operations which gives us the desired time complexity of $O(k \log n)$.

Question 3 *Counting Inversions Between Arrays*

[20 marks] Let A and B be two arrays of length n, each containing a random permutation of the numbers from 1 to n. An inversion between the two permutations A and B is a pair of values (x, y) where the index of x is less than the index of y in array A, but the index of x is more than the index of y in array B.

Design an algorithm which counts the total number of inversions between A and B that runs in $O(n \log n)$ time.

Observation: For any values x and y, swapping x and y in both arrays does not change the number of inversions. This is because when we swap x and y in both arrays, the order of x and y change in both arrays, which means the inversion relationship between them would preserve. For example, suppose x and y are inverted in arrays A and B, where the order of x, y is (x, y) in A and (y, x) in B. After swapping, the order of x, y becomes (y, x) in A and (x, y) in B and they are still inverted.

For simplicity, let's perform replacements such that A is sorted, that is, $A = [1, 2, \cdots, n]$. Then the number of inversions between the permutations is the number of values (x, y) such that $x < y$ and $B[x] > B[y]$. Apply standard divide and conquer inversion counting gives us the time complexity of $O(nlogn)$.

Proof. Suppose the index of x in A is $A[x]$, the inversion of A and B can be found by looking at the sign of $(A[x]–A[y])(B[x]–B[y])$; if $(A[x]–A[y])(B[x]–B[y]) < 0$ then A and B are inverted, otherwise if $(A[x]–A[y])(B[x]–B[y]) > 0$, then A and B are not inverted. Doing the replacement means we assign $A[x]$ with the original value of $A[y]$ and $A[y]$ with the original value of $A[x]$, and same for $B[x]$ and $B[y]$. Then the equation becomes $(A[y]–A[x])(B[y] – B[x]) = -(A[x]–A[y]) \times -(B[x]–B[y]) = (A[x]–A[y])(B[x]–B[y])$, which is the same as before swapping.

Question 4 *Red & Yellow Flowers*

[20 marks] You are given an array of n flowers that are either red or yellow. Your goal is to find the number of subarrays where there are more red flowers contained within the subarray, than there are yellow flowers outside the subarray.

Subarrays must be contiguous.

4.1 [6 marks] Describe a method that runs in $O(n)$ time, which pre-processes the input array such that you can then calculate the number of red flowers within any contiguous subarray in $O(1)$ time.

We want to create a prefix sum which counts the number of red flowers in a given subarray. Begin by creating an array `is_red` which mirrors the given array, except we'll use 1 to represent red flowers and 0 to represent not red flowers. Using `is_red`, we then create a prefix sum array that we'll denote as `count_red`.

For instance, if

`input`	=	[		Y,	R,	R,	Y,	Y,	R,	R,	R	]
`is_red`	=	[		0,	1,	1,	0,	0,	1,	1,	1	]
`count_red`	=	[	0,	0,	1,	2,	2,	2,	3,	4,	5	]

Then, to calculate the number of red flowers within any subarray $[l..r]$, we can simply calculate

$$\texttt{num\_red(l, r)} = \texttt{count\_red}[r] - \texttt{count\_red}[l-1].$$

4.2 **[14 marks]** Design an algorithm that achieves your goal in $O(n \log n)$ time.

Solution 1:

First note that if you have a subarray which satisfies the condition, you can always extend the subarray while maintaining the condition. This is because extending the subarray will never cause the number of red flowers inside the array to decease, nor the number of yellow flowers outside the array to increase.

Using the method in Question 4.1, we can calculate the number of red flowers within any subarray $[l..r]$ using `num_red(l, r)`. Similarly, create another prefix sum array to calculate `num_yellow(l, r)`, the number of yellow flowers within any subarray $[l..r]$. Note that this calculation also takes $O(1)$.

Then, for every element in the given array, we want to find the smallest subarray that begins at that element which satisfies the condition. This is because every largest array will also satisfy the condition. We can do this using binary search, with pivot condition being:

$$\texttt{num\_red(l, m)} > \texttt{num\_yellow(1, l - 1)} + \texttt{num\_yellow(m + 1, n)}$$

where m is our pivot index. Once the condition is met, we add to our count the $n - m$.

In terms of time complexity, we perform n binary searches, each of which takes $O(\log n)$ since the pivot condition can be calculated in $O(1)$. Hence the total time complexity is $O(n \log n)$.

Solution 2:

Let T_{yellow} denote the total number of yellow flowers.

Notice that the problem can be reduced as follows:

Count the number of subarrays $[l..r]$, such that

$$\begin{aligned} & \texttt{num\_red(l, r)} > T_{yellow} - \texttt{num\_yellow(l, r)} \\ \implies & \texttt{num\_red(l, r)} + \texttt{num\_yellow(l, r)} > T_{yellow} \\ \implies & \text{size of } [l..r] > T_{yellow} \\ \implies & r - l + 1 > T_{yellow} \end{aligned}$$

Which means that we're really just looking for the number of subarrays with length greater than T_{yellow}.

We know that there are n subarrays of length 1; $n-1$ subarrays of length 2; $n-2$ subarrays of length 3; and so on. In general, the number of subarrays of length m is $n - m + 1$. Since we know T_{yellow}, we can use the sum of an arithmetic series to find the answer in $O(1)$.

$$\texttt{subarrays of length n} + \cdots + \texttt{subarrays of length } (T_{yellow} + 1)$$

$$= (n - n + 1) + \cdots + (n - (T_{yellow} + 1) + 1)$$
$$= (1) + (2) + \cdots + (n - T_{yellow})$$
$$= \frac{(n - T_{yellow})(n - T_{yellow} + 1)}{2}$$

You may also optionally observe that $n - T_{yellow} = T_{red}$.

In total, we can calculate T_{yellow} in $O(n)$ using a linear search and calculate the sum of the arithmetic series in $O(1)$, for a total time complexity of $O(n)$.