4(c) Algo: 1. Given the array A,
sum all its elements → SUM

2. Sum of numbers from [a, a+n]
$= a(n+1) + \frac{n(n+1)}{2}$ → Total

Missing element = Total − SUM
Answer

Complexity → O(n) [Summing the elements of A]
Space Comp. → O(1)

[3 marks]

4(a)
(i) BST [BST: O(log n); Heaps: O(n)]
ii) same O(log n)
iii) same O(log n)
iv) BST [BST: O(log n); Heaps: O(n)]

3 marks
−1 for each incorrect answer.
0 if all incorrect

4(b) $\sum_{i=1}^{n} h_i$: Done in the class $= O\left[\sum_{i=1}^{n} \text{level}[i]\right]$ level[i] → length of the path to inorder successor.

$= O(\#\text{edges}) = O(n)$

O(n) or Θ(n) is acceptable [3 marks]

Only 1 mark if complexity is given without the explanation.

4(d) Idea: Find all elements larger than x, store in a new array. Heapify this array (min-heap)

B ← output array Count = 0
H ← Heap Size

outer call: larger (x, 1, H, size)

function: larger (x, pos, H, size)

{
if (pos > size) return;
if (H[pos] <= x) return; // all descendants will also be $\leq x$
B[count] = H[pos]; count++;
larger (x, 2*pos, H, size);
larger (x, 2*pos+1, H, size);
}

→ 5 marks for pseudo code
→ 1 mark for complexity
for an O(n) algo
⇒ 3 marks may be given if correct

Complexity
→ O(n)

Consider the knapsack problem where you are allowed to take items more than once, have a total capacity of 10 and have the following items available:

Item	*Weight*	*Value*
A	*2*	*4*
B	*3*	*5*
C	*4*	*9*
D	*5*	*10*
E	*7*	*16*

What is the best combination of items to take?

Running the dynamic program covered in class we get the following table:

Capacity	0	1	2	3	4	5	6	7	8	9	10
Value	0	0	4	5	**9**	10	13	16	**18**	20	**22**

Working backwards, we find that 22 can be achieved either as a copy of A and items of weight 8 or item C plus items of weight 6. We consider the former case. The 18 for items of weight 8 can only be achieved by a copy of item C plus items of weight 4, and the optimum there can only be achieved by a single copy of C. Thus, the best combination of items is one copy of A and two copies of C.

Rose and Greg are stranded on opposite sides of Graphania. The map of Graphania is given by a directed graph where each edge in the graph takes 1 minute to traverse. Each minute each of the two travellers can either traverse an edge or stay where they are. Give an algorithm that given the map of Graphania along with the initial locations of Rose and Greg computes the minimum amount of time required for them to meet at the same location. For full credit, your algorithm should run in linear time or better.

We first run BFS starting from Rose's location computing distances $d_R(v)$ for each vertex v. We then run BFS from Greg's location computing distances $d_G(v)$. For each vertex v we then compute $\max(d_R(v), d_G(v))$ and return the smallest value obtained.

The runtime is $O(|V| + |E|)$ to run BFS twice and then $O(|V|)$ to compute $\max(d_R(v), d_G(v))$ for each vertex and find a minimum. So the runtime is $O(|V| + |E|)$.

Give an algorithm that given two sets A and B each consisting of n real numbers finds the minimum distance between a point of A and a point of B. In particular, it computes the minimum possible value of $|a - b|$ over all pairs $a \in A$ and $b \in B$. For full credit, your algorithm should run in time $O(n \log(n))$.

For example, if $A = \{0, 1, 8\}$ and $B = \{11, 3, 5\}$, then the closest pair would be 1 and 3 with distance 2.

The algorithm is the following:

```
Sort A U B keeping track of which point came from which set.
For each pair of consecutive elements x,y in the sorted list
  if x and y came from different sets, record |x-y|.
Return the smallest number recorded.
```

The runtime of the first step here is $O(n \log(n))$. The second and third steps are clearly linear time, so the full runtime is $O(n \log(n))$.

To show correctness, it is not hard to see that the closest such pair of points a, b will be consecutive points in the sorting of $A \cup B$. This is because otherwise there is some point in between them, either an $a' \in A$ or a $b' \in B$, and in that case either $|a' - b|$ or $|a - b'|$ will be less than $|a - b|$. Therefore, we only need to check all pairs of consecutive points, which we do.