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

Question 1

You have recently been hired to work for a large social media company. The company has staff shortages due to a recent firing spree, and you have been tasked with designing the recommendation algorithm used on the front page! After thinking for several days, you have figured out a way to measure a user's popularity, and now need to design an efficient algorithm to calculate it.

For a given user with n posts, you have access to an integer array $V = [v_1, v_2, \ldots, v_n]$, where v_i is the number of views the user's i^{th} post has in thousands. Since these view counts are used to order posts on a user's profile page, they are already sorted, in *descending* order. The popularity of a user is the largest value of p such that the user has at least p posts that each have at least p thousand views.

For example, suppose a user has $n = 6$ posts, and a view count array $V = [7, 6, 5, 4, 4, 1]$. Then they have at least 4 posts each with at least 4,000 views (indeed, they have five such posts), so $p = 4$ satisfies the criterion. You can confirm that the criterion is also satisfied for $p = 3$, but it is not satisfied for $p = 5$ since there are only three qualifying posts. The user's popularity index is in fact 4, because $p = 4$ is the largest value for which the criterion holds.

1.1 [7 marks] Before you are given full access to the codebase, you need to try and implement your new popularity measure on top of the existing code. Given the integer n and the integer array V for a user as well as an integer value $p \leq n$, you must determine whether that user's popularity is *at least* p. This snippet of code runs every single time the page gets loaded, so it needs to be very efficient.

Design an algorithm that runs in $O(1)$ time and checks whether a user's popularity is at least p.

Using 1-indexing, we check $V[p]$.

[A] If $V[p] < p$, then since the array is sorted in descending order, there are less than p posts for this user with at least p thousand views, so decide no.

[B] If $V[p] \geq p$, then there are at least p posts with at least p thousand views, so it is possible for the user to have popularity p, or possibly higher. In this case, decide yes.

The algorithm involves one random access into an array and one integer comparison, so the time complexity is constant.

1.2 [13 marks] The code you modified was written by a now-fired intern, and would simply try every value of p until it found one that worked, for every user. Now that the company has grown to serve millions of users, your boss has demanded that you develop a faster algorithm. Given the integer n and the integer array V for a user, you must efficiently compute the user's popularity index p.

Design an algorithm that runs in $O(\log n)$ time and computes p for a given user.

We must find the largest p such that $V[p] \geq p$. We will use binary search, with an initial search space of $[0, n]$, and maintaining a variable q to store the tentative answer, i.e. the largest known q satisfying $V[q] \geq q$. At each stage, run the algorithm from question 1.1 on the midpoint of the search space (say m).

[A] If the algorithm decides that the user's popularity is at least m, then update q to m.

The actual popularity might be even higher than m, so continue searching on higher popularity values.

[B] Otherwise, the popularity must be less than m, so continue searching on lower popularity values.

Once the algorithm reduces the search space to zero, variable q will store the answer, since it satisfies $V[q] \geq q$ and all larger values do not.

Instead of maintaining q, one could instead recurse until the search space is a single integer p, at which point either $p-1$ or p is the user's popularity value. To see this, notice that if the popularity value is used to decide between lower and higher popularity, then the above algorithm will always choose to search for higher values (assuming there are any left to search), and all following decisions will look for lower values, eventually stopping on the popularity value just above the true value. Using the algorithm in question 1.1 once more on p, either p is the answer, or otherwise, $p-1$ must be the answer. Binary search works here due to the monotonic nature of the searching function.

The search range is initially $[0, n]$ and halves at every stage of the binary search, so there are $O(\log n)$ stages. Each stage is completed in $O(1)$ using the algorithm developed in 1.1. Therefore the time complexity is $O(\log n)$.

Question 2

DAC Investments is a finance company that has a team of k bankers. The ith banker manages $M[i]$ dollars worth of investments, and has a performance rating of $P[i]$. The bankers are indexed in *decreasing* order of performance rating.

The profitability of DAC Investments is given by

$$(M[1] + \ldots + M[k]) \times \min(P[1], \ldots, P[k]).$$

For example, if $M = [4, 8, 2, 7, 1]$ and $P = [9, 6, 3, 3, 2]$, then the profitability is

$$(4 + 8 + 2 + 7 + 1) \times \min(9, 6, 3, 3, 2) = 22 \times 2 = 44.$$

2.1 **[2 marks]** DAC Investments has recently received a letter from the CEO, advising them that his nephew is looking for work. Although the nephew's performance rating is no better than anyone in the team, DAC Investments have been *strongly encouraged* to hire him in place of one of the current team members.

You are given integer arrays $M[1..(k+1)]$ and $P[1..(k+1)]$, representing the amounts managed and performance ratings respectively. In each array, the first k entries are for the current team members and the last entry is for the CEO's nephew. The performance ratings are listed in decreasing order.

In order to hire the CEO's nephew, DAC Investments will have to remove one of their bankers. How should they select which banker to remove, while maximising profitability?

Since the CEO's nephew has the lowest performance, selecting him fixes the minimum performance of the team. This means we can only control the total amount managed, which is maximised by removing the employee with the lowest amount managed.

For the remaining subquestions, suppose DAC Investments has been acquired by a large bank. There are now n employees available, so you are given integer arrays $M[1..n]$ and $P[1..n]$, the latter of which is again in decreasing order. From these n candidates, DAC Investments would like to select k team members.

2.2 **[10 marks]** Design an algorithm that runs in $O(nk)$ time and determines the set of k employees that maximise the profitability of DAC Investments.

See the answer for 2.3, but instead of using a heap, use a length k array and search for the employee to kick out of the team in $O(k)$ time instead. Solving the problem this way also simplifies saving the team with maximum profitability, since we can instead just copy the array in $O(k)$ time.

2.3 **[8 marks]** Design an algorithm that runs in $O(n \log k)$ time and determines the set of k employees that maximise profitability.

Begin by creating a min-heap containing the first k employees, being those with the highest performance ratings, in $O(k)$ time. The min-heap will sort based on managed amount, so popping from the heap will remove the employee in the heap with the lowest managed amount.

Keep a running sum of the k chosen employees' managed amounts, and calculate the profitability using these initial k employees, which we try to maximise. Now, iterate the remaining $n-k$ employees, and for each employee, pop the current heap root, and add the new employee, in $O(\log k)$ time. Recalculate the running sum and profitability in $O(1)$ time, using the current employee's performance (which we know to be minimal by the order of P), checking whether this is a new maximum profitability.

To recover the set of employees, one might keep an array of length n which contains the first iteration of the loop where the employee was added, and the last iteration when they got removed. Keeping track of the iteration where the maximum profitability was achieved, we can run through the array to look for the employees in the heap at that iteration in $O(n)$ time.

Overall complexity is $O(k \log k)$ initialising the heap, $(n-k) \log k$ for all the heap replacements, and $O(n)$ for the final employee recovery, totalling $O(n \log k)$.

The correctness relies on the observation of 2.1: each iteration we try to add an employee whose performance rating is the minimum amongst all previously considered employees, so we must kick out the employee with the minimum amount of managed funds. The other $k-1$ employees in the heap at any iteration will be the $k-1$ employees with maximum managed amount amongst all previously considered employees. So, on the iteration considering employee i, we will have the maximum profitability amongst all possible combinations of the first i employees that contains employee i. If the optimal solution has index i as its worst-performing employee (based on index in P), then that selection of employees will be exactly the selection of iteration i, so this algorithm will produce a globally optimal selection of employees.

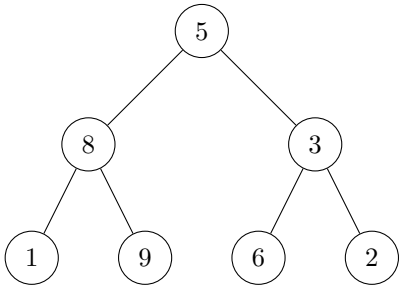
Question 3

A *traversal* of a binary tree is an order in which one can visit all the nodes.

- In-order traversal first recurses on the left subtree, then visits the root node, and recurses on the right subtree.
- Pre-order traversal first visits the root node, then recurses on the left subtree, and finally recurses on the right subtree.
- Post-order traversal first recurses on the left subtree, then recurses on the right subtree, and finally visits the root node.

For example, the tree given below has:

- in-order traversal of $[1, 8, 9, 5, 6, 3, 2]$,
- pre-order traversal of $[5, 8, 1, 9, 3, 6, 2]$, and
- post-order traversal of $[1, 9, 8, 6, 2, 3, 5]$.



3.1 **[8 points]** Given an array P that contains the pre-order traversal of a binary **search** tree of n nodes with distinct integer values, design an algorithm that runs in $O(n)$ time and computes the post-order traversal of the tree.

In order to produce the post-order traversal, we will first construct the original binary search tree. First note that the first element of the pre-order traversal is the root of the tree. We wish to insert nodes in the same order as the array P. To tell if a node can be inserted in the current position, we need to ensure it satisfies the range property of the BST. That is, it must be smaller than any element to its right, and larger than any element to its left. The method proceeds as follows:

1. Initially, set $\min = -\infty$, $\max = \infty$.
2. Keep track of current progress through the P array with variable p. Initially $p = 1$.
3. If $\min < P[p] < \max$, then create a new node with value $P[p]$ and let $p = p+1$ since the current element of P has now been used. If not, then simply return from the function.
4. Recurse to the left child with max equal to the value of the node just created.
5. Recurse to the right child with min equal to the value of the node just created.
6. Return the node just created.

Once the tree has been constructed, the post-order traversal can be easily obtained by recursively traversing the tree in the required order.

Correctness:
Nodes are inserted in the order of root, left, right just as they are in the pre-order traversal. Since the pre-order traversal of a BST is unique, and the constructed tree has the same pre-order traversal P, then it must be the same tree.

Time complexity:
The recursion visits each of the n values of the P array precisely once and does at most $O(1)$ work for each value (creating a new node, recursing into the left and right child slots). This means the time complexity is simply $O(n)$ as was required.

3.2 **[12 points]** A binary tree is said to be *height-balanced* if:

- it has zero or one nodes, or
- the heights of its left and right subtrees differ by at most one, and both subtrees are height-balanced.

Given two arrays I and P which contain the in-order and pre-order traversals of a height-balanced binary tree of n nodes with distinct integer values, design an algorithm that runs in $O(n \log n)$ time and computes the post-order traversal of the tree.

As with the previous part, in order to produce the post-order traversal, we will first construct the original binary tree. First we make two important observations:

1. The first element in the pre-order array is always the root of the tree.
2. In the in-order array every element to the left of the root belongs to the left subtree, and every element to the right of the root belongs to the right subtree.

These observations motivate a divide & conquer approach, where we split the in-order array up into left and right components about the root node. The method proceeds as follows:

1. Keep track of current progress through the P array with variable p. Initially $p = 1$.
2. Create a new node, root, with value $v = P[p]$. Let $p = p + 1$ since the current element of P has now been used.
3. Find the index i of the value v in I.
4. Recurse to assign the left child of root with $I_L = I[L, \ldots, i-1]$, where L is the index of leftmost element of the current root.
5. Recurse to assign the right child of root with $I_R = I[i+1, \ldots, R]$, where R is the index of rightmost element of the current root.

Once again, since the tree has been constructed, the post-order traversal can be easily obtained by recursively traversing the tree in the required order.

Correctness:
Correctness of the splitting procedure follows from the observation that every element to the left of the root belongs in the left subtree, and every element to the right of the root belongs in the right subtree. To see that the root values in each recursive call are chosen in the correct order from P, see that the recursive structure follows the same call order as a pre-order traversal. That is, we first assign a value to the root, then recursively to the left child, then recursively to the right child.

Time complexity:
The work done to split the subproblems is $O(n)$ as we must find the index of v in I. The work to recombine is $O(1)$ as we simply assign the left and right children. Since the tree is height balanced, the difference in the number of nodes from the left subtree to the right subtree can be no worse than a ratio of 2:1. This means the time complexity is no worse than the solution to the recursion $T(n) = T(n/3) + T(2n/3) + O(n)$. As seen in the lectures, this recursion has a solution that is $O(n \log n)$. Finally, the time complexity of the post-order traversal is $O(n)$, so the overall time complexity is $O(n \log n)$ as required.