(a)

General Idea and Code

We will use the `MomSelect` and `Partition` in the problem. (They are listed in the textbook). We will write `R ← [1..n]` to declare an array of size `n`. We use `|A|` to denote the size of an array `A`.

The general idea is that for the list of h indices `H[1..h]` we could choose the median of the `H`. Once we find the median of `H` (call it `m`), we use the `MomSort` to find the corresponding `m`th element in the `A`, and the partition the `A` to have two parts `left` and `right` that is smaller and bigger than the `m`th element, respectively. Then it's true that the indices smaller than `m` is all in the `left` part and indices that is larger than `m` is all in `right` part, this effectively divide and conquer the problem. The pseudocode is shown below:

```
SelectK(A[1..n], H[1..h]):
    R ← [1..h]
    SelectKRecursive(A, H, R)
    return R

SelectKRecursive(A[1..n], H[1..h], R[1..h]):
    medianH ← MomSelect(H, ceil(h/2))
    hElement ← MomSelect(A, medianH)

    R[medianH] ← hElement

    leftA, rightA ← Partition(A, hElement)
    leftH, rightH ← Partition(H, medianH)

    for i ← 1 to |rightH|:
        rightH[i] -= medianH

    if leftH is not empty:
        SelectKRecursive(leftA, leftH, R)
    if rightH is not empty:
        SelectKRecursive(rightA, rightH, R)
```

Time Complexity Analysis

The main point of concern is the `SelectKRecursive` function which calls itself recursively. We see that the the for the base case when `h = 1`. (`leftH` and `rightH` is empty so there won't be any recursion in this case) The work done on the node is:

- `MomSelect` on `H` and `A`, it is $O(k)$ and $O(n)$, respectively.
- `Partition` on `H` and `A`, it is also $O(k)$ and $O(n)$, respectively.
- For loop to subtract `medianH` from `rightH`, it is $O(k)$.

In this case, the time complexity is mainly dominated by $O(n)$. Therefore, the base case has $T(n, k = 1) = O(n)$ time complexity.

Then, we see the inductive case, when A is get partition by `hElement`, we could assume there exist an $0 \leq l \leq 1$, and the partitioned `leftA` and `rightA` part have approximately size of `ln` and `(1-l)n`, respectively. (The `floor`/`ceil`/`+1/-1` doesn't really matter here), and since the `medianH` is the median of the `H`, the partitioned `leftH` and `rightH` both have approximately size of `h/2`. Then we see that the two subcases have time complexity of $T(ln, h/2)$ and $T((1-l)n, h/2)$. We also see that for the work done on the node, it has two parts

- `MomSelect` on `H` and `A`, it is $O(k)$ and $O(n)$, respectively.
- `Partition` on `H` and `A`, it is also $O(k)$ and $O(n)$, respectively.

So we see that work done on node is $O(n)$. Therefore, we could write the recurrence:

$$\begin{aligned} &T(n,h) = O(n) + T(ln, h/2) + T((1-l)n, h/2) \quad (0 \leq l \leq 1) \\ &T(n,1) = O(n) \end{aligned}$$

We could use the same approach on our GPS. (List the table first)

Levels	**(Work Per Node, Number Of Nodes)**	**(Case, Number of Case)**	**Total Work**
Level 0	$(n, 1)$	$(T(n,h)$	n
Level 1	$(ln, 1)$ $((1-l)n, 1)$	$(T(ln, h/2), 1)$ $(T((1-l)n, h/2), 1)$	n
Level 2	$(l^2n, 1)$ $(l(1-l)n, 2)$ $((1-l)^2n, 1)$	$(T(l^2n, h/4), 1)$ $(T((1-l)ln, h/4), 2)$ $(T((1-l)^2n, h/4), 1)$	n

We see the trend that for level `L`, the work is always n, a constant sequence. (The variable l we introduced just get canceled) Therefore, the total time complexity is `workPerLevel * numberOfLevels`. We see that the the $T(n,h)$ will go to base case when its h goes to 1. From the table, we see that the level L will have all subcases $T(\_, h/2^L)$ (the _ means we don't care about the first parameter in T). Therefore we see that $h/2^L = 1$ and $L = \log_2 h$. We have $L = \log_2 h$ levels. Therefore, the total time complexity is just $O(n) \cdot O(\log h) = O(n \log h)$ as we desired.

(b)

The pseudocode for the algorithm is shown below, we use the `A[a..b]` to slice from `A[a]` to `A[b]` in the array, inclusive. (If `a > b`, we will get empty array). we do this by referencing, but not copying, so this operation could be done in $O(1)$.

We see that we effectively change the problem of size k into $3k/4$ by each recursion. For base case when $k = 1$, the function returns and has time complexity of $O(1)$. In the inductive case, we see that the work done on the node is still $O(1)$. We see that in each case $T(k)$ will be $T(3k/4)$ (Notice that the empty array doesn't matter since in this case we set the corresponding element to `Infinity` to avoid it from being selected). Therefore, we see

$$\begin{aligned} &T(k) = T(3k/4) + O(1) \\ &T(1) = O(1) \end{aligned}$$

Each level will have $O(1)$ work, and we see there are $L = \log_{3/4} k$ levels for $T(k)$, and then $T(k) = O(\log k)$ and $G(k) = T(k) + O(\log n) = O(\log n)$ as we desired.

```
SelectFromFour(A1[1..a1], A2[1..a2], A3[1..a3], A4[1..a4], k):
    if k = 1:
        return min(A1[1], A2[1], A3[1], A4[1])

    index ← ceil(k/4)

    indexA1 ← max(|A1|, index)
    indexA2 ← max(|A2|, index)
    indexA3 ← max(|A3|, index)
    indexA4 ← max(|A4|, index)

    A1Element ← if A1 is not empty then A1[indexA1] else Infinity
    A2Element ← if A2 is not empty then A2[indexA2] else Infinity
    A3Element ← if A3 is not empty then A3[indexA3] else Infinity
    A4Element ← if A4 is not empty then A4[indexA4] else Infinity

    minElement ← min(A1Element, A2Element, A3Element, A4Element)

    if A1Element = minElement:
        return SelectFromFour(A1[(indexA1 + 1)..a1], A2, A3, A4, k - indexA1)
    if A2Element = minElement:
        return SelectFromFour(A1, A2[(indexA2 + 1)..a2], A3, A4, k - indexA2)
    if A3Element = minElement:
        return SelectFromFour(A1, A2, A3[(indexA3 + 1)..a3], A4, k - indexA3)
    if A4Element = minElement:
        return SelectFromFour(A1, A2, A3, A4[(indexA4 + 1)..a4], k - indexA4)
```