Problem 1

(a) Algorithm for sorting a k-swapped array.[1]

```
1:  function SortKSwapped(A, n, k)
2:      Create priority queue q of size k+1.                          ▷ O(1)
3:      for i = 1 to k+1 do          ▷ Insert the first k+1 elements into the p. queue.
4:          Insert(q, A[i])                                           ▷ O(log k)
5:      end for
6:      for i = 1 to n-k-1 do                          ▷ Sort the first n-k-1 elements.
7:          A[i] ← ExtractMin(q, k+1)  ▷ O(log k). Remove and return the min. element.[2]
8:          Insert(q, A[k+i])          ▷ O(log k). Insert a new element from the array.
9:      end for
10:     for i = n-k to n do                            ▷ Sort the last k+1 elements.
11:         A[i] ← ExtractMin(q, k+1)                                 ▷ O(log k)
12:     end for
13: end function
```

The algorithm first creates a priority queue of $k+1$ elements. This yields Insert and ExtractMin operations of $O(\log k)$ complexity. We initialise the queue by inserting the first $k+1$ elements of A.

We then get to the fun part... sorting. This is split into two loops. The first loop sorts the first $n-k-1$ elements *and* inserts the remaining $n-k-1$ elements of the array (alternating between extract and insert each iteration). The second loop sorts the last $k+1$ elements.

And the array is sorted.

(b) Note that the 1-st smallest element is within $A[max(1,i-k)\ldots min(n,i+k)]$. Since the priority queue q can hold up to $k+1$ elements, so the i-th time we extract, we get the i-th smallest element of the entire array. We do this for the whole array and take care to insert the remaining elements at the same time as sorting the front (in the second loop).

(c) Note that although the heap operations actually take $O(\log(k+1))$, we simplify this to

[1] Inspired by: `https://www.geeksforgeeks.org/nearly-sorted-algorithm/`

[2] The lecture notes aren't exactly clear on the signature of ExtractMin. I've taken the liberty to express it as a function that takes the p. queue q and the size of the queue $k+1$, and returns the minimum element (before removal).

$O(\log k)$.

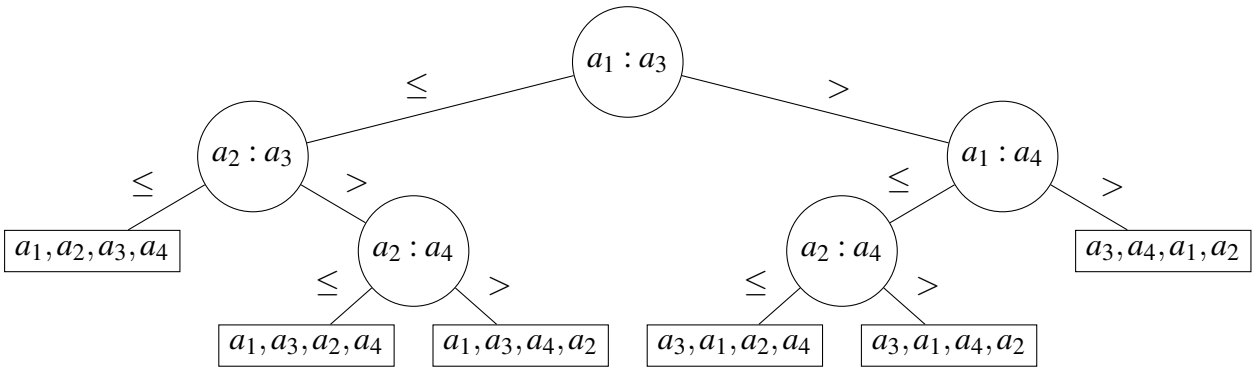
The first loop (lines 3-5) takes $O(k \log k)$ time since it runs k iterations of an $O(\log k)$ operation (INSERT). The second loop takes $O(2(n-k)\log k)$ time since it runs $n-k$ iterations of two $O(\log k)$ operations (EXTRACTMIN followed by INSERT). The final loop takes $O(k \log k)$ operations since it runs k iterations of an $O(\log k)$ operation (EXTRACTMIN).

Altogether, SORTKSWAPPED runs in $O(k \log k + 2(n-k)\log k + k \log k) = O(n \log k)$ time.

(d) The comparison-based algorithm makes no assumptions on the elements of the array and assumes $n!$ leaves in the decision tree. With a k-swapped array, it is assumed that $A[i] < A[k+i]$ for all $1 \leq i \leq n-k$, so this eliminates some leaves in the decision tree and is what allows us to use a priority queue of size k instead of n.

(e) We can separate the array into blocks of less than or equal to k. These blocks are localised and all elements in one block are smaller than all elements in the next block. The lower bound of leaves is then $(k!)^{n/k}$ which yields $\Omega(n \log k)$ by Stirling's approximation.

Problem 2


$a_1 : a_3$
$\leq$
$>$
$a_2 : a_3$
$a_1 : a_4$
$\leq$
$>$
$\leq$
$>$
a_1, a_2, a_3, a_4
$a_2 : a_4$
$a_2 : a_4$
a_3, a_4, a_1, a_2
$\leq$
$>$
$\leq$
$>$
a_1, a_3, a_2, a_4
a_1, a_3, a_4, a_2
a_3, a_1, a_2, a_4
a_3, a_1, a_4, a_2

Problem 3

(a)

$$[29681, 53846, 43521, 39427, 32433, 35700, 30764, 16892, 52608, 19583]$$

$$[35700, 29681, 43521, 16892, 32433, 19583, 30764, 53846, 39427, 52608]$$

$$[35700, 52608, 43521, 39427, 32433, 53846, 30764, 29681, 19583, 16892]$$

$$[39427, 32433, 43521, 19583, 52608, 29681, 35700, 30764, 53846, 16892]$$

$$[30764, 32433, 52608, 43521, 53846, 35700, 16892, 39427, 19583, 29681]$$

$$[16892, 19583, 29681, 30764, 32433, 35700, 39427, 43521, 52608, 53846]$$

(b)

$$[8b74, 73f1, aa01, 41fc, 7eb1, 4c7f, 782c, d256, 9a03, cd80]$$

$$[cd80, 73f1, aa01, 7eb1, 9a03, 8b74, d256, 41fc, 782c, 4c7f]$$

$$[aa01, 9a03, 782c, d256, 8b74, 4c7f, cd80, 7eb1, 73f1, 41fc]$$

$$[41fc, d256, 73f1, 782c, aa01, 9a03, 8b74, 4c7f, cd80, 7eb1]$$

$$[41fc, 4c7f, 73f1, 782c, 7eb1, 8b74, 9a03, aa01, cd80, d256]$$

Problem 4

(a) We sort the input by the endpoints f_j then use induction to show that a minimal $A \subseteq \{f_1, \ldots, f_n\}$ exists.

By induction, suppose we have the base case of one interval I_1. Trivially, $\{f_1\}$ is a minimal cover. Now assume that the first k intervals can be minimally covered by some $A \subseteq \{f_1, \ldots, f_k\}$.

We now introduce the $k+1$-th interval, I_{k+1}. Since we've sorted the input by the endpoints, so $f_k \leq f_{k+1}$. There are two cases, either I_{k+1} does not overlap with any previous intervals, or it does overlap with some. If it doesn't overlap, then we simply add f_{k+1} to A so that I_{k+1} is covered. On the other hand, if it does overlap, then there exists a set of endpoints J such that $f_j \in I_{k+1}$ for all $f_j \in J$. If none of f_j are in A (i.e. $J \cap A = \varnothing$), then we add f_{k+1} to A; otherwise, A already covers I_{k+1}.

In any circumstance, A now remains a minimal cover and $A \subseteq \{f_1, \ldots, f_{k+1}\}$. Thus there exists a minimal cover A such that $A \subseteq \{f_1, \ldots, f_{k+1}\}$.

(b)

```
1:  function GreedyCovering(I, n)
2:      Sort intervals by f_i.                          ▷ O(n log n) worst case complexity.
3:      A ← ∅
4:      j ← 1
5:      while j ≤ n do
6:          t ← I[j].f                                  ▷ Get the endpoint f_j
7:          A ← A ∪ {t}                                 ▷ Add f_j to the covering
8:          j ← j + 1                                   ▷ Skip this interval, we know it covers
9:          while j ≤ n and I[j].s ≤ t and t ≤ I[j].f do   ▷ Advance while t can cover I[j]
10:             j ← j + 1
11:         end while
12:     end while
13:     return A
14: end function
```

(c) Suppose that our greedy solution (GREEDY) is not optimal and there exists an optimal solution (OPT).

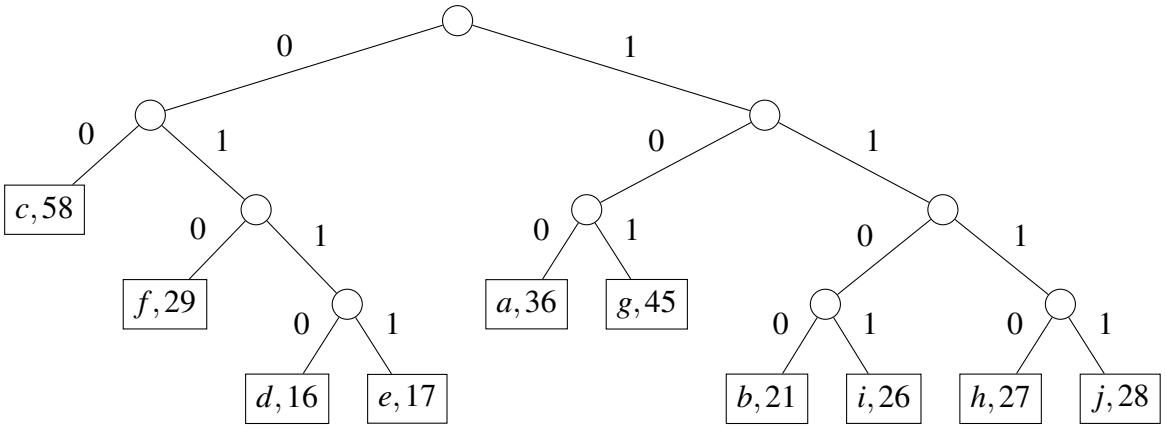
We collect the times returned by each solution. Let $g = \langle g_1, \ldots, g_m \rangle$ and $h = \langle h_1, \ldots, h_n \rangle$ be sorted arrays containing the points returned by GREEDY and OPT respectively (with $n < m$).

For the base case of $i = 1$, since g_1 is an endpoint, so $h_1 \leq g_1$. (If $h_1 > g_1$, then h_1 won't cover I_1.) Now assume that the first j intervals have been covered by k points ($k < n < m$), and consider the next interval I_{j+1} with points g_{k+1} and h_{k+1}. The situation repeats as with the base case. It is imperative that $h_{k+1} \leq g_{k+1}$, otherwise I_{j+1} won't be covered. In the end, we have $n = m$, contradicting our initial assumption that GREEDY is not optimal. Thus GREEDY is optimal.

(d) The time complexity is bounded by sorting. The rest of the code is just a loop that runs in $O(n)$ time. Each interval is only considered once. Each interval is either added (lines 6-8) or was covered (lines 9) by some previous finish time.

Problem 5

(a)



(b)

a	100
b	1100
c	00
d	0110
e	0111
f	010
g	101
h	1110
i	1101
j	1111

(c)

c	00
a	100
f	010
g	101
b	1100
d	0110
e	0111
h	1110
i	1101
j	1111