General rules to save everyone time:

- You do not need to attach the question sheet.
- You do not need to restate the questions.
- If you use an algorithm defined in class, you do not need to restate it or prove its performace.
- If you modify an algorithm defined in class, you only need to state the modification and how it (might) affect the running time.

Total: 15 points.

1. (a) (5 points)

   1. Run deterministic K-Select, to find the kth smallest number.
   2. Scan the array, output all numbers that are $\leq$ the kth smallest.

   Time: $O(n)$ for BSelect (deterministic K-Select), as taught in class plus $O(n)$ for scanning the array equals $O(n)$ total running time.

   Grading: There is no need to specify how the deterministic K-Select algorithm works, only that it runs in worst-case (i.e., deterministic) $O(n)$. The solution is, therefore, very short and simple.

   Some students copied the K-Select recursive implementation. It is possible to make small modifications to B-Select (the deterministic version), so it would not only return the kth smallest element, but also all elements smaller than it. However, if this is what you decide to do, make sure to make this modification!

   Also note, that using a randomized pivot in this case, would give us $\Omega(n^2)$ in the worst case, which is not good enough.

   (b) (5 points) First, proving $\binom{n}{k} \geq (n/k)^k$:

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k!(n-k)!} = \frac{(n-k+1)(n-k+2)\cdots n}{k!} \\ &= \frac{n-k+1}{1}\frac{n-k+2}{2}\cdots\frac{n}{k} \geq \frac{n}{k}\frac{n}{k}\cdots\frac{n}{k} = (n/k)^k\end{aligned}$$

   Each combination of k out of n elements of A is a possible output. There are $\binom{n}{k}$ such combinations:

$$\#\text{ leaves in decision tree} \geq \#\text{possible outputs} = \binom{n}{k}$$

   It is $\geq$, because an output may appear in multiple leaves, but every possible output must appear in at least one leaf.

   Assuming all elements are distinct, each comparison has two possible outcomes ("<" or ">"), therefore the branching factor in the tree is two. Therefore:

$$\#\text{leaves in the decision tree} \leq 2^{\text{tree height}}.$$

Putting it all together:

$$(n/k)^k \leq \binom{n}{k} \leq \#\text{leaves} \leq 2^{\text{height}} \Rightarrow \log((n/k)^k) \leq \log 2^{\text{height}} \Rightarrow \text{height} \geq k\log(n/k)$$

Therefore, in the worst case *any* algorithm must make $k\log(n/k)$ comparisons.

This lower bound is to loose. For example, for $k = 1$ this is the problem of finding the minimal element. In reality we need $\Omega(n)$ time because we must scan the entire array, but for $k = 1$ the lower bound is $\Omega(k\log(n/k)) = \Omega(\log n)$, which is much lower.

Grading: 3 points for proving the decision tree lower bound, 1 point for correctly proving the inequality, and 1 point for proving the bound is loose. Just stating it is loose is not enough, unfortunately. Remember: the decision tree technique is *not* for proving a lower bound for a specific algorithm; it is for proving a lower bound for *any* (comparison-based) algorithm that solves the problem. Therefore, the proof should have nothing to do with the specific algorithm you found. You cannot assume that at the root node, or at any other node, the algorithm will compare specific elements $(A[i], A[j])$, because different algorithms will choose to compare different pairs of elements, and the proof should hold for *any* algorithm. Also note: in this question, the number of leaves in the decision tree may be smaller than the number of orderings of the input array. There are $n!$ possible orderings (=permutations) of the input array, but only $\binom{n}{k} < n!$ possible outputs.

2. (5 points)

In this question, we need to board people to the spaceship by order of decreasing priorities, until we can't board the next person, because the weight limit k would be acceeded. Notice, however, that there may remain a thinner person who would not violate the weight limit, but we will not board this person because a person with a higher priority would be left on Earth.

If all weights are 1, we just need to find the k people with the highest priorities, and we could have used the algorithm in question 1. However, the weights are different.

Also notice: according to the way the question was asked, it is possible to return either the list of people that will board, or the priority p_m of the highest-priority person that is rejected. Both return values are ok, and it is easy and quick ($O(n)$) to find one from the other.

Notice that higher priorities are represented by higher numbers, *not* by lower numbers.

Algorithm MoonFlyers(A,k)
(A is an array of n people with priorities & weights, k is the weight limit)
1. If $|A| = 0$, return A (the empty set).
2. Find a "good" pivot p (see more below), from the array of priorities.
3. Split A by priorities:
   H = people with priority higher than p
   L = people with priority lower than p.
   (Notice: $|H| + 1 + |L| = n$.)
4. Calculate: W_H = sum of the weights of the people in H.
5. If $k \leq W_H$ return MoonFlyers(H,k)
6. If $k \leq W_H + w_p$ (the weight of the pivot), return $H \cup \{p\}$.
7. return $H \cup \{p\} \cup$ MoonFlyers(L,$k - W_H - w_p$).

Finding a good pivot: The simplest way is to use BSelect, the deterministic K-Select algorithm, to select the person with the median priority from A. This ensures that: 1) finding the pivot takes $O(n)$ time; and 2) the pivot is "good" – no matter if we recurse on H or on L, the size of the problem will decrease by half.

Time: Define $T(n)$ to be the running time of MoonFlyers(A,k), when $|A| = n$ (and for any k). Step 1 takes $O(1)$. Step 2 takes $O(n)$ (running deterministic BSelect). Step 3 takes $O(n)$ (partitioning). Step 4 takes $O(|H|)$, and $|H| < n$. Step 5-7: Since both $|H|, |L| \leq n/2$, and we recurse either on H or on L but not on both, the recursion part takes at most $T(n/2)$, and the non-recursive parts take $O(n)$. Therefore, the total gives us: $T(n) \leq T(n/2) + O(n)$, i.e. $T(n) \leq T(n/2) + c \cdot n$, which implies $T(n) = O(n)$ (from the formula of the sum of a decaying geometric series). Notice that this upper bounds the worst-case running time and *not* the expected running time. If we used a random pivot, we would get $O(n^2)$ in the worst-case scenario, and this is not good enough here.

Notice: For the pivot, we used BSelect to find the median priority, in $O(n)$ worst-case time. It is also possible to use BSelect's pivot function (divide to groups of 5, and find the median of the $n/5$ medians). In both cases the pivot is "good". In the first case we are guaranteed that $|H|, |L| \leq n/2$, and in the second that $|H|, |L| \leq 0.7n$. In both cases the pivot-finding function runs in $O(n)$ and we recurse with a subproblem which is "small enough", i.e. $T(n) = T(n/2) + O(n)$ or $T(n) = T(0.7n) + O(n)$, so the total running time becomes a decaying geometric series, and therefore the total running time is $O(n)$ (make sure you understand this). For analysis purposes, using BSelect for the pivot is more comfortable, because it can guarantee a very nice pivot (the median), so the analysis is simpler. From a practical perspective, using the pivot function directly is simpler (no need to implement BSelect itself) and might be faster, because we find a "good" pivot and don't waste time on finding the "ideal" pivot that BSelect provides (the median). On the down side, the recursion will probably take longer, because the pivot is less ideal: we are guaranteed that $|H|, |L| \leq 0.7n$ instead of $|H|, |L| \leq n/2$. Furthermore, in a real-life situation, a probabilistic $O(n)$ expected-time algorithm would probably suffice, in which case we can simply pick a pivot at random.

For the grading, it is important to mention that the pivot is taken among the priorities, the function you are using for the pivot (e.g. BSelect), and what it guarantees. But it is not necessary to re-prove what was proved in class.

Stopping conditions: checking if A is empty is required. It may be initially empty, or it may be empty when we recurse on small subsets. Writing "if $|A| = 1$ return $A[1]$" is a mistake, because the single person in A might be too heavy.

Another idea that popped up was to find the maximal weight $w_{\max}$. Then we know that the $k/w_{\max}$ people with the highest priority can go. (I'm leaving out the rest of the algorithm's details.) In practice, this may work. But in theory, how many iterations are needed? What happens if the weights are: $W = (1, 2^1, 2^2, 2^3, \cdots)$?

And, of course, there is another solution, which is the most recommended: dieting.