1. The procedure is not a correct heapify solution. There are different ways to show that the order is important. For example, consider array $A = [0, 1, 2, 3, 4, 5, 6]$. Applying bubble-down as indicated in the code results in array $[2, 4, 6, 3, 1, 5, 0]$ which is not a max-heap.

2. part(a): basic calculation shows that MST has size $\sqrt{5} + 3 + 2 + 2 + 1 \approx 10.23$.
part(b): we use a reduction from sorting. Given an instance $A = \{x_1, x_2, \ldots, x_n\}$ of the sorting problem, we create an instance of MST as follows. For each $x_i (1 \leq i \leq n)$, we plot a point $(x_i, 0)$ in the plane. The MST is formed by segments that connect consecutive numbers in the sorted array. So, if there is an algorithm that solves MST in $O(n \log n)$, then there is an algorithm that solves the sorting problem in $O(n \log n)$. This contradicts the lower bound of $\Omega(n \log n)$ for comparison-based sorting algorithms.

3. part (a): $i_{max} = \binom{n}{2}$. This happens if and only if every pair of indices (i, j) is an inversion, which happens if and only if the array is sorted in decreasing order.
$i_{min} = 0$. This happens if and only if no pair of indices (i, j) is an inversion, which happens if and only if the array is sorted in increasing order.

   part(b): Fix i and j where $1 \leq i < j \leq n$. For any permutation $\pi = \pi_1 \pi_1 \ldots \pi_n$, define π^* by interchanging π_i and π_j . The mapping $\pi \mapsto \pi^*$ is a bijection defined on the set of all $n!$ permutations. If $\pi_i > \pi_j$, then (i, j) is an inversion for π but not for π^*. If $\pi_i < \pi_j$, then (i, j) is an inversion for π^* but not for π. Therefore, for any π, (i, j) is an inversion for exactly one of π and π^*. Hence, the number of permutations for which (i, j) is an inversion is $n!/2$ and the probability that (i, j) is an inversion is $1/2$.

   part(c): There are $\binom{n}{2}$ pairs of indices (i, j) (where $i < j$). For each such pair of indices (i, j), we proved in part (c) that the probability that (i, j) is an inversion is $1/2$. It follows that $i_{avg} = 1/2 \times \binom{n}{2} = n(n-1)/4$.

   part (d): If any two adjacent elements in an array are exchanged, then the number of inversions in the array either increases by one or decreases by one. An array sorted in increasing order has 0 inversions. Therefore, the number of exchanges of adjacent elements required to sort A is at least the number of inversions in A. In part (c), we proved that the average number of inversions in an array is $n(n-1)/4$, which is $\Theta(n^2)$. It follows that the average-case complexity of the sorting algorithm is $\Theta(n^2)$. This immediately implies that the worst-case complexity is also $\Theta(n^2)$.

4. part(a): $\Theta(n)$, which happens when the first random item is the desired one; it takes $\Theta(n)$ to check this.
part(b): ∞, which happens when the randomly selected items are never the desired one.
part(c) Let cn be the linear time spent for selecting and checking a random index. We

will have:

$$T(n) = cn + (n-1)/n \times T(n)$$
$$1/n\ T(n) = cn$$
$$T(n) = cn^2 \in \Theta(n^2)$$

5. part(a): $T(n) = cn + T(n-i-1) + T(i)$ where $1 \leq i \leq n$ and $T(1) = d$, where c and d are constants. Finding the mean and partition take cn time in total.

   part(b): Consider the algorithm is called with m elements $a + tk, a + (t+1)k, \ldots a + (t+m-1)k$. For the mean of these elements, we have:

$$mean = 1/m \times \sum_{j=t}^{t+m-1} (a + jk) = a + k/m \times \sum_{j=t}^{t+m-1} j = a + k(t + m/2)$$

   Hence, the mean of the elements is roughly $(m/2)$'th element, i.e., the pivot is the median of the elements **in every call of the algorithm**. Consequently, the best case happens (as discussed in the class) and the algorithm runs in $\Theta(n \log n)$.

   part(c): Consider items to be $1, 2, 4, \ldots, 2^n$. In every call, the mean of the subarray is close to the last item (precisely, the $m-1$th item will be selected). Hence, the complexity becomes $\Theta(n^2)$.