1. You know from the lectures that a heap can be built from an array of n integers in $O(n)$ time. Heap is ordered such that each parent node has a key that is bigger than both children's keys. So it seems like we can sort an array of n arbitrary integers in $O(n)$ time by building a heap from it. Is it true?

1 / 1 point

- (●) No
- ( ) Yes

Correct

Correct! Although the key of each parent node is bigger than the keys of the children, the keys can be not ordered from the biggest to the smallest. For example, with just three numbers 312 the head element 3 is bigger than both children 1 and 2, but their relative order is wrong. Also, you should recall from the Algorithmic Toolbox class that it is impossible to sort n objects based only on results of comparisons of pairs of objects faster than in $O(n \log n)$ time.

2. You've organized a party, and your new robot is going to meet and greet the guests. However, you need to program your robot to specify in which order to greet the guests. Of course, guests who came earlier should be greeted before those who came later. If several guests came at the same time or together, however, you want to greet first the older guests to show them your respect. You want to use a min-heap in the program to determine which guest to greet next. What should be the comparison operator of the min-heap in this case?

0 / 1 point

◉
```
def GreetBefore(A, B):
  if A.arrival_time != B.arrival_time:
    return A.arrival_time < B.arrival_time
  return A.age < B.age
```

○
```
def GreetBefore(A, B):
  if A.arrival_time != B.arrival_time:
    return A.arrival_time < B.arrival_time
  return A.age > B.age
```

○
```
def GreetBefore(A, B):
  if A.arrival_time != B.arrival_time:
    return A.arrival_time > B.arrival_time
  return A.age > B.age
```

○
```
def GreetBefore(A, B):
  if A.arrival_time != B.arrival_time:
    return A.arrival_time > B.arrival_time
  return A.age < B.age
```

⊗ **Incorrect**

With this comparison, if two guests come at the same time, the younger one will be greeted before the older one.

3. You want to implement a Disjoint Set Union data structure using both path compression and rank heuristics. You also want to store the size of each set to retrieve it in $O(1)$. To do this, you've already created a class to store the nodes of DSU and implemented the $Find$ method using the path compression heuristic. You now need to implement the $Union$ method which will both use rank heuristics and update the size of the set. Which one is the correct implementation?

1 / 1 point

○
```
def Union(a, b):
  pa = Find(a)
  pb = Find(b)
  pa.parent = pb
  pb.size += pa.size
```

○
```
def Union(a, b):
  pa = Find(a)
  pb = Find(b)
  if pa.rank <= pb.rank:
    pa.parent = pb
    if pa.rank == pb.rank:
      pb.rank += 1
  else:
    pb.parent = pa
```

○
```
def Union(a, b):
  pa = Find(a)
  pb = Find(b)
  if pa.rank <= pb.rank:
    pa.parent = pb
    pb.size += pa.size
  else:
    pb.parent = pa
    pa.size += pb.size
```

◉
```
def Union(a, b):
  pa = Find(a)
  pb = Find(b)
  if pa.rank <= pb.rank:
    pa.parent = pb
    pb.size += pa.size
    if pa.rank == pb.rank:
      pb.rank += 1
  else:
    pb.parent = pa
    pa.size += pb.size
```

Correct

Correct! You need to determine the new root using the rank heuristics, and then not forget to update the size stored in the new root.