1. Illustrate the operation of Heap_extract_max on the heap A=<15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>

We will move the max to end of array, and remove it. So we will have,

```
1, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2
```

The we will need to do heapify on position 1, l: 2, r: 3, (we start from index 1).

```
13, 1, 9, 5, 12, 8, 7, 4, 0, 6, 2
```

On position 2, 4, 5

```
13, 12, 9, 5, 1, 8, 7, 4, 0, 6, 2
```

On position 5, 10, 11

```
13, 12, 9, 5, 6, 8, 7, 4, 0, 1, 2
```

And we done.

2. You are given an array of n elements, and you notice that some of these elements are duplicates; that is they appear more than one time in the array. Devise an algorithm to remove all duplicates from the array in time O(n log n).

We sorted it at first. It cost O(n log n) time. Than we scan the array again, and remove all items which have the same value as the previous one, which cost O(n).

```
prev <- a[0]
int i = 1;
while (i < a.length - 1)
  if (a[i] === prev)
    remove a[i]
  else
    prev <- a[i]
  i++
```

3. A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

The special cost will at sequence.

```
1, 2, 4, 8, 16, 32 ....
```

There are (log2 n) + 1 times, that i will equal to power of 2.

The total cost will be,

```
// for all other constant cost
sum <- (n - log2 n) * 1

// for special cost
for (i <- 1 to (log2 n) - 1)
  sum <- sum + 2^i
```

The total cost will be small than `n + 2*n = 3*n`. That is O(n) for total cost, and O(1) amortized cost per operation.

4. A sequence of stack operations is performed on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Everytime we do a operation, we can image we do the same thing on virtual "backup" stack, and after k operations, we will "actually" do those operations to create real new "backup" stack.

Therefore, we just only need to double the cost of all operation from original one, so the new total cost is `2*(cost of original operations)`. Because amortized cost of stack with operations POP and PUSH is O(n), the total cost will still be O(n) with backup mechanism. (Assume create new backup stack will be constant cost, and overall cost is still O(n)).

5. Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Show how to implement a counter as an array of bits so that any sequence of increment or reset operations takes time O(n) on an initially zero counter.

We will need to introduce another fields `max_d` in memory to store the max digits of counter. When we execute increment, we need to update the `max_d`.

Originally, we assume it cost 2 for each increment, since when we flip 0 to 1 which cost 1, we need to save 1 for future when we need to flip 1 to 0 in other increments.

Now, we assume it will cost 4 for each increment at most, 2 is the same before; 1 for update `max_d` so we know how far we need to check to flip all digit to 0 when we reset; another 1 for fliping that digit to 0 in resetting operation. Those 2 cost happens only when `max_d` need to be updated. Therefore, we only need 1 cost when we reset the counter, that is for reseting the `max_d`. All other costs are covered by increment.

In short, the total cost is small than 4n, and it is still O(n).

6. What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s0 objects and finishes with sn objects?

In the lecture, we have cost 2 for PUSH, and 0 for POP and MULTIPOP, and we define the potential function be `φ(Sn)`, where Sn is the number of elements.

And we have two following equations.

For PUSH

```
cost of operation + (φ(Sn) - φ(Sn-1)) = 2
```

For POP & MULTIPOP

```
cost of operation + (φ(Sn) - φ(Sn-1)) = 0
```

Therefore, for n operation, and objects change from S0 to Sn, we have total cost of operation + S0 - Sn <= 2n.

In other words, Total cost of operation <= `2n + Sn - S0`

7. Illustrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table has 9 slots, and let the hash function be h(k) = k mod 9.

```
[0]
[1]
[2]
[3]
[4]
[5] 5
[6]
[7]
[8]

[0]
[1] 28
[2]
[3]
[4]
[5] 5
[6]
[7]
[8]

[0]
[1] 28 -> 19
[2]
[3]
[4]
[5] 5
[6]
[7]
```

```
[8]

[0]
[1] 28 -> 19
[2]
[3]
[4]
[5] 5
[6] 15
[7]
[8]

[0]
[1] 28 -> 19
[2] 20
[3]
[4]
[5] 5
[6] 15
[7]
[8]

[0]
[1] 28 -> 19
[2] 20
[3]
[4]
[5] 5
[6] 15 -> 33
[7]
[8]

[0]
[1] 28 -> 19
[2] 20
[3] 12
[4]
[5] 5
[6] 15 -> 33
[7]
[8]

[0]
[1] 28 -> 19
[2] 20
[3] 12
[4]
[5] 5
[6] 15 -> 33
[7]
[8] 17


[0]
[1] 28 -> 19 -> 10
[2] 20
[3] 12
```

```
[4]
[5] 5
[6] 15 -> 33
[7]
[8] 17
```