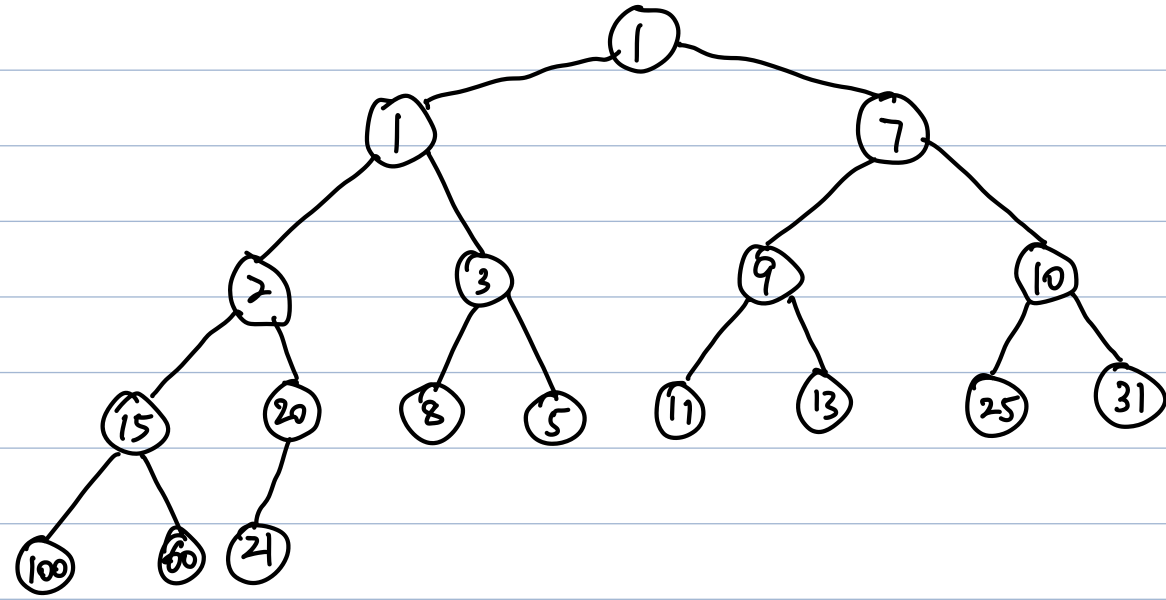
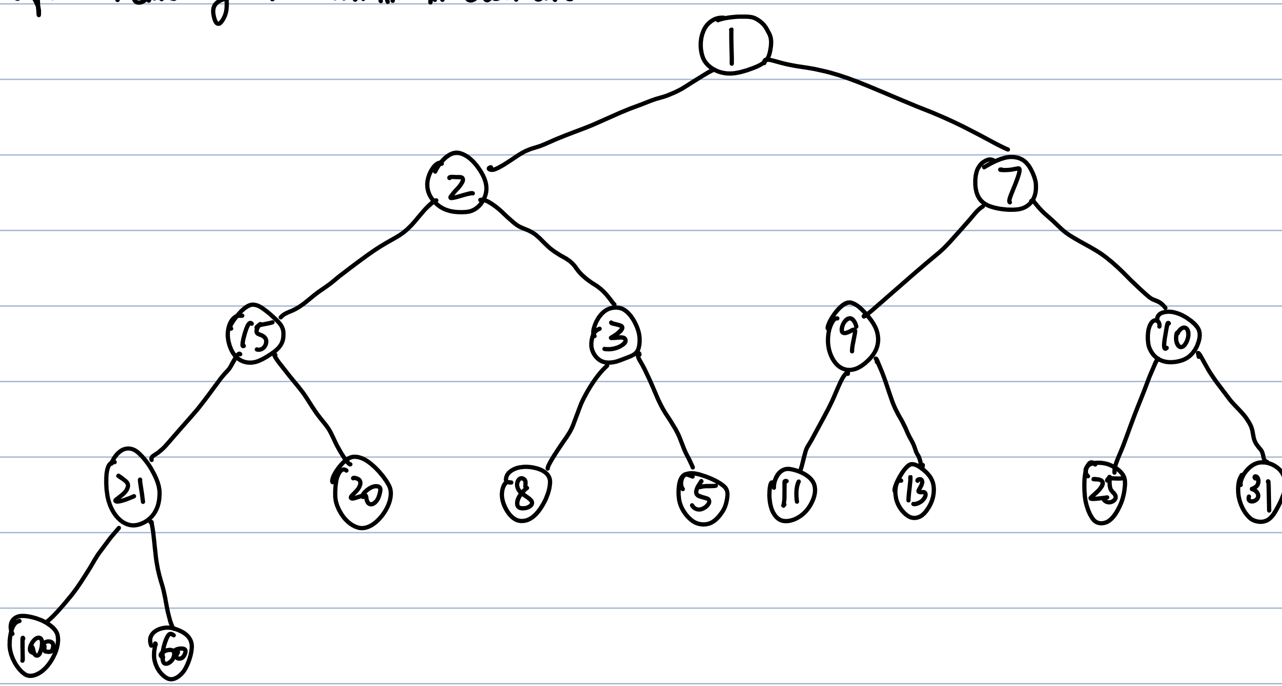
1.



1	1	7	2	3	9	10	15	20	8	5	11	13	25	31	100	60	21
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

After removing the minimum element:



1	2	7	15	3	9	10	21	20	8	5	11	13	25	31	100	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2. (a)

Auxillary functions:

```
heapifyUP(H, i)
{ if i>0:
      parent = ⌊(i-1)/2⌋
      if H[i] < H[parent]:
          temp = H[i]
          H[i] = H[parent]
          H[parent] = temp
          heapifyUP(H, parent)
}

// heapify down the ith-node with length n array
heapifyDown(H, i, n)
{
    left = 2i+1
    right = 2i+2
    if left ≥ n-1
        return
    else if left ≤ n-2:
        j = min{H[left], H[right]}
    else if left = n-2:
        j = left
    if H[j] < H[i]
        swap H[j] and H[i]
        heapifyDown(H, j, n)
}
```

Median getter:

```
getMedian(A, n)
{
    if n < 1:                              ← O(1)
        return // A contains nothing
    let H = []
    for i in range(0, n)                   ← O(n)
    {
        H[i] = A[i]                        ← O(1)
        heapifyUP(H, i)                    ← O(log n)
    } // now H is a min heap of A.
    for j in range(0, n-1)                 ← O(n)
    {                                      ← O(1)
        temp = H[n-1-j] // last element
        H[n-1-j] = H[0]                    ← O(1)
        H[0] = temp                        ← O(1)
        heapifyDown(H, 0, n-j-1)           ← O(log n)
    } // now H is a decreasing sorted array. *

    if n % 2 == 0:                         ← O(1)
        return ((H[n/2]+H[n/2-1])/2.0)
    else:                                  ↖ O(1)
        return (H[(n-1)/2])
                                           ↖ O(1)
}
```

So in total, O(n log n) time.

(b) //after the line ✗ in (a), we have a sorted array H containing all
// elements in A. The above processes is O(n log n).
// then:

```
if  n <2 :            ← O(1)
    return  // A does not have at last 2 elements, terminate
let dist = +∞         ← O(1)
let minIdx = 0        ← O(1)
for i in range(0, n-2):      ← O(n)
    if H[i]-H[i+1] < dist :   //since H in decreasing order, H[i]-H[i+1]≥0.
        dist = H[i]-H[i+1]     ← O(1)
        minIdx = i
element_1 = H[i]     // element_1 & element_2 are the 2 elements.
element_2 = H[i+1]   // if want to return them, we can store them in an array
                     // and return the pointer points to the first element in the array.
```

In total, $O(n\log n)+O(n)=O(n\log n)$

(c) // namely we want to find 2 distinct i, j s.t. $(y_i-y_j)/(x_i-x_j)$ is maximized.

```
struct point //we define a struct called point
{            // to contain the coordinate values.
    int x;
    int y;
    bool operator<(const point& other) const
    { // overload the "<" operator
        if( this->x < other.x)
        { return true; }
        return false
    }
};
```

```
point ptArr[n]; // declare an array to contain points.
for(int i=0; i<n ; i++)
{
    ptArr[i].x = x_i ;
    ptArr[i].y = y_i ;
}
// now since we overloaded the "<" operator,
// we can use the algorithm in (a)
// to sort ptArr, such that for each
// element in ptArr, their x coordinate is
// in decreasing order.   ← O(n log n) till here
```

```
//then for the sorted ptArr:
    if (n <2)                  ← O(1)
    { return;}                 ← O(1)
    int maxSlope= -∞ ;         ← O(1)
    for (int i=0 ; i< n-1; i++)   ← O(n)
    {
        if (ptArr[i] .x - ptArr[j] .x == 0)  ← O(1)
        {
            return ∞; // verticle line   ← O(1)
        }
        if ((ptArr[i].y - ptArr[j].y)/(ptArr[i].x - ptArr[i].x) > maxSlope)  ← O(1)
        {
            maxSlope = (ptArr[i].y - ptArr[j].y)/(ptArr[i].x - ptArr[i].x);  ← O(1)
        }
    }
    return maxSlope;   ← O(1) .
```

In total: $O(n \log n)$

4. Idea: we can build an $O(n)$ function, killable(n, k, t, R), to check whether n aliens each with $R[i]$ HP can be killed by k bombs within a fixed time t. Then we binary search on $t \in (0, n)$ to find the minimal t.

For killable(n,k,t,R), since we want t to be minimized, we need to use every bomb most efficiently. Now, think of an array $A = [R[0], R[1], \cdots, R[n-1]]$, to use any bomb in the optimal way, we look for the left most non-zero element, say it has index i, then we plant the bomb at $A[\min\{i+\frac{t}{2}, n-1]$. Then the HP for all $A[i]$ to $A[\min\{i+t-1, n-1\}]$ decrease by 1, repeatedly. Based on the idea, we have our pseudo-code:

```
bool killable(n, k, t, R){
    Let A = [R[0], R[1], ..., R[n-1]]
    int remainBomb = k
  ☆ for i in range(0, n):              ← O(n)
        if remainBomb = 0:            ← O(1)
            break
        elif A[i] <= 0:
            continue
        elif A[i] != 0:
            if remainBomb < A[i]:     ← O(1)
                return false
            for j in range(i, min{n, i+t})
                A[j] -= A[i]          ← O(1)
            remainBomb -= A[i]        ← O(1)
    for i in A:                       ← O(n)
        if i > 0:
            return false              in total, O(n)
    return true }
```

in each killable, this inner loop will be executed for k times at most. So $O(k \cdot t) = O(k \cdot n)$ regardless of the outer for loop ☆

So in total, killable is still $O(n)$

```
int shortestTime(n, k, R){
    curTime = n
    while(killable(n, k, R, curTime)):   ← binary search on curTime, O(log n)
        if curTime == 1:   ← O(1)
            return 1   // aliens can be killed with charging 1 minute.
        curTime = ⌈curTime / 2⌉   ← O(1)
    return curTime }
```

killable is $O(\log n)$

In total $O(n \log(n))$

Thus the algorithm is $O(n \log(n))$