3.1 For any k, the height of recursive ~~function~~ quicksort function is ~~maximum~~ at most $\log(\frac{n}{k})$. Therefore quicksort's time complexity is $O(n \log \frac{n}{k})$.

Time compexity of the insertion sort is $O(n^2)$

We will move every element no more than k times. ~~O(nk~~ Now our answer is $O(nk)$.

~~n log n/k part is definitely too less than nk,~~

~~but we can get it~~

$$\alpha n \log n \geq \beta nk + \alpha n \log \frac{n}{k} \quad / \alpha n$$

$$\log n \geq \frac{\beta}{\alpha} k + \log n - \log k$$

$$\log k \geq \frac{\beta}{\alpha} k$$

To maximize the performance, we can pick k using binary search.

3.2 ~~1~~) insert to the rightmost leaf which results in runtime $\sum_{i=1}^{n} i \in \Theta(n^2)$

~~2~~2) the size of two subtrees will not exceed 1. therefore the height is $\Theta(\log n)$. runtime $\sum_{i=1}^{n} \lg n \in \Theta(n \lg n)$

3) linear time since each list insertion will take const time

4) in the worst-case, random will choose rightmost leaf everytime, so the answer is the same as in 1. $\Theta(n^2)$

Expected time: since there is equal chance $(\frac{1}{2})$ to get to the left or right child, the height will be $\Theta(\log n)$, and the time complexity $\Theta(n \log n)$

3.3 1) Parent (i)

    return (i-2)/d + 1;

  child (i, j)

    return d * (i - 1) + j + 1;

2) since each node has d children, the height will be $\Theta(\log_d n)$

3) extractMax(N)

```
if(N.size < 1) throw "error";
mx = N.data[1];
N.data[1] = N.data[N.size];
N.heapsize--;
MaxHeapify(N, 1);
return mx
```

```
MaxHeapify (N, i)
    mx = i;
    for k in ... d
        if (child(k, i) ≤ N.sz &&
                        N.data[child(k,i)] >
                                N.data[i]
            if N.data[child(k,i)] > mx
                mx = N.data[child[k,i]];
        if mx != i
            swap N.data[i], N.data[mx];
            MaxHeapify(N, mx);
```

Max Heapify calls itself "height" times

d so, the runtime is $O(d \log_d n)$

e) insert(N, key)

```
    N.sz++;
    N[N.sz] = key
    i = N.sz;
    while (i>1 && N.data[parent(i)] < A[i]
        swap them
        i = parent(i);
```

runs at most "height" times, so $O(\log_d n)$

5) increaseKey (N, i, key)

```
if (key < N.data[i]) throw "cannot decreasekey";
N.data[i] = key;
while i > 1 && N[parent(i)] < N[i]
    swap them
    i = parent(i)
```

runs the same as 4), so $O(\log_d n)$