Problem 1

Because my array begins with index 0, father nodes and son nodes have following relation. This is a d-ary heap.

father = (son - 1) / d

father * d + 1 <= son <= father * d + d

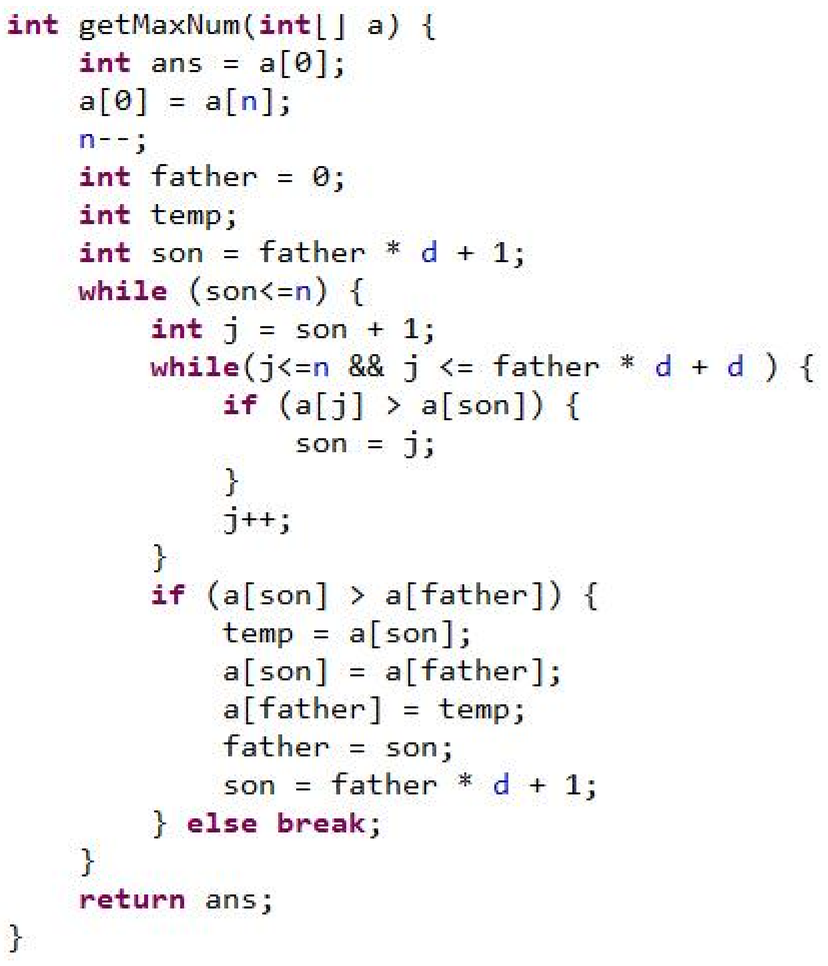
Problem 2

A full d-ary heap, the height is h. So, it has d^h-1 nodes. Suppose a d-ary heap with n elements and the height is h. Then

d ^ (h - 1) <= n <= d ^ h - 1

So, h = ⌈logd(n)⌉

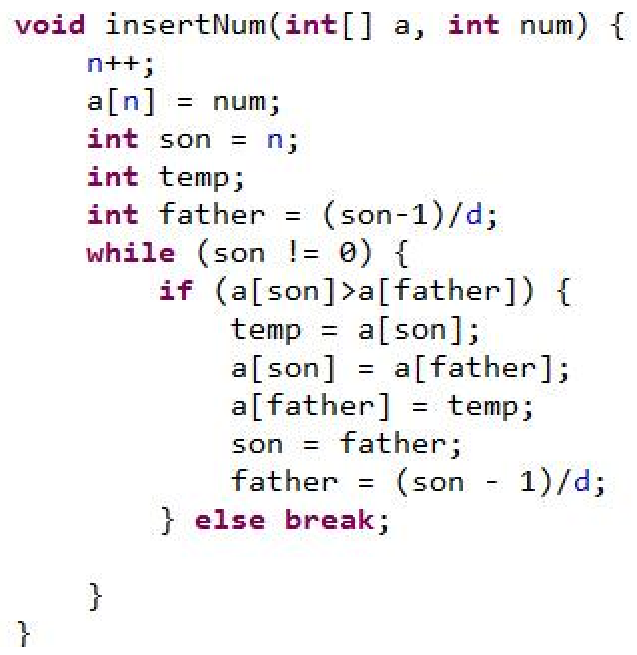
Problem 3



```
int getMaxNum(int[] a) {
    int ans = a[0];
    a[0] = a[n];
    n--;
    int father = 0;
    int temp;
    int son = father * d + 1;
    while (son<=n) {
        int j = son + 1;
        while(j<=n && j <= father * d + d ) {
            if (a[j] > a[son]) {
                son = j;
            }
            j++;
        }
        if (a[son] > a[father]) {
            temp = a[son];
            a[son] = a[father];
            a[father] = temp;
            father = son;
            son = father * d + 1;
        } else break;
    }
    return ans;
}
```

The height is logd(n). So, there are at most logd(n) times compare between father and son. Every time we need choose the largest son. Because every father has at most d sons, we need at most d times for searching the largest son. So, the running time for this method is d * logd(n).

Problem 4

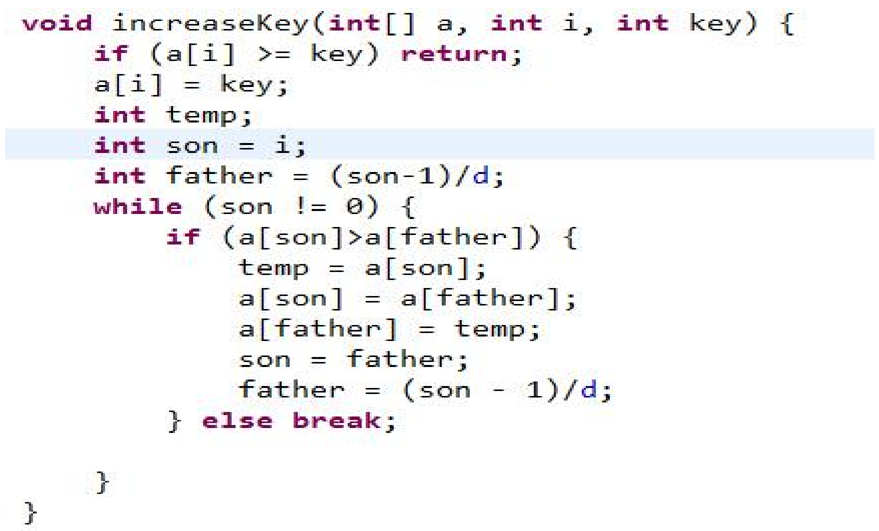


```
void insertNum(int[] a, int num) {
    n++;
    a[n] = num;
    int son = n;
    int temp;
    int father = (son-1)/d;
    while (son != 0) {
        if (a[son]>a[father]) {
            temp = a[son];
            a[son] = a[father];
            a[father] = temp;
            son = father;
            father = (son - 1)/d;
        } else break;

    }
}
```

The height is logd(n). So, there are at most logd(n) times compare between father and son. Every time we just need compare the current son and his father. So, the running time for this method is logd(n)

Problem 5



```
void increaseKey(int[] a, int i, int key) {
    if (a[i] >= key) return;
    a[i] = key;
    int temp;
    int son = i;
    int father = (son-1)/d;
    while (son != 0) {
        if (a[son]>a[father]) {
            temp = a[son];
            a[son] = a[father];
            a[father] = temp;
            son = father;
            father = (son - 1)/d;
        } else break;

    }
}
```

This is similar to the method insertNum(). And the running time is logd(n).

Problem 6

About insertNum()

n is the index of the last element currently. Every insertion, increase n, let a[n] be the insertion number. Then, we need keep the heap is a max heap——heapify(from son to father). We just need compare the current son and his father. If the son is larger than father, then exchange, and the new father with larger than all his son. If the son is small than father or equal to father, then heapify over.

About increaseKey()

This method is similar to insertNum(). If a[i] is not change, we do nothing. If a[i] is changed, it is to say a[i] is larger than all his son. So we just need heapify from son to father. And it is same as insertNum().

About getMaxNum()

Because this is a max heap, the root is the max number. We get and store it. In order to get the rest max number next time, we need delete the root. The total number of the elements reduce(n will reduce), that is to say a[n] will lose. So, before reduce, let the root(a[0]) be a[n], then n-1, this is delete. Then we need heapify from father to son, started at root. In order to make the father larger than all his son, we should choose a largest son for the current father. If father is smaller than son, then exchange. Because this is the largest son, we can make sure that the new father is larger than all his son. If father is larger than son or equal to son, then heapify over. At last, return the max numbr stored before.

13302010079
Qin Haifeng