Problem 1

Several functions occur in the implementation of Problem 1. These include insert, search, delete, and sort. Insertion alone would be O(1), as all that really happens is hepe[k] = kelem; with the **while** (kelem < hepe[(k-1)/childct]&&k > 0) taking roughly O(k) time, where k is the size of the heap.

An additional level of complexity is added if the heap has to be doubled – that is, the array size is exceeded and its elements have to be copied to a new one. This depends on O(k) as well, as only k elements get copied and array instantiation is O(1). Thus, this operation in total is O(2k) which is roughly O(k).

```
19    public void doubleArray()
20    {
21        int[] hepe1=new int[hepe.length*2];
22        for(int i=0;i<hepe.length;i++)
23        {
24            hepe1[i]=hepe[i];
25        }
26        hepe=hepe1;
27    }
```

Delete is O(k^2), as only k elements at worst need to be traversed to find the needed heap entry, but this occurs twice due to a nested loop.

```
54        while (1+i*childct < heapdimen)
55        {
56            int chil = 1+i*childct;
57            int tpe=2;
58            int locat = tpe+i*childct;
59            while ((tpe <= childct)&&(locat < heapdimen))
60            {
61                if (hepe[locat] < hepe[chil])
62                    chil = locat;
63                locat = tpe+1+i*childct;
64            }
65            cle=chil;
66            if (hepe[cle] < kelem)
67                hepe[i] = hepe[cle];
68            else
69                break;
70            i = cle;
71        }
```

The heapsort depends on the tree height. This is log-base-n of k, where n is the number of children allowed and k is the number of elements in the heap. At worst, the tree must be run up and down k times, along its lognk height. This is klogk, which is O(nlogn).

```
124       while(i>=0)
125       {
126           int store = hepe[0];
127           hepe[0]=hepe[i];
128           hepe[i]=store;
129           recursort(i, 0);
130           i--;
131       }
```