Binary Search

* public static unique(int[] v, int len)
Que retira os elementos repetidos no array v, com dimensão len. No final do método os elementos presentes no array devem estar contiguos. O método retorna o numero de elementos presentes no array.

```
public static unique(int[] v, int len){
        int i=0,j=1;
        for(;j<len;++j){
                if (v[j]!=v[j-1]){v[i++]=v[j];}
        }
        return i;
}
```

* public static int potencia(int a, int n)
que retorna o calculo de a^n

```
public static int potencia(int a, int n){
        if (n == 0) return 1;
        int x= potencia(a,n/2);
        return (n%2 != 0)? a*x*x : x*x;
}
```

* public static int lowerbound(int[] v, int l, int r, int key)
Que dado um array ordenado de N elementos, retorna o primeiro indicie i tal que a[i] >= key

```
public static int lowerbound(int[] v, int l, int r, int key){
        int mid;
        while (l<=r){
                mid=(l+r)/2;
                if (v[mid] >= key)
                        r=mid-1;
                else
                        l=mid+1;
        }
        return l;
}
```

* public static boolean issubsetof(int[] v, int l1, int r1, int[] v2, int l2, int r2)
que dados dois sub-arrays ordenados retorna true se os elementos do primeiro estão contidos no segundo

```
public static boolean issubsetof(int[] v, int l1, int r1, int[] v2, int l2, int r2){
        int result=0;

        for (int i=l1;i<=r1;++i){
                if (v1[i]> v2[l2]) return false;
                result=lowerbound(v2,l2,r2,v[i]);
                if result == -1 ) return false;
                l2=result;
        }
        return true;
}
```

HEAP

* public int heapHeight(int[] v, int len)
Assumir que o len existe com posição no array

```
public int heapHeight(int[] v, int len){
        int h=0;
        while (len > 0){
                ++h;
                len=(len-1)/2;
        }
        return h;
}
```

* public int heapChangeValue(int[] v, int len, int x, int nval)
Assumir que v está em Max-Heap, len é uma posição no array

```
public int heapChangeValue(int[] v, int len, int x, int nval){
        v[x] =nval;
        if (v[x] >= v[(x-1)/2]
                return changekey(v,x,len);
        else
                return maxHeapify(v,x,len);
}
```

*public static int minimum(int[] v, int len)
Retorna o menor elemento do max-heap de dimensão len, representado por v.

```
public static int minimum(int[] v, int len) {
        int i= (len-1)/2 +1;
        int min=v[i];
        while(++i < len){
                if (min > v[i])
                        min = v[i];
        }
        return min;
}
```

Iterable

```
public static Iterable<E> XXX(final Iterable<E> x){
        return new Iterable<E>(){
                public iterator<E> iterator(){
                        return new Iterator<E> (){
                                public boolean hasnext(){
                                }
                                public E next(){
                                }
                                public void remove(){
                                        throw new
UnssuportedOperationException();
                                }
                        };
                }
        };
}
```

Listas

HashTables

BST

```
public static boolean isMirror(Node<E> root1, Node<E> root2){
        if (root1 == null || roo2 == null) return false;
        return ( root1.item.compareTo(item2.item)) &&
                        isMirror(root1.left, root2.right) &&
                        isMirror(root1.right, root2.left);
}
public static int altura(Node root){
        if (root == null) return -1;
        int hl=altura(root.left);
        int hr=altura(root.right);

        return 1 + (hl>hr)?hl:hr;
}
```

* public static<E> Node<E> search(Node <E> root, E e)
Não assumir parent, elementos não null e e não null

```
public static<E> Node<E> search(Node <E> root, E e){
        if (root == null ) return null;
        if (root.item.equal(e)) return root;
        Node <E> a= search(root.left,e);
        return (a != null) ? a: search(root.right,e);
}
public static <E> Node <E> search (Node <E> root, E e, Comparator cmp){
                if (root == null ) return null;
                int i;
                while ((i=cmp.compare(root.item,e))!=0)
                        root=(i>0)?root.right:root.left;
                return root;
}
```

Implemente o método estático retorne true sse a arvore de pesquisa contem algum elemento no intervalo [l,r];

```
public static boolean contains (Node<Integer> root, Integer l, Integer r){
        if (root == null) return false;
        if (root.value >= l && root.value<=r) return true;
        if (root.value < l)
                return contains (root.right,l,r);
        if (root.value>r)
                return contains (root.left, l,r);

}
```

Implemente o método estático que retorne um array contendo os K menores elementos prensentes na árvore binária de pesquisa com raiz root.

```
public static int getKleast(Node<Integer> root, Integer [] pos, int idx){
        if (idx == pos.lenght || root == null) return idx;
        idx=getKleast(root.left, pos,idx);
        pos[idx++] =  root.item;
        idx=getKleast(root.right, pos,idx);
        return idx;
}
```