public abstract void sort(int[] A, int lo, int hi)
{
    int s = split(A, lo, hi);
    sort(A, lo, s - 1);
    sort(A, s, hi + 1);
}

Recall the abstract class Abstract in the handout.

Sorting by Divide and Conquer
Selection Sort

The smallest element in the array is swapped with the element in the smaller partition.

- splitt() divides the array into two partitions of size I and n - I.

Selection Sort is a hard-split, easy-join method.
{ } //

return i + 1;

int tmp = a[i][j];
int temp = a[j][i];
swap tmp with a[j][i]; //

// if this makes A[i][j] the minimum of A[i][j:]
//
// also invariant still holds;
//
on loop exit: i = i + 1;
// { 
++i;
// invariant is maintained;

s = i;

if (a[i][j] > a[i-1][j])
while (i => j) do
    scan A to find minimum //

j = j + 1;

i = i + 1;

s = i;

}

public int split(int[] A, int i, int j)
Selection Sort take \( O(n^2) \) steps.

- \( \text{sort}() \) does nothing!
which to insert the new element.

- `sort()` starts with a sorted array and finds the correct position in

```java
    {
        return hi;
    }
```

- `split()` is trivial.

**Insertion Sort** is an easy-split, hard-join method.
{ and A[j+1:n] is sorted. //

if A[j] > key then A[j] = key; //
A[j] is sorted, A[j-1] is sorted because:

A[j] = key;

also invariant is still true //

on loop exit: j = lo or A[j-1] > key.

// invariant is maintained.

j = j - 1;

A[j] = A[j-1];

} ( ) ( )

while ( ) ( ) ( ) { // to the "right" to make room for key.

shift elements of A[j-1] that are greater than key //

key > all elements of A[j+1:n]. //

Invariant: A[j+1:n] and A[j+1:n] are sorted and

int key = A[j+1];

int s = A[j]; // remember s = A[j].

int j = A[j];

}

public void join(int A, int lo, int s, int hi)
• Insertion sort take $O(n^2)$ steps.