Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Sorting

	0	1	2	3	4	5	n
A	50	30	10	60	20	40	

(before)

	0	1	2	3	4	5	n
A	10	20	30	40	50	60	

(after)

To *sort* is to rearrange values according to some defined order.

Sorting an array is a fundamental operation, and a way to do so is built into every language.

We study sorting to illustrate these principles:

- Creativity in code development can be inspired by starting with an invariant.
- Different invariants lead to different algorithms, some better than others.
- Algorithms based on Divide and Conquer can have superior performance.
- Algorithms based on everyday experience can have inferior performance.
- Divide-and-Conquer approaches are naturally implemented by recursive procedures.
- Fast algorithms are not necessarily harder to code than slow algorithms.
- Implementations often draw on established code patterns.
- Precise specifications support careful reasoning during implementation.

	0	1	2	3	4	5	n
A	50	30	10	60	20	40	

(before)

	0	1	2	3	4	5	n
A	10	20	30	40	50	60	

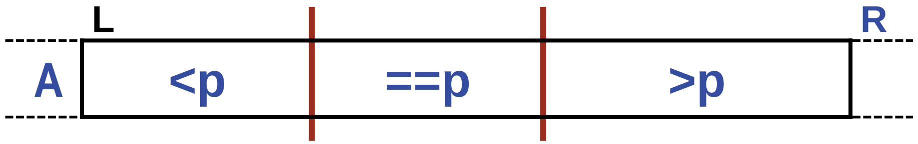
(after)

The specification for sorting an array is:

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
```

We consider four implementations of this specification:

- QuickSort
- Merge Sort
- Selection Sort
- Insertion Sort



Recall that Partitioning divides an array segment A[L..R-1] into “<p”, “==p”, and “>p” regions.

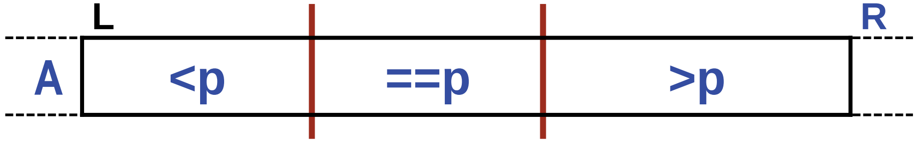
```
/* Rearrange A[L..R-1] into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
   ⟨body of Partition⟩
   } /* Partition */
```

All values in the “<p” region are less than p, which is less than all values in the “>p” region.

Also, on average, appropriate choice of pivot yields “<p” and “>p” regions of near equal size.

This is a basis for a Divide and Conquer algorithm.

☞ **Consider Divide and Conquer when designing an algorithm.**

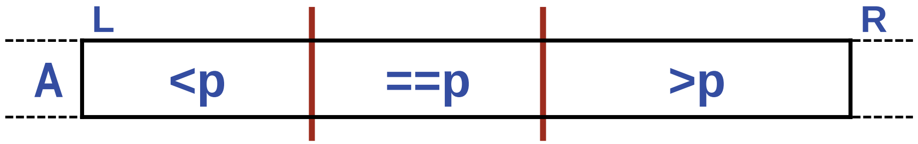


Start with the code for Partition, and morph it into QuickSortAux:

```
/* Choose a pivot p and rearrange A[L..R-1] into <p, ==p, and >p regions. */
static void QuickSortAux( int A[], int L, int R, int p ) {
   ⟨body of Partition⟩
   } /* QuickSortAux */
```

☞ **Don’t type if you can avoid it; clone. Cut and paste, then adapt.**

Change the name and header comment.

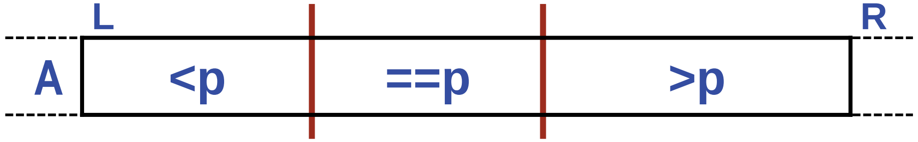


Start with the code for Partition, and morph it into QuickSort:

```
/* Choose a pivot p and rearrange A[L..R-1] into <p, ==p, and >p regions. */
static void QuickSortAux( int A[], int L, int R ) {
   int p = /* value of pivot */ ;
   ⟨body of Partition⟩
   } /* QuickSortAux */
```

☞ **Don’t type if you can avoid it; clone. Cut and paste, then adapt.**

Move pivot parameter p into the body of QuickSortAux.

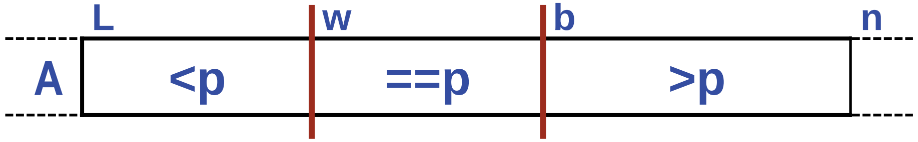


Start with the code for Partition, and morph it into QuickSort:

```
/* Choose a pivot p and rearrange A[L..R-1] into <p, ==p, and >p regions. */
static void QuickSortAux( int A[], int L, int R ) {
   if ( R-L > 1 ) {
      int p = /* value of pivot */ ;
      ⟨body of Partition⟩
      }
   } /* QuickSortAux */
```

☞ **Don’t type if you can avoid it; clone. Cut and paste, then adapt.**

Introduce the base case for regions of size 1, which perforce is sorted.

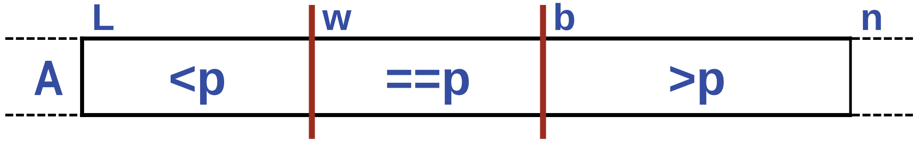


Recursively sort the “<p” and “>p” regions.

```
/* Choose a pivot p and rearrange A[L..R-1] into <p, ==p, and >p regions. */
static void QuickSortAux( int A[], int L, int R ) {
   if ( R-L > 1 ) {
      int p = /* value of pivot */ ;
      ⟨body of Partition⟩
      QuickSortAux(A, L, w);
      QuickSortAux(A, b, R);
      }
   } /* QuickSortAux */
```

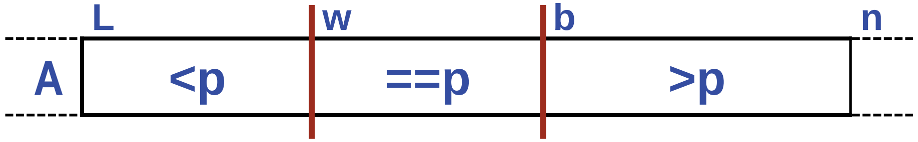
☞ **Consider recursion when designing an algorithm.**

QuickSort



Compute pivot p (designed to produce near-equal size “<p” and “>p” regions, on average.

```
/* Choose a pivot p and rearrange A[L..R-1] into <p, ==p, and >p regions. */
static void QuickSortAux( int A[], int L, int R ) {
   if ( R-L > 1 ) {
      int p = (A[L]+A[R-1])/2 ;
      ⟨body of Partition⟩
      QuickSortAux(A, L, w);
      QuickSortAux(A, b, R);
      }
   } /* QuickSortAux */
```



Invoke QuickSortAux from the top-level routine QuickSort.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
static void QuickSort( int A[], int n) {
   QuickSortAux(A, 0, n);
   } /* QuickSort */
```

Performance: Pivots computed as (A[L]+A[R-1])/2

- Best case. On each iteration, pivot is (serendipitously) the median of `A[L..R-1]`, so region sizes reduced by ½, leading to recursion depth log *n.* At each level of recursion, total partitioning cost is linear in n. Total effort: Proportional to *n* log *n*.

- Worst case. On each iteration, pivot is (serendipitously) the min or max of `A[L..R-1]`, so region sizes reduced by 1, leading to recursion depth *n.* Total effort: $n + (n-1) + (n-2) + \ldots + 1 = n \cdot (n-1)/2$, i.e., quadratic in n.

- Average case, i.e., summed over all permutations of values in `A[0..n-1]`. Total effort: Proportional to *n* log *n.*

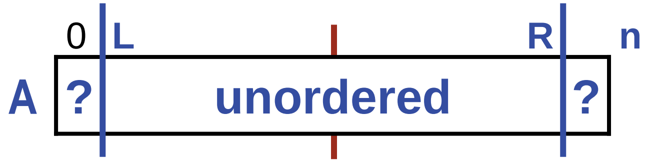
QuickSort recursively partitions, but region sizes are unpredictable. In contrast, MergeSort divides regions into (approximate) halves, quarters, eighths, etc.

```
/* Rearrange values of A[L..R] into non-decreasing order. */
static void MergeSortAux(int A[], int L, int R) {
   } /* MergeSortAux */
```

Note: In analogy with Binary Search, R is changed to the index of the last element of the region rather than one passed the last.

MergeSort divides (unordered) regions (approximately) in half at each recursion, sorts the halves, and collates those (ordered) halves into an (ordered) whole.

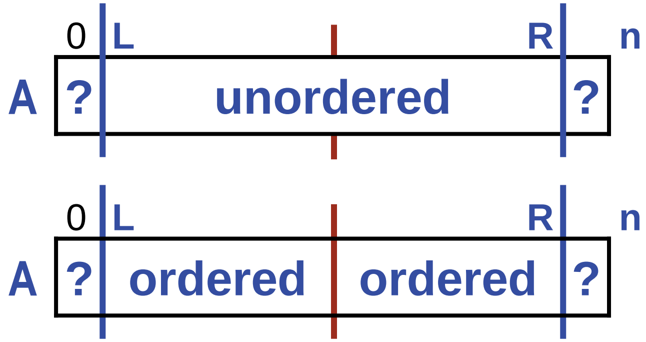
```
/* Rearrange values of A[L..R] into non-decreasing order. */
static void MergeSortAux(int A[], int L, int R) {
   if ( R>L ) {
      int m = (L+R)/2;
      MergeSortAux(A, L, m);
      MergeSortAux(A, m+1, R);
      /* Given A[L..m] and A[m+1..R], both already
         in non-decreasing order, collate them so
         A[L..R] is in non-decreasing order. */
      }
   } /* MergeSortAux */
```



Merge Sort

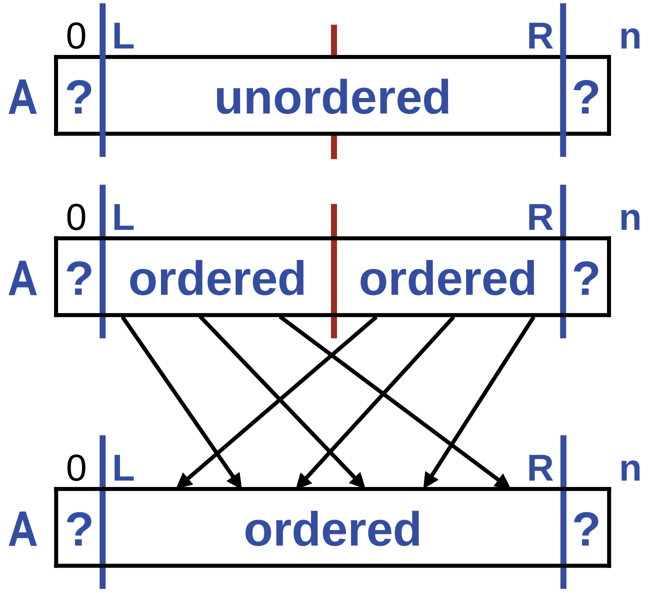
MergeSort divides (unordered) regions (approximately) in half at each recursion, sorts the halves, and collates those (ordered) halves into an (ordered) whole.

```
/* Rearrange values of A[L..R] into non-decreasing order. */
static void MergeSortAux(int A[], int L, int R) {
   if ( R>L ) {
      int m = (L+R)/2;
      MergeSortAux(A, L, m);
      MergeSortAux(A, m+1, R);
      /* Given A[L..m] and A[m+1..R], both already
         in non-decreasing order, collate them so
         A[L..R] is in non-decreasing order. */
      }
   } /* MergeSortAux */
```



MergeSort divides (unordered) regions (approximately) in half at each recursion, sorts the halves, and collates those (ordered) halves into an (ordered) whole.

```
/* Rearrange values of A[L..R] into non-decreasing order. */
static void MergeSortAux(int A[], int L, int R) {
   if ( R>L ) {
      int m = (L+R)/2;
      MergeSortAux(A, L, m);
      MergeSortAux(A, m+1, R);
      /* Given A[L..m] and A[m+1..R], both already
         in non-decreasing order, collate them so
         A[L..R] is in non-decreasing order. */
      }
   } /* MergeSortAux */
```

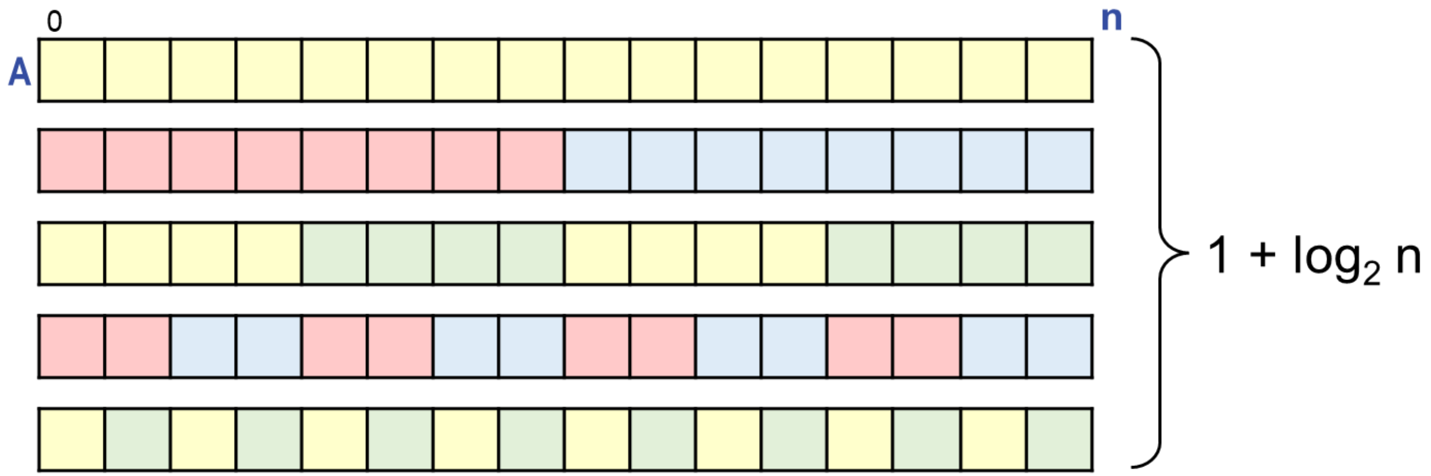


Invoke MergeSortAux from the top-level routine MergeSort.

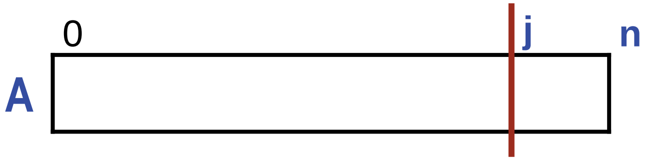
```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
static void MergeSort( int A[], int n) {
   MergeSortAux(A, 0, n-1);
   } /* MergeSort */
```

Performance:

- All cases. On each iteration, region sizes reduced by (approximately) ½, leading to recursion depth (approximately) log *n.* At each level of recursion, total collation cost is linear in n. Total effort: Proportional to *n* log *n*.



Positive: Guaranteed *n* log *n* performance. Negative: Not *in situ*.



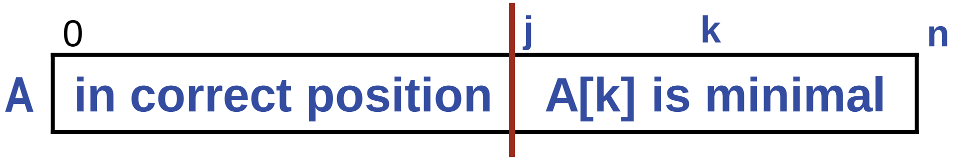
Selection Sort scans across array A from left to right with index `j`.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; ___; j++) _________
```

Selection Sort

INVARIANT: Values in `A[0..j-1]` are in their correct and final positions.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; ___; j++) _________
```



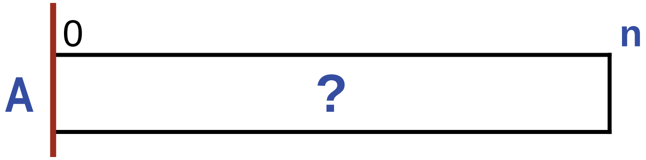
To maintain the **INVARIANT** as `j` is increased by 1, guarantee that `A[j]` is also in its final position.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; ___; j++) {
      /* Let k be s.t. A[k] is a minimal value in A[j..n-1]. */
      /* Swap A[j] and A[k]. */
      }
```

If A[0..n-2] are in their correct and final positions, so too is A[n-1].

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; j<(n-1); j++) {
      /* Let k be s.t. A[k] is a minimal value in A[j..n-1]. */
      /* Swap A[j] and A[k]. */
      }
```

Selection Sort



When `j==0`, the **INVARIANT** that all values in `A[0..-1]` are in their correct and final positions is trivially true.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 0; j<(n-1); j++) {
      /* Let k be s.t. A[k] is a minimal value in A[j..n-1]. */
      /* Swap A[j] and A[k]. */
      }
```

Selection Sort

The first step in the loop body is an application of Find Minimal (from Chapter 7).

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 0; j<(n-1); j++) {
      /* Let k be s.t. A[k] is a minimal value in A[j..n-1]. */
         int k = j;
         for (int i=j+1; i<n; i++)
            if ( A[i]<A[k] ) k = j;
      /* Swap A[j] and A[k]. */
      }
```

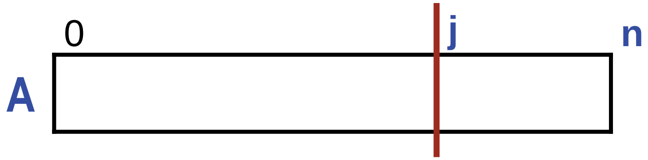
Swap is standard.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 0; j<(n-1); j++) {
      /* Let k be s.t. A[k] is a minimal value in A[j..n-1]. */
         int k = j;
         for (int i=j+1; i<n; i++)
            if ( A[i]<A[k] ) k = j;
      /* Swap A[j] and A[k]. */
         int temp = A[j]; A[j] = A[k]; A[k] = temp;
      }
```

Performance: Quadratic in n.

- *All cases*. The sum of the successive efforts to find the minimal value in `A[j..n-1]` is n +(n-1) + (n-2) + … + 2 = n·(n-1)/2 – 1, i.e., proportional to n^2.

Insertion Sort



Insertion Sort scans across array A from left to right with index `j`.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; ___; j++) _________
```

Insertion Sort

INVARIANT: Values in `A[0..j-1]` are in non-decreasing order.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; ___; j++) __________
```

To maintain the **INVARIANT** as `j` is increased by 1, insert `A[j]` into `A[0..j]` appropriately.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; ___; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
      }
```

Insertion Sort

The last element of `A[0..n-1]` may have to move, just like the others.

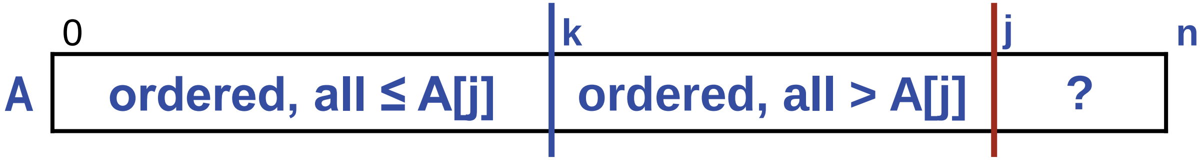
```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = ___; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
      }
```

Insertion Sort

When j==1, the **INVARIANT** that all values in A[0..0] is ordered is trivially true.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
      }
```

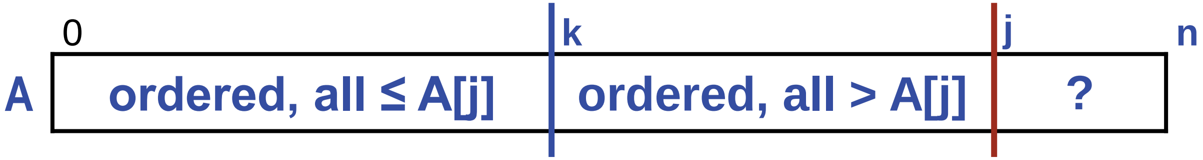
Insertion Sort



Right-shift values of `A[0..j-1]` that are larger than `A[j]`. Then insert `A[j]` appropriately.

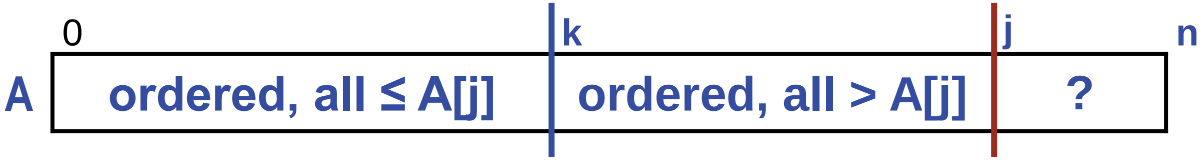
```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
         A[k] = temp;
      }
```

Insertion Sort



Treat the inner loop as a right-to-left search for rightmost k s.t. `A[k]≤A[j]`.

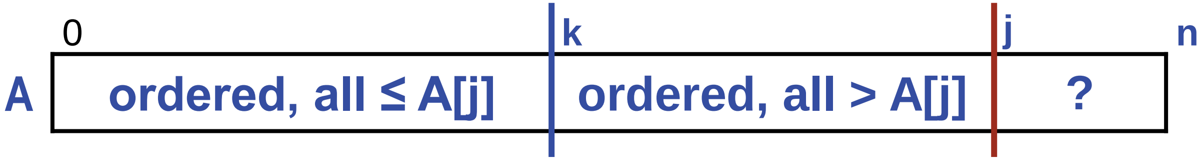
```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
            int k = ____;
            while ( ______ ) {
               A[ ____ ] = A[ ____ ];
               k--;
               }
         A[k] = temp;
      }
```



Treat loop as a right-to-left search for rightmost k s.t. A[k]≤A[j].

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
            int k = j;
            while ( ______ ) {
               A[ ____ ] = A[ ____ ];
               k--;
               }
         A[k] = temp;
      }
```

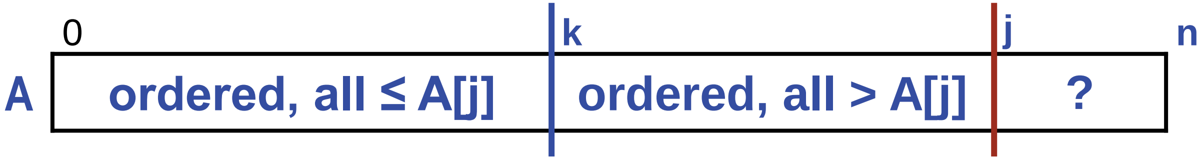
Insertion Sort



Treat loop as a right-to-left search for rightmost k s.t. A[k]≤A[j].

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
            int k = j;
            while ( A[k-1] ____ temp ) {
               A[ ____ ] = A[ ____ ];
               k--;
               }
         A[k] = temp;
      }
```

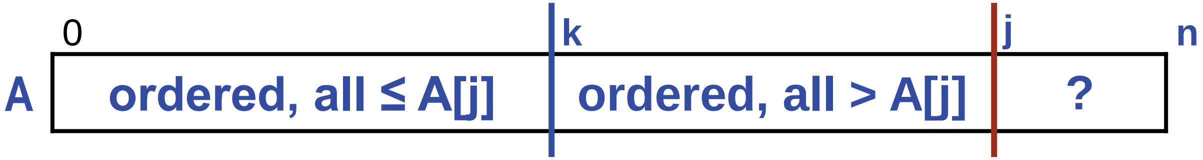
Insertion Sort



Treat loop as a right-to-left search for rightmost k s.t. A[k]≤A[j].

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
            int k = j;
            while ( A[k-1] > temp ) {
               A[ ____ ] = A[ ____ ];
               k--;
               }
         A[k] = temp;
      }
```

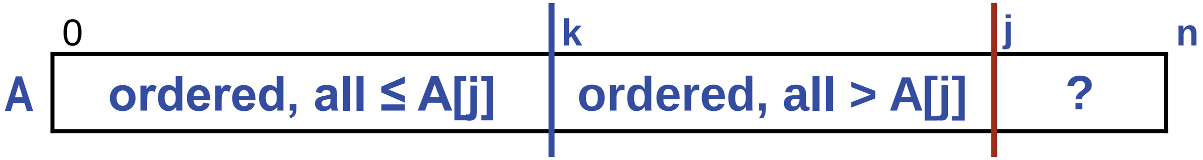
Insertion Sort



Allow for A[j] being minimum.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
            int k = j;
            while ( k>0 && A[k-1] > temp ) {
               A[ ____ ] = A[ ____ ];
               k--;
               }
         A[k] = temp;
      }
```

Insertion Sort



Do the shift at the same time as the search. Could end up putting A[j] right back where it started.

```
/* Rearrange values of A[0..n-1] into non-decreasing order. */
   for (int j = 1; j<n; j++) {
      /* Given A[0..j-1] ordered in non-decreasing order, rearrange
         values of A[0..j] so it is ordered. */
         int temp = A[j];
         /* Shift A[k..j-1] right one place, where k is the largest
            integer s.t. A[k-1]≤temp, or 0 if temp is smallest. */
            int k = j;
            while ( k>0 && A[k-1] > temp ) {
               A[k] = A[k-1];
               k--;
               }
         A[k] = temp;
      }
```

Insertion Sort

Performance: Quadratic in n.

- *Worst case*. Array starts out in non-increasing order. The sum of the successive shifts is $1 + 2 + \ldots + (n-2) + (n-1) = n \cdot (n-1)/2$, i.e., proportional to n^2.
- Best case. Array starts out already ordered. Linear in n.