Arrays and Pointers in C

In today's lecture we first take a detour from data structure, and discuss pointers and arrays in C.

Arrays and Pointers in C

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}
```

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}

void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printArray(A, 3);
}
```

```
1 3 5
```

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}

void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printArray(A, 5);
}
```

```
1 3 5 7 11
```

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}

void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printArray(A+3, 4);
}
```

```
7 11 13 17
```

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}

void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printArray(A+7, 2);
}
```

```
23 29
```

```
void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printf("%d\n", A[2]);
   printf("%d\n", (A+3)[0]);
   printf("%d\n", (A+5)[2]);
}
```

```
5
7
23
```

```
void main() {
  int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
  int *p; p = A;
  printf("%d\n", p[2]);
  printf("%d\n", (p+3)[0]);
  printf("%d\n", (p+5)[2]);
}
```

C treat A as constant

```
5
7
23
```

```
void main() {
  int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
  int *p; A = p;
  printf("%d\n", p[2]);
  printf("%d\n", (p+3)[0]);
  printf("%d\n", (p+5)[2]);
}
```

```
error: array type 'int [10]' is not assignable
    int *p; A = p;
            ~ ^
```

```c
void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printf("%d\n", A[0]);
   printf("%d\n", *A);
   printf("%d\n", A);
}
```

```
1
1
-296015264
Process finished with exit code 11
```

```c
void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   printf("%d\n", A[2]);
   printf("%d\n", *(A+3));
   printf("%d\n", A+3);
}
```

```
5
7
-390227348
Process finished with exit code 11
```

What will be printed?

```
void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   int i = (A+5)[2];
   printf("%d\n", i);
}
```

What will be printed?

```
void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   int *p;
   p = A;
   printf("%d\n", p[3]);
   p = A+3;
   printf("%d\n", p[2]);
}
```

What will be printed?

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}

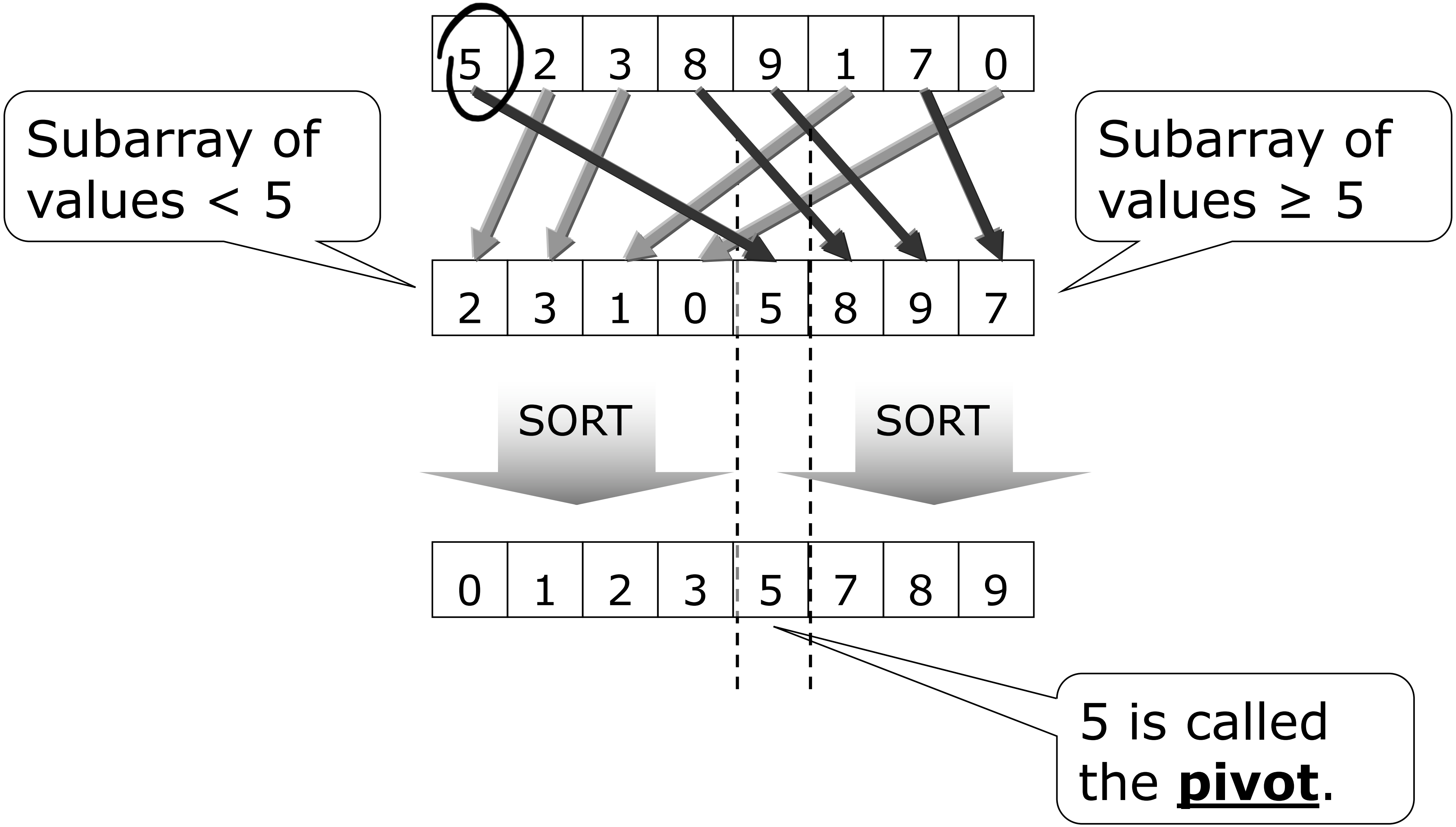
void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   int *p; p = A; printArray(p, 4);
}
```

What will be printed?

```
void printArray(int A[], int n) {
   for (int i=0; i<n; i++) printf("%d ", A[i]);
   printf("\n");
}

void main() {
   int A[10] = {1, 3, 5, 7, 11, 13, 17, 23, 29, 31};
   int *p; p = A; printArray(p+3, 4);
}
```

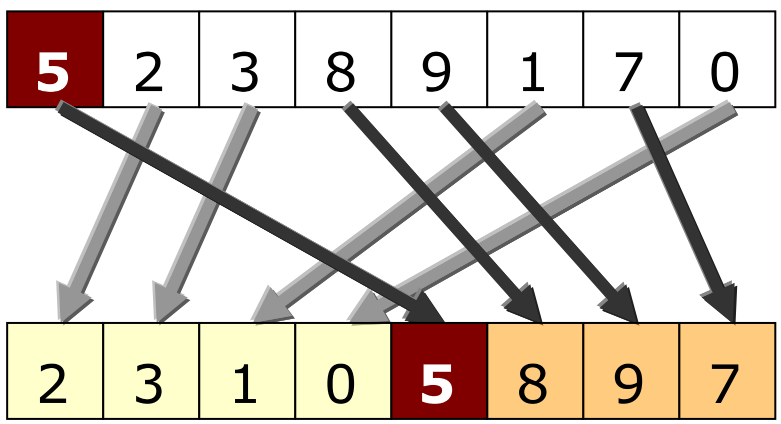
The Quicksort Algorithm



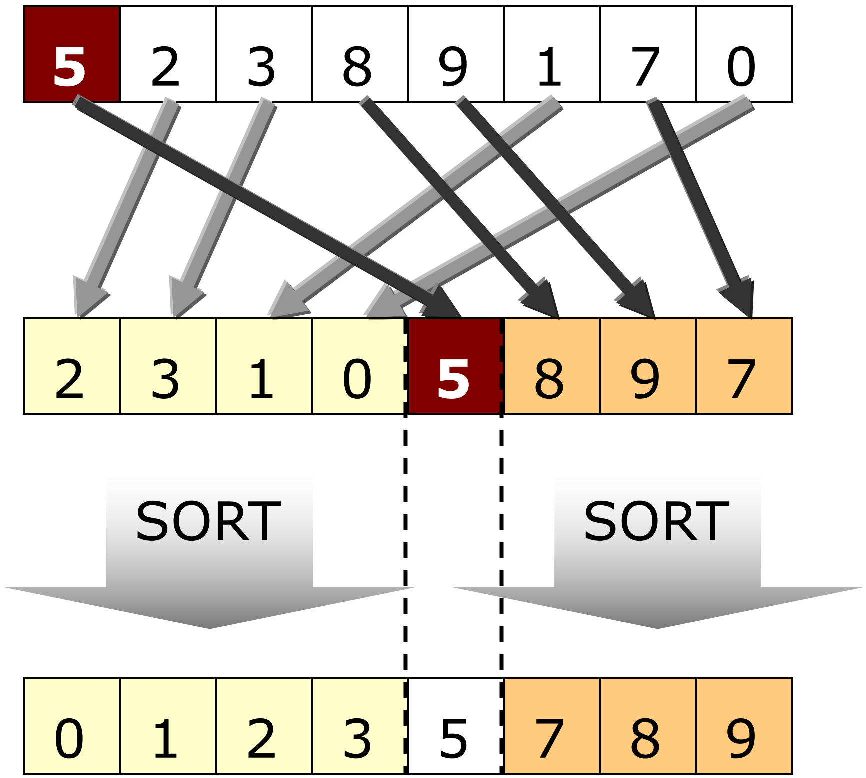
This is the basic idea of the **quicksort** algorithm:

- First, if the array is empty, or has only 1 element, then it is already sorted.

- Otherwise, choose a **pivot** and rearrange the elements in the array so that large elements are moved toward the end and small elements towards the beginning.



- Finally, sort the two subarrays.



We usually simply take the first element of the array as the pivot.

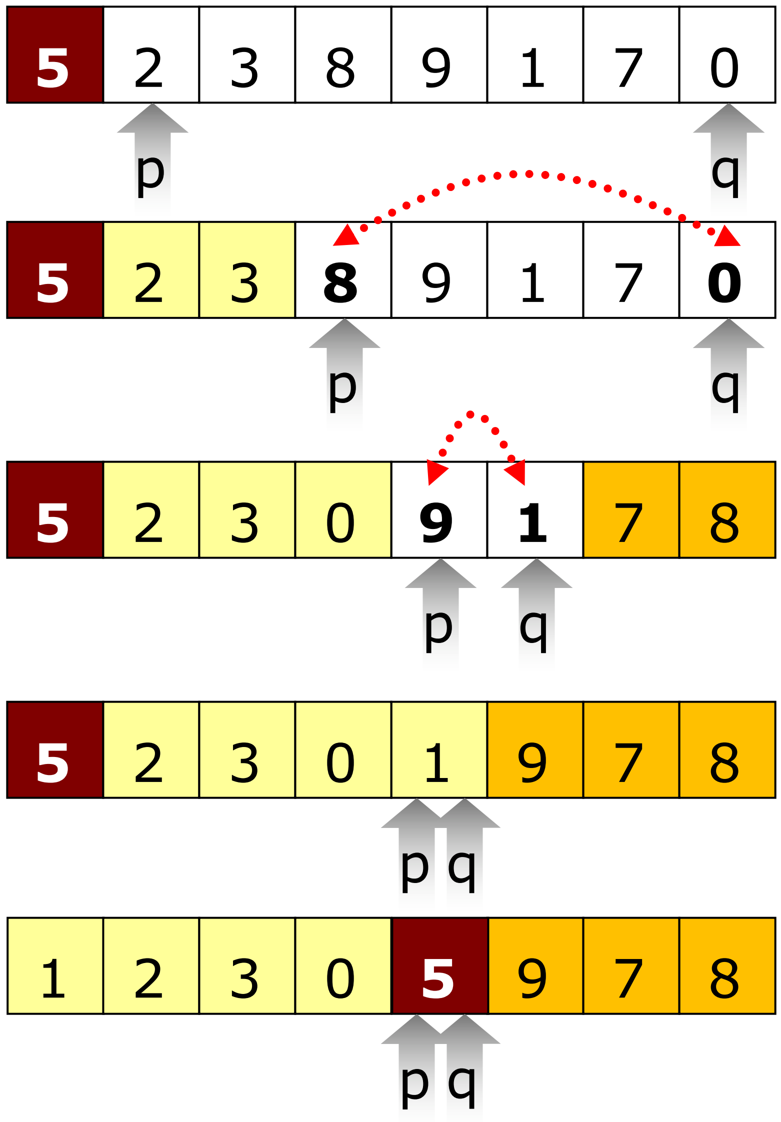
5	2	3	8	9	1	7	0

To arrange the array, we use the following method:

Initially, p=1 and q=n−1.

REPEAT
q: moves left until array[q] < pivot or meets p
p: moves right until array[p] >= pivot or meets q
If p meets q, then break
else swap array[p] and array[q].

Finally, put the pivot into the position where p and q meet.

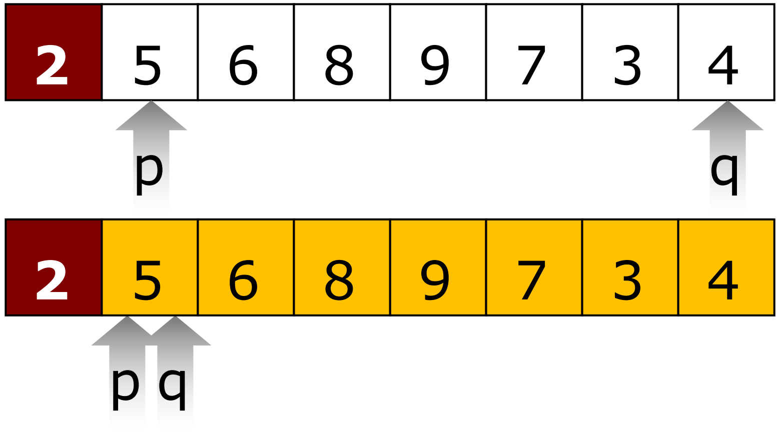


Another Example:

Initially, p=1 and q=n−1.

REPEAT
q: moves left until
    array[q] < pivot
    or meets p
p: moves right until
    array[p] >=
    pivot or meets q
If p meets q, then
    break
else
    swap array[p]
    and array[q].

Finally, put the pivot into the position where p and q meet.



*

```
static int Partition(int array[], int n) {

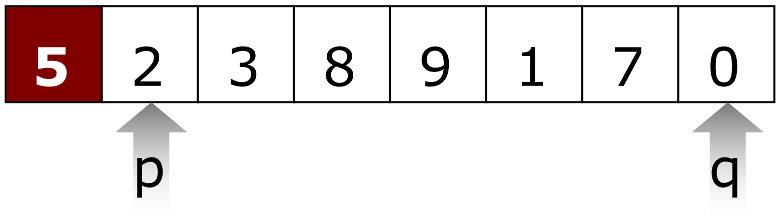
  int p, q, pivot;
  pivot=array[0]; p=1; q=n-1;
  while (1) {
    while (p<q && array[q]>=pivot) q--;
    while (p<q && array[p]<pivot) p++;
    if (p==q) break;
    {int tmp; tmp=array[p]; array[p]=array[q]; array[q]=tmp;}
  }
  if (array[p] >= pivot) return(0);
  array[0]=array[p]; array[p]=pivot;
  return(p);
}
```

```
static int Partition(int array[], int n) {
    int p, q, pivot;
    pivot=array[0]; p=1; q=n-1;
    while (1) {
        while (p<q && array[q]>=pivot) q--;
        while (p<q && array[p]<pivot) p++;
        if (p==q) break;
        {int tmp; tmp=array[p]; array[p]=array[q]; array[q]=tmp;}
    }
    if (array[p] >= pivot) return(0);
    array[0]=array[p]; array[p]=pivot;
    return(p);
}
```

Note:

- **pivot=array[0]; p=1; q=n-1;**

Initially, p=1 and q=n-1.

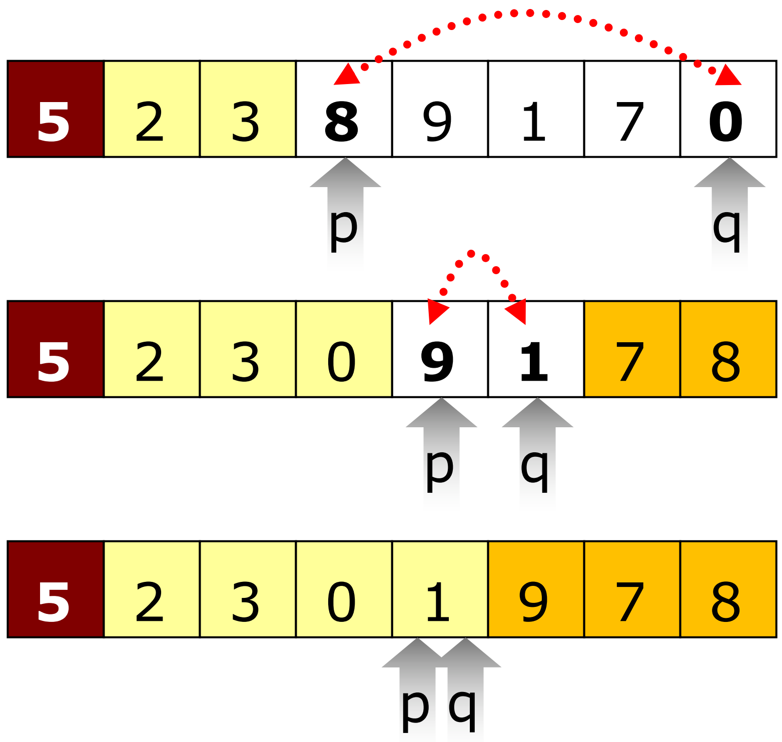


```
static int Partition(int array[], int n) {
    int p, q, pivot;
    pivot=array[0]; p=1; q=n-1;
    while (1) {
        while (p<q && array[q]>=pivot) q--;
        while (p<q && array[p]<pivot) p++;
        if (p==q) break;
        {int tmp;tmp=array[p];array[p]=array[q];array[q]=tmp;}
    }
    if (array[p] >= pivot) return(0);
    array[0]=array[p]; array[p]=pivot;
    return(p);
}
```

- ```
  while (1) {
      while (p<q && array[q]>=pivot)
          q--;
      while (p<q && array[p]<pivot)
          p++;
      if (p==q) break;
      {int tmp; tmp=array[p];array[p]=array[q];array[q]=tmp;}
  }
  ```

block

REPEAT
q: moves left until array[q] < pivot or meets p
p: moves right until array[p] >= pivot or meets q
If p meets q, then break
else
swap array[p] and array[q].

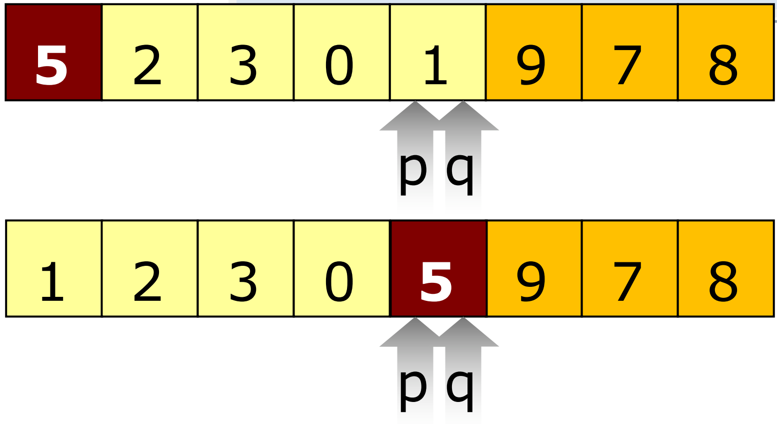

  ```

```
static int Partition(int array[], int n) {
    int p, q, pivot;
    pivot=array[0]; p=1; q=n-1;
    while (1) {
        while (p<q && array[q]>=pivot) q--;
        while (p<q && array[p]<pivot) p++;
        if (p==q) break;
        {int tmp; tmp=array[p]; array[p]=array[q]; array[q]=tmp;}
    }
    if (array[p] >= pivot) return(0);
    array[0]=array[p]; array[p]=pivot;
    return(p);
}
```

- **if (array[p] >= pivot)**
  return(0);
  array[0]=array[p];
  array[p]=pivot;
  return(p);

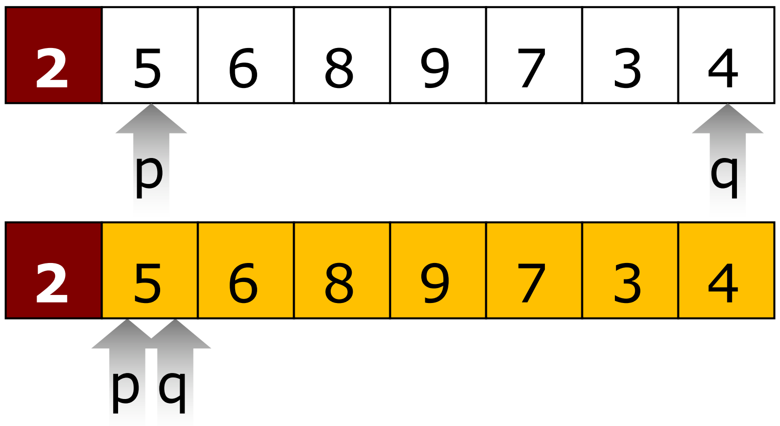
Finally, put the pivot into the position where p and q meet.

return(p);

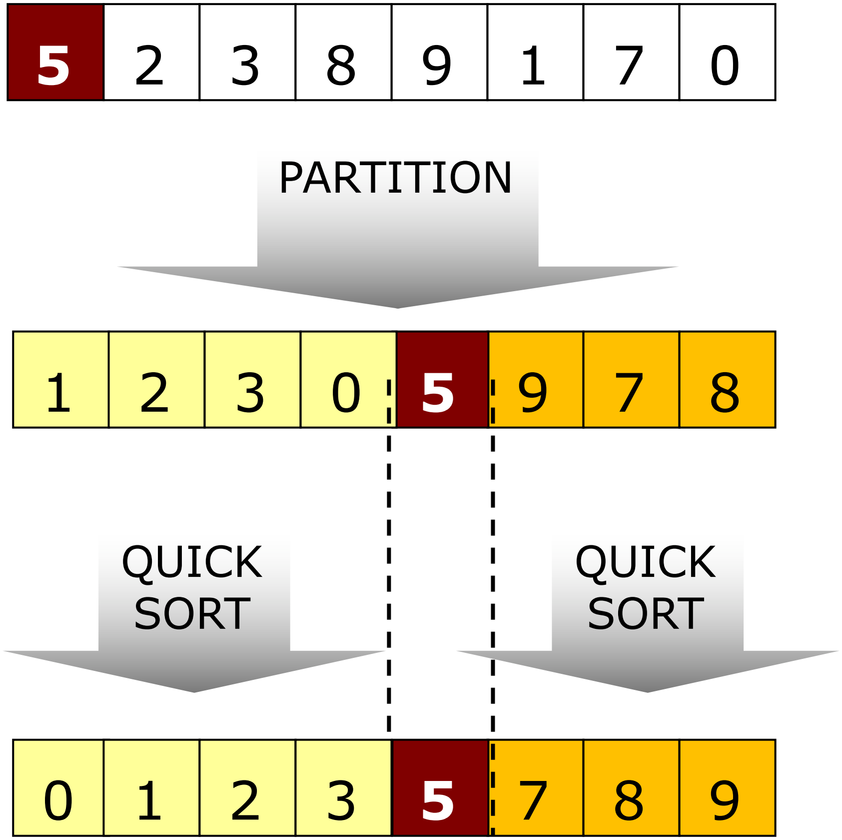


Finally, put the pivot into the position where p and q meet.

return(0);



Reminder: the Quicksort algorithm



*

```
void QuickSort(int array[], int n) {

  int pivotPosition;

  if (n<=1) return;

  pivotPosition = Partition(array, n);

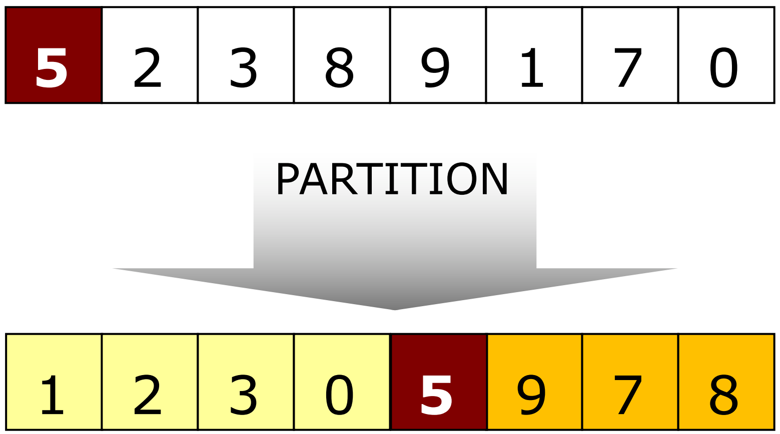
  QuickSort(array, pivotPosition);
  QuickSort(array+PivotPosition+1, n–PivotPosition–1);
}
```

```
void QuickSort(int array[], int n) {
    int pivotPosition;
    if (n<=1) return;
    pivotPosition = Partition(array, n);
    QuickSort(array, pivotPosition);
    QuickSort(array+PivotPosition+1, n–PivotPosition–1);
}
```

Note:

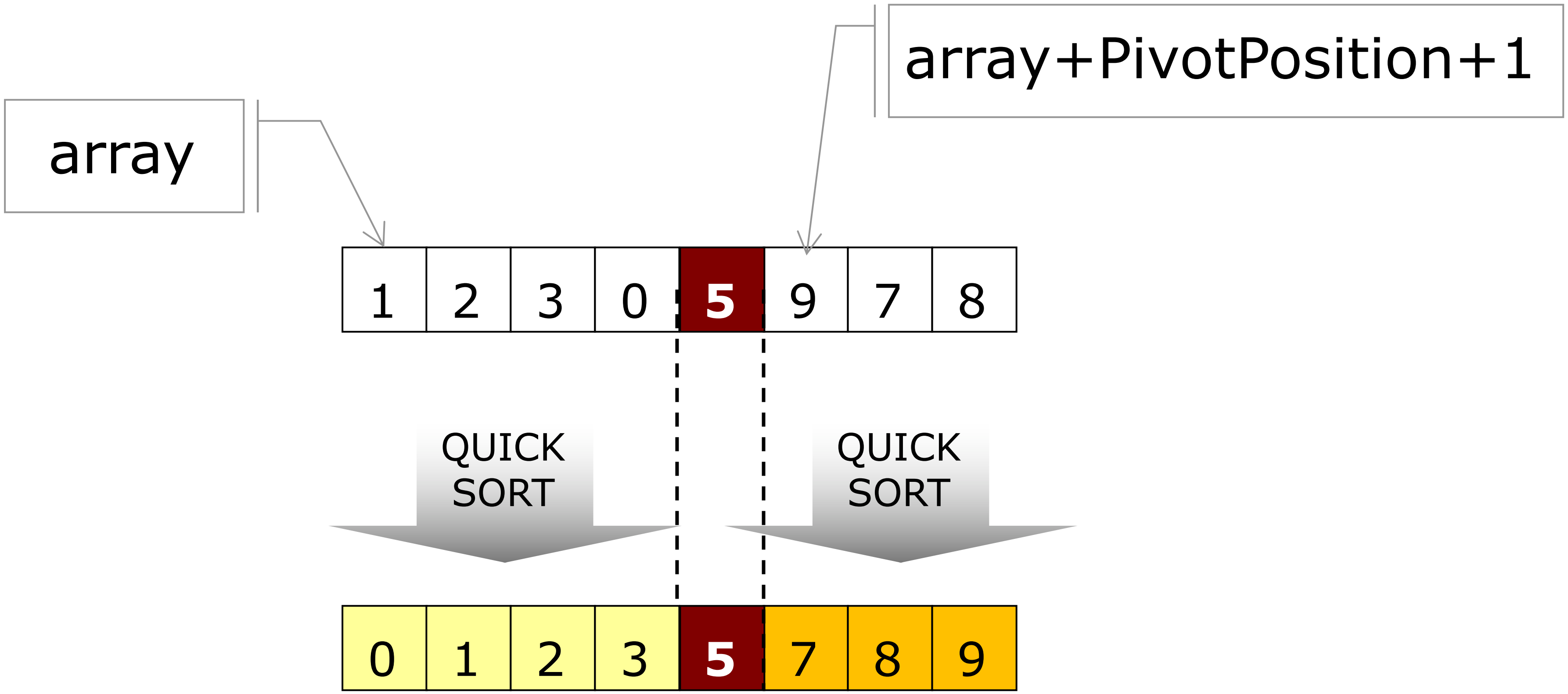
- **if (n<=1) return;**
  If the array is empty, or contains only 1 element, then it is already sorted. There is no need to do anything.

- **pivotPosition = Partition(array, n);**
  In this example, pivotPosition = 4.



```
void QuickSort(int array[], int n) {
    int pivotPosition;
    if (n<=1) return;
    pivotPosition = Partition(array, n);
    QuickSort(array, pivotPosition);
    QuickSort(array+PivotPosition+1, n–PivotPosition–1);
}
```

- **QuickSort(array, pivotPosition); QuickSort(array+PivotPosition+1, n–PivotPosition–1);**

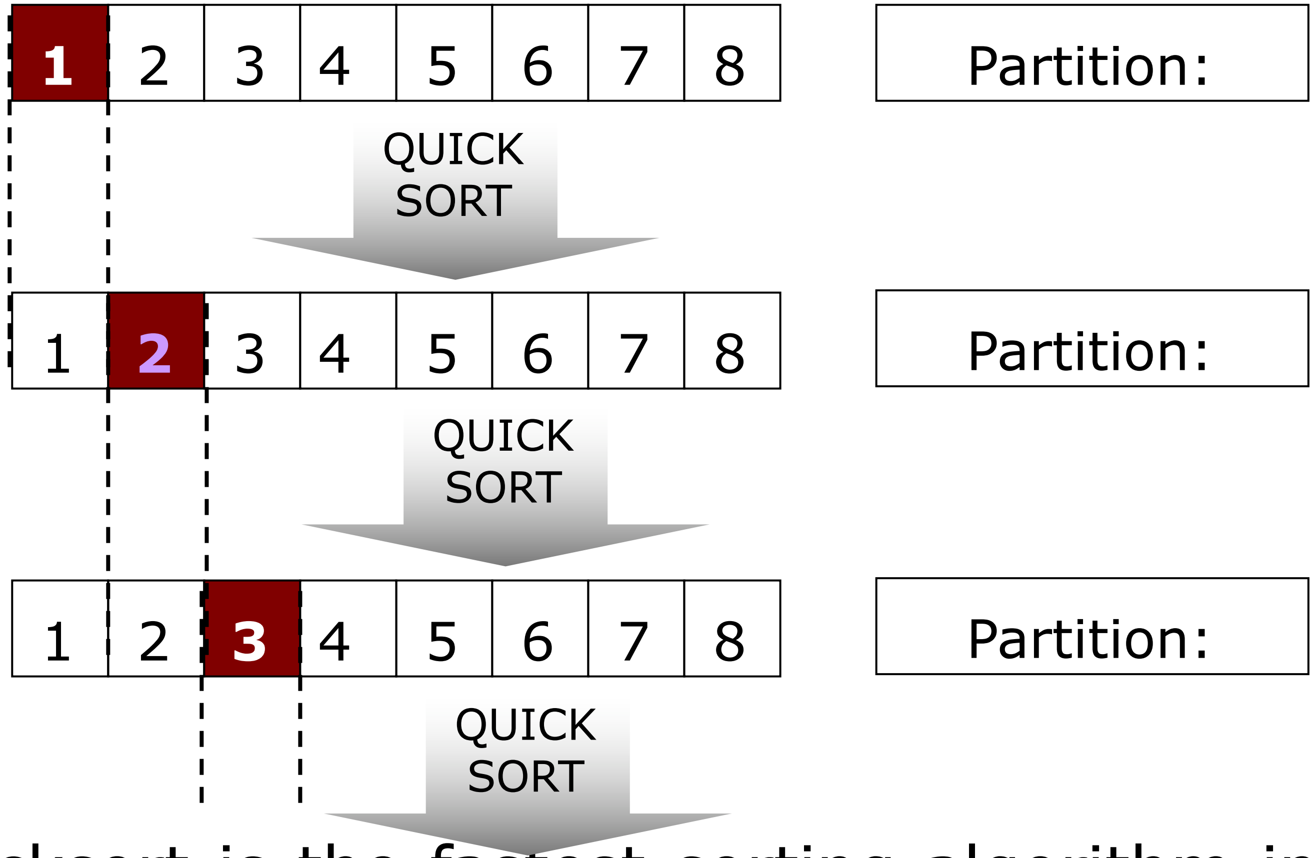


Complexity of Quicksort

The following is simply stated and we do not show how to derive:

The worst-case complexity of Quicksort is $O(N^2)$.
The average-case complexity of Quicksort is $O(N \log N)$.

The worst case happens when the array is already sorted.



On average, Quicksort is the fastest sorting algorithm in practice.