COMP2012H
Spring 2014
Dekai Wu

Sorting

(more on sorting algorithms: mergesort, quicksort, heapsort)

Merge Sort

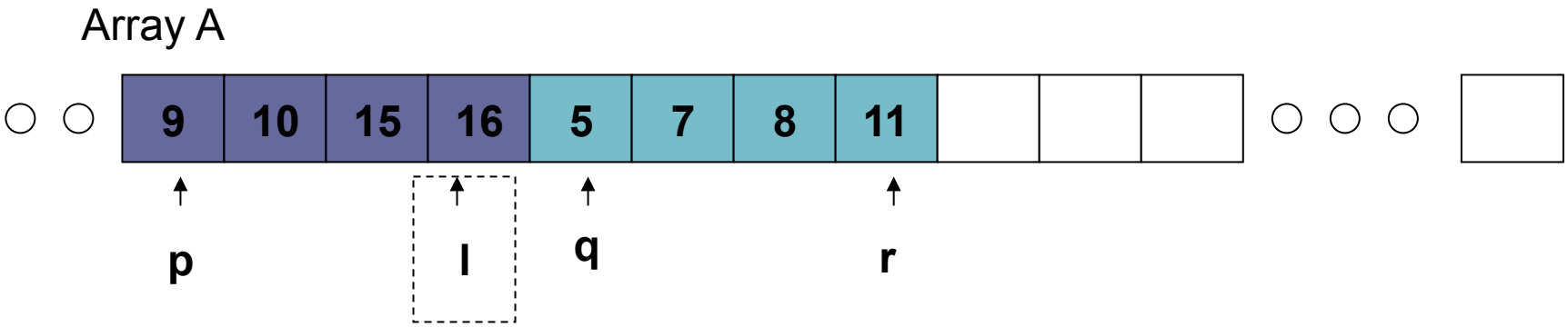
```
void mergesort(vector<int> & A, int left, int right)
{
   if (left < right) {
      int center = (left + right)/2;
      mergesort(A,left,center);
      mergesort(A,center+1,right);
      merge(A,left,center+1,right);
   }
}
```

Recursive sorting strategy.
Let's look at merge(. .) first.

Algorithm *merge*(A, p, q, r)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
($* \ T$ is a temporary array. $*$)

```
1.   k = p; i = 0; l = q − 1;
2.   while p ≤ l and q ≤ r
3.       do if A[p] ≤ A[q]
4.              then T[i] = A[p]; i = i + 1; p = p + 1;
5.              else  T[i] = A[q]; i = i + 1; q = q + 1;
6.   while p ≤ l
7.       do T[i] = A[p]; i = i + 1; p = p + 1;
8.   while q ≤ r
9.       do T[i] = A[q]; i = i + 1; q = q + 1;
10.  for i = k to r
11.      do A[i] = T[i − k];
```

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



Because of the recursive nature of the mergesort, we know that the end of the first sub-array is just before the beginning of the second array, so . . **l = q -1**.

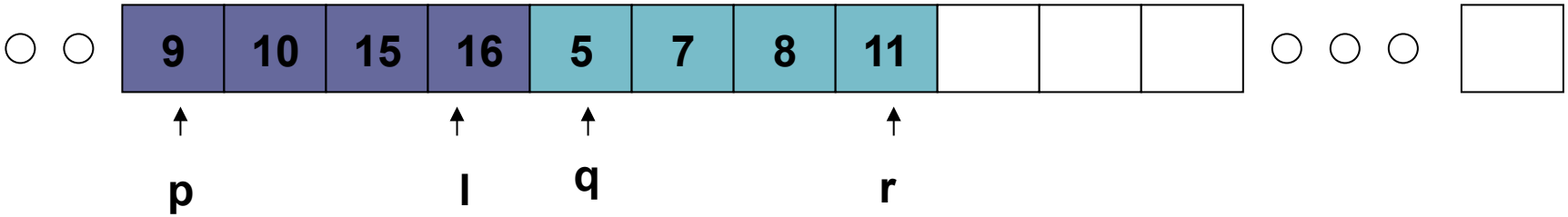
Temporary array T

Algorithm *merge*(A, p, q, r)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
($*$ T is a temporary array. $*$)

```
1.   k = p; i = 0; l = q − 1;
2.   while p ≤ l and q ≤ r
3.       do if A[p] ≤ A[q]
4.              then T[i] = A[p]; i = i + 1; p = p + 1;
5.              else  T[i] = A[q]; i = i + 1; q = q + 1;
6.   while p ≤ l
7.       do T[i] = A[p]; i = i + 1; p = p + 1;
8.   while q ≤ r
9.       do T[i] = A[q]; i = i + 1; q = q + 1;
10.  for i = k to r
11.      do A[i] = T[i − k];
```

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**

Array A



k=p

Just compare elements at **p** and **q** and copies the smaller into position **i** in the temp array T.

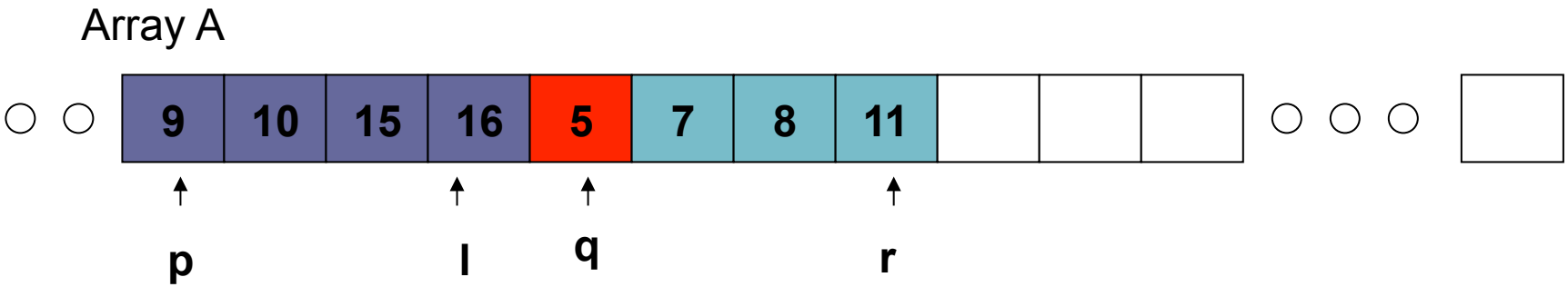
If A[p] <= A[q] p=p+1 else q=q+1.
i = i + 1

```
1.  k = p; i = 0; l = q − 1;
2.  while p ≤ l and q ≤ r
3.     do if A[p] ≤ A[q]
4.           then T[i] = A[p]; i = i + 1; p = p + 1;
5.           else  T[i] = A[q]; i = i + 1; q = q + 1;
```

Temporary array T

i=0

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



k=p

A[q] is smallest

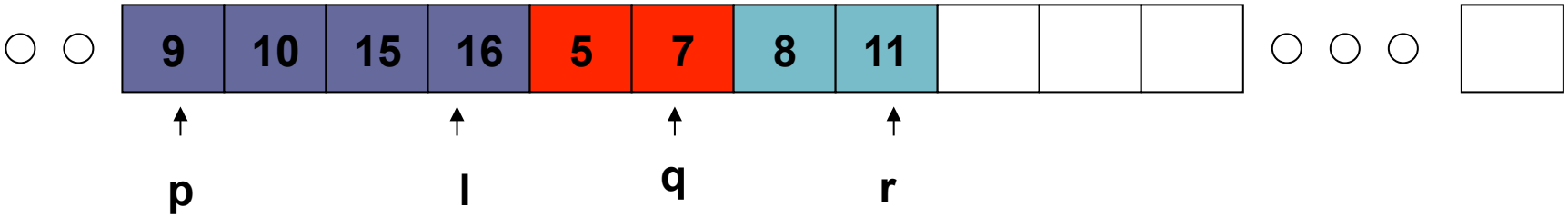
T[i] = A[q]
i++, q++

Temporary array T

i=0

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**

Array A



k=p

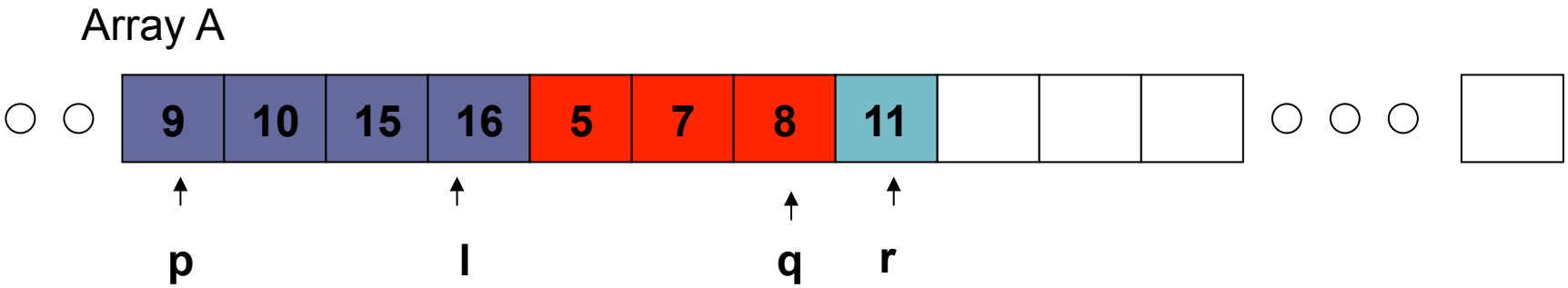
A[q] is smallest

T[i] = A[q]
i++, q++

i=1

Temporary array T

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



k=p

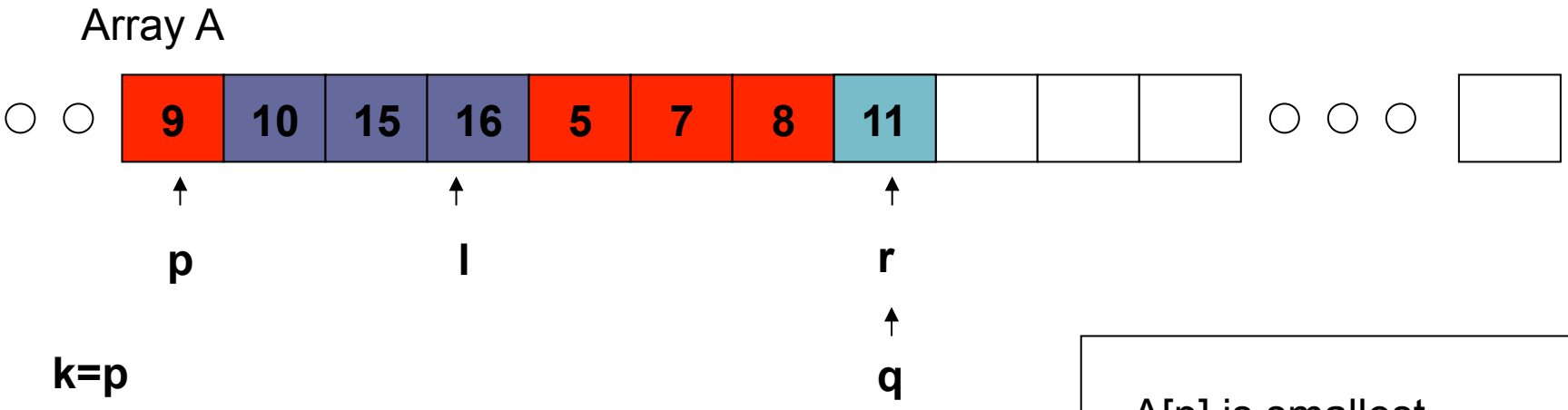
A[q] is smallest

T[i] = A[q]
i++, q++

i=2

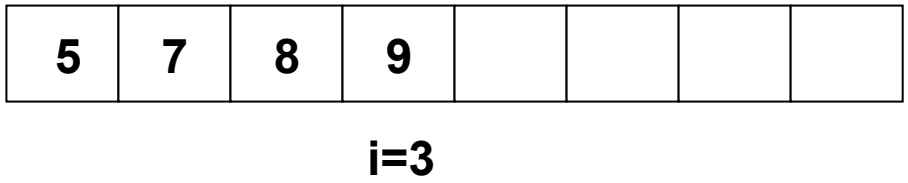
Temporary array T

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



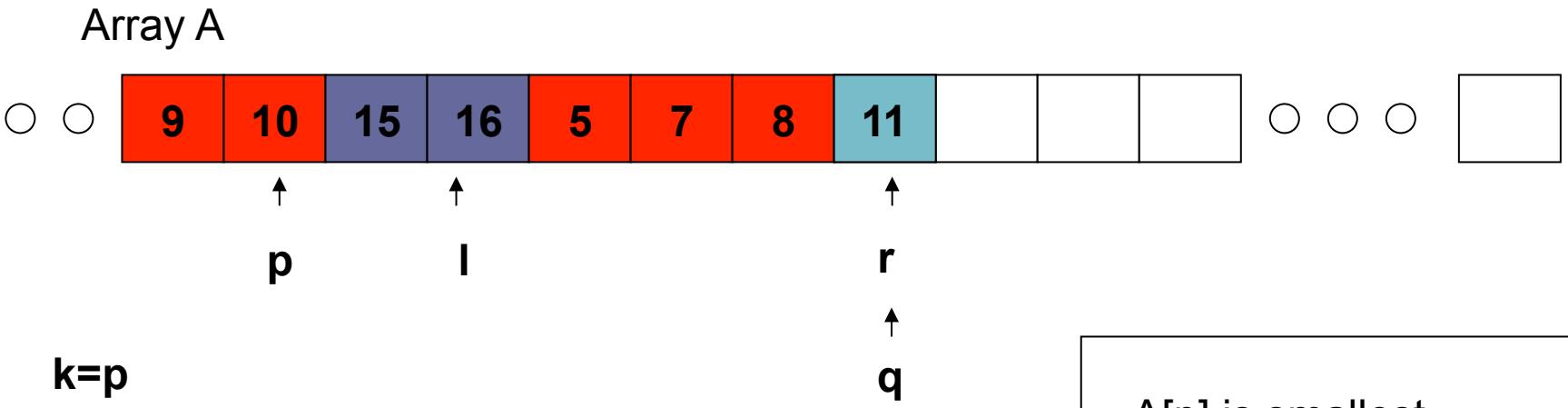
A[p] is smallest

T[i] = A[p]
i++, p++



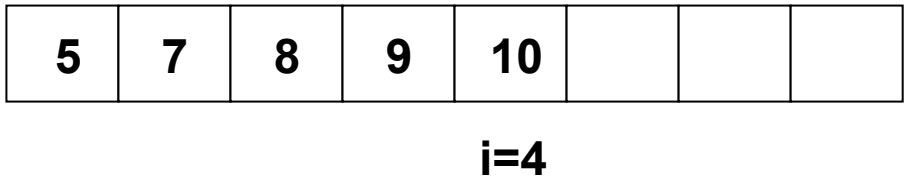
Temporary array T

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



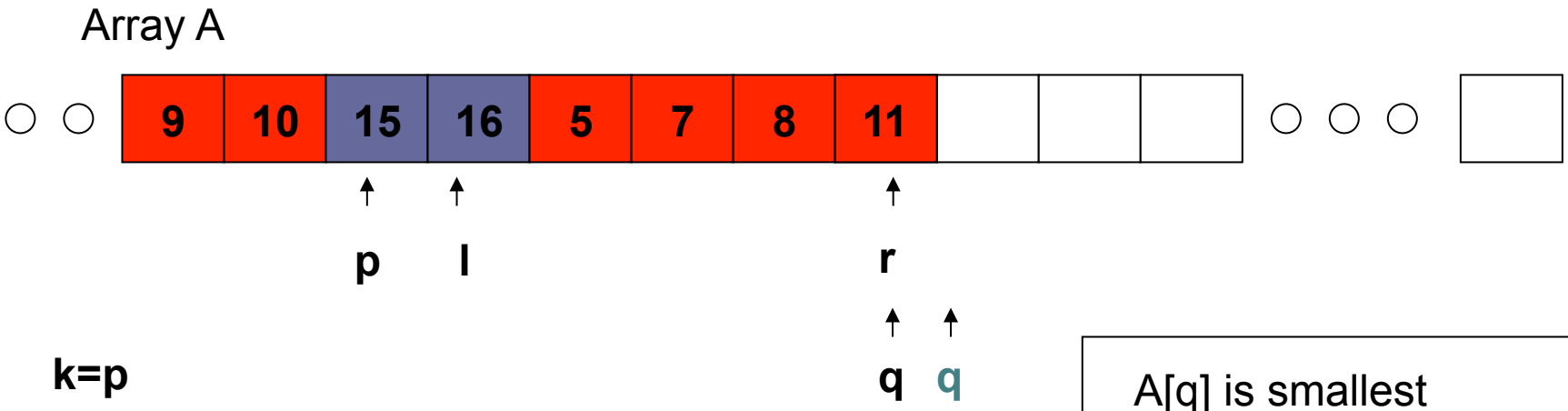
A[p] is smallest

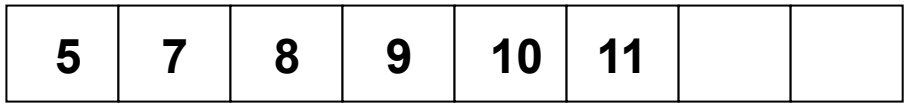
T[i] = A[p]
i++, p++



Temporary array T

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**





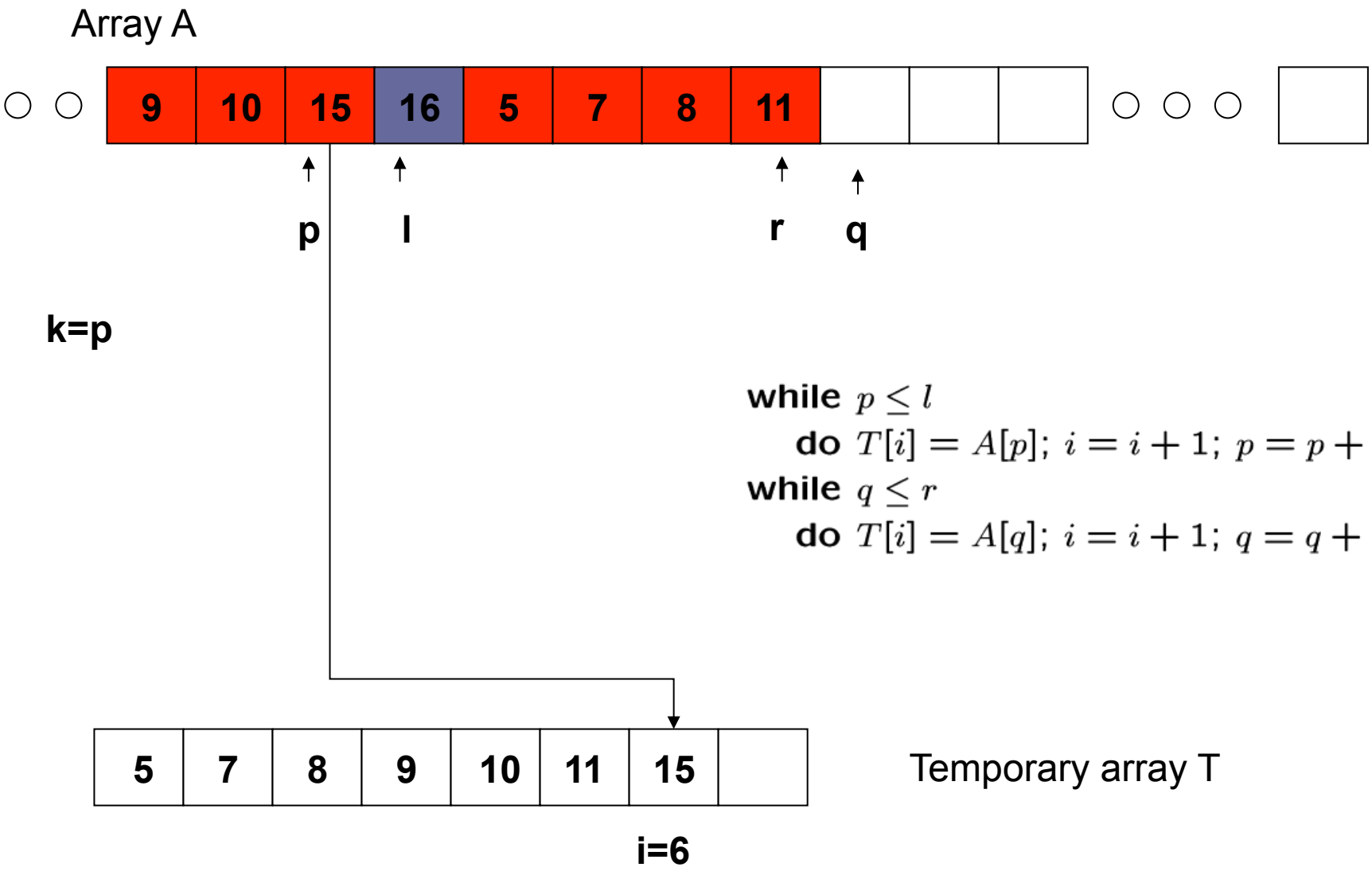
Temporary array T

Algorithm *merge*(A, p, q, r)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
$(*$ T is a temporary array. $*)$

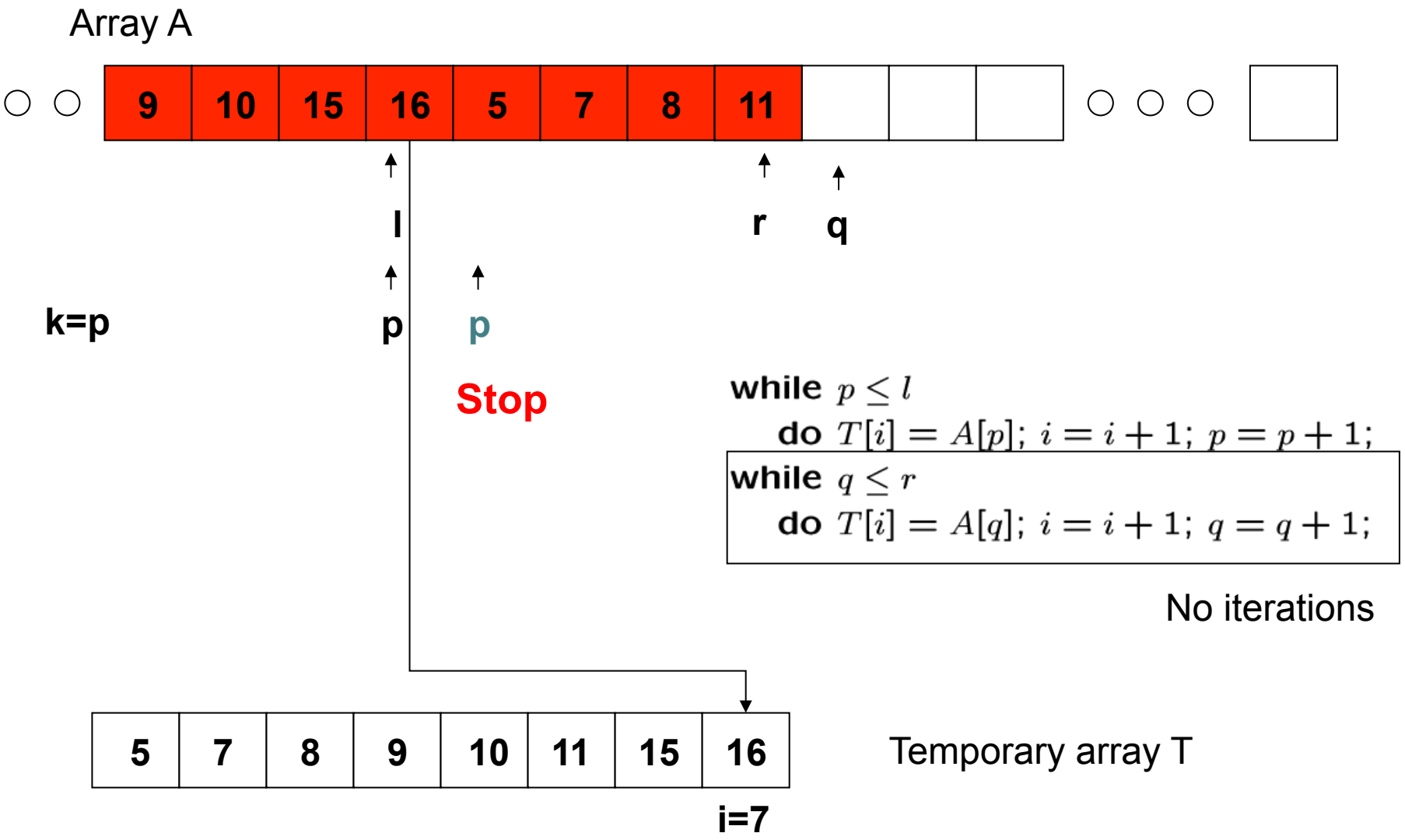
```
1.   k = p; i = 0; l = q − 1;
2.   while p ≤ l and q ≤ r
3.       do if A[p] ≤ A[q]
4.              then T[i] = A[p]; i = i + 1; p = p + 1;
5.              else  T[i] = A[q]; i = i + 1; q = q + 1;
6.   while p ≤ l
7.       do T[i] = A[p]; i = i + 1; p = p + 1;
8.   while q ≤ r
9.       do T[i] = A[q]; i = i + 1; q = q + 1;
10.  for i = k to r
11.      do A[i] = T[i − k];
```

COPY Remainder of arrays (if anything is there)

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**



Algorithm *merge*(A, p, q, r)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
($*$ T is a temporary array. $*$)

```
1.   k = p; i = 0; l = q − 1;
2.   while p ≤ l and q ≤ r
3.       do if A[p] ≤ A[q]
4.              then T[i] = A[p]; i = i + 1; p = p + 1;
5.              else  T[i] = A[q]; i = i + 1; q = q + 1;
6.   while p ≤ l
7.       do T[i] = A[p]; i = i + 1; p = p + 1;
8.   while q ≤ r
9.       do T[i] = A[q]; i = i + 1; q = q + 1;
10.  for i = k to r
11.      do A[i] = T[i − k];
```

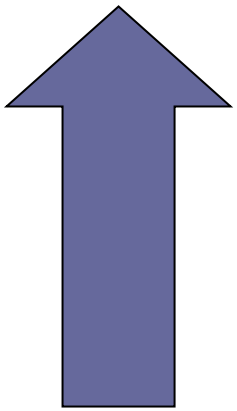
COPY Temp back to A

MERGING TWO SORTED Sub-Arrays . . . Parameters passed **Merge(A, p, q, r)**

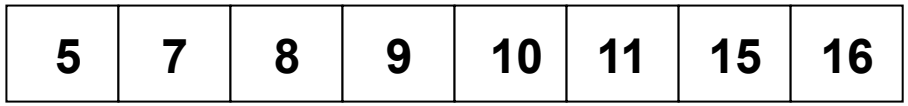
Array A

k=p

r



for $i = k$ to r
do $A[i] = T[i - k]$;



i=7

Temporary array T

Iterative Mergesort

- There is an iterative version of mergesort
  - Bottom-up version of the recursive mergesort
  - Merge in groups of single element
  - Then merge in groups of 2 elements
  - Then in groups of 4 elements
  - Then in groups of 8 elements
  - Etc.
- Merge groups of size s from `a` to `b` (temporary array), and then merge groups of size `2s` from `b` to `a`
- Need to worry about the last group with incomplete elements
  - It may be larger than s but smaller than `2s` → do merging on them
  - It may be smaller `s` → directly copy that segment to the other array
- `msort.h, msort.cpp`

QuickSort

- As the name implies, it is quick. This is the algorithm generally preferred for sorting.

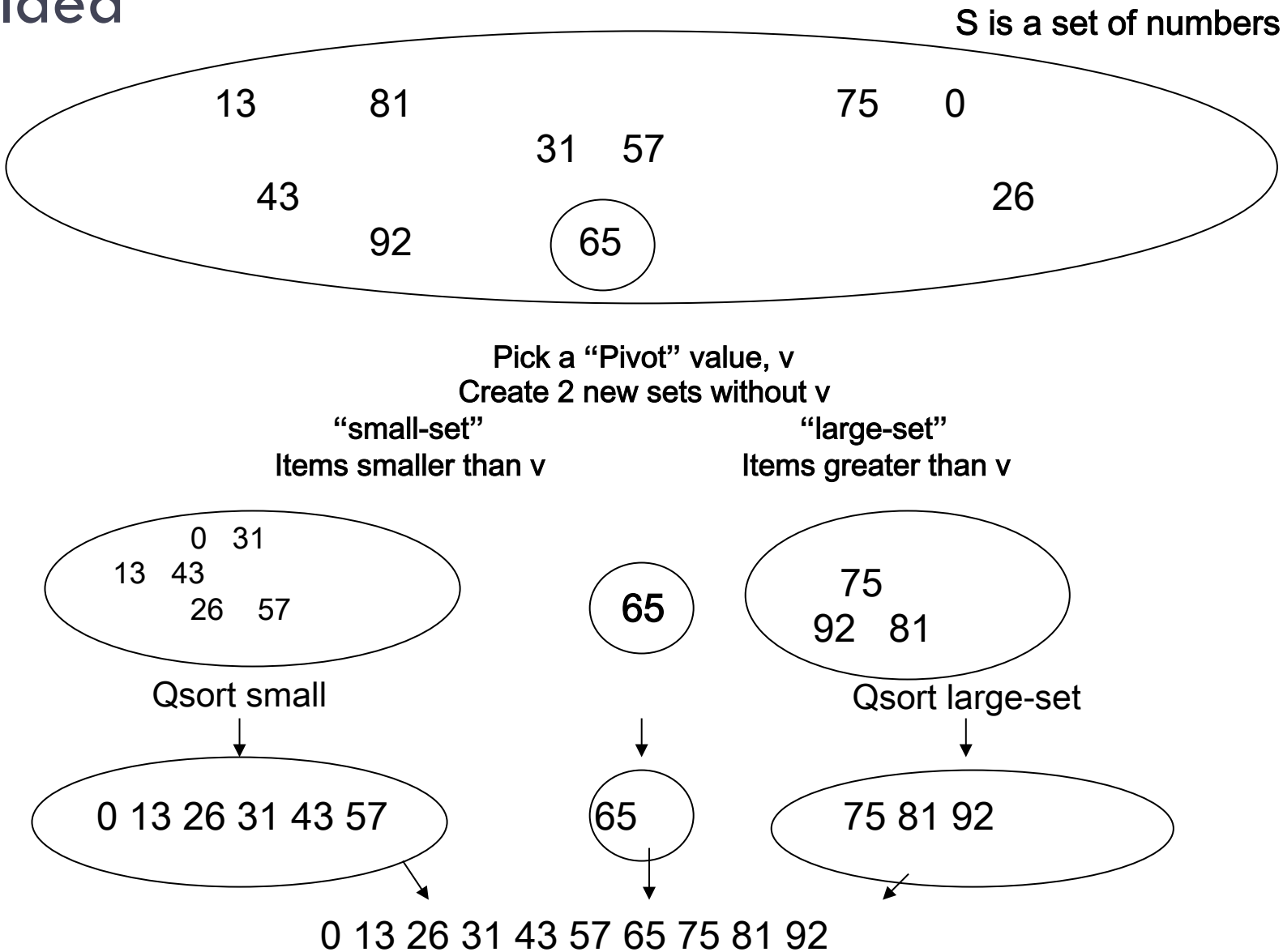
1. If the number of elements in set S is 0 or 1, then return
2. Pick any element v in S. This is the pivot
3. Partition S – {v} (the remaining element in S without v) into two disjoint groups, S1 and S2 such that:

   $S1 = \{x \in S - \{v\} \mid x \leq v\}$, and $S2 = \{ x \in S - \{v\} \mid v \leq x\}$.

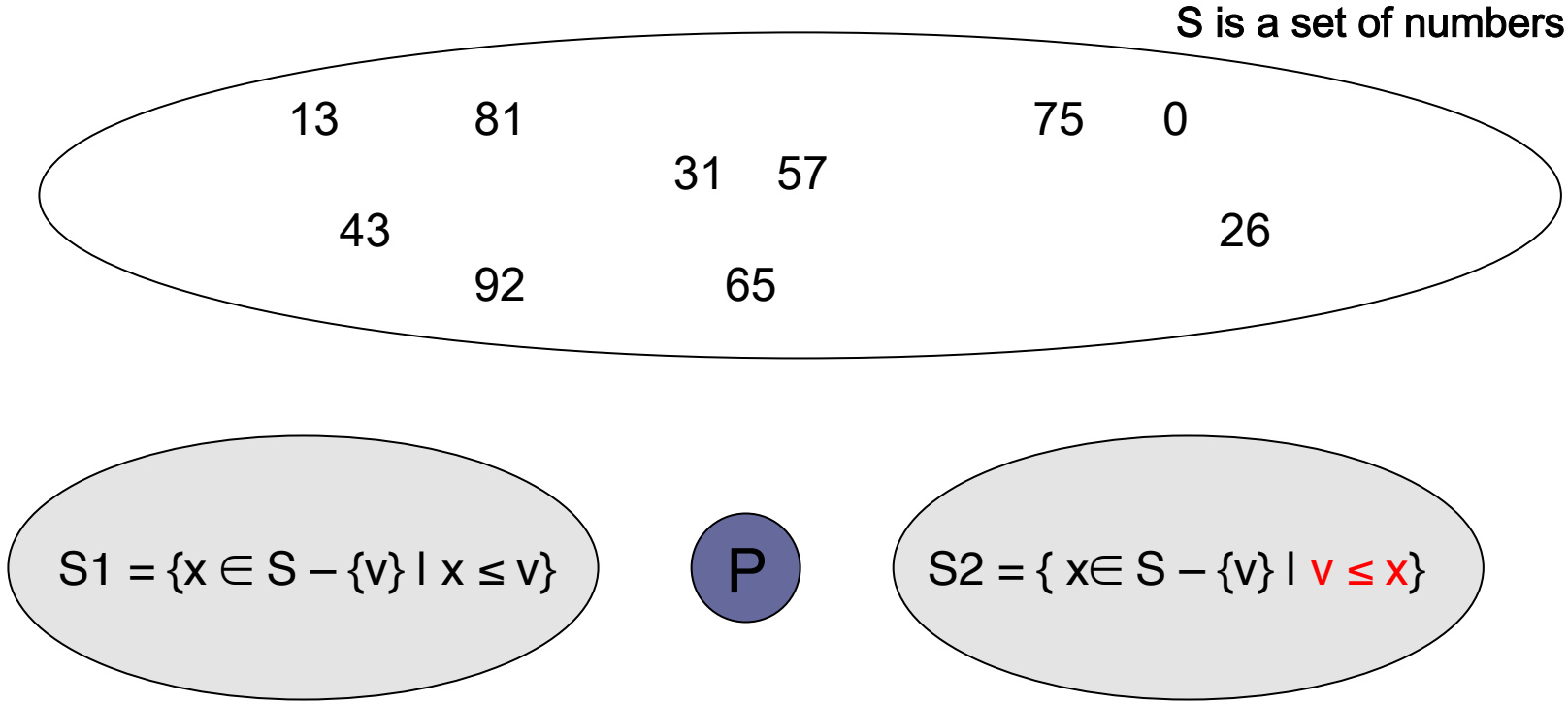
4. Return {quicksort(S1), v, quicksort(S2)}

   (that is the results of qsort(s1) followed by v followed by the results of quicksort(S2))

QSort Idea


S is a set of numbers
13 81 75 0
31 57
43 26
92 65
Pick a "Pivot" value, v
Create 2 new sets without v
"small-set"
Items smaller than v
"large-set"
Items greater than v
0 31
13 43
26 57
65
75
92 81
Qsort small
Qsort large-set
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

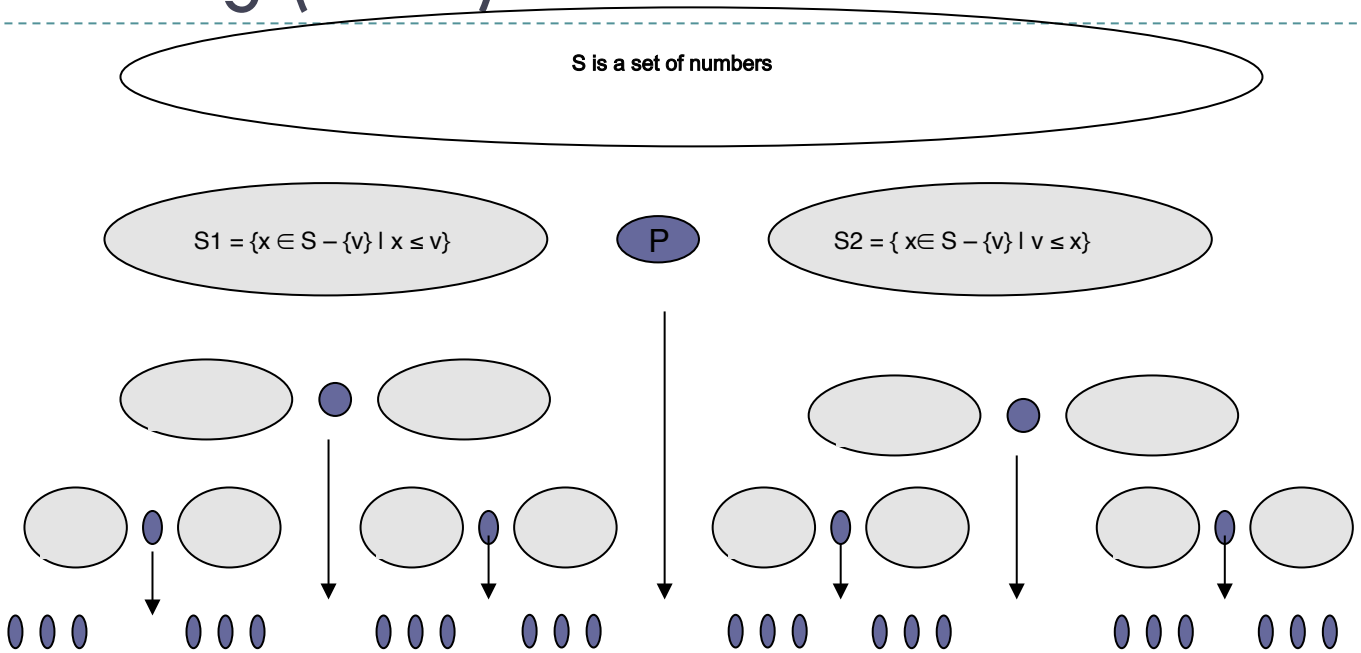
Partitioning (Pivot)



We'd like to pick a pivot that divides set S into two equal sized sets, S1 and S2.

Why?

Partitioning (Pivot)



If we had equal-sized sets, quick sort would break the problem into 2 equal-sized sub-problems at each recursive step.

Quicksort algorithm

Assume that we are sorting the subarray $A[l..r]$
l is left and r is right

- Divide step: Pick an element x from $A[l..r]$. Use x to partition the subarray into $A[l..q]$ and $A[q+1..r]$ such that

  each element in $A[l..q] \leq x$
  each element in $A[q+1..r] \geq x$

- Conquer step: Recursively sort the subarrays $A[l..q]$ and $A[q+1..r]$.
- Combine step: nothing extra needs to be done.

Is this fast?

- It should be clear that this works.
- The question is whether this is efficient as compared to mergesort

- This seems a little like merge-sort.
  Only the divide-and-conquer steps does not have to be of equal size.
  Also, the arrays have been partitioned into largest, smallest.

- We will see that the advantage of this approach is that we can sort "in-place", without the need for a temporary buffer.

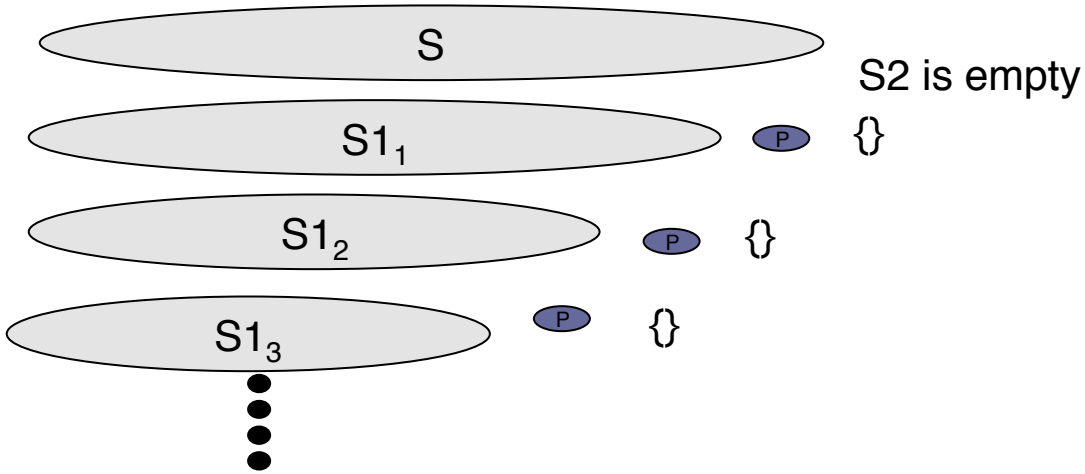
  Also, you should see there is no need for a merge of sub-arrays.

How bad could happen?

- What if we chose the worst pivot at each recursive level?

  -Bad pivot results in only one set, the other is empty { }.

  Thus the recursive call only reduces the problem by 1 each call.

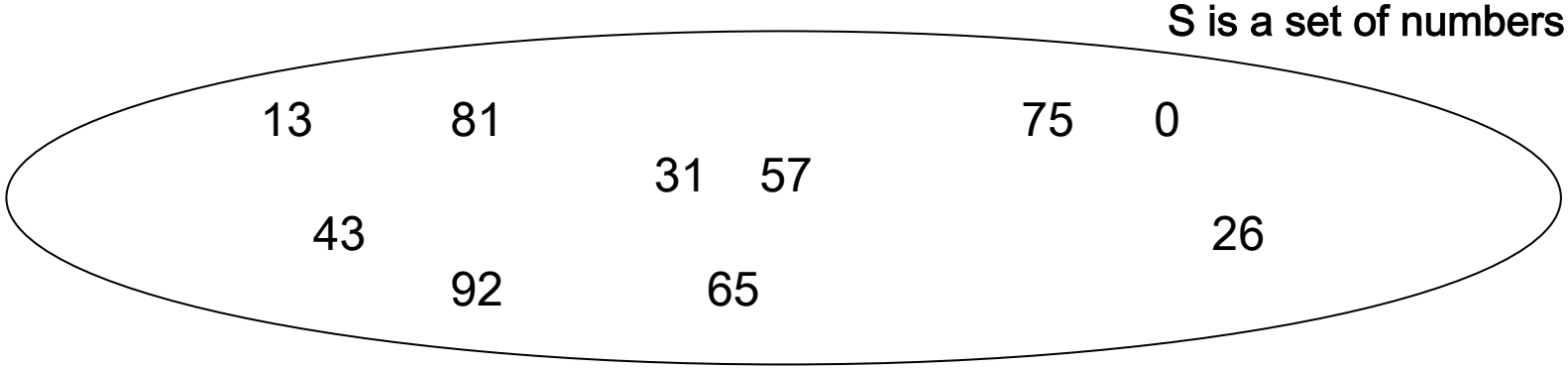


How many times could this happen?
n times.

Things to consider

- How to pick the “pivot” value from a set.
- How to partition quickly
- Tricks to speed things up.

What is a good pivot?



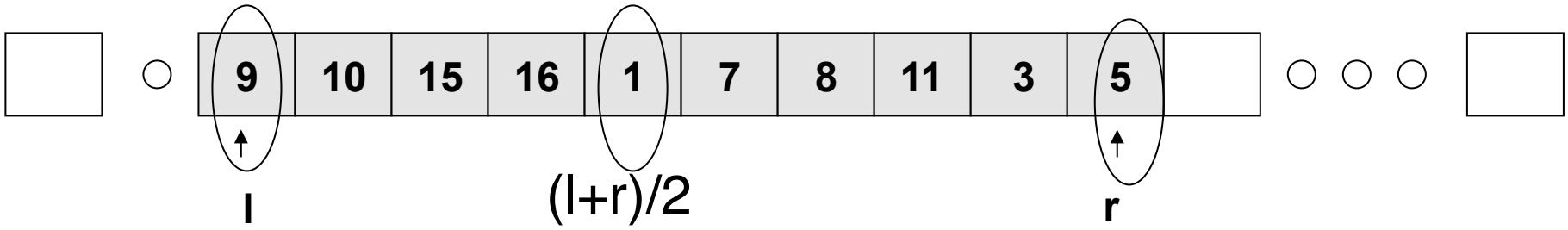
0, 13, 26, 31, 43, 57, 65, 75, 81, 92

The median is a good choice.

But, it is hard to find.

The set should already be sorted to find the median.

Maybe we can *estimate* the median



Pick a pivot?

How about "Median" of 3 Numbers?

A[l], A[r], A[(l+r)/2]

9, 1, 5

Is this a good choice?

What if the data is already sorted?

What if it is random?

Quicksort C++ Code

```
        /**
         * Quicksort method that makes recursive calls.
         * Uses median-of-three partitioning and a cutoff of 10.
         * a is an array of Comparable items.
         * left is the left-most index of the subarray.
         * right is the right-most index of the subarray.
         */
        template <class Comparable>
        void quicksort( vector<Comparable> & a, int left, int right )
        {
/* 1*/      if( left + 10 <= right )
            {
/* 2*/          Comparable pivot = median3( a, left, right );

                    // Begin partitioning
/* 3*/          int i = left, j = right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( a[ ++i ] < pivot ) { }
/* 6*/              while( pivot < a[ --j ] ) { }
/* 7*/              if( i < j )
/* 8*/                  swap( a[ i ], a[ j ] );
                    else
/* 9*/                  break;
                }

/*10*/          swap( a[ i ], a[ right - 1 ] );   // Restore pivot

/*11*/          quicksort( a, left, i - 1 );      // Sort small elements
/*12*/          quicksort( a, i + 1, right );     // Sort large elements
            }
            else  // Do an insertion sort on the subarray
/*13*/          insertionSort( a, left, right );
        }
```

C++ Code

```
/**
 * Standard swap
 */
template <class Comparable>
inline void swap( Comparable & obj1, Comparable & obj2 )
{
    Comparable tmp = obj1;
    obj1 = obj2;
    obj2 = tmp;
}
```

Swap.

```
/**
 * Return median of left, center, and right.
 * Order these, (the pivot is placed in a[right-1])
 */
template <class Comparable>
const Comparable & median3( vector<Comparable> & a, int left,
int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
    swap( a[ center ], a[ right - 1 ] );
    return a[ right - 1 ];
}
```

Sort the three numbers: Put the smallest of the three on the left, largest on the right, the median in the middle

Median of 3.

Examine one pass of quicksort

```
        /**
         * Quicksort method that makes recursive calls.
         * Uses median-of-three partitioning and a cutoff of 10.
         * a is an array of Comparable items.
         * left is the left-most index of the subarray.
         * right is the right-most index of the subarray.
         */
        template <class Comparable>
        void quicksort( vector<Comparable> & a, int left, int right )
        {
/* 1*/      if( left + 10 <= right )
            {
/* 2*/          Comparable pivot = median3( a, left, right );

                // Begin partitioning
/* 3*/          int i = left, j = right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( a[ ++i ] < pivot ) { }
/* 6*/              while( pivot < a[ --j ] ) { }
/* 7*/              if( i < j )
/* 8*/                  swap( a[ i ], a[ j ] );
                    else
/* 9*/                  break;
                }

/*10*/          swap( a[ i ], a[ right - 1 ] );  // Restore pivot

/*11*/          quicksort( a, left, i - 1 );     // Sort small elements
/*12*/          quicksort( a, i + 1, right );    // Sort large elements
            }
            else  // Do an insertion sort on the subarray
/*13*/          insertionSort( a, left, right );
        }
```

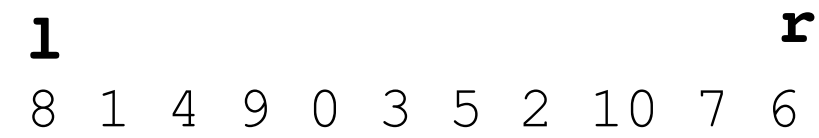
First call Median3(..)

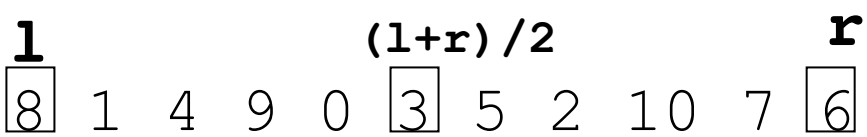
Input A[l..r]

l											r
8	1	4	9	0	3	5	2	10	7	6	

Median of 3

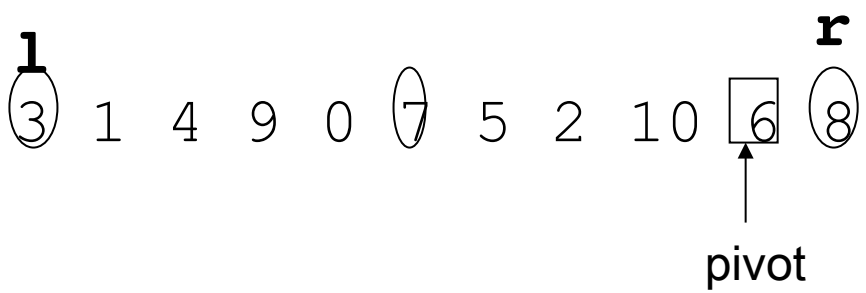
Input A[l..r]





Call median3()
finds median, places smallest a A[l],
largest at A[r], and pivot at A[r-1]

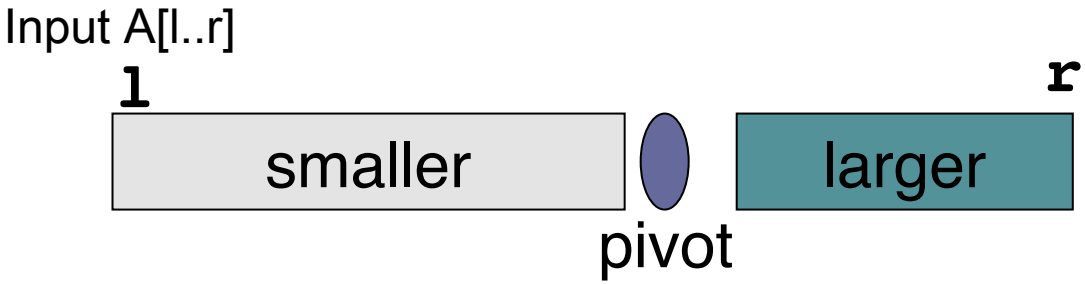
A[l..r] after median3()




```
median3(...) {

int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
  swap( a[ left ], a[ center ] );

if( a[ right ] < a[ left ] )
  swap( a[ left ], a[ right ] );

if( a[ right ] < a[ center ] )
  swap( a[ center ], a[ right ] );


// Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );


return a[ right - 1 ];

}
```

Partitioning

- After you pick the pivot
- You need to
  1. place elements smaller than the pivot value on one side of the array
  2. Place elements larger than the pivot value on the other side of the pivot
  3. Place the pivot between the two sets.



C++ Code

```
        /**
         * Quicksort method that makes recursive calls.
         * Uses median-of-three partitioning and a cutoff of 10.
         * a is an array of Comparable items.
         * left is the left-most index of the subarray.
         * right is the right-most index of the subarray.
         */
        template <class Comparable>
        void quicksort( vector<Comparable> & a, int left, int right )
        {
/* 1*/      if( left + 10 <= right )
            {
/* 2*/          Comparable pivot = median3( a, left, right );

                // Begin partitioning
/* 3*/          int i = left, j = right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( a[ ++i ] < pivot ) { }
/* 6*/              while( pivot < a[ --j ] ) { }
/* 7*/              if( i < j )
/* 8*/                  swap( a[ i ], a[ j ] );
                    else
/* 9*/                  break;
                }

/*10*/          swap( a[ i ], a[ right - 1 ] );  // Restore pivot

/*11*/          quicksort( a, left, i - 1 );     // Sort small elements
/*12*/          quicksort( a, i + 1, right );    // Sort large elements
            }
            else  // Do an insertion sort on the subarray
/*13*/          insertionSort( a, left, right );
        }
```

Partitioning Code

Partitioning

pivot=6

(0*) **l** **r**

3 1 4 9 0 7 5 2 10 6 8

i **j**

(1)

3 1 4 9 0 7 5 2 10 6 8

move i until A[i] >= pivot → **i** **j**

(2)

3 1 4 9 0 7 5 2 10 6 8

i **j** move juntil A[j] <= pivot

(3)

3 1 4 2 0 7 5 9 10 6 8

i **j**

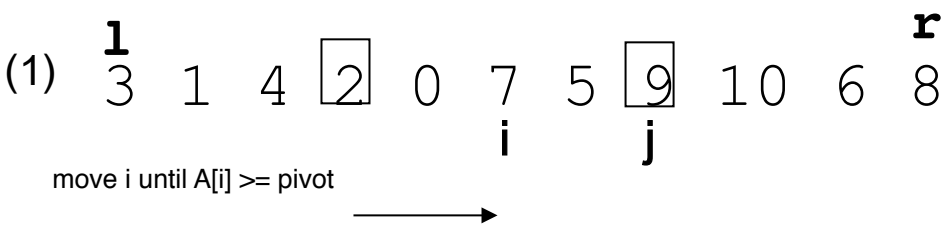
if (i < j) Yes
swap(A[i],A[j])

[repeat for(;;) back to (1)]

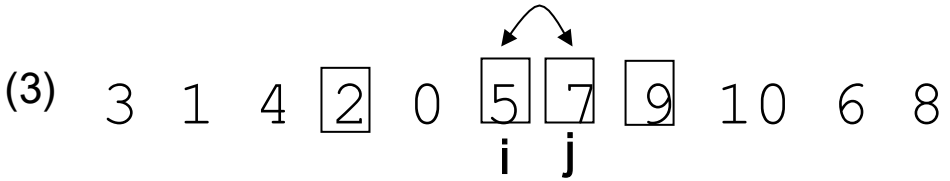
```
(0) int i = left, j = right - 1;
    for( ; ; )
    {
(1)  while( a[ ++i ] < pivot ) { }
(2)  while( pivot < a[ --j ] ) { }
(3)  if( i < j )
       swap( a[ i ], a[ j ] );
     else
       break;

     }

(4) swap( a[ i ], a[ right - 1 ] );
```

*[No need to start j at right, since the median3 function made sure A[right] is larger than the pivot

Partitioning



pivot=6



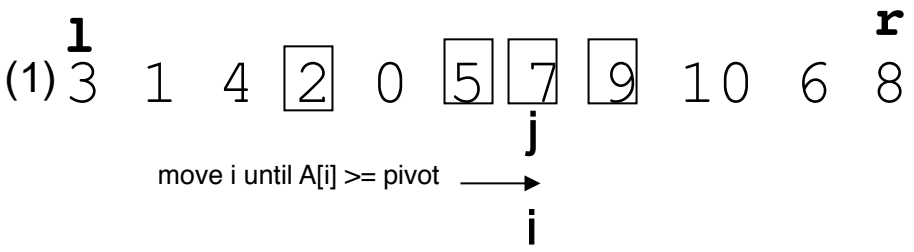
if (i < j) Yes
swap(A[i],A[j])

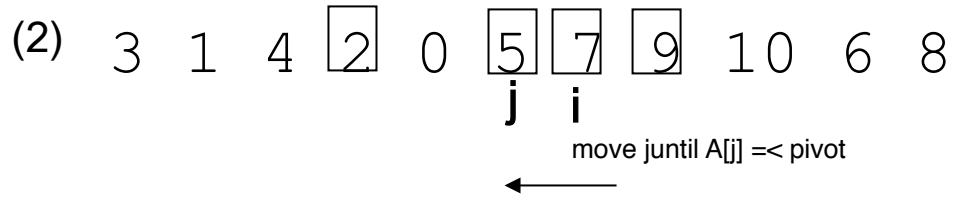
[repeat for(;;) back to (1)]

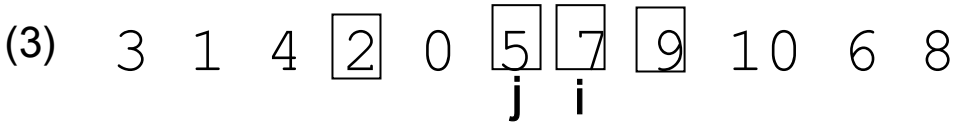
```
(0) int i = left, j = right - 1;
    for( ; ; )
    {
(1)  while( a[ ++i ] < pivot ) { }
(2)  while( pivot < a[ --j ] ) { }
(3)  if( i < j )
       swap( a[ i ], a[ j ] );
     else
       break;

     }
(4) swap( a[ i ], a[ right - 1 ] );
```

Partitioning

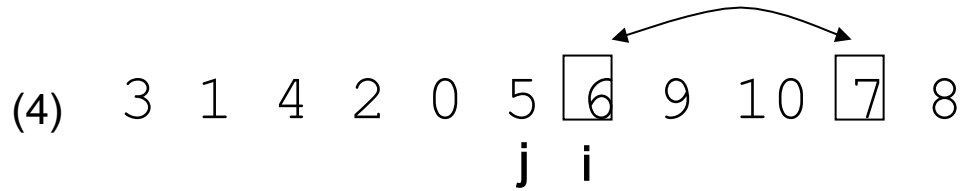
pivot=6





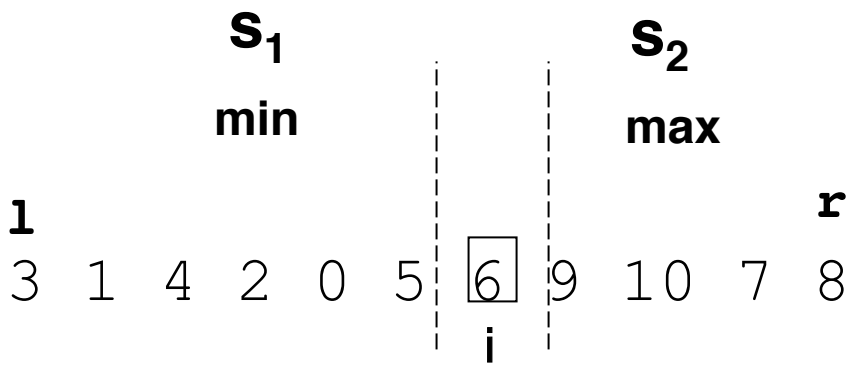


if (i < j) NO STOP




```
(0) int i = left, j = right - 1;
    for( ; ; )
    {
(1)  while( a[ ++i ] < pivot ) { }
(2)  while( pivot < a[ --j ] ) { }
(3)  if( i < j )
       swap( a[ i ], a[ j ] );
     else
       break;

     }
(4) swap( a[ i ], a[ right - 1 ] );
```

Swap pivot to i.
Remember pivot is at A[r-1]
(median3 placed the pivot there)



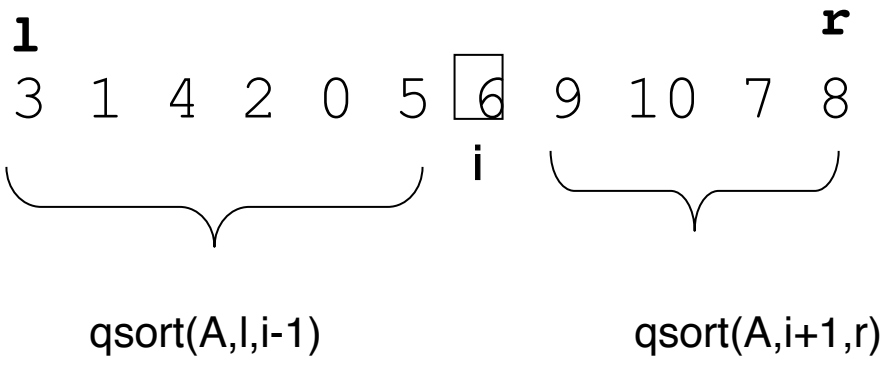
Now look. MAGIC.

A[l..i-1] is < pivot
A[i+1..r] is > pivot

and

The pivot is in its sorted position!

Now,recursively call qsort. . . . .



Heapsort

- Go back to the notes

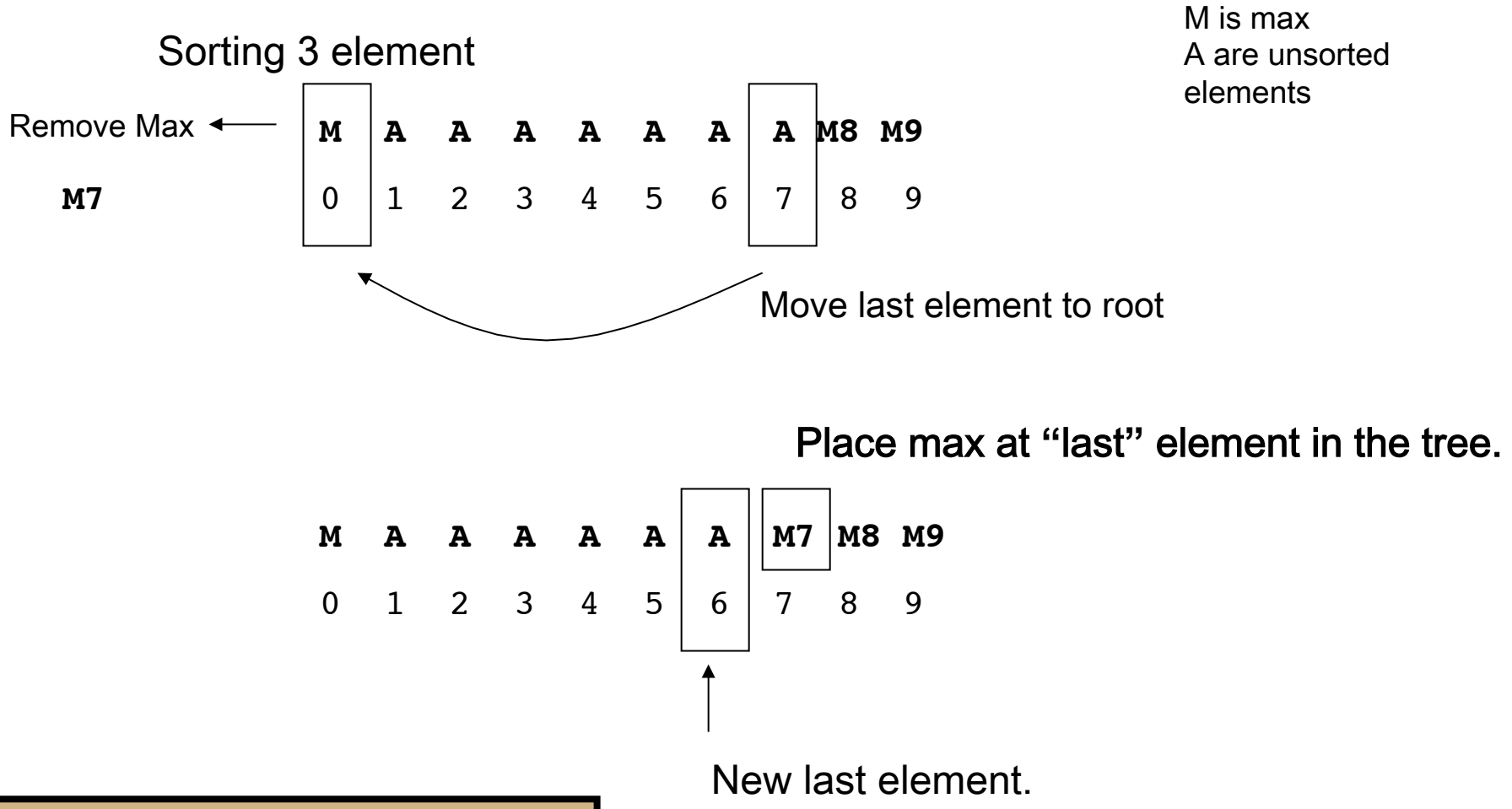
Heapsort: Sorting with Max-Heaps

•An in-place sorting algorithm

1. Build a max-heap
2. Remove max -> swap it with the last element in the heap
3. Rebuild the heap

This will result in a sorted list from smallest to largest.

Remove Max, place in the slot of the removed element



The result. When you are done, the array is sorted smallest to largest. No need for temporary buffer.