Partitioning

CS 350 – Algorithms and Complexity
Paul Doliotis – Adjunct Assistant Professor
Portland State University

Partitioning

- Partitioning takes three arguments:
  - An array a[].
  - A left index L.
  - A right index R.
- Partitioning rearranges array elements a[L], a[L+1], ..., a[R].
  - Elements before L or after R are not affected.
- Partitioning returns an index i such that:
  - When the function is done, a[i] is what a[R] was before the function was called.
    - We move a[R] to a[i].
  - All elements between a[L] and a[i-1] are not greater than a[i].
  - All elements between a[i+1] and a[R] are not less than a[i].

Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- What does partition(a, 0, 9) do in this case?

Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- What does partition(a, 0, 9) do in this case?
- The last element of the array is 35. 35 is called the **pivot**.
  - Array a[] has:
  - 3 elements less than the pivot.
  - 6 elements greater than the pivot.

Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- What does partition(a, 0, 9) do in this case?
- The last element of the array is 35. 35 is called the **pivot**.
  - Array a[] has:
  - 3 elements less than the pivot.
  - 6 elements greater than the pivot.
- Array a[] is rearranged so that:
  - First we put all values less than the pivot (35).
  - Then, we put the pivot.
  - Then, we put all values greater than the pivot.

Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- What does partition(a, 0, 9) do in this case?
- The last element of the array is 35. 35 is called the **pivot**.
  - Array a[] has:
  - 3 elements less than the pivot.
  - 6 elements greater than the pivot.
- Array a[] is rearranged so that:
  - First we put all values less than the pivot (35).
  - Then, we put the pivot.
  - Then, we put all values greater than the pivot.
- partition(a, 0, 9) **returns the new index of the pivot**, which is 3.

Partitioning

- Example: suppose we have this array a[]:

position	0	1	2	3	4	5	6	7	8	9
value	17	90	70	30	60	40	45	80	10	35

- How does array a[] look after we call partition(a, 0, 9)?

position	0	1	2	3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70

- Note that:
  - Items at positions 0, 1, 2 are not necessarily in sorted order.
  - However, items at positions 0, 1, 2 are all <= 35.
  - Similarly: items at positions 4, ..., 9 are not necessarily in sorted order.
  - However, items at positions 4, ..., 9 are all >= 35.

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

- Partitioning can be used to solve the k-th Median problem

Finding Median

- We can obviously find the median by sorting the array, and then picking the k-th element
- How much work is that (in average case)?

Finding Median

- We can obviously find the median by sorting the array, and then picking the k-th element
- How much work is that (in average case)?
  - O(n)
  - O(n lg n)
  - O(n2)
  - something else

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

- Partitioning can be used to solve the k-th Median problem
- What is “k” after partition(a,0,9)

Finding Median

- Array a after partition(a, 0,9)

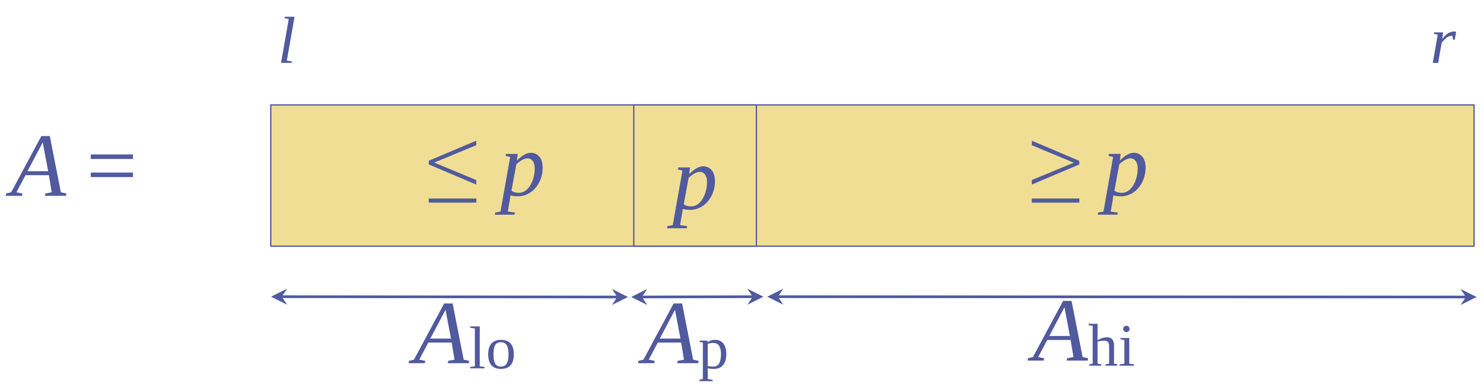
position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

- Partitioning can be used to solve the k-th Median problem
- What is “k” after partition(a,0,9)
  - items at positions 0, 1, 2 are all <= 35.
  - items at positions 4, ..., 9 are all >= 35.

Finding Median

- Array a after partition(a, 0,9)

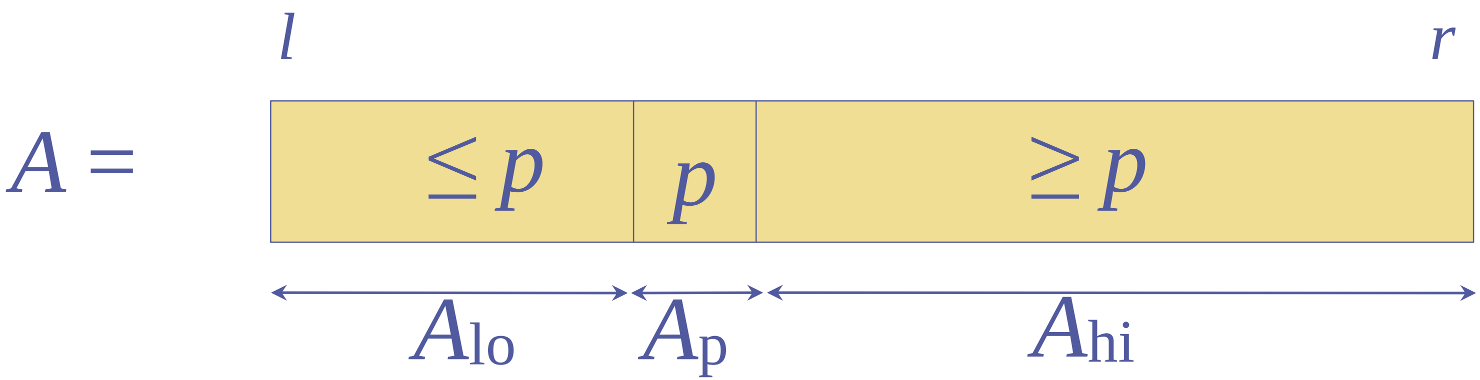
position	0	1	2	3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70



Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

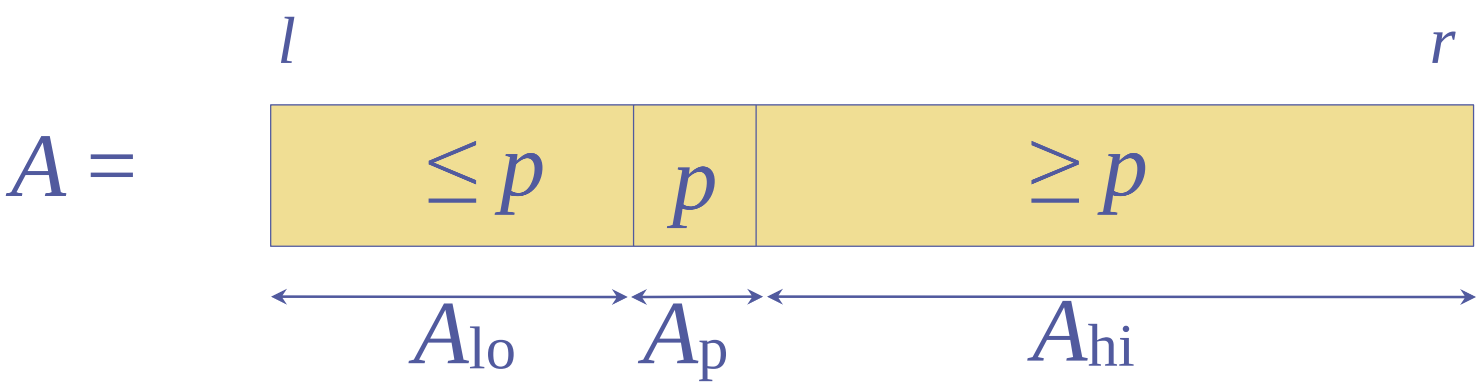


|Alo| = 3
|Ap| = 1
|Ahi| = 6

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70



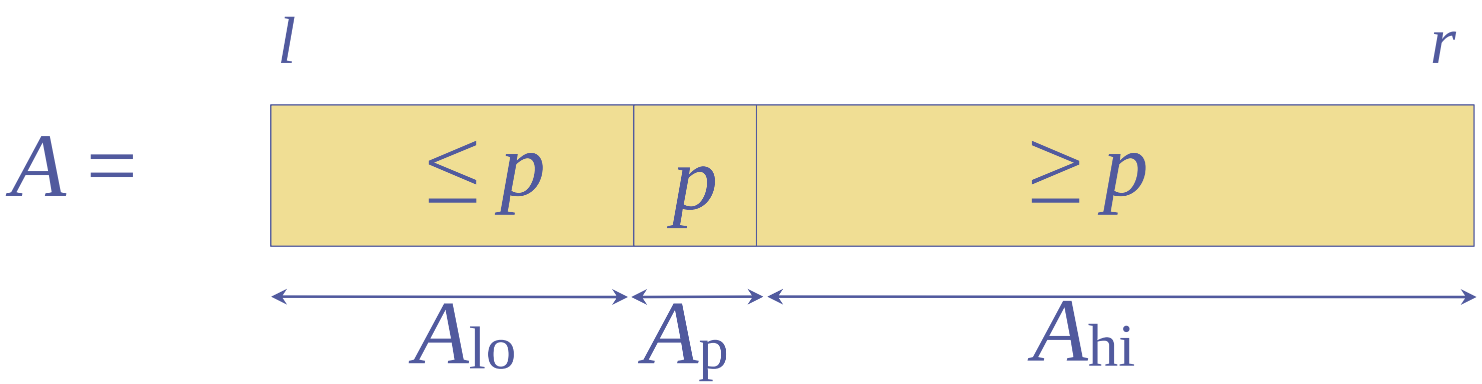
|Alo| = 3
|Ap| = 1
|Ahi| = 6

What if we were looking for 8^{th} median?

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70



|Alo| = 3
|Ap| = 1
|Ahi| = 6

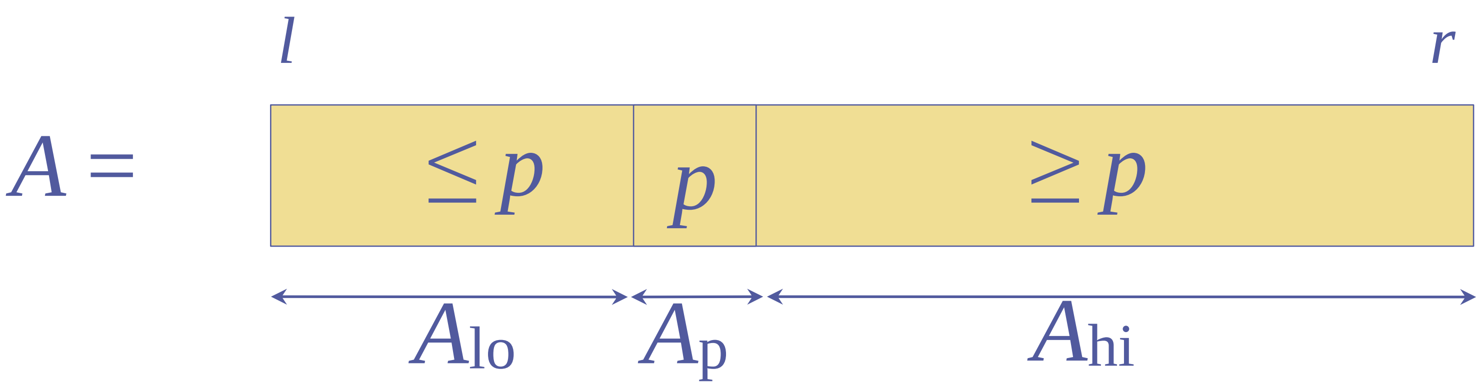
What if we were looking for 8^{th} median?

- Look into subarray Ahi [60,40,45,80,90,70] for its 4^{th} median

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70



|Alo| = 3
|Ap| = 1
|Ahi| = 6

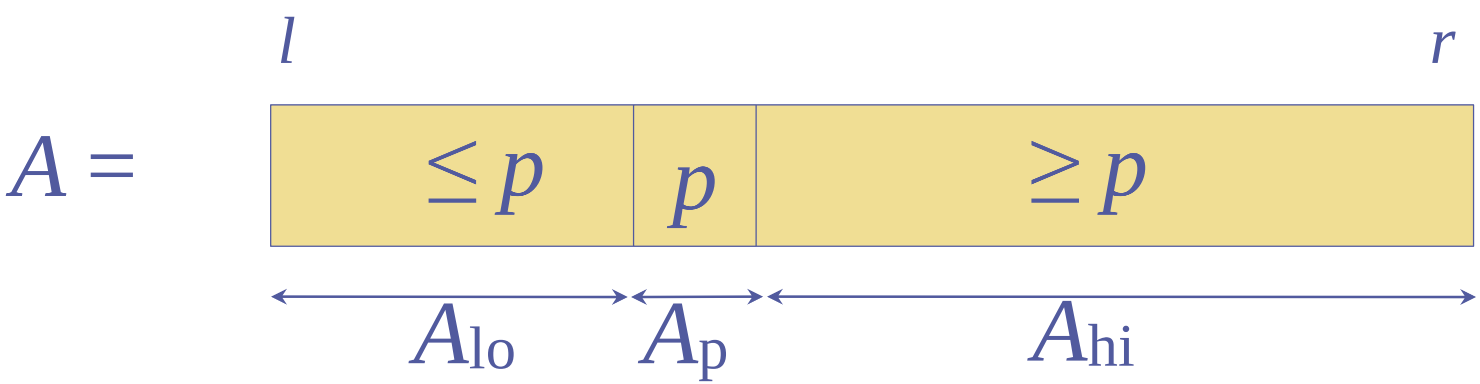
What if we were looking for 8^{th} median?

- Look into subarray Ahi [60,40,45,80,90,70] for its 4^{th} median
- Problem reduced by |Alo| + |Ap|

Finding Median

- Array a after partition(a, 0,9)

position	0	1	2	3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70



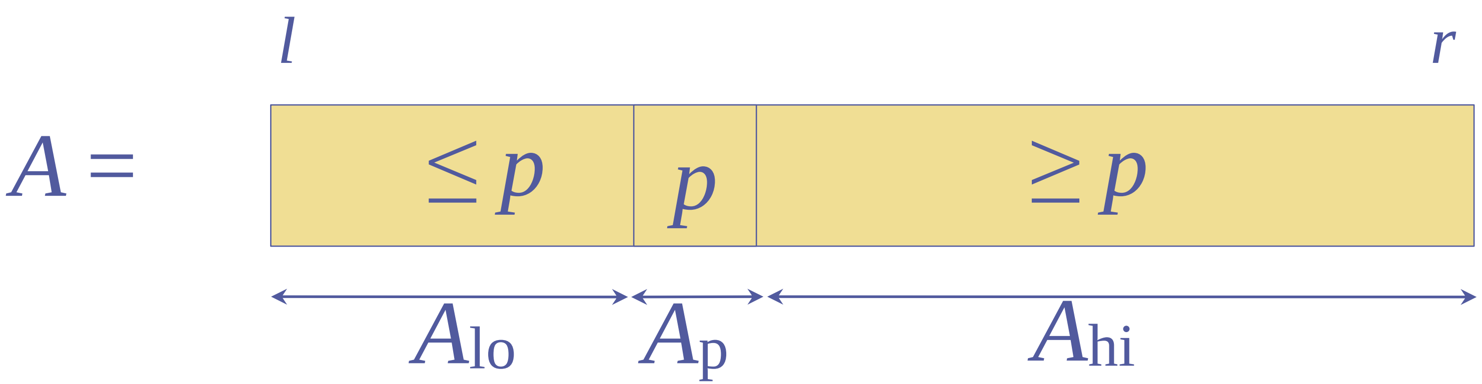
|Alo| = 3
|Ap| = 1
|Ahi| = 6

What if we were looking for 2nd median?

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70



|Alo| = 3
|Ap| = 1
|Ahi| = 6

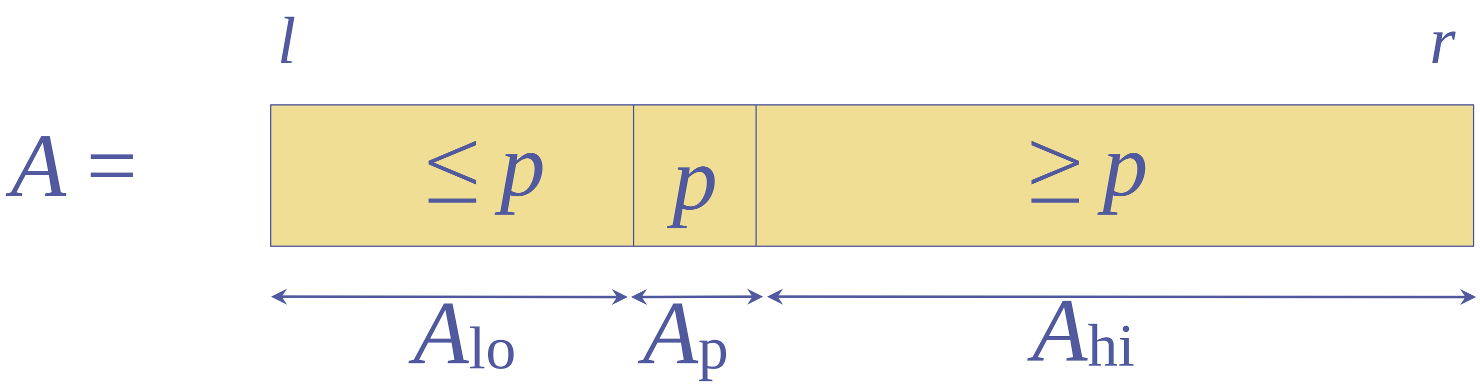
What if we were looking for 2^{nd} median?

- Look into subarray Alo [17,10,30] for 2^{nd} median

Finding Median

- Array a after partition(a, 0,9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70



|Alo| = 3
|Ap| = 1
|Ahi| = 6

What if we were looking for 2nd median?

- Look into subarray Alo [17,10,30] for 2nd median
- Problem reduced by |Ahi| + |Ap|

Median – Variable Size Decrease

- What's the connection?
- suppose that we have A[1:20] and are looking for the 7th-smallest element:
- run partition, find p = 9, say
- Where do we look for the 7th-smallest element?
  - A: A[1..20]
  - B: A[1..8]
  - C: A[1..9]
  - D: A[10..20]

Median – Variable Size Decrease

- What's the connection?
- suppose that we have A[1:20] and are looking for the 7th-smallest element:
- run partition, find p = 3, say
- Where do we look for the 7th-smallest element?
  - A: A[1..3]
  - B: A[1..4]
  - C: A[3..20]
  - D: A[4..20]

What about efficiency?

- Dasgupta’s analysis shows that:
  - if we can do the partition in O(n) time,
    then we can select the kth element in O(n) time

- How can we do partition in O(n) time?
  - Hoare Partition
  - Lomuto Partition

Back to Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- What does partition(a, 2, 6) do in this case?

Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- What does partition(a, 2, 6) do in this case?
- a[6] = 45. 45 is the <u>**pivot**</u>.
  – Array a[2, ..., 6] has:
  – 2 elements less than the pivot.
  – 2 elements greater than the pivot.
- Array a[2, 6] is rearranged so that:
  – First we put all values less than the pivot (45).
  – Then, we put the pivot.
  – Then, we put all values greater than the pivot.
- partition(a, 2, 6) returns the new index of the pivot, which is 4.

Partitioning

- Example: suppose we have this array a[]:

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- How does array a[] look after we call partition(a, 2, 6)?

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	40	30	45	70	60	80	10	35

- Note that:
  - Items at positions 2,3 are not necessarily in sorted order.
  - However, items at positions 2, 3 are all <= 45.
  - Similarly: items at positions 5, 6 are not necessarily in sorted order.
  - However, items at positions 5, 6 are all >= 45.
  - Items at positions 0, 1 and at positions 7, 8, 9, are not affected.

Partitioning Code

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = -1
- j = 9

```
int partition(Item a[], int l, int r)
{
 int i = l-1, j = r;
 Item v = a[r];
 for (;;)
 {
  while (less(a[++i], v)) ;
  while (less(v, a[--j])) if (j == l) break;
  if (i >= j) break;
  exch(a[i], a[j]);
 }
 exch(a[i], a[r]);
 return i;
}
```

How Partitioning Works

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

- partition(a, 0, 9):
- v = a[9] = 35
- ~~i = -1~~
- j = 9

- i = 0
- a[i] = 17 < 35
- i = 1;
- a[i] = 90. 90 is not < 35, break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	i=1	2	3	4	5	6	7	8	9
value	17	90	70	30	60	40	45	80	10	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 1
- ~~j = 9~~

- j = 8
- a[j] = 10. 35 is not < 10, break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	i=1	2	3	4	5	6	7	j=8	9
value	17	10	70	30	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 1
- j = 8

- i is not >= j, we don't break.
- swap values of a[i] and a[j].
- a[i] becomes 10.
- a[j] becomes 90.

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	**0**	**1**	**i=2**	**3**	**4**	**5**	**6**	**7**	**j=8**	**9**
value	17	10	70	30	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- ~~i = 1~~
- j = 8

- i = 2
- a[i] = 70. 70 is not < 35, break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	1	i=2	3	4	5	j=6	7	8	9
value	17	10	70	30	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 2
- ~~j = 8~~

- j = 7
- a[j] = 80.
- j = 6
- a[j] = 45

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	**0**	**1**	**i=2**	**j=3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	70	30	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 2
- ~~j = 8~~

- j = 5, a[j] = 40
- j = 4, a[j] = 60
- j = 3, a[j] = 30. 30 < 35, break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	1	i=2	j=3	4	5	6	7	8	9
value	17	10	30	70	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 2
- j = 3

- i is not >= j, we don't break.
- swap values of a[i] and a[j].
- a[i] becomes 30.
- a[j] becomes 70.

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	**0**	**1**	**2**	**i=j=3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	70	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- ~~i = 2~~
- j = 3
- i = 3
- a[i] = 70 > 35, break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	1	j=2	i=3	4	5	6	7	8	9
value	17	10	30	70	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 3
- ~~j = 3~~

- j = 2
- a[j] = 30 > 35, break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	1	j=2	i=3	4	5	6	7	8	9
value	17	10	30	70	60	40	45	80	90	35

- partition(a, 0, 9):
- v = a[9] = 35
- i = 3
- j = 2

- i >= j, we break!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

How Partitioning Works

position	0	1	j=2	i=3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70

- partition(a, 0, 9):
- v = a[9] = 35
- i = 3
- j = 2

- a[i] becomes 35
- a[r] becomes 70
- we return i, which is 3.
- DONE!!!

```
int partition(Item a[], int l, int r)
{
  int i = l-1, j = r;
  Item v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

Quicksort

```
void quicksort(Item a[], int length)
{
  quicksort_aux(a, 0, length-1);
}

void quicksort_aux(Item a[], int l, int r)
{
  int i;
  if (r <= l) return;
  i = partition(a, l, r);
  quicksort_aux(a, l, i-1);
  quicksort_aux(a, i+1, r);
}
```

To sort array a, quicksort works as follows:

- Do an initial partition of a, that returns some position i.
- Recursively do quicksort on:
  - a[0], ..., a[i-1]
  - a[i+1], ..., a[length-1]
- What are the base cases?

Quicksort

```
void quicksort(Item a[], int length)
{
  quicksort_aux(a, 0, length-1);
}

void quicksort_aux(Item a[], int l, int r)
{
  int i;
  if (r <= l) return;
  i = partition(a, l, r);
  quicksort_aux(a, l, i-1);
  quicksort_aux(a, i+1, r);
}
```

To sort array a, quicksort works as follows:

- Do an initial partition of a, that returns some position i.
- Recursively do quicksort on:
  - a[0], ..., a[i-1]
  - a[i+1], ..., a[length-1]
- What are the base cases?
  - Array length 1 (r == l).
  - Array length 0 (r < l).

How Quicksort Works

Before partition(a, 0, 9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	90	70	30	60	40	45	80	10	35

quicksort_aux(a, 0, 9);

    3 = partition(a, 0, 9);

How Quicksort Works

After partition(a, 0, 9)

position	0	1	2	3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70

quicksort_aux(a, 0, 9);

3 = partition(a, 0, 9);

How Quicksort Works

After partition(a, 0, 9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
```

How Quicksort Works

Before partition(a, 0, 2)

position	0	1	2	3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70

quicksort_aux(a, 0, 9);

    3 = partition(a, 0, 9);

    quicksort_aux(a, 0, 2);

        2 = partition(a, 0, 2)

    quicksort_aux(a, 4, 9);

How Quicksort Works

After partition(a, 0, 2) (no change)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
        2 = partition(a, 0, 2)
    quicksort_aux(a, 4, 9);
```

How Quicksort Works

After partition(a, 0, 2) (no change)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	17	10	30	35	60	40	45	80	90	70

quicksort_aux(a, 0, 9);

    3 = partition(a, 0, 9);

    quicksort_aux(a, 0, 2);

        2 = partition(a, 0, 2)

        quicksort_aux(a, 0, 1);

        quicksort_aux(a, 3, 2);

    quicksort_aux(a, 4, 9);

How Quicksort Works

Before partition(a, 0, 1)

position	0	1	2	3	4	5	6	7	8	9
value	17	10	30	35	60	40	45	80	90	70

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
        2 = partition(a, 0, 2)
        quicksort_aux(a, 0, 1);
            0 = partition(a, 0, 1);
        quicksort_aux(a, 3, 2);
   quicksort_aux(a, 4, 9);
```

How Quicksort Works

After partition(a, 0, 1)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	60	40	45	80	90	70

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
        2 = partition(a, 0, 2)
        quicksort_aux(a, 0, 1);
            0 = partition(a, 0, 1);
        quicksort_aux(a, 3, 2);
   quicksort_aux(a, 4, 9);
```

How Quicksort Works

After partition(a, 0, 1)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	60	40	45	80	90	70

quicksort_aux(a, 0, 9);

3 = partition(a, 0, 9);

quicksort_aux(a, 0, 2);

2 = partition(a, 0, 2)

quicksort_aux(a, 0, 1);

0 = partition(a, 0, 1);

quicksort_aux(a, 0, -1); quicksort_aux(a, 1, 1);

Base cases.

Nothing to do.

quicksort_aux(a, 3, 2);

quicksort_aux(a, 4, 9);

How Quicksort Works

After partition(a, 0, 1)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	60	40	45	80	90	70

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
        2 = partition(a, 0, 2)
        quicksort_aux(a, 0, 1);
            0 = partition(a, 0, 1);
        quicksort_aux(a, 3, 2);
    quicksort_aux(a, 4, 9);
```

Base case.
Nothing to do.

How Quicksort Works

Before partition(a, 4, 9)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	60	40	45	80	90	70

quicksort_aux(a, 0, 9);

    3 = partition(a, 0, 9);

    quicksort_aux(a, 0, 2);

    quicksort_aux(a, 4, 9);

        7 = partition(a, 4, 9);

How Quicksort Works

After partition(a, 4, 9)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	60	40	45	70	90	80

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
        7 = partition(a, 4, 9);
```

How Quicksort Works

After partition(a, 4, 9)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	60	40	45	70	90	80

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
        7 = partition(a, 4, 9);
        quicksort_aux(a, 4, 6);
        quicksort_aux(a, 8, 9);
```

How Quicksort Works

Before partition(a, 4, 6)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	60	40	45	70	90	80

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
        7 = partition(a, 4, 9);
        quicksort_aux(a, 4, 6);
            5 = partition(a, 4, 6);
        quicksort_aux(a, 8, 9);
```

How Quicksort Works

After partition(a, 4, 6)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	40	45	60	70	90	80

```
quicksort_aux(a, 0, 9);
  3 = partition(a, 0, 9);
  quicksort_aux(a, 0, 2);
  quicksort_aux(a, 4, 9);
    7 = partition(a, 4, 9);
    quicksort_aux(a, 4, 6);
      5 = partition(a, 4, 6);
    quicksort_aux(a, 8, 9);
```

How Quicksort Works

After partition(a, 4, 6)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	40	45	60	70	90	80

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
        7 = partition(a, 4, 9);
        quicksort_aux(a, 4, 6);
            5 = partition(a, 4, 6);
            quicksort_aux(a, 4, 4);  quicksort_aux(a, 6, 6);
        quicksort_aux(a, 8, 9);
```

Base cases.
Nothing to do.

How Quicksort Works

Before partition(a, 8, 9)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	40	45	60	70	90	80

```
quicksort_aux(a, 0, 9);
  3 = partition(a, 0, 9);
  quicksort_aux(a, 0, 2);
  quicksort_aux(a, 4, 9);
    7 = partition(a, 4, 9);
    quicksort_aux(a, 4, 6);
    quicksort_aux(a, 8, 9);
      8 = partition(a, 8, 9);
```

How Quicksort Works

After partition(a, 8, 9)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	40	45	60	70	80	90

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
        7 = partition(a, 4, 9);
        quicksort_aux(a, 4, 6);
        quicksort_aux(a, 8, 9);
            8 = partition(a, 8, 9);
```

How Quicksort Works

After partition(a, 8, 9)

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	40	45	60	70	80	90

```
quicksort_aux(a, 0, 9);
    3 = partition(a, 0, 9);
    quicksort_aux(a, 0, 2);
    quicksort_aux(a, 4, 9);
        7 = partition(a, 4, 9);
        quicksort_aux(a, 4, 6);
        quicksort_aux(a, 8, 9);
            8 = partition(a, 8, 9);
            quicksort_aux(a, 8, 7);  quicksort_aux(a, 9, 9);
```

Base cases.
Nothing to do.

How Quicksort Works

After partition(a, 8, 9)

position	0	1	2	3	4	5	6	7	8	9
value	10	17	30	35	40	45	60	70	80	90

quicksort_aux(a, 0, 9);

3 = partition(a, 0, 9);

quicksort_aux(a, 0, 2);

quicksort_aux(a, 4, 9);

7 = partition(a, 4, 9);

quicksort_aux(a, 4, 6);

quicksort_aux(a, 8, 9);

8 = partition(a, 8, 9);

quicksort_aux(a, 8, 7); quicksort_aux(a, 9, 9);

Done!!!

All recursive calls have returned.

The array is sorted.

Base cases.

Nothing to do.

Worst-Case Time Complexity

- The **worst-case** of quicksort is interesting:
- Quicksort has the slowest running time when the input array **is already sorted**.

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	42	50	60	70	80	90

- partition(a, 0, 9):
  - scans 10 elements, makes no changes, returns 9.
- partition(a, 0, 8):
  - scans 9 elements, makes no changes, returns 8.
- partition(a, 0, 7):
  - scans 8 elements, makes no changes, returns 7.
- Overall, **worst-case** time is $N+(N-1)+(N-2)+\ldots+1 = \Theta(N^2)$.

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.
- The **best case** time complexity for quicksort is when the array is partitioned in a **perfectly balanced** way.
- I.e., when the pivot is always the median value in the array.
- Let T(N) be the best-case running time complexity for quicksort.
- $T(N) = N + 2 * T(N/2)$
- Why? Because to sort the array:
  - We do N operations for the partition.
  - We do to recursive calls, and each call receives half the data.

Best-Case Time Complexity

- For convenience, let N = 2^n.
- Assuming that the partition always splits the set into two equal halves, we get:
- $T(2^n) = 2^n + 2 * T(2^{n-1})$

$= 1*2^n + 2^1 * T(2^{n-1})$ step 1

$= 2*2^n + 2^2 * T(2^{n-2})$ step 2

$= 3*2^n + 2^3 * T(2^{n-3})$ step 3

…

$= i*2^n + 2^i * T(2^{n-i})$ step i

…

$= n*2^n + 2^n * T(2^{n-n})$ step n

$= \lg N * N + N * T(0)$

$= \Theta(N \lg N)$.

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N \lg N)$.
- It turns out that the average time complexity is also $\Theta(N \lg N)$.
- On average, quicksort performance is close to that of the best case.
- Why? Because, usually, the pivot value is "close enough" to the 50-th percentile to achieve a reasonably balanced partition.
  - For example, half the times the pivot value should be between the 25-th percentile and the 75th percentile.

Improving Performance

- The basic implementation of quicksort that we saw, makes a partition using the rightmost element as pivot.
  - This has the risk of giving a pivot that is not that close to the 50th percentile.
  - When the data is already sorted, the pivot is the 100th percentile, which is the worst-case.

Improving Performance

- We can improve performance by using as pivot the median of three values:
  - The leftmost element.
  - The middle element.
  - The rightmost element.
- Then, the pivot has better chances of being close to the 50th percentile.
- If the file is already sorted, the pivot is the median.
- Thus, already sorted data is:
  - The worst case (slowest running time) when the pivot is the rightmost element.
  - The best case (fastest run time) when the pivot is the median of the leftmost, middle, and rightmost elements.