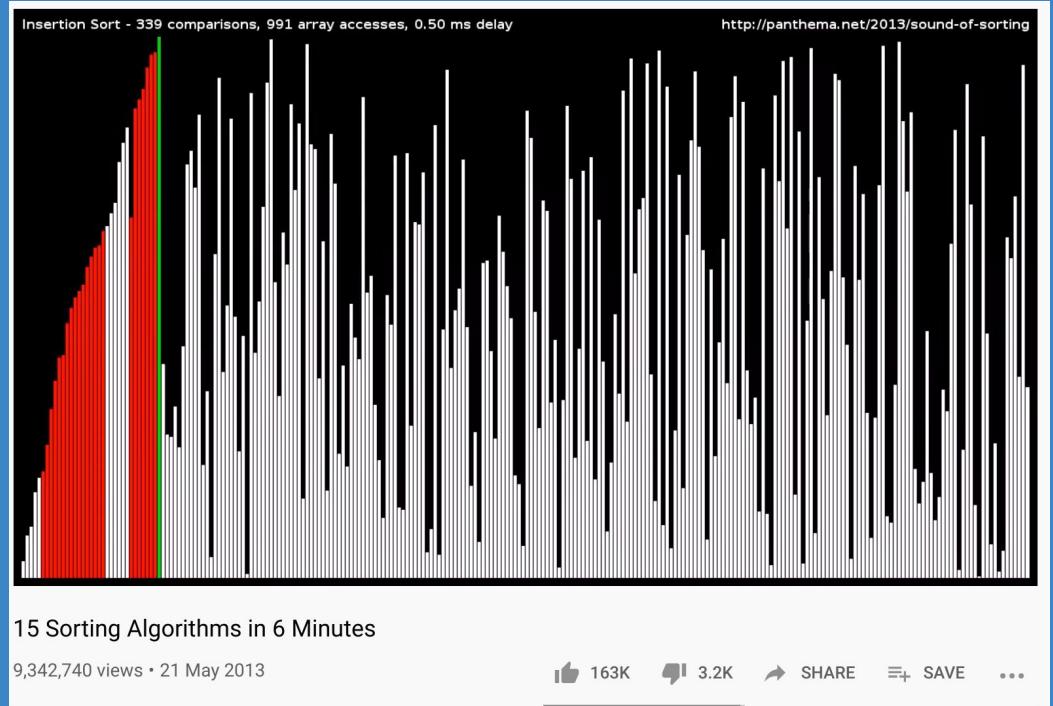


Sorting

... in ascending order ;)



<https://www.youtube.com/watch?v=kPRA0W1kECg>

Insertion Sort

Space Complexity:

2 auxiliary variables (i, j) $\Rightarrow O(1)$ ($O(n)$ total, store the array)

So for small arrays,

Or nearly sorted arrays

Insertion sort is pretty good!

Time Complexity:

Best case: Fully sorted! 1 2 3 4 5 6 7 8 9
 $\Rightarrow O(n)$ comparisons, $O(1)$ swaps.

2nd Best case: Nearly sorted 1 2 3 4 5 6 7 9 8
 \Rightarrow approx $O(n)$ comparisons, $O(1)$ swaps (non-rigorous)

Worst case: Reversed array 9 8 7 6 5 4 3 2 1
 $\Rightarrow O(n^2)$ comparisons, $O(n^2)$ swaps.

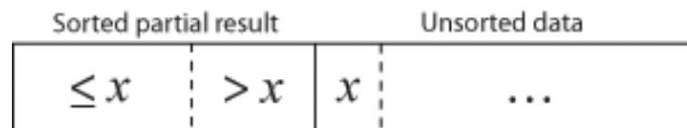
Average case: $O(n^2)$ comparisons, $O(n^2)$ swaps

Insertion Sort

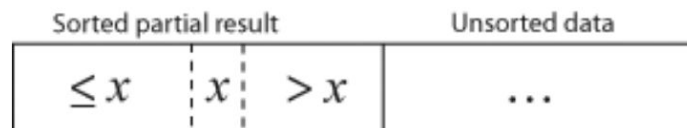
- Builds sorted array one element at a time.

Steps

1. Pick the first unsorted element, x
2. Shift it towards the left, until it is "in place"
 - "In place": element left $\leq x$
3. Repeat until there are no more unsorted elements



becomes



Insertion Sort - Algorithm

```
void insertion_sort(int A[], int n) {  
    for (int i = 1; i < n; i++) {      // i : first 'not sorted' index  
        for (int j = i; j > 0; j--) {  // j : element being sorted  
            if (A[j - 1] <= A[j]) {    // A[j] is in place?  
                break;                // Yep, onto next element.  
            }  
            int_swap(A + j - 1, A + j); // No, move A[j] one left:  
                                         // swap A[j] and A[j-1].  
        }  
    }  
}
```

Modified from <https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/insertionsort.c>

Insertion Sort - Algorithm

```
void insertion_sort(int A[], int n) {  
    for (int i = 1; i < n; i++) {  
        for (int j = i; j > 0 && A[j - 1] > A[j]; j--) {  
            int_swap(A + j - 1, A + j);  
        }  
    }  
}
```

Insertion Sort

Pros:

- Rather simple to implement
- Easy to interpret - it's the same way one sorts a hand of cards.
- Low memory usage
- Fast insertion for one element
- Very efficient for small datasets.
 - More efficient than mergesort / quicksort there

Cons:

- Slow for larger arrays $O(n^2)$

Selection Sort

- Builds sorted array one element at a time. .. but less efficiently.

Steps

1. Find the maximum value that is unsorted, x
2. Swap x with the first value of the unsorted region.
3. Repeat until there are no more unsorted elements

5 3 4 1 2

Selection Sort

Selection Sort

*Find the max in the (unsorted) array, swap it to the end. Repeat.
(Or, find min and swap to the start.)*

```
void selection_sort(int A[], int n) {  
    for (int i = n; i > 1; i--)  
        int_swap(&A[find_max_index(A, i)], &A[i - 1]);  
}  
int find_max_index(int A[], int n) { // 'argmax'  
    int max_i = 0;  
    for (int i = 1; i < n; i++)  
        if (A[i] > A[max_i]) max_i = i;  
    return max_i;  
}
```


Selection Sort

Space Complexity:

3 auxilliary variables (i, j) => $O(1)$ ($O(n)$ total, store the array)

Time Complexity:

All cases: $O(n^2)$ comparisons, $O(n)$ swaps.

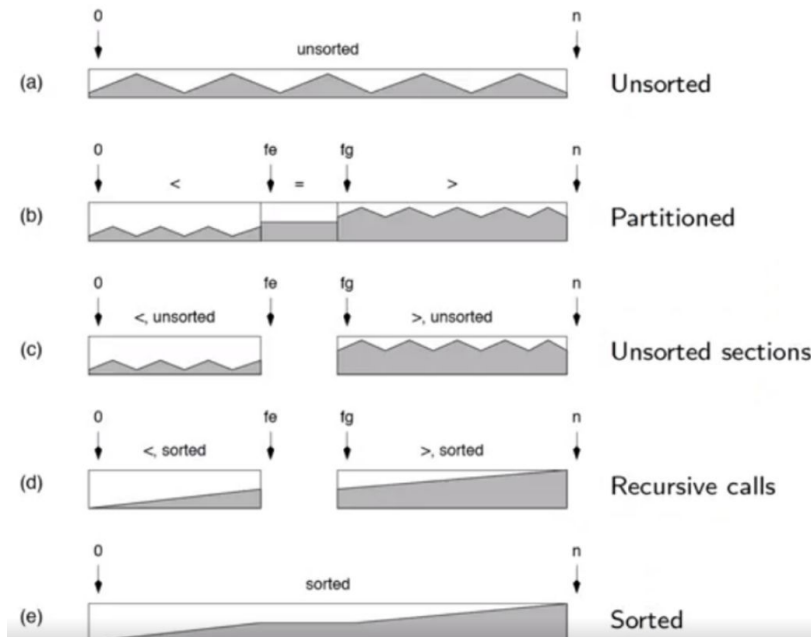
At least it's consistent.

Pros:

- Simple
- Sometimes has performance advantages (low auxilliary memory usage)
- Fewer swaps than insertion sort, minimal memory utilisation. (writes slower than reads?)

Quicksort (Hoare, 1959)

Recursively partition array so that we have 'smaller' items on the left of our 'pivot' and 'larger' items on the right of our pivot.



Quicksort Implementation

```
void quicksort(int *A, int n){
    if (n <= 1) return; // base case -- sorted
    int fe, fg;          // index: first equal=, first greater>
    int pivot = choose_pivot(A); // pick a pivot (value)
    partition(A, n, &fe, &fg, pivot); // partition the array. (b)
    quicksort(A, fe);      // length of 1st partition: fe
    quicksort(A + fg, n - fg); // length of 2nd partition: (n - fg)
    // equals are in place -- we're done!
}
```

Quicksort

```
void quicksort(int *A, int n){
    if (n <= 1) return; // base case -- sorted
    int fe, fg; // "first equal =", "first greater >"
    int pivot = choose_pivot(A); // pick a pivot (value of the pivot, not index)
    partition(A, n, &fe, &fg, pivot); // partition the array. O(n)
    quicksort(A, fe); // length of first partition is index of fe
    quicksort(A + fg, n - fg); // length of second partition is size n - index of fg
    // equals are in place -- we're done!
}
```

9	6	5	3	2	8	7	2	5
---	---	---	---	---	---	---	---	---

<			=		>			
2	3	2	5	5	7	8	6	9
			^fe		^fg			

Now run quicksort on both partitions.

9	6	5	3	2	8	7	2	5
---	---	---	---	---	---	---	---	---

Partition:

2	3	2	5	5	7	8	6	9
---	---	---	---	---	---	---	---	---

\wedge_{fe}

\wedge_{fg}

Note: the resulting order within each partition (red / orange) doesn't matter - it depends on the partition scheme used. Here I'm using Hoare from the lectures, so I get this order.

2	2	3
---	---	---

<empty>

2	2
---	---

<empty>

<empty>

6	7	9	8
---	---	---	---

\wedge_{fe}

<empty>

<empty>

\wedge_{fg}

8	9
---	---

<empty>

9

<empty>

<empty>

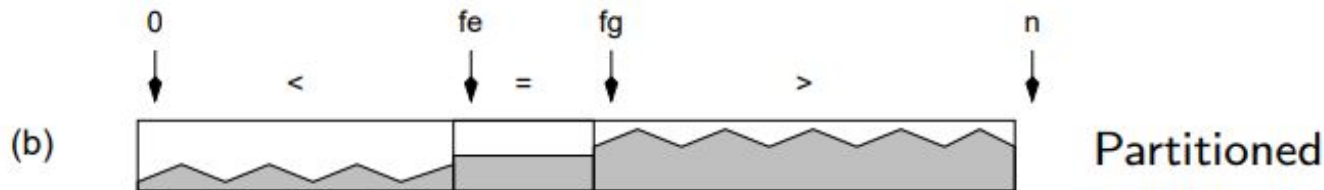
Result:

2	2	3	5	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Quicksort (Hoare, 1959)

Recursively partition array so that we have 'smaller' items on the left of our 'pivot' and 'larger' items on the right of our pivot.

0. **Base case:** if the array has 0 or 1 elements, it is sorted.
1. Pick a pivot. (to partition the array with)
2. Partition the array into <smaller>, <equal> and <greater> partitions.
3. Recurse: quicksort the <smaller> partition, and quicksort the <greater> partition.



Quicksort Correctness

Quicksort:

```
if  $n \leq 1$ 
    return
else
    p: pivot
     $p \leftarrow$  any element in  $A[0 \dots n - 1]$ 
    assert:  $n > 1$  and  $p \in A[0 \dots n - 1]$ 
     $(fe, fg) \leftarrow \text{partition}(A, n, p)$ 
    assert:  $R$ 
    quicksort( $A[0 \dots fe - 1]$ )
    quicksort( $A[fg \dots n - 1]$ )
    assert:  $A[0 \dots fe - 1]$  is sorted and  $A[fg \dots n - 1]$  is sorted
           and  $R \implies A[0 \dots n - 1]$  is sorted
```

Partition:

Initialization:

$next, fe, fg \leftarrow 0, 0, n$

assert: P

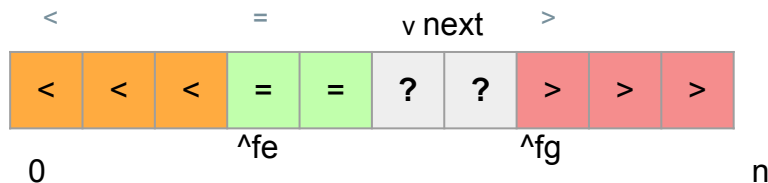
Loop:

```
while  $next < fg$ 
    assert:  $P$  and  $next < fg$ 
    if  $A[next] < p$ 
        swap  $A[fe]$  and  $A[next]$ 
         $fe, next \leftarrow fe + 1, next + 1$ 
        assert:  $P$ 
    else if  $A[next] > p$ 
        swap  $A[next]$  and  $A[fg - 1]$ 
         $fg \leftarrow fg - 1$ 
        assert:  $P$ 
    else
         $next \leftarrow next + 1$ 
        assert:  $P$  and  $fe < next$ 
```

Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```



Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

v next

9	6	5	3	2	8	7	2	5
---	---	---	---	---	---	---	---	---

^fe

^fg

Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

9 > 5

v next

9	6	5	3	2	8	7	2	5
---	---	---	---	---	---	---	---	---

^fe

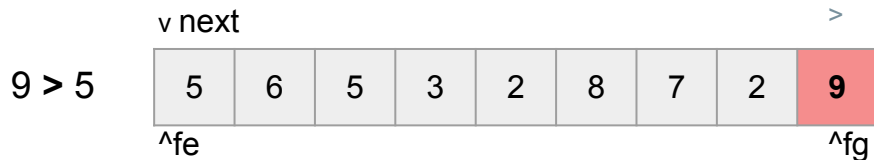
^fg

Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5



Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++;    next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

5 = 5

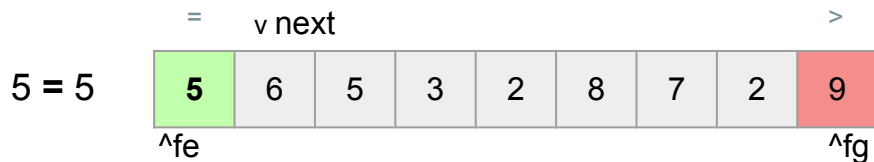
v next								>	
5	6	5	3	2	8	7	2	9	
^fe								^fg	

Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++;    next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

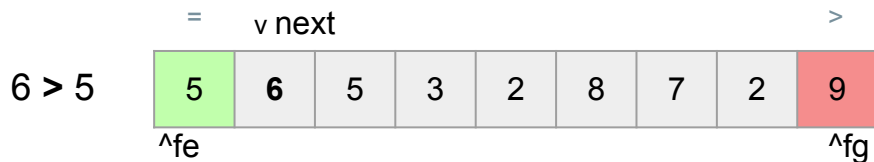


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

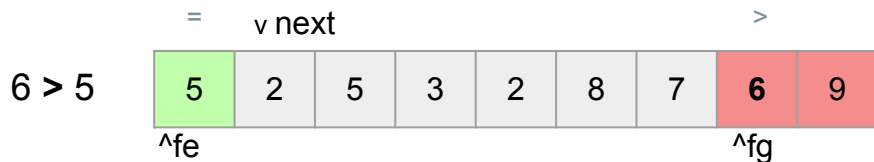


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

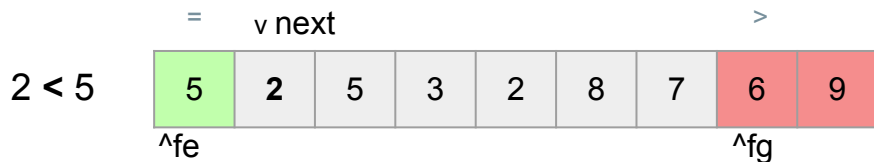


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

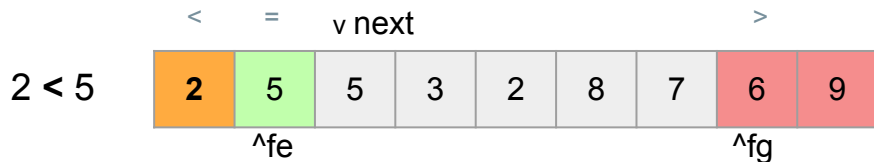


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: **5**

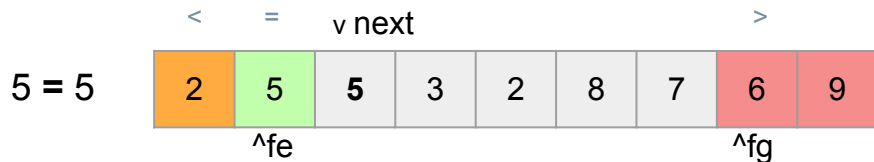


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

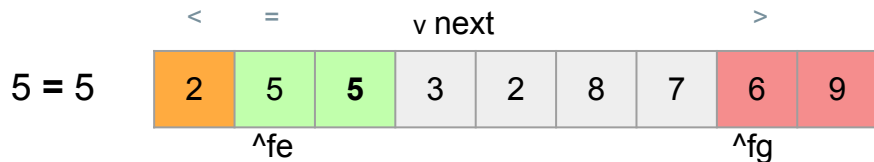


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

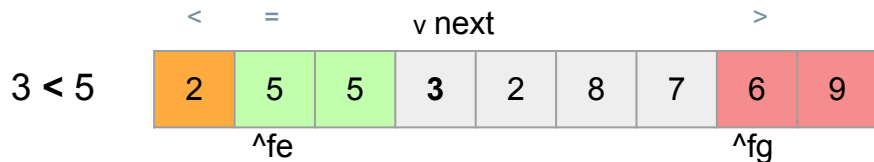
Pivot: 5



Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {  
    int next = 0; // next item to partition  
    *fe = 0; // first equals, will go ->  
    *fg = n; // first greater, will go <-  
  
    while (next < *fg) {  
        if (A[next] < pivot) {  
            int_swap(A + *fe, A + next);  
            (*fe)++;    next++;  
        }  
        else if (A[next] > pivot) {  
            (*fg)--;  
            int_swap(A + *fg, A + next);  
        }  
        else { // if (A[next] == pivot)  
            next++;  
        }  
    }  
}
```

Pivot: 5

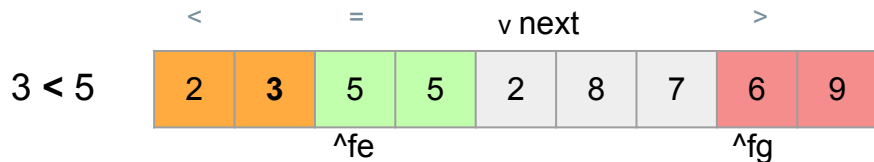


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

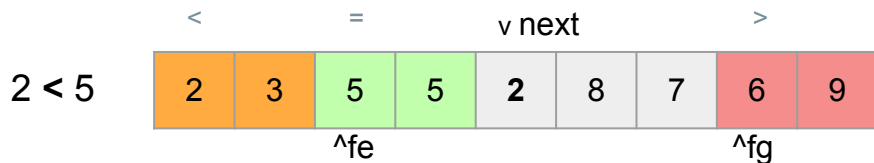
Pivot: 5



Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {  
    int next = 0; // next item to partition  
    *fe = 0; // first equals, will go ->  
    *fg = n; // first greater, will go <-  
  
    while (next < *fg) {  
        if (A[next] < pivot) {  
            int_swap(A + *fe, A + next);  
            (*fe)++;    next++;  
        }  
        else if (A[next] > pivot) {  
            (*fg)--;  
            int_swap(A + *fg, A + next);  
        }  
        else { // if (A[next] == pivot)  
            next++;  
        }  
    }  
}
```

Pivot: 5

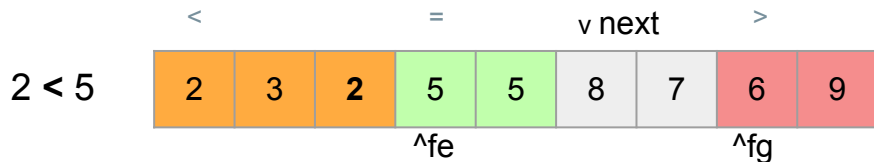


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

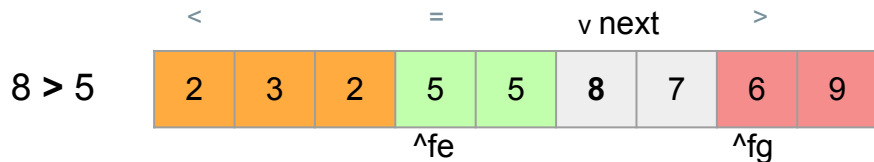


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

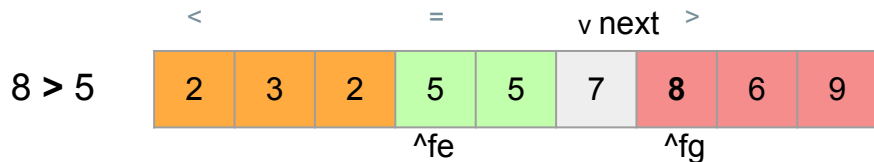


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

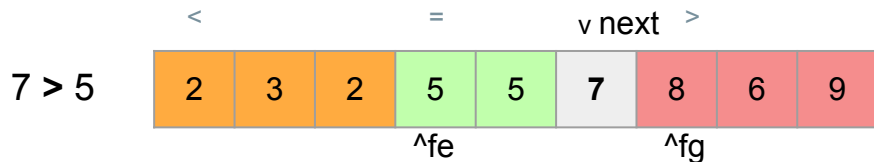


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

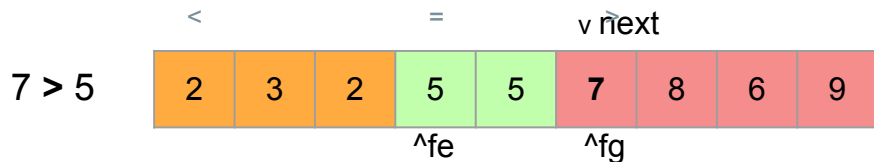


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {
    int next = 0; // next item to partition
    *fe = 0; // first equals, will go ->
    *fg = n; // first greater, will go <-

    while (next < *fg) {
        if (A[next] < pivot) {
            int_swap(A + *fe, A + next);
            (*fe)++; next++;
        }
        else if (A[next] > pivot) {
            (*fg)--;
            int_swap(A + *fg, A + next);
        }
        else { // if (A[next] == pivot)
            next++;
        }
    }
}
```

Pivot: 5

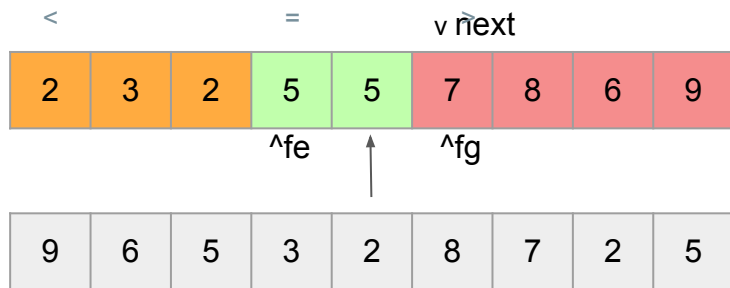


Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {  
    int next = 0; // next item to partition  
    *fe = 0; // first equals, will go ->  
    *fg = n; // first greater, will go <-
```

```
    while (next < *fg) {  
        if (A[next] < pivot) {  
            int_swap(A + *fe, A + next);  
            (*fe)++;    next++;  
        }  
        else if (A[next] > pivot) {  
            (*fg)--;  
            int_swap(A + *fg, A + next);  
        }  
        else { // if (A[next] == pivot)  
            next++;  
        }  
    }  
}
```

Pivot: 5



Finished!

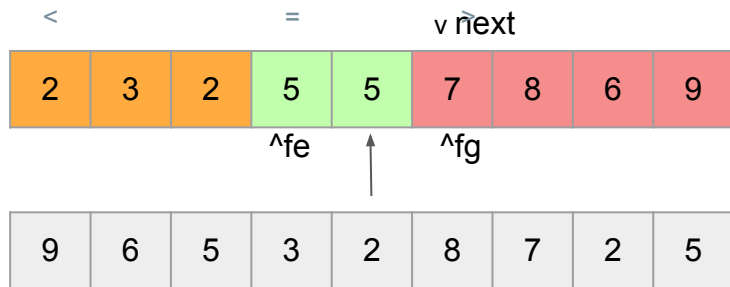
How many operations was that?

Hoare Partitioning Implementation

```
void partition(int *A, int n, int* fe, int* fg, int pivot) {  
    int next = 0; // next item to partition  
    *fe = 0; // first equals, will go ->  
    *fg = n; // first greater, will go <-
```

```
    while (next < *fg) {  
        if (A[next] < pivot) {  
            int_swap(A + *fe, A + next);  
            (*fe)++; next++;  
        }  
        else if (A[next] > pivot) {  
            (*fg)--;  
            int_swap(A + *fg, A + next);  
        }  
        else { // if (A[next] == pivot)  
            next++;  
        }  
    }  
}
```

Pivot: 5



Finished!
How many operations was that?

$O(n)$

Quicksort Implementation

```
void quicksort(int *A, int n){  
    if (n <= 1) return; // base case -- sorted  
    int fe, fg; // "first equal =", "first greater >"  
    int pivot = choose_pivot(A); // pick a pivot (value of the pivot, not index)  
    partition(A, n, &fe, &fg, pivot); // partition the array. (b)  
    quicksort(A, fe); // length of first partition is index of fe  
    quicksort(A + fg, n - fg); // length of second partition is size n - index of fg  
    // equals are in place -- we're done!  
}
```

How does choosing a pivot change the complexity of quicksort?
What is the best choice of pivot?

Quicksort Performance

... depends on our choice of pivot!

eg: 2 4 6 8 7 5 3 1

Best case quicksort

sort => (< 2 3 1) (= 4) (> 7 8 6)

Partitioning is n operations

sort left => (< 1) (= 2) (> 3)

$n/2$ ops

sort left => (<) (= 1) (>) (done)

$n/4$ ops

sort right => (<) (= 3) (>) (done)

$n/4$ ops

sort right => (< 6) (= 7) (> 8)

$n/2$ ops

sort left => (<) (= 6) (>) (done)

$n/4$ ops

sort right => (<) (= 8) (>) (done)

$n/4$ ops

Total: $n + 2*n/2 + 4*n/4 = 3n = \mathbf{O(n \log n)}$
($\log_2(8) = 3$)

Quicksort Performance

... depends on our choice of pivot!

eg: 2 4 6 8 7 5 3 1

Worst case quicksort...? Just wait.

Choosing a pivot

1. First element

1 2 3 4 5 6 7 8

^ pivot

Any issues?

Choosing a pivot

1. First element

1 2 3 4 5 6 7 8

^ pivot

Any issues? **Yep.**

If the array is already sorted, we will get worst case time complexity.

Partitioning:

(recall: Hoare partitioning reverses > elements)

(<)	(=) 1	(>) <u>8</u> 7 6 5 4 3 2
(<) <u>7</u> 6 5 4 3 2	(=) 8	(>)
(<) <u>6</u> 5 4 3 2	(=) 7	(>)
(<) <u>5</u> 4 3 2	(=) 6	(>)
(<) <u>4</u> 3 2	(=) 5	(>)
(<) <u>3</u> 2	(=) 4	(>)
(<) <u>2</u>	(=) 3	(>)
(<)	(=) 2	(>)

Choosing a pivot

1. First element

1 2 3 4 5 6 7 8

^ pivot

Any issues? **Yep.**

If the array is already sorted, we will get worst case time complexity.

Partitioning:

(<) 1	(=) 1	(>) <u>8</u> 7 6 5 4 3 2
(<) <u>7</u> 6 5 4 3 2	(=) 8	(>)
(<) <u>6</u> 5 4 3 2	(=) 7	(>)
(<) <u>5</u> 4 3 2	(=) 6	(>)
(<) <u>4</u> 3 2	(=) 5	(>)
(<) <u>3</u> 2	(=) 4	(>)
(<) <u>2</u>	(=) 3	(>)
(<)	(=) 2	(>)

(recall: Hoare partitioning reverses > elements)

n partitions,
partitioning is $O(n)$

Total time complexity:
 $n * O(n) = \mathbf{O(n^2)}$

Quicksort Complexity – Pivot choice

Pivot: **choose first item**

	1	2	3	4	5	6	7
$O(n)$	1	2	3	4	5	6	7
$O(n)$	1	2	3	4	5	6	7
$O(n)$	1	2	3	4	5	6	7
$O(n)$	1	2	3	4	5	6	7
...							
$O(n)$	1	2	3	4	5	6	7

$n * O(n) \Rightarrow O(n^2)$, worst case

This is $O(n)$ under insertion sort!!

Choosing a pivot

2. Last element

Any issues?

Choosing a pivot

2. Last element

Any issues?

8 7 6 5 4 3 2 1
 ^ pivot

If the array is in reverse order, we will get worst case time complexity.

Partitioning:

(<) 1	(=) 1	(>) 2 3 4 5 6 7 <u>8</u>	} n partitions, partitioning is $O(n)$ $n * O(n) = O(n^2)$
(<) 2 3 4 5 6 <u>7</u>	(=) 8	(>)	
(<) 2 3 4 5 <u>6</u>	(=) 7	(>)	
(<) 2 3 4 <u>5</u>	(=) 6	(>)	
(<) 2 3 <u>4</u>	(=) 5	(>)	
(<) 2 3	(=) 4	(>)	
(<) 2	(=) 3	(>)	
(<)	(=) 2	(>)	

Choosing a pivot

3. Middle element

eg: 5 7 2 9 3 2 5 7

Any issues?

Choosing a pivot

3. Middle element

eg: 5 7 2 **9** 3 2 5 7

Any issues? Not with that one, but consider: **2 4 6 8 7 5 3 1**

Partitioning:

(<)	2 4 6 <u>7</u> 5 3 1	(=)	8	(>)
(<)	2 4 <u>6</u> 5 3 1	(=)	7	(>)
(<)	2 4 <u>5</u> 3 1	(=)	6	(>)
(<)	2 <u>4</u> 3 1	(=)	5	(>)
(<)	2 <u>3</u> 1	(=)	4	(>)
(<)	<u>2</u> 1	(=)	3	(>)
(<)	<u>1</u>	(=)	2	(>)
(<)		(=)	1	(>)

n partitions,
partitioning is $O(n)$

$$n * O(n) = O(n^2)$$

Choosing a pivot

Seems like wherever we pick a pivot, we're always getting $O(n^2)$ some way.

Can we do better?

Choosing a pivot

Seems like wherever we pick a pivot, we're always getting $O(n^2)$ some way.

Can we do better?

4. Random pivot

- On each partition, choose a random index in the array as the pivot.

eg: 2 4 6 8 7 5 3 1

- There's a $2/n$ chance of choosing the **worst** case. (1 or 8)
- There's a $2/n$ chance of choosing the **best** case. (4 or 5)
- And we can have everything in between.

Turns out, this ensures $O(n \log n)$ as long as there isn't some prior distribution of the array.

Choosing a pivot

```
int choose_pivot(int *A, int n) {  
    return A[rand() % n];    // Random pivot (best option, why?)  
    // return A[0];          // first item  
    // return A[n-1];        // last item  
    // return A[n/2];        // middle item  
}
```

Quicksort Complexity

Space Complexity:

$O(n)$ - 2 partition; $O(\log n)$ - 3 partitions.

Time Complexity - depends on how the array is partitioned.

Best case: Pivot always splits the array in half.
 $\Rightarrow O(n \log n)$

9 8 7 6 5 4 3 2 1 \rightarrow choose 5.

Worst case: Pick pivot that is largest / smallest
 $\Rightarrow O(n^2)$

9 8 7 6 5 4 3 2 1 \rightarrow choose 1 or 9.

Average case: Pivot could be anywhere in between. Proof in DoA / ADS
 $\Rightarrow O(n \log n)$

Partition is **$O(n)$** . We partition $O(\log n)$ times if our pivot choice is good $\Rightarrow O(n \log n)$.

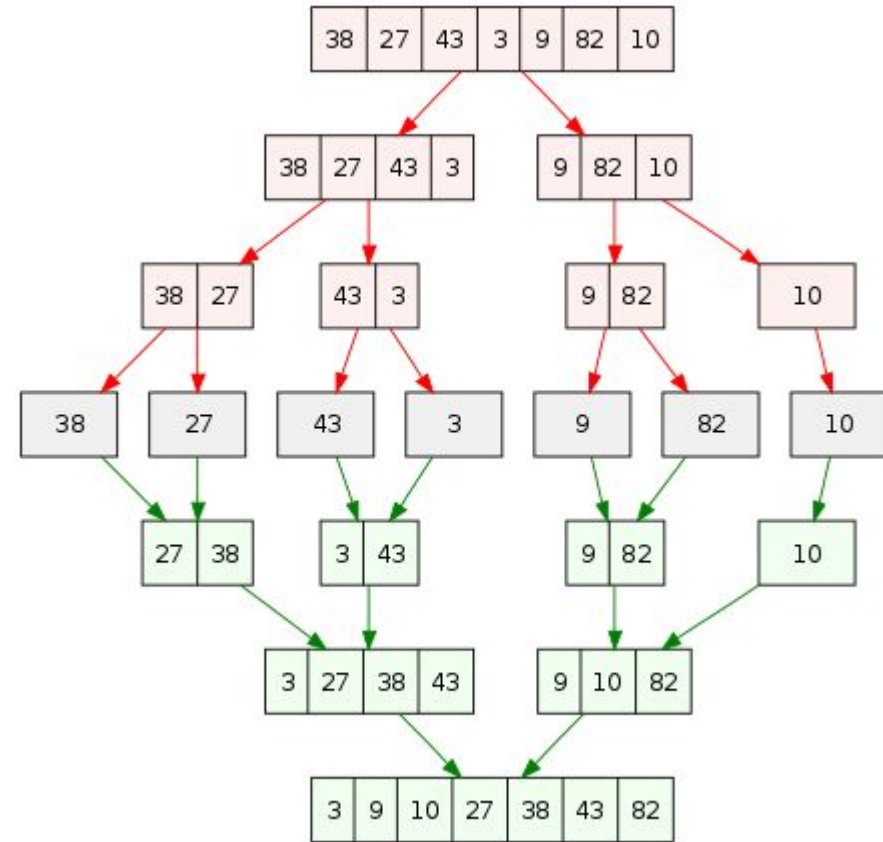
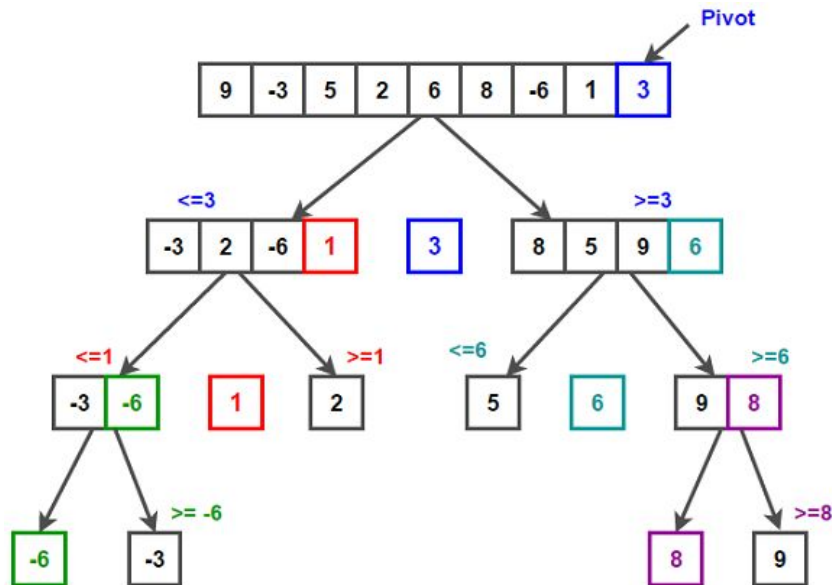
Comparing sorting algorithms

	Insertion sort	Quicksort
Idea	Builds sorted array sequentially, swapping elements towards the end	Recursively partition array into two parts, to sort independently (divide and conquer)
Notes	<i>Easy to implement / understand. Efficient for nearly sorted arrays, and small arrays.</i>	Complexity depends on pivot choice. Choose random pivot for best results. Hoare Partitioning is $O(n)$
Time Complexity	$O(n)$ best (already sorted!) $O(n^2)$ average and worst	$O(n^2)$ worst (picking worst pivot) $O(n \log n)$ average and best.
Space Complexity	$O(1)$ (auxiliary)	$O(n)$ (2 partition) $O(\log n)$ (3 partition)

Divide and Conquer

- Break problem into smaller instances (**divide**)
- solve the instances (recursively?) (**conquer**)
- **combine solutions** to create a solution to the original problem

mergesort, quicksort, subset sums, hanoi, binary search



Mergesort

Mergesort

$O(n \log n)$ worst / best case

but $O(n)$ space.

