Quicksort

❖ Quicksort

Previous sorts were all $O(n^k)$ (where $k>1$).

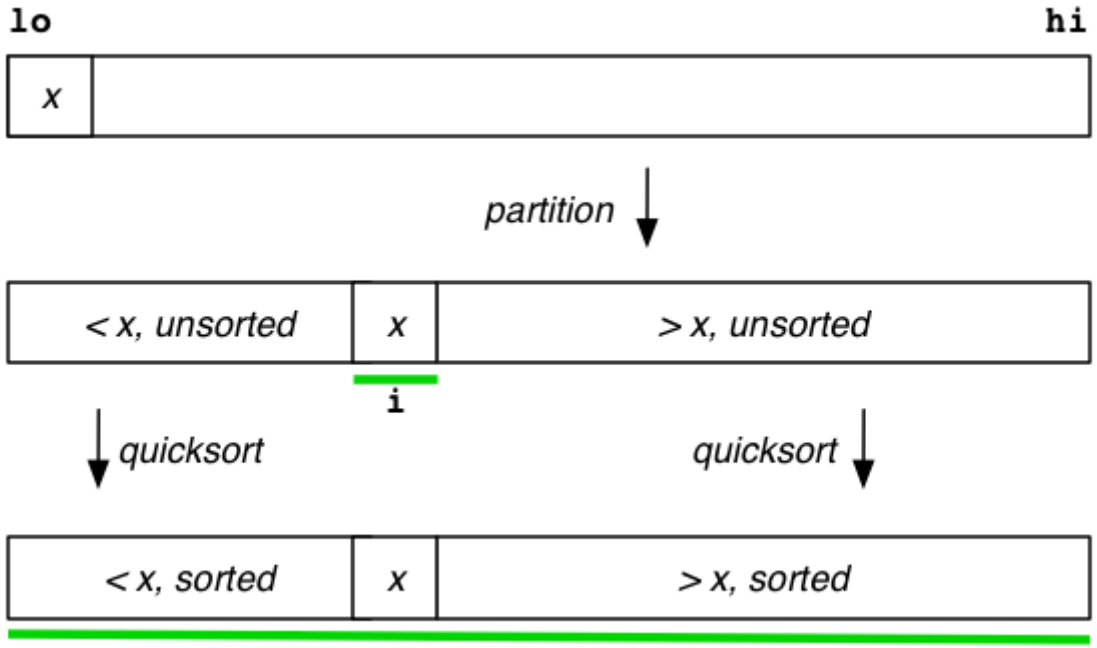
We can do better ...

Quicksort: basic idea

- choose an item to be a "pivot"
- re-arrange (partition) the array so that
  - all elements to left of pivot are smaller than pivot
  - all elements to right of pivot are greater than pivot
- (recursively) sort each of the partitions

❖ ... Quicksort

Phases of quicksort:



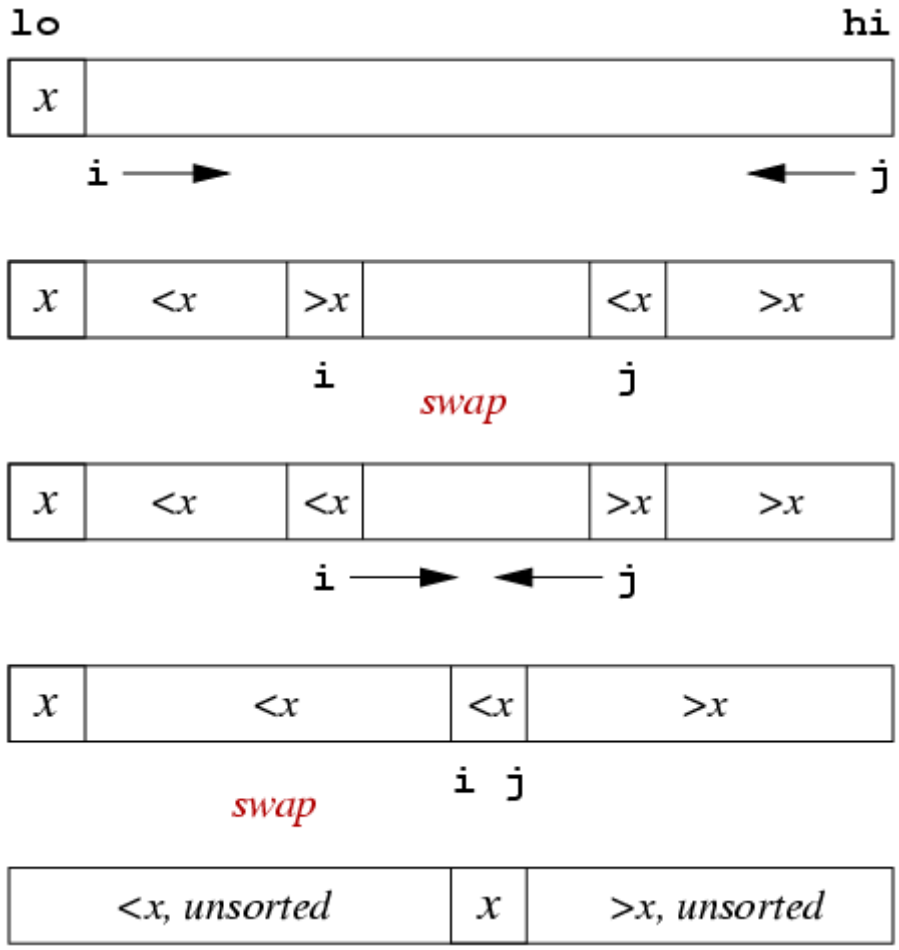
❖ Quicksort Implementation

Elegant recursive solution ...

```
void quicksort(Item a[], int lo, int hi)
{
   int i; // index of pivot
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

❖ ... Quicksort Implementation

Partitioning phase:



❖ ... Quicksort Implementation

Partition implementation:

```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];  // pivot
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

❖ Quicksort Performance

Best case: $O(n \log n)$ comparisons

- choice of pivot gives two equal-sized partitions
- same happens at every recursive level
- each "level" requires approx n comparisons
- halving at each level $\Rightarrow \log_2 n$ levels

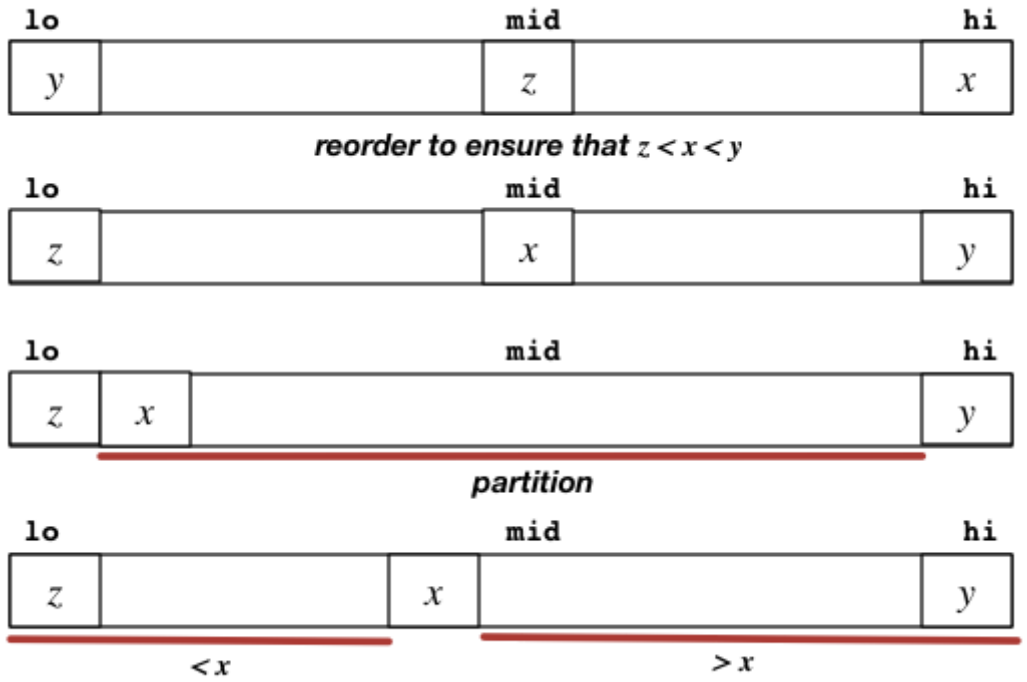
Worst case: $O(n^2)$ comparisons

- always choose lowest/highest value for pivot
- partitions are size 1 and $n-1$
- each "level" requires approx n comparisons
- partitioning to 1 and $n-1 \Rightarrow n$ levels

❖ Quicksort Improvements

Choice of pivot can have significant effect:

- always choosing largest/smallest ⇒ worst case
- try to find "intermediate" value by median-of-three



❖ ... Quicksort Improvements

Median-of-three partitioning:

```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   // now, we have a[lo] < a[mid] < a[hi]
   // swap a[mid] to a[lo+1] to use as pivot
   swap(a, mid, lo+1);
}
void quicksort(Item a[], int lo, int hi)
{
   if (hi <= lo) return;
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

❖ ... Quicksort Improvements

Another source of inefficiency:

- pushing recursion down to very small partitions
- overhead in recursive function calls
- little benefit from partitioning when size < 5

Solution: handle small partitions differently

- switch to insertion sort on small partitions, or
- don't sort yet; use post-quicksort insertion sort

❖ ... Quicksort Improvements

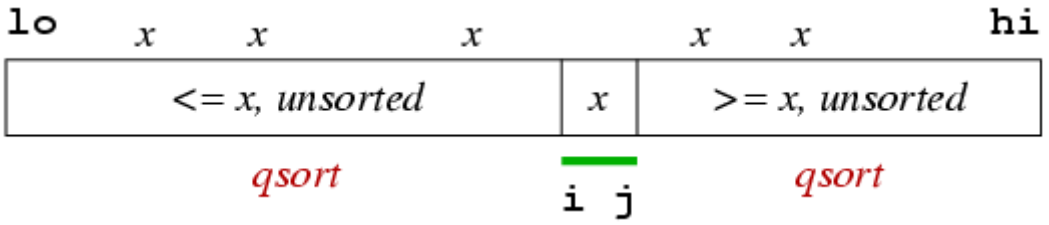
Quicksort with thresholding ...

```
void quicksort(Item a[], int lo, int hi)
{
   if (hi-lo < Threshhold) {
      insertionSort(a, lo, hi);
      return;
   }
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

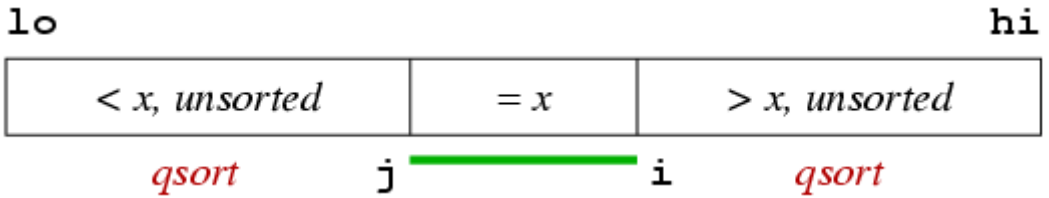
❖ ... Quicksort Improvements

If the array contains many duplicate keys

- standard partitioning does not exploit this

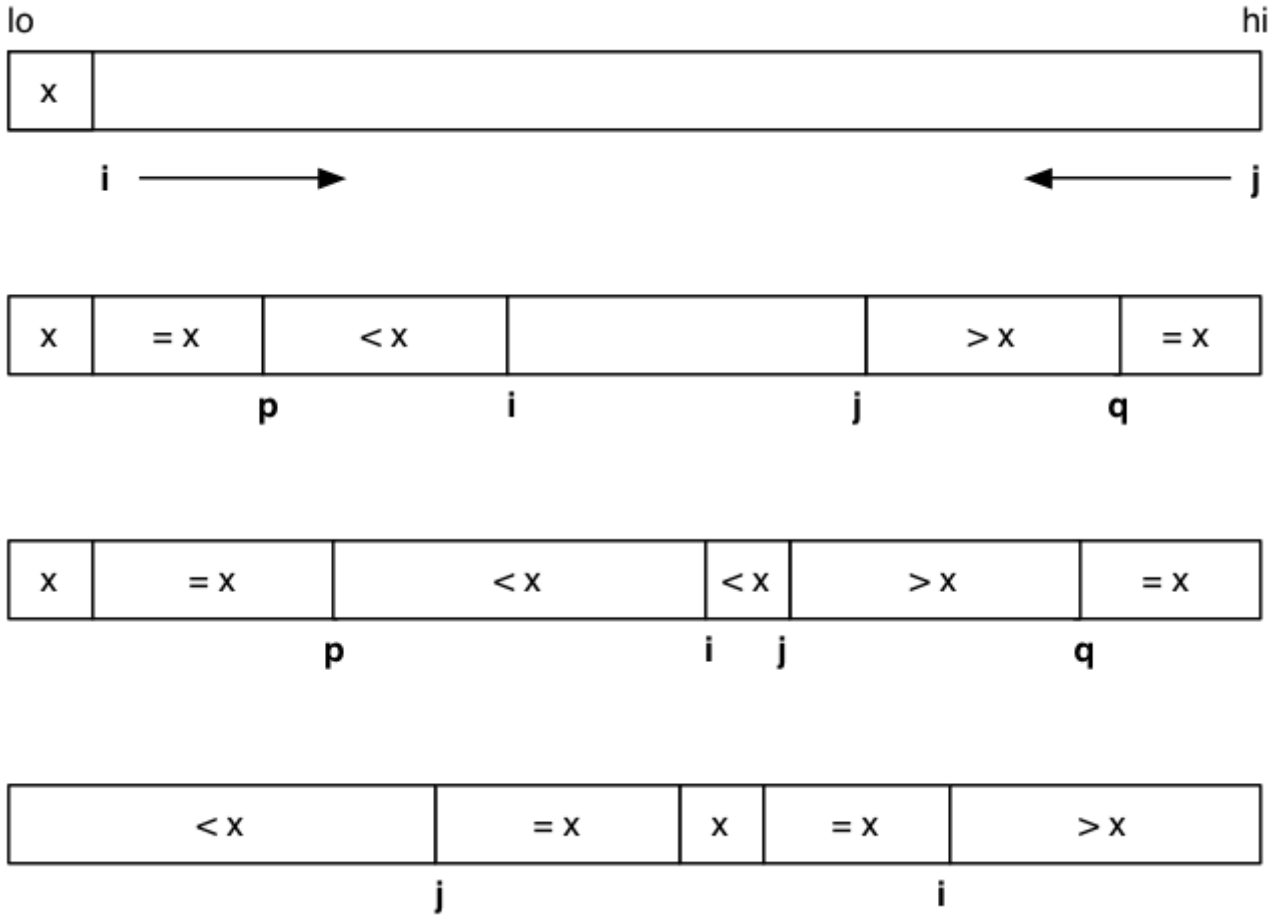


- can improve performance via three-way partitioning



❖ ... Quicksort Improvements

Bentley/McIlroy approach to three-way partition:



❖ Non-recursive Quicksort

Quicksort can be implemented using an explicit stack:

```
void quicksortStack (Item a[], int lo, int hi)
{
   Stack s = newStack();
   StackPush(s,hi); StackPush(s,lo);
   while (!StackEmpty(s)) {
      lo = StackPop(s);
      hi = StackPop(s);
      if (hi > lo) {
         int i = partition (a,lo,hi);
         StackPush(s,hi); StackPush(s,i+1);
         StackPush(s,i-1); StackPush(s,lo);
      }
   }
}
```

Produced: 20 Jul 2020