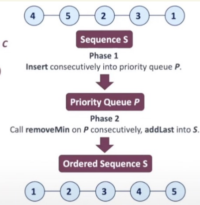
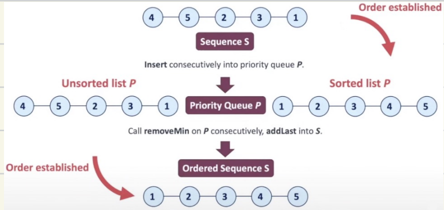
Video 1. Sorting with Priority Queues

```
P ← priority queue using C
while ¬ S.isEmpty ()
    e ← S.remove(S.first ())
    P.insert (e, ∅)

while ¬ P.isEmpty()
    e ← P.removeMin()
    S.addLast(e)
```





Selection Sort (unsorted list PQ)

Sorting n elements.
Unsorted list priority queue P.

Phase 1: insertion
Insertion into priority queue.

Phase 2: removal
Repeated removal of minimal key from P.

	Sequence S	Priority Queue P
Input	(7,4,8,2,5,3,9)	( )
Phase 1		
(a)	(4,8,2,5,3,9)	(**7**)
(b)	(8,2,5,3,9)	(7,**4**)
(c)	(2,5,3,9)	(7,4,**8**)
(d)	(5,3,9)	(7,4,8,**2**)
(e)	(3,9)	(7,4,8,2,**5**)
(f)	(9)	(7,4,8,2,5,**3**)
(g)	( )	(7,4,8,2,5,3,**9**)
Phase 2		(7,4,8,**2**,5,3,9)
(a)	(2)	(7,4,8,5,**3**,9)
(b)	(2,3)	(7,**4**,8,5,9)
(c)	(2,3,4)	(7,8,**5**,9)
(d)	(2,3,4,5)	(**7**,8,9)
(e)	(2,3,4,5,7)	(**8**,9)
(f)	(2,3,4,5,7,8)	(**9**)
(g)	(2,3,4,5,7,8,9)	()

$$nc + n(c-1) + \dots + 2c + 1c$$
$$= c\sum_{i=1}^{n} i = \frac{cn(n-1)}{2} \quad O(n^2)$$

Insertion Sort (sorted list PQ)

Sorting n elements.
Sorted list priority queue P.

Phase 1: insertion
Repeated insertions into priority queue P.
Entries inserted at final sorted position.

Phase 2: removal
Repeated removal of minimal key from P.

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(**7**)
(b)	(8,2,5,3,9)	(**4**,7)
(c)	(2,5,3,9)	(4,7,**8**)
(d)	(5,3,9)	(**2**,4,7,8)
(e)	(3,9)	(2,4,**5**,7,8)
(f)	(9)	(2,**3**,4,5,7,8)
(g)	()	(2,3,4,5,7,8,**9**)
Phase 2		
(a)	(2)	(**3**,4,5,7,8,9)
(b)	(2,3)	(**4**,5,7,8,9)
(c)	(2,3,4)	(**5**,7,8,9)
(d)	(2,3,4,5)	(**7**,8,9)
(e)	(2,3,4,5,7)	(**8**,9)
(f)	(2,3,4,5,7,8)	(**9**)
(g)	(2,3,4,5,7,8,9)	()

$$1c + 2c + \dots + (n-1)c + nc =$$
$$= c\sum_{i=1}^{n} i = \frac{cn(n-1)}{2} \quad O(n^2)$$

Sorting n elements.
Sorted list priority queue P.

Space complexity:
Depends on implementation.

If S is linked list modified by insert/remove.
Then space is $O(1)$.

If S is an array of capacity n (or any linked list that does not shrink, elements are overriden).
Then we need $O(n)$ space for P.

Space complexity is $O(n)$ with array.

	Sequence S Insert/remove (array list)	Sequence S Read/override (array or linked list)	Priority queue P
Input:	(7,4,8,2,5,3,9)	(7,4,8,2,5,3,9)	( )
Phase 1			
(a)	(,4,8,2,5,3,9)	(7,4,8,2,5,3,9)	(**7**)
(b)	(, ,8,2,5,3,9)	(7,4,8,2,5,3,9)	(**4**,7)
(c)	(, , ,2,5,3,9)	(7,4,8,2,5,3,9)	(4,7,**8**)
(d)	(, , , ,5,3,9)	(7,4,8,2,5,3,9)	(**2**,4,7,8)
(e)	(, , , , ,3,9)	(7,4,8,2,5,3,9)	(2,4,**5**,7,8)
(f)	(, , , , , ,9)	(7,4,8,2,5,3,9)	(2,**3**,4,5,7,8)
(g)	(, , , , , ,)	(7,4,8,2,5,3,9)	(2,3,4,5,7,8,**9**)
Phase 2			
(a)	(2, , , , , ,)	(**2**,4,8,2,5,3,9)	(**3**,4,5,7,8,9)
(b)	(2,3, , , , ,)	(**2**,**3**,8,2,5,3,9)	(**4**,5,7,8,9)
(c)	(2,3,4, , , ,)	(**2**,**3**,**4**,2,5,3,9)	(**5**,7,8,9)
(d)	(2,3,4,5, , ,)	(**2**,**3**,**4**,**5**,5,3,9)	(**7**,8,9)
(e)	(2,3,4,5,7, ,)	(**2**,**3**,**4**,**5**,**7**,3,9)	(**8**,9)
(f)	(2,3,4,5,7,8,)	(**2**,**3**,**4**,**5**,**7**,**8**,9)	(**9**)
(g)	(2,3,4,5,7,8,9)	(**2**,**3**,**4**,**5**,**7**,**8**,**9**)	( )

In-place implementation

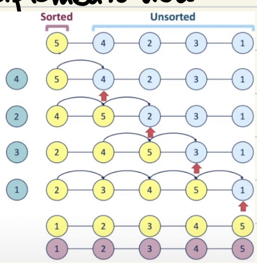
In-place insertion sort

Insertion sort implemented **in-place**.

Only $O(1)$ space used (in addition to the sequence being sorted).

A portion of the input sequence itself serves as the priority queue P.

In-place insertion sort:
Iterate left to right, one index i at a time (where $i = 1$ up to $i = n - 1$).
- Iterate backwards within $0, \dots, i-1$:
  - If element larger than the element at index i, shift a position to the right.
- Insert element at its correct position.



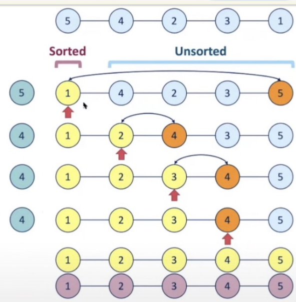
In-place selection sort

Selection sort implemented **in-place**.

Only $O(1)$ space used (in addition to the sequence being sorted).

A portion of the input sequence itself serves as the priority queue P.

In-place selection sort:
Iterate left to right, one index i at a time (where $i = 0$ up to $i = n - 2$).
- Find index m of minimum element within indexes $i, \dots, n-1$.
- Swap elements at i and m.



Video 2. Heap Sort

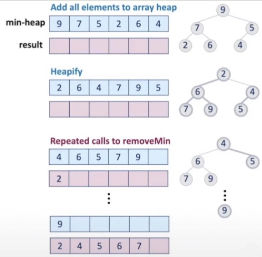
Heap sort

Sorting n elements (non-decreasing).
Heap-based priority queue P (min-heap).

Phase 1: insertion
The i^{th} insertion takes $O(\log_2 i)$ time, since it performs upheap on the heap with i entries.
Takes $O(n \log_2 n)$ time for all insertions.
Can be $O(n)$ using bottom-up construction.

Phase 2: removal
The j^{th} removeMin is $O(\log_2(n-j+1))$, since it downheaps heap of $n-j+1$ entries.
Takes $O(n \log_2 n)$ time for all removals.

Heap sort time complexity
$O(n \log_2 n)$

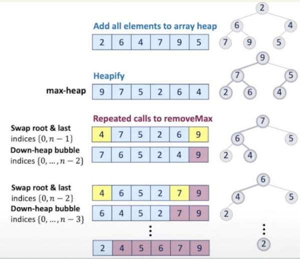


In-place heap sort

To sort n elements in non-decreasing order in-place using a heap:

Use max-heap instead!
Comparator produces reverse outcome of a min-heap, such that:
- Maximal key at the root.
- Parent's key larger than or equal to its children's keys.

If we repeatedly remove the largest, we can place it at the position made free by the removal, at end of the array.



Video 3. Merge Sort

Divide-and-conquer is an **algorithmic design pattern** consisting of 3 steps:

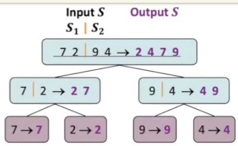
1. **Divide:**
   [Small input: base case]
   If input is small (e.g. 1-2 elements), solve problem directly.
   [Larger input: recurrence]
   Divide the input into two or more disjoint subsets.
2. **Conquer:** Recursively solve the subproblems associated with the subsets.
3. **Combine:** Take the solutions of the subproblems and merge them into a solution to the larger problem.

Execution of merge sort depicted by its recursion tree.

Warning: This is not a tree data structure!

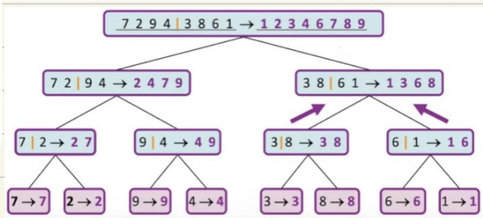
Each node represents a recursive call of merge sort with:
- unsorted sequence before the execution and its partition (**left**);
- sorted sequence at the end of the execution (**right**).

Root contains initial call, and final result.
Leaves are calls on sequences of size 1.



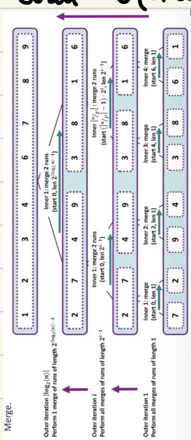
Merge sort uses divide-and-conquer to sort a sequence S with n elements:

1. **Divide:**
   [Base case]
   If S has less than 2 element(s), return S (already sorted).
   [Recurrence]
   If S has at least 2 element(s), split elements of S into 2 sequences S_1 and S_2.
   S_1 and S_2 contain each about half of the elements: S_1 the first $\lfloor n/2 \rfloor$, S_2 the remaining $\lceil n/2 \rceil$.
2. **Conquer:** Recursively sort sequences S_1 and S_2.
3. **Combine:** Put elements back into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.



level	#seqs	size	time
i	2^i	$n/2^i$	$O(n)$

total: $O(n \log_2 n)$




```java
// Merge contents of arrays S1 and S2 into properly sized array S.
public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
    int i = 0, j = 0;
    while (i + j < S.length) {
        // if no more elements in S2, or still elements in S1 and next element in S1 smaller than next element in S2
        if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
            S[i+j] = S1[i++];          // copy ith element of S1 into S and increment i
        else
            S[i+j] = S2[j++];          // copy jth element of S2 into S and increment j
    }
}
// Merge sort elements in array S.
public static <K> void mergeSort(K[] S, Comparator<K> comp) {
    int n = S.length;
    if (n < 2) return;                 // base case, array is trivially sorted
    // divide
    int mid = n/2;
    K[] S1 = Arrays.copyOfRange(S, 0, mid);      // copy of first half
    K[] S2 = Arrays.copyOfRange(S, mid, n);      // copy of second half
    // conquer (with recursion)
    mergeSort(S1, comp);
    mergeSort(S2, comp);
    // merge results
    merge(S1, S2, S, comp);            // merge sorted halves back into original
}
```

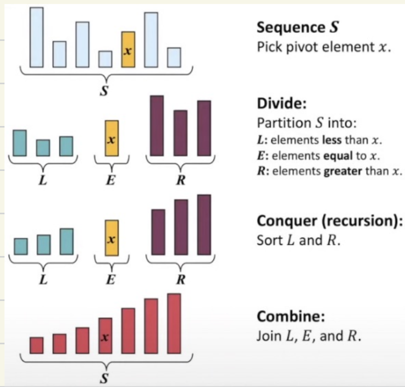
Algorithm	Time (worst)	Time (average)	Time (best)	Space	Properties
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Slow. In-place. For small datasets (< 1K).
Insertion	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	Generally slow. In-place. For small datasets (< 1K). Can be $O(n)$ time for nearly sorted sequences.
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)^1$	$O(1)$	Fast. In-place. For large datasets (1K — 1M). [1]Best $O(n)$ time only if all elements are equal!
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)^2$	$O(n)^3$	Fast. Sequential data access. For huge data sets (> 1M). [2]Can be made to have best $O(n)$ time, but only if sequence is sorted. [3]Difficult to implement in-place, beyond scope of this course.

Video 4. Quick Sort

Quick sort

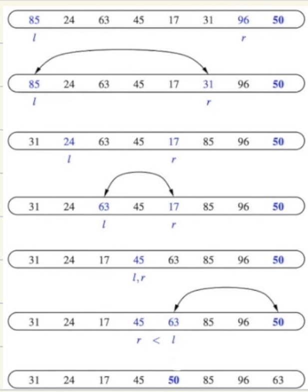
Quick sort uses divide-and-conquer to sort a sequence S with n elements.
In quick sort the hard work is mostly done before the recursive calls.

1. **Divide:**
   [Base case]
   If S has less than 2 element(s), return (nothing to do).
   [Recurrence]
   If S has at least 2 element(s), select a specific element from S, called the **pivot**. E.g. choose **pivot** as the last element in S (other choices possible: e.g. middle).
   Remove all elements from S and split them into 3 sequences:
   - **Sequence L:** elements from S that are **smaller** than **pivot**.
   - **Sequence E:** elements from S that are **equal** to **pivot**. (If all elements in S are unique, then only the pivot)
   - **Sequence R:** elements from S that are **larger** than **pivot**.
2. **Conquer:** Recursively sort sequences L and R.
3. **Combine:** Put elements back into S as follows: **first** all elements of L, **then** elements of E, finally elements of R.



	time	space
worst case:	$O(n^2)$	$O(n)$
average case:	$O(n \log n)$	$O(\log n)$

```
/** Sort the subarray S[a..b] inclusive. */
private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
                                                         int a, int b) {
  if (a >= b) return;          // subarray is trivially sorted
  int left = a;
  int right = b-1;
  K pivot = S[b];
  K temp;                  // temp object used for swapping
  while (left <= right) {
    // scan until reaching value equal or larger than pivot (or right marker)
    while (left <= right && comp.compare(S[left], pivot) < 0) left++;
    // scan until reaching value equal or smaller than pivot (or left marker)
    while (left <= right && comp.compare(S[right], pivot) > 0) right--;
    if (left <= right) {       // indices did not strictly cross
      // so swap values and shrink range
      temp = S[left]; S[left] = S[right]; S[right] = temp;
      left++; right--;
    }
  }
  // put pivot into its final place (currently marked by left index)
  temp = S[left]; S[left] = S[b]; S[b] = temp;
  // make recursive calls
  quickSortInPlace(S, comp, a, left - 1);
  quickSortInPlace(S, comp, left + 1, b);
}
```



Algorithm	Time	Properties
Selection sort	$O(n^2)$	In-place. Slow. OK for small input, but insertion sort is typically better.
Insertion sort	$O(n^2)$	In-place. Slow, good for small input. Can be $O(n)$ for nearly sorted input.
Quick sort	$O(n \log_2 n)^*$	In-place. Randomized. Fastest (good for large inputs). Worst-case $O(n^2)$.
Heap sort	$O(n \log_2 n)$	In-place. Fast (good for large inputs).
Merge sort	$O(n \log_2 n)$	Sequential data access. Fast (good for huge inputs).