RUNNING TIME AND MEMORY USAGE

- What influence has the size of the input on the running time?
- What influence has the ordering of the input (is it randomly distributed, is it nearly sorted, is sorted the wrong way, etc.) on the running time.
- What is worst case, average case and best case?
- Is auxiliar memory needed?

SELECTION SORT

[Quadratic sort → $O(n^2)$]

- Sorts an array in passes
  - ↳ Each pass selects the next smallest element
  - ↳ At the end of the pass, places it where it belongs

LOGIC: array is considered into 2 parts: unsorted & sorted.
Initially whole array is unsorted

SELECTION
SWAPPING
COUNTER SHIFT

• IMPLEMENTATION

1. for fill = 0 to n-2 do // steps 2-6 form a pass
2. set pos_min = fill
3. for next = fill+1 to n-1 do
4. if item at next < item at pos_min
5. set pos_min to next
6. Exchange item at pos_min with one at fill

- Performs:
  - → $O(n^2)$ comparisons
  - → $O(n)$ exchanges (swaps)
  - → In place sorting (no extra memory)

35 65 30 60 20

num = 35 → 35 < 60 True
35 < 30 False → num = 30
30 < 60 True
30 < 20 False → num = 20

20 | 65 30 60 35
num = 65 → 65 < 30 False → num = 30
30 < 60 True
30 < 35 True

20 30 | 65 60 35
num = 65 → 65 < 60 False → num = 60
60 < 35 False → num = 35

20 30 35 | 60 65
num = 60 → 60 < 65 True

20 30 35 60 65

```
for(j=0; j<n-1; j++){
  int iMin = j;
  for(i=j+1; i<n; i++){
    if(a[i] < a[iMin]
      iMin = i;
  if(iMin != j)
    swap(a[j], a[iMin])
```

Why Quadratic?

First pass does n-1 comparisons, second does n-2, and so forth until there are 2 elements left. Mathematical: $n-1+n-2+\dots+2+1$

Gauss: $s = \frac{n(n+1)}{2}$ we have to subtract n: $s = \frac{n(n+1)}{2} - n = \frac{n^2+n}{2} - n = \frac{n^2}{2} + \frac{n}{2} - \frac{2n}{2} = \frac{n^2}{2} - \frac{n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$

↳ Quadratic

BUBBLE SORT [Quadratic sort → $O(n^2)$]

- Compares adjacent array elements
  - ↳ Exchanges their values if they are out of order.
- Smaller values bubble up to the top of the array
  - ↳ Larger values sink to the bottom.

[60,42,75,83,27] → [42,60,75,27,83] → [42,60,27,75,83] → [42,27,60,75,83] → [27,42,60,75,83]

60 < 42 False ↓ [42,60,75,83,27]
60 < 75 True
75 < 83 True
83 < 27 False

42 < 60 True
60 < 75 True
75 < 27 False

42 < 60 True
60 < 27 False

42 < 27 False ↓
27 < 42 True
42 < 60 True
60 < 75 True
75 < 83 True
SORTED

• IMPLEMENTATION

```
for i=n down to 2 do
    for j=1 to i-1 do
        if A[j] > A[j+1]
            then A[j] ↔ A[j+1]
        end if
    end for
end for
```

• ANALYSIS

- Works best when array is nearly sorted to begin with
- Worst case n° of comparisons: $O(n^2)$
- Worst case n° of exchanges: $O(n^2)$
- Best case n° of comparisons: $O(n)$
- Best case n° of exchanges: $O(1)$ → none actually

INSERTION SORT

[Quadratic sort → $O(n^2)$]

- Based on technique of card players to arrange a hand.
  - ↳ Player keeps cards picked up so far in sorted order
    - ↳ Makes room for the new car
      - ↳ Then inserts it in its proper place

• ANALYSIS

* Max n° of comparisons: $O(n^2)$
* Best case n° of comparisons: $O(n)$
* In place sorting

→ Fast for small and nearly sorted arrays (used by more complex sorting algorithms).

• ALGORITHM

* For each element from 2nd (next_pos = 1) to last:
  - ↳ Insert element at next_pos where it belongs
  - ↳ Increases sorted subarray size by 1.

→
1. Work left to right
2. Examine each item and compare it to items on its left.
3. Insert the item in the correct position in the array

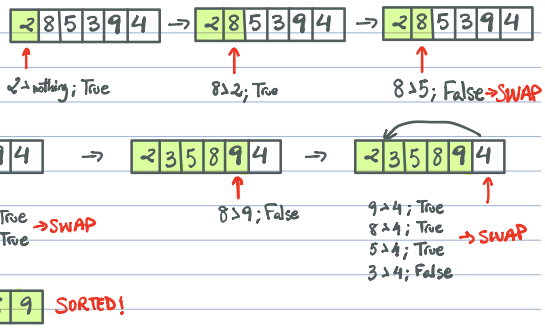
* The array will form sorted and unsorted positions.

* To make room:
  - ↳ Hold next_pos value in a variable
  - ↳ Shuffle elements to the right until gap at right place.

```
for (int i=1; i < A.length-1; i++) {
    j = i
    while j>0 && A[j-1] > A[j]
        swap A[j] ↔ A[j-1]
        j = j-1
```

Example:

sorted items



• COMPARISON OF QUADRATIC SORTS

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

- None good for large arrays

Insertion Sort
↳ Small arrays
↳ Almost sorted arrays

Shell Sort
↳ Very large arrays

SHELL SORT: A BETTER INSERTION SORT

- Divide and conquer approach to insertion sort
  - ↳ Sort many smaller subarrays using insertion sort
  - ↳ Sort progressively larger arrays
  - ↳ Finally sort the entire array
- These arrays are elements separated by a gap
  - ↳ Start with large gap
  - ↳ Decrease the gap on each pass

- Avg. performance: $O(n^{3/2})$ or better

• ANALYSIS

- Performance depends on sequence of gap values.
  - ↳ For sequence $2^k \rightarrow O(n^2)$
  - ↳ Hibbard's sequence $(2^k-1) \rightarrow O(n^{3/2})$
- We start with n/2 and repeateadly divide by 2.2
  - ↳ Empirical results show this is $O(n^{5/4})$ or $O(n^{7/6})$
  - ↳ No theoretical basis (proof) that this holds

• Implementation ??

MERGE SORT

- Performed in 2 sequences of data
  - → items in both sequences use same compare
  - → Both sequences in ordered of this compare

Merge sort merges longer and longer sequences

GOAL: combine the two sorted sequences in one larger sorted sequence

• IMPLEMENTATION

1. Access the first item from both sequences
2. While neither sequence is finished
   1. Compare the current items of both
   2. Copy smaller current item to the output
   3. Access next item from that input sequence
3. Copy any remaining from first sequence to output
4. Copy any remaining from second to output

• ANALYSIS

- 2 input sequences → Must move each element to the output → Merge time is O(n)
- Must store both input and output sequences → An array cannot be merged in place ?? → Additional space needed: O(n)

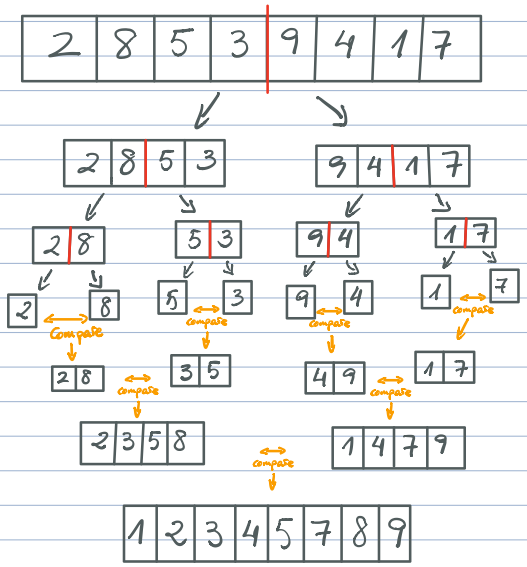
• IMPLEMENTATION

↳ Split array into 2 halves
↳ Sort the left half (recursively)
↳ Sort the right half (recursively)
↳ Merge the 2 sorted halves

```
mergeSort(array a)
  if(n==1)
    return
  arrayOne = a[0] ... a[n/2]
  arrayTwo = a[n/2+1] ... a[n]
  arrayOne = mergeSort(arrayOne)
  arrayTwo = mergeSort(arrayTwo)
  return merge(arrayOne, arrayTwo)
```

```
merge(array a, array b)
  array c
  while(a and b have elements)
      if(a[0] > b[0])
          add b[0] to the end of c
          remove b[0] from b
      else
          add a[0] to the end of c
          remove a[0] from a
  //At this point either a or b is empty
  while(a has elements)
      add a[0] to the end of c
      remove a[0] from a

  while(b has elements)
      add b[0] to the end of c
      remove b[0] from b
  return c
```

Example:



• ANALYSIS

* Splitting/copying n elements to subarrays: O(n)
* Merging back into original array: O(n)
* Recursive calls: 2, each of size n/2
  ↳ Their total non-recursive work again O(n)
* Size sequence: n, n/2, n/4, ..., 1
  ↳ N° of levels = log n
  ↳ Total work: O(n log n)

- Looking at the while loop, we see we have to visit n items
- log n comes from the maximum height of the binary tree we create

HEAPSORT

- Works in place ; no additional storage → Merge Sort time is O(n log n) ; but requires (temporarily) n extra storage items.
- Performance : O (n log n)

- Idea (not quite in-place)
  - ↳ Insert each element into a priority queue
  - ↳ Repeatedly remove from priority queue to array (array slots go from 0 to n-1)

• IMPLEMENTATION

- Build heap starting from unsorted array
- While the heap is not empty
  - ↳ Remove the first item from the heap:
    - ↳ Swap it with the last item
  - ↳ Restore the heap property

EXAMPLE + PSEUDOCODE ??

• ANALYSIS

- Insertion cost is log i for heap of size i
  - ↳ Total insertion cost = log (n) + log (n-1) + ... + log (1)
  - ↳ This is O(n log n)

- Removal cost is also log i for heap of size i
  - ↳ Total removal cost = O (n log n)

- Total cost is O(n log n)

QUICKSORT

- Given a pivot value
  - ↳ Rearranges array into 2 parts
    - ↳ Left part ≤ pivot value
    - ↳ Right part > pivot value

- Average case is O(n log n)
- Worst case is O(n^2)

How do we choose the pivot ?

Huge impact in the performance of the algorithm.

- Popular method: median-of-three
  In this method we look at the last, middle and first of the array, sort them properly and choose the middle item as our pivot. We are making a guess that the middle of these 3 items will be close to the median of the array.

2	6	5	3	8	7	1	0	4

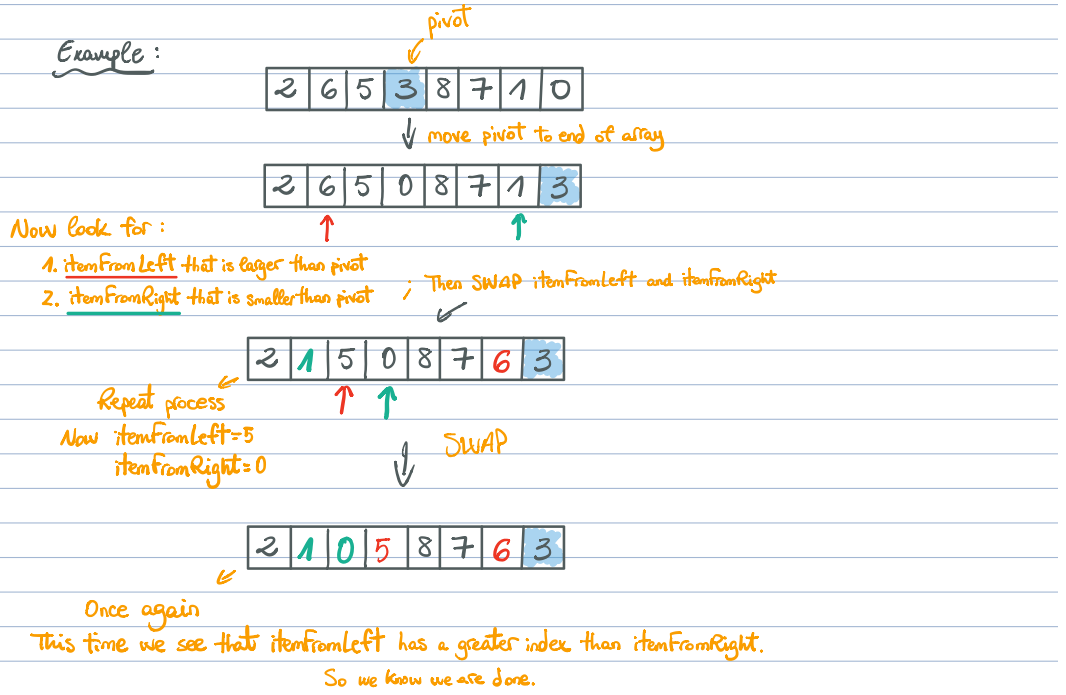
↓

2	6	5	3	4	7	1	0	8

• IMPLEMENTATION

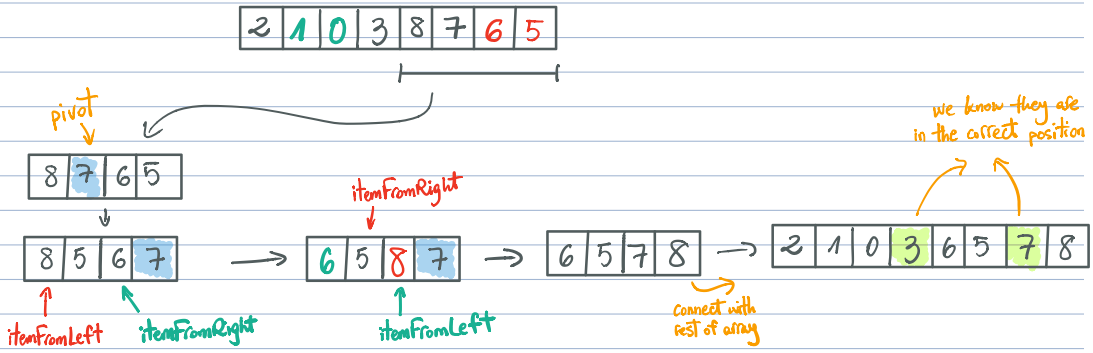
```
Quicksort (A as array, low as int, high as int)
    if (low < high)
        pivot_location = Partition (A, low, high)
        Quicksort (A, low, pivot_location)
        Quicksort (A, pivot_location+1, high)

Partition (A as array, low as int, high as int)
    pivot = A[low]
    leftwall = low

    for i = low+1 to high
        if (A[i] < pivot) then
            swap (A[i], A[leftwall])
            leftwall = leftwall+1

    swap (pivot, A[leftwall])
    return (leftwall)
```

Example :



Stop when index of itemFromLeft > index of itemFromRight

Swap itemFromLeft with our pivot



TESTING SORTING ALGORITHMS

- Need to use a variety of test cases
  - ↳ Small and large arrays
  - ↳ Arrays in random order
  - ↳ Arrays that are already sorted (and reverse order)
  - ↳ Arrays with duplicate values
- Compare performance on each type of array

SUMMARY OF PERFORMANCE

Number of Comparisons			
	Best	**Average**	**Worst**
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$