1 Insertion Sort 4
  1.1. Sorted Insert (Forward) ........................................ 4
  1.2. Sorted Insert (Reverse) ........................................ 5
  1.3. Out-of-Place Insertion Sort ................................. 6
  1.4. In-Place Insertion Sort ....................................... 7

2 Selection Sort 8
  2.1. Find Minimum .............................................. 8
  2.2. Out-of-Place Selection Sort ............................... 9
  2.3. In-Place Selection Sort .................................... 10

3 Merge Sort 11
  3.1. Merge ...................................................... 11
  3.2. Merge Sort ................................................ 13
  3.3. Hybrid Merge/Insertion Sort .............................. 16

4 Quick Sort 18
  4.1. Partition .................................................. 18
4.2. In-Place Quick Sort
• We will explore a variety of different kinds of algorithms:
  – **Out-of-Place Algorithms**: Gradually copy elements from input array into a temporary array; by the end the temporary array is sorted; $O(N)$ space complexity
  – **In-Place Algorithms**: Keep all elements stored in the input array; use input array for intermediate results; no temporary storage is required; $O(1)$ space complexity
  – **Iterative Algorithms**: Use iteration statements to implement an iterative sorting strategy
  – **Recursive Algorithms**: Use recursion to implement a divide-and-conquer sorting strategy
  – **Hybrid Algorithms**: Initially use one algorithm, but switch to a different algorithm sometime during the sorting process

• For each algorithm, we will ...
  – start by exploring a helper function
  – use this helper function to implement a sorting function

• For each function, we will use ...
  – cards to build intuition behind algorithm
  – pseudocode to make algorithm more concrete
  – visual pseudocode to precisely illustrate algorithm
  – complexity analysis
1. Insertion Sort

- sorted_insert helper function (forward and reverse variants)
- Call sorted_insert for every element in input array

1.1. Sorted Insert (Forward)

- Insert new element into sorted array such that array remains sorted
- Search array in the forward direction

```python
1 def sorted_insert_fwd( a, begin, end, v ):
2     # find where to insert new value
3     idx = begin
4     while (idx < end) and (v > a[idx]):
5         idx = idx + 1
6
7     # move all elements down to make room
8     tmp = v
9     for i in idx to end
10        swap( a[i], tmp )
11     a[end] = tmp
```
1. Insertion Sort

1.2. Sorted Insert (Reverse)

- Insert new element into sorted array such that array remains sorted
- Search array in the reverse direction

```python
def sorted_insert_rev( a, begin, end, v ):
    a[end] = v
    for i in reverse( begin to end ):
        if ( a[i+1] < a[i] ):
            swap( a[i+1], a[i] )
        else:
            break
```

---

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```
1.3. Out-of-Place Insertion Sort

- For each element in input array, use `sorted_insert` to **insert** it into a temporary output array
- Copy temporary array back into input array

```
1. def isort_op_fwd( a, size ):
2.
3. set tmp to an empty array with size elements
4. for i in 0 to size:
5.     sorted_insert_fwd( tmp, 0, i, a[i] )
6.
7. for i in 0 to size:
8.     a[i] = tmp[i]
```
1.4. In-Place Insertion Sort

- Divide input array into sorted and unsorted partitions
- Use sorted insert to **insert** elements from unsorted to sorted partition

```python
def isort_ip_rev( a, size ):
    for i in 0 to size:
        sorted_insert_rev( a, 0, i, a[i] )
```

2. Selection Sort

- find_min helper function
- Call find_min for every element in output array

2.1. Find Minimum

- Return index of the minimum value in an array

```python
def sorted_insert_fwd( a, begin, end ):
    min_value = a[begin]
    min_idx   = begin

    for i in begin to end:
        if a[i] < min_value:
            min_value = a[i]
            min_idx   = i

    return min_idx
```
2.2. Out-of-Place Selection Sort

- Use `find_min` to select elements to add to temporary output array
- Reset selected elements to avoid selecting them twice
- Copy temporary array back into input array

```python
def ssort_op( a, size ):
    set tmp to an empty array with size elements
    for i in 0 to size:
        idx = find_min( a, 0, size )
        tmp[i] = a[idx]
        a[idx] = MAX_INT
    for i in 0 to size:
        a[i] = tmp[i]
```
2.3. In-Place Selection Sort

- Divide input array into sorted and unsorted partitions
- Use \texttt{find\_min} to \textit{select} elements to move from unsorted to sorted partitions

```python
def ssort_ip( a, size ):
    for i in 0 to size:
        idx = find_min( a, i, size )
        swap( a[i], a[idx] )
```

3. Merge Sort

- merge helper function
- Recursively divide array into partitions, merge sorted partitions

3.1. Merge

- Merge two sorted input arrays into separate output array
- Ensure output array is also sorted

```python
def merge( c, a, begin0, end0, b, begin1, end1 ):
    size = ( end0 - begin0 ) + ( end1 - begin1 )
    assert len(c) == size

    idx0 = begin0
    idx1 = begin1

    for i in 0 to size:
        # done with array a
        if ( idx0 == end0 ):
            c[i] = b[idx1]
            idx1 += 1

        # done with array b
        elif ( idx1 == end1 ):
            c[i] = a[idx0]
            idx0 += 1

        # front of array a is less than front of array b
        elif ( a[idx0] < b[idx1] ):
            c[i] = a[idx0]
            idx0 += 1

        # front of array by is less than front of array a
        else:
            c[i] = b[idx1]
            idx1 += 1
```
3.2. Merge Sort

- Recursively partition input array into halves
- Base case is when a partition contains a single element
- After recursive calls return, use merge to merge sorted partitions

```python
def msort_op_h( a, begin, end ):
    size = end - begin
    if ( size == 1 ):
        return

    mid = ( begin + end ) / 2
    msort_op_h( a, begin, mid )
    msort_op_h( a, mid, end )

    set tmp to an empty array with size elements
    merge( tmp, a, begin, mid, a, mid, end )

    # copy temporary array to input array
    j = 0
    for i in begin to end:
        a[i] = tmp[j]
        j += 1

def msort_op( a, size ):
    msort_op_h( a, 0, size )
```
• Show contents of `a` for each recursive call
• Show contents of `tmp` for each merge
• Worst-case time complexity analysis

• Space complexity analysis
3.3. Hybrid Merge/Insertion Sort

- Once array becomes small enough, use $O(N^2)$ sort

```python
1  msort_hybrid_h( a, begin, end )
2      size = end - begin
3  if ( size <= 4 ):
4      return isort_op( a, begin, end )
5  ...
```
• Worst-case time complexity analysis
4. Quick Sort

- Use partition helper function to recursively partition array

4.1. Partition

- Choose an element as the *pivot* and *partition* based on pivot
- Move all elements less than the pivot to front of the array
- Move all elements greater than the pivot to end of the array
- Pivot’s final location is in between these two partitions

```python
def partition( a, begin, end ):  
pivot = a[end-1]  
idx = begin  
for i in begin to end:  
    if ( a[i] <= pivot ):  
        swap( a[i], a[idx] )  
        idx += 1  
return idx-1
```
4.2. In-Place Quick Sort

- Recursively partition input array using \texttt{partition}
- Base case is when a partition contains a single element

```python
1 def qsort_ip_h( a, begin, end ):
2     if ( begin >= end ):
3         return
4
5     p = partition( a, begin, end )
6     qsort_ip_h( a, begin, p )
7     qsort_ip_h( a, p + 1, end )
8
9 def qsort_ip( a, size ):
10     qsort_ip_h( a, 0, size )
```
4. Quick Sort

4.2. In-Place Quick Sort
4. Quick Sort

4.2. In-Place Quick Sort

- Time complexity analysis