1 Insertion Sort 3
  1.1. Out-of-Place Insertion Sort 3
  1.2. In-Place Insertion Sort (forward search) 5
  1.3. In-Place Insertion Sort (reverse search) 7
  1.4. Activity 9

2 Selection Sort 10
  2.1. Out-of-Place Selection Sort 10
  2.2. In-Place Selection Sort 12
  2.3. Activity 15

3 Merge Sort 16
  3.1. Hybrid Merge/Insertion Sort 19

4 Quick Sort 21
  4.1. Out-of-Place Quick Sort 21
  4.2. In-Place Quick Sort 24
  4.3. In-Place Quick Sort 25
4.4. Hybrid Quick/Insertion Sort ........................................... 25
4.5. Activity ................................................................. 25

5 Radix Sort ................................................................. 25
5.1. Out-of-Place Radix Sort ............................................. 25
5.2. Activity ................................................................. 25
• We will explore three 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
  – **Hybrid Algorithms:** Initially use one algorithm, but switch to a different algorithm sometime during the sorting process

• For each algorithm 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**

• Take elements out of input array and insert them into a sorted output array such that the output array remains sorted

1.1. **Out-of-Place Insertion Sort**
```python
isort_op( a, size )
  set tmp to an empty array of with size elements
  for i in 0 to size-1 (inclusive) # iterate over input
    temp = a[i]
    for j in 0 to i-1 (inclusive) # iterate over output
      if temp < tmp[j]
        swap( temp, tmp[j] )
    tmp[i] = temp
  return tmp
```

- Show contents of tmp for each iteration of outer loop
• Space complexity is $O(N)$ due to temporary array $b$
• Worst-case time complexity analysis
  – Assume $C_0$ is time spent in one iteration of outer loop excluding any time spent in inner loop
  – Assume $C_1$ is time spent in one iteration of inner loop

1.2. In-Place Insertion Sort (forward search)
1. Insertion Sort

1.2. In-Place Insertion Sort (forward search)

```
isort_ip_fwd( a, size )
  for i in 0 to size-1 (inclusive) # iterate over input
    temp = a[i]
    for j in 0 to i-1 (inclusive) # iterate over sorted
      if temp < a[j]: # region
        swap( temp, a[j] )
      a[i] = temp
  return a
```

- Show contents of `a` for each iteration of outer loop
1. Insertion Sort

1.3. In-Place Insertion Sort (reverse search)

- Space complexity is $O(1)$, no temporary array
- Worst-case time complexity analysis
  - Assume $C_0$ is time spent in one iteration of outer loop excluding any time spent in inner loop
  - Assume $C_1$ is time spent in one iteration of inner loop

1.3. In-Place Insertion Sort (reverse search)
1. Insertion Sort

1.3. In-Place Insertion Sort (reverse search)

```python
isort_ip_rev( a, size )
  for i in 1 to size-1 (inclusive)
    for j in i-1 to 0 (inclusive)
      if a[j+1] < a[j]
        swap( a[j], a[j+1] )
      else # stop once new value
        break # is in the right position
  return a
```

• Space complexity is $O(1)$, no temporary array
• Worst-case time complexity analysis
  – Assume $C_0$ is time spent in one iteration of outer loop excluding any time spent in inner loop
  – Assume $C_1$ is time spent in one iteration of inner loop

• Best-case time complexity analysis
  – Assume input array is already sorted
1.4. Activity

- Use in-place insertion sort
- Show contents of a for each iteration of outer loop
2. Selection Sort

- Select minimum element from input array and add this element to the end of the sorted output array such that the output array remains sorted

2.1. Out-of-Place Selection Sort
ssort_op( a, size )
set tmp to an empty array of with size elements
for i in 0 to size-1 (inclusive)
    min_value = a[0]  # Find the minimum
    min_idx   = 0
    for j in 0 to size-i-1 (inclusive)
        if min_value > a[j]
            min_value = a[j]
            min_idx   = j
    for j in min_idx to size-i-2 (inclusive)  # Remove the minimum
        a[j] = a[j+1]
    tmp[i] = min_value  # Put minimum in output
return tmp

- Show contents of a and b for each iteration of outer loop
2. Selection Sort

• Space complexity is $O(N)$ due to temporary array $b$
• Worst-case time complexity analysis
  – Assume $C_0$ is time spent in one iteration of outer loop excluding any time spent in inner loops
  – Assume $C_1$ is time spent in one iteration of first inner loop
  – Assume $C_2$ is time spent in one iteration of second inner loop

2.2. In-Place Selection Sort
2. Selection Sort

2.2. In-Place Selection Sort

```python
ssort_ip( a, size )
    for i in 0 to size-1 (inclusive)

        min_value = a[i]    # Find the minimum
        min_idx   = i
        for j in i to size-1 (inclusive)
            if min_value > a[j]
                min_value = a[j]
                min_idx   = j

        swap( a[min_idx], a[i] )    # Swap the minimum

    return a
```

- Show contents of `a` for each iteration of outer loop

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Space complexity is $O(N)$ due to temporary array $b$

• Worst-case time complexity analysis
  – Assume $C_0$ is time spent in one iteration of outer loop excluding any time spent in the inner loop
  – Assume $C_1$ is time spent in one iteration of inner loop
2.3. Activity

- Use in-place selection sort
- Show contents of a for each iteration of outer loop
3. Merge Sort

- Recursively partition input array into smaller partitions
- Base case is when one element is in a partition
- Merge two sorted partitions into a larger partition

```python
msort_op( a, size )
if ( size == 1 )
    return a
left_size = size/2
right_size = size - left_size
left = msort2_op( a[0:left_size], left_size )
right = msort2_op( a[left_size:size], right_size )
set tmp to an empty array of with size elements
left_idx = 0; right_idx = 0
for k in 0 to size-1 (inclusive)
    # Done with left array
    if ( left_idx == left_size )
        tmp[k] = right[right_idx]; right_idx += 1
    # Done with right array
    elif ( right_idx == right_size )
        tmp[k] = left[left_idx]; left_idx += 1
    # Front of left is less than front of right
    elif ( left[left_idx] < right[right_idx] )
        tmp[k] = left[left_idx]; left_idx += 1
    # Front of right is less than front of less
    else:
        tmp[k] = right[right_idx]; right_idx += 1
return tmp
```
3. Merge Sort

- Show contents of a for each recursive call
- Show contents of tmp for each merge
3. Merge Sort

- Worst-case time complexity analysis
  - Assume $C_0$ is time spent in one iteration of merge loop
  - Assume $C_1$ is time spent in base case

- Space complexity analysis
3.1. Hybrid Merge/Insertion Sort

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

```c
1 msort_hybrid(a, size)
2   if (size <= 4)
3       return isort_op(a, size)
4   ...
```
• Worst-case time complexity analysis
  – Assume $C_0$ is time spent in one iteration of merge loop
  – Assume $C_2$ is time spent in one iteration of isort inner loop
4. Quick Sort

4.1. Out-of-Place Quick Sort

- Pick a pivot element to partition input array into two partitions
- All elements less than the pivot are in first partition
- All elements more than the pivot are in second partition
- Now know position of pivot
- Recursively sort each of the two partitions

```python
def qsort_op(a, size):
    if size <= 1:
        return a

    pivot = a[size-1]

    left = []
    left_size = 0
    right = []
    right_size = 0

    for i in range(size-2):
        if a[i] < pivot:
            left.append(a[i])
            left_size += 1
        else:
            right.append(a[i])
            right_size += 1

    left = qsort_op(left, left_size)
    right = qsort_op(right, right_size)

    return left + [pivot] + right
```
• Time complexity analysis
4.2. In-Place Quick Sort

```
partition( A, lo, hi )
  pivot = a[hi]
  i = lo - 1
  for j in lo to hi-1 (inclusive)
    if ( a[j] < pivot )
      i = i + 1
      swap( a[j], a[i] )
    if ( a[hi] < a[i+1] )
      swap( a[hi], a[i+1] )
  return i + 1

qsort_ip_h( a, lo, hi )
  if ( lo < hi )
    p = partition( a, lo, hi )
    qsort_ip_h( a, lo, p - 1 )
    qsort_ip_h( a, p + 1, hi )

qsort_ip( a, size )
  qsort_ip_h( a, 0, size-1 )
  return a
```
5. Radix Sort

5.1. Out-of-Place Radix Sort

5.2. Activity