# Topic 9: Sorting Algorithms

School of Electrical and Computer Engineering
Cornell University

revision: 2020-10-12-10-10

## 1 Insertion Sort

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

## 2 Merge Sort

2.1. Merge ................................................................. 7
2.2. Merge Sort .......................................................... 8
2.3. Hybrid Merge/Insertion Sort .......................... 12

## 3 Quick Sort

3.1. Partition ............................................................. 14
3.2. In-Place Quick Sort ............................................. 14

## 4 Comparing Sorting Algorithms

17
**Additional Topics Covered in zyBook**

While the zyBook covers the topics in lecture, it also contains additional material not covered in lecture. Students are responsible for all material covered in lecture and in the zyBook. Material from both lecture and the zyBook will be assessed in the exams. Examples of additional material covered in the zyBook but not lecture include:

- selection sort
- radix 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; $O(N)$ heap space complexity
  – **In-Place Algorithms**: Keep all elements stored in the input array; use input array for intermediate results; no temporary storage is required; $O(1)$ heap 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
  – complexity analysis

• A quick preliminary reminder about finite sequences

<table>
<thead>
<tr>
<th>sequence</th>
<th># elements</th>
<th>sum of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, ..., N-1, N</td>
<td>$N + 1$</td>
<td>$\frac{N+1}{2}(N)$</td>
</tr>
<tr>
<td>0, 1, 2, ..., N-2, N-1</td>
<td>$N$</td>
<td>$\frac{N}{2}(N-1)$</td>
</tr>
<tr>
<td>1, 1, 2, ..., N-1, N</td>
<td>$N$</td>
<td>$\frac{N}{2}(N+1)$</td>
</tr>
<tr>
<td>1, 1, 2, ..., N-2, N-1</td>
<td>$N - 1$</td>
<td>$\frac{N-1}{2}(N)$</td>
</tr>
</tbody>
</table>
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
def sorted_insert_fwd( a, begin, end, v ):
    # find where to insert new value
    idx = begin
    while (idx < end) and (v > a[idx]):
        idx = idx + 1

    # move all elements down to make room
    tmp = v
    for i in idx to end
        swap( a[i], tmp )
    a[end] = tmp
```

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

```python
def isort_op_fwd( a, size ):
    # set tmp to an empty array with size elements
    tmp = [0] * size
    for i in range(size):
        sorted_insert_fwd( tmp, 0, i, a[i] )
    for i in range(size):
        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
1 def isort_ip_rev( a, size ):  
2   for i in 0 to size: 
3     sorted_insert_rev( a, 0, i, a[i] )
```
2. Merge Sort

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

2.1. Merge

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

```
1 def merge( c, a, begin0, end0, b, begin1, end1 ):
2     size = ( end0 - begin0 ) + ( end1 - begin1 )
3     assert len(c) == size
4
5     idx0 = begin0
6     idx1 = begin1
7
8     for i in 0 to size:
9
10        # done with array a
11        if ( idx0 == end0 ):
12           c[i] = b[idx1]
13           idx1 += 1
14
15        # done with array b
16        elif ( idx1 == end1 ):
17           c[i] = a[idx0]
18           idx0 += 1
19
20        # front of array a is less than front of array b
21        elif ( a[idx0] < b[idx1] ):
22           c[i] = a[idx0]
23           idx0 += 1
24
25        # front of array b is less than front of array a
26        else:
27           c[i] = b[idx1]
28           idx1 += 1
```
2.2. 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
• Time complexity analysis
• Space complexity analysis
## 2.3. Hybrid Merge/Insertion Sort

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

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

- Use partition helper function to recursively partition array

3.1. 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
```

3.2. In-Place Quick Sort

- Recursively partition input array using *partition*
- Base case is when a partition contains a single element

```python
def qsort_ip_h( a, begin, end ):
    if ( begin >= end ):
        return

    p = partition( a, begin, end )
    qsort_ip_h( a, begin, p )
    qsort_ip_h( a, p + 1, end )

def qsort_ip( a, size ):
    qsort_ip_h( a, 0, size )
```
• Time complexity analysis
### 4. Comparing Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Worst Case</td>
<td>Avg Case</td>
</tr>
<tr>
<td>insertion</td>
<td>op</td>
<td>ip</td>
</tr>
<tr>
<td>insertion</td>
<td>op</td>
<td>ip</td>
</tr>
<tr>
<td>selection</td>
<td>op</td>
<td>ip</td>
</tr>
<tr>
<td>selection</td>
<td>ip</td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td>op</td>
<td></td>
</tr>
<tr>
<td>quick</td>
<td>ip</td>
<td></td>
</tr>
<tr>
<td>radix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>