ECE 2400 Computer Systems Programming
Fall 2017

Topic 10: Sorting Algorithms

School of Electrical and Computer Engineering
Cornell University

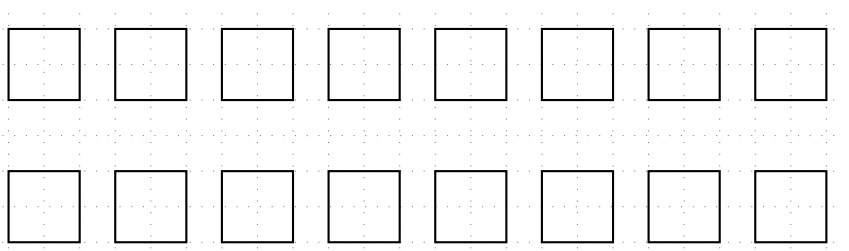
revision: 2017-10-18-17-51

- 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

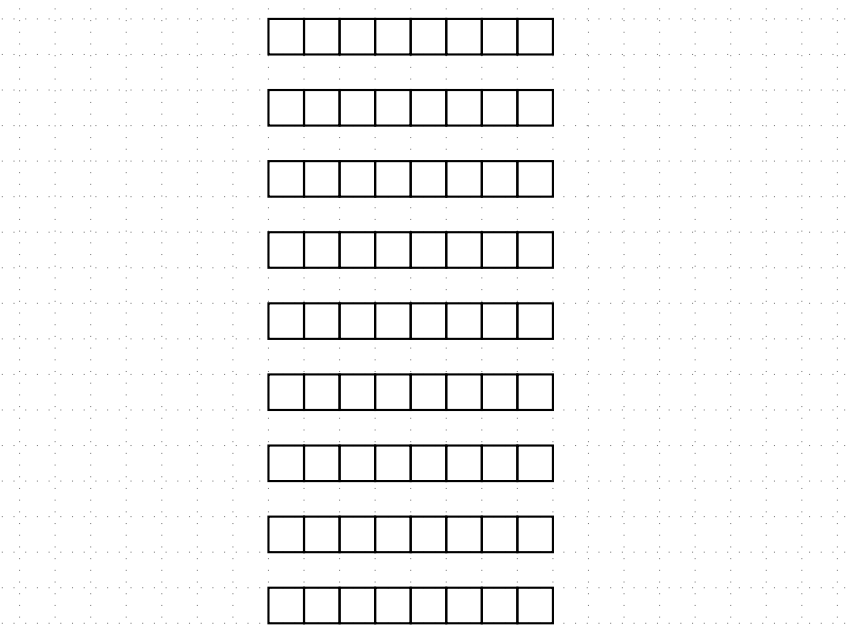
- Take elements out of input array and insert them into a sorted output array such that the output array remains sorted

1.1. Out-of-Place Insertion Sort



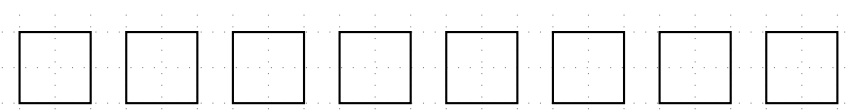
```
1 isort_op( a, size )
2   set tmp to an empty array of with size elements
3   for i in 0 to size-1 (inclusive) # iterate over input
4     temp = a[i]
5     for j in 0 to i-1 (inclusive)  # iterate over output
6       if temp < tmp[j]
7         swap( temp, tmp[j] )
8     tmp[i] = temp
9   return tmp
```

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



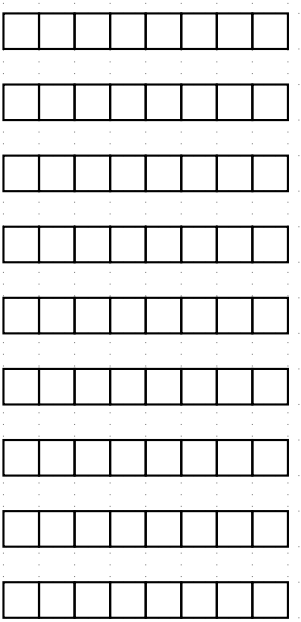
- Space complexity is $O(N)$ due to temporary array `b`
- Worst-case time complexity analysis
  - Assume C_0 is time spent in one iteration of outer loop excluding any time spent in inner loop
  - Assume C_1 is time spent in one iteration of inner loop

1.2. In-Place Insertion Sort (forward search)



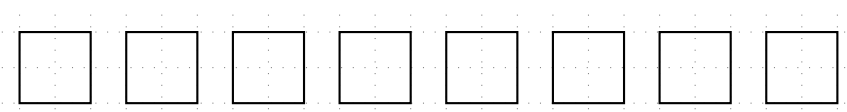
```
1 isort_ip_fwd( a, size )
2   for i in 0 to size-1 (inclusive) # iterate over input
3     temp = a[i]
4     for j in 0 to i-1 (inclusive)  # iterate over sorted
5       if temp < a[j]:              # region
6         swap( temp, a[j] )
7     a[i] = temp
8   return a
```

- Show contents of a for each iteration of outer loop



- 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)



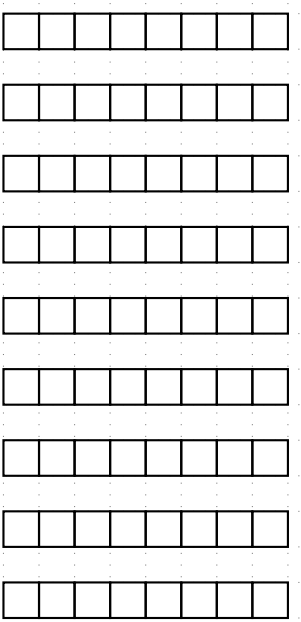
```
1 isort_ip_rev( a, size )
2   for i in 1 to size-1 (inclusive)
3     for j in i-1 to 0 (inclusive)
4       if a[j+1] < a[j]
5         swap( a[j], a[j+1] )
6       else                    # stop once new value
7         break                 # is in the right position
8   return a
```

- 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

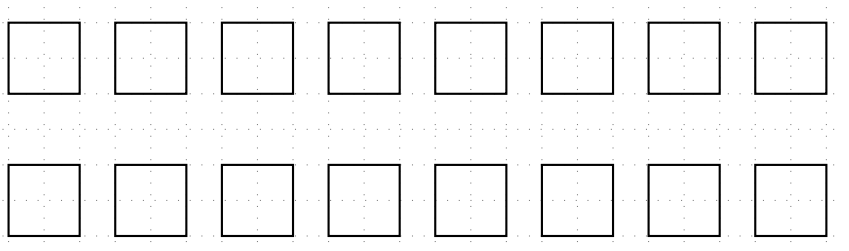
- Use in-place insertion sort
- Show contents of a for each iteration of outer loop



2. Selection Sort

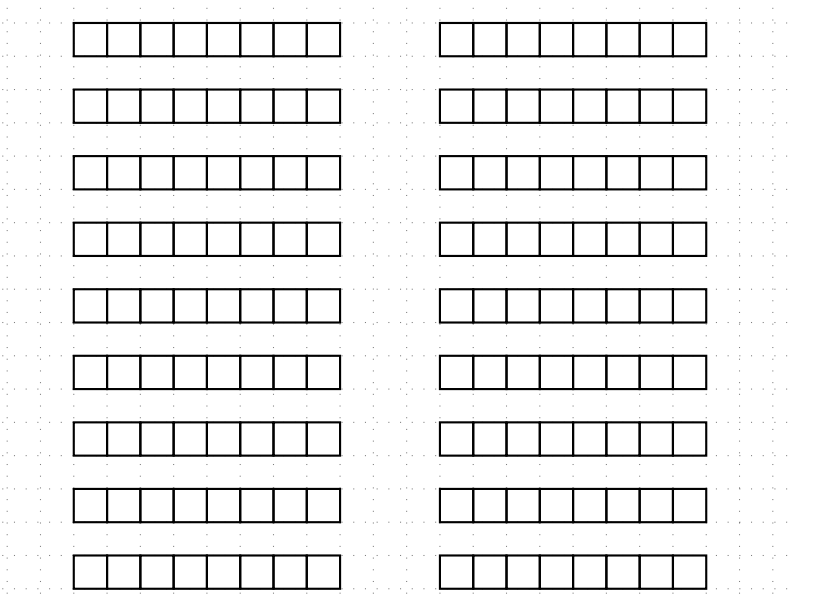
- Select minimum element from input array and add this element to the end of the sorted output array such that the output array remains sorted

2.1. Out-of-Place Selection Sort



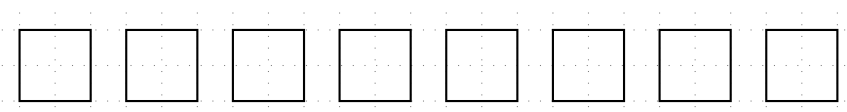
```
1  ssort_op( a, size )
2    set tmp to an empty array of with size elements
3    for i in 0 to size-1 (inclusive)
4      min_value = a[0]                  # Find the minimum
5      min_idx   = 0
6      for j in 0 to size-i-1 (inclusive)
7        if min_value > a[j]
8          min_value = a[j]
9          min_idx   = j
10     for j in min_idx to size-i-2 (inclusive)  # Remove the minimum
11       a[j] = a[j+1]
12     tmp[i] = min_value                # Put minimum in output
13   return tmp
```

- Show contents of a and b for each iteration of outer loop



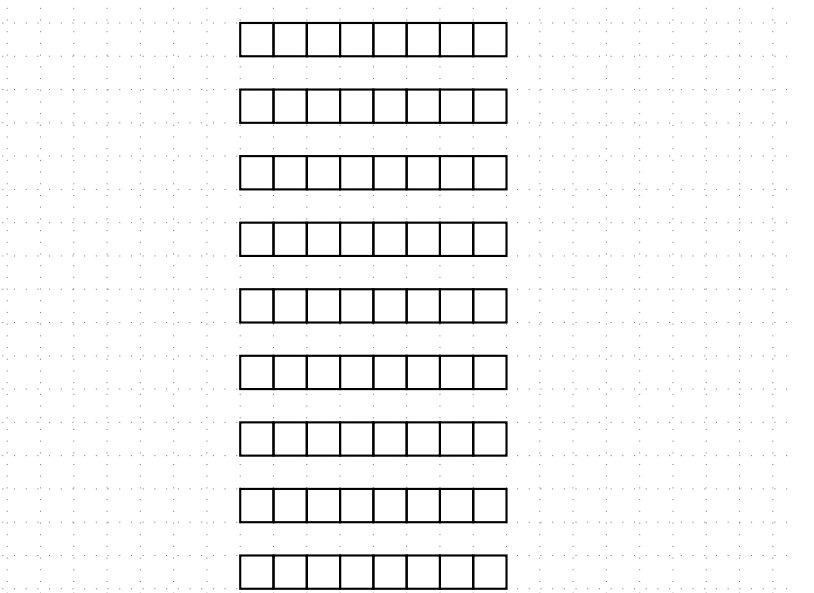
- Space complexity is $O(N)$ due to temporary array `b`
- Worst-case time complexity analysis
  - Assume C_0 is time spent in one iteration of outer loop excluding any time spent in inner loops
  - Assume C_1 is time spent in one iteration of first inner loop
  - Assume C_2 is time spent in one iteration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

2.2. In-Place Selection Sort



```
1  ssort_ip( a, size )
2    for i in 0 to size-1 (inclusive)
3
4      min_value = a[i]                  # Find the minimum
5      min_idx   = i
6      for j in i to size-1 (inclusive)
7        if min_value > a[j]
8          min_value = a[j]
9          min_idx   = j
10
11     swap( a[min_idx], a[i] )          # Swap the minimum
12
13   return a
```

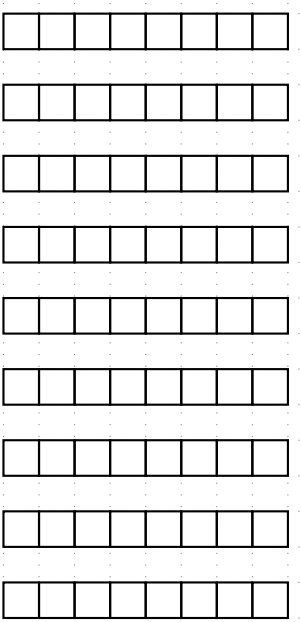
- Show contents of a for each iteration of outer loop



- 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 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

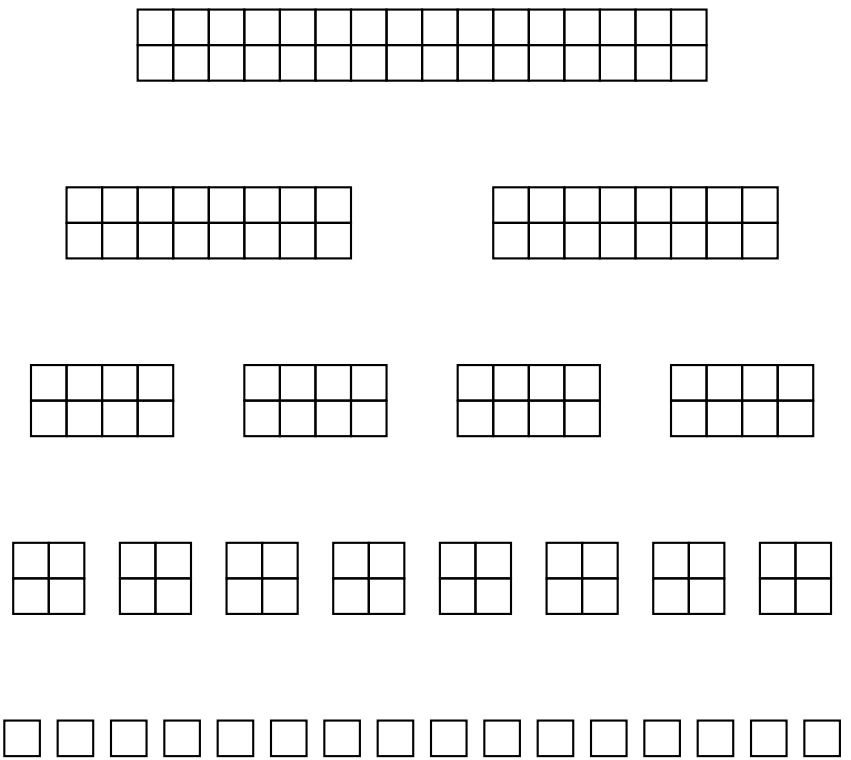


3. Merge Sort

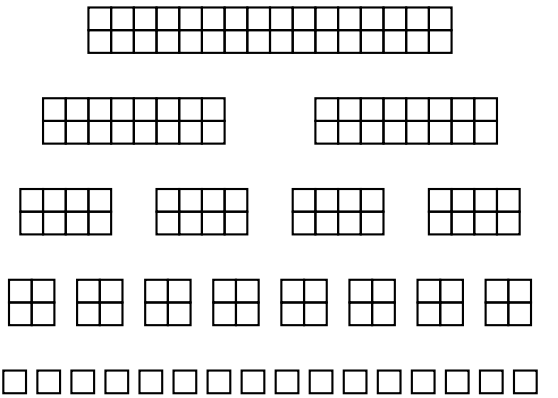
- Recursively partition input array into smaller partitions
- Base case is when one element is in a partition
- Merge two sorted partitions into a larger partition

```
1  msort_op( a, size )
2
3    if ( size == 1 )
4      return a
5
6    left_size  = size/2
7    right_size = size - left_size
8    left  = msort2_op( a[0:left_size],    left_size  )
9    right = msort2_op( a[left_size:size], right_size )
10
11   set tmp to an empty array of with size elements
12   left_idx = 0; right_idx = 0
13
14   for k in 0 to size-1 (inclusive)
15
16     # Done with left array
17     if ( left_idx == left_size )
18       tmp[k] = right[right_idx]; right_idx += 1
19
20     # Done with right array
21     elif ( right_idx == right_size )
22       tmp[k] = left[left_idx]; left_idx += 1
23
24     # Front of left is less than front of right
25     elif ( left[left_idx] < right[right_idx] )
26       tmp[k] = left[left_idx]; left_idx += 1
27
28     # Front of right is less than front of less
29     else:
30       tmp[k] = right[right_idx]; right_idx += 1
31
32   return tmp
```

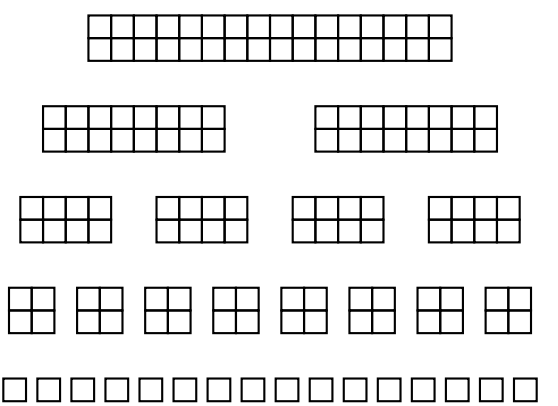
- Show contents of `a` for each recursive call
- Show contents of `tmp` for each merge



- 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



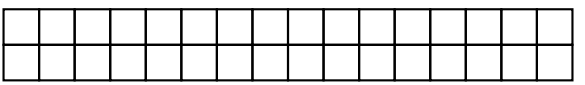
- Space complexity analysis

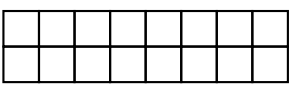


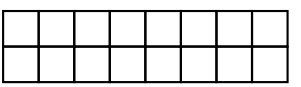
3.1. Hybrid Merge/Insertion Sort

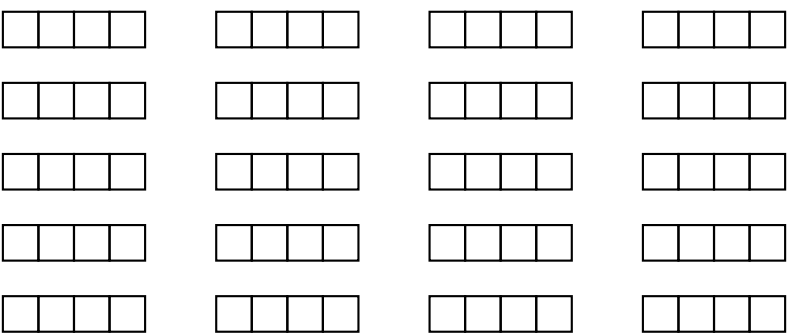
- Once array becomes small enough, use $O(N^2)$ sort

```
1 msort_hybrid( a, size )
2   if ( size <= 4 )
3     return isort_op( a, size )
4   ...
```

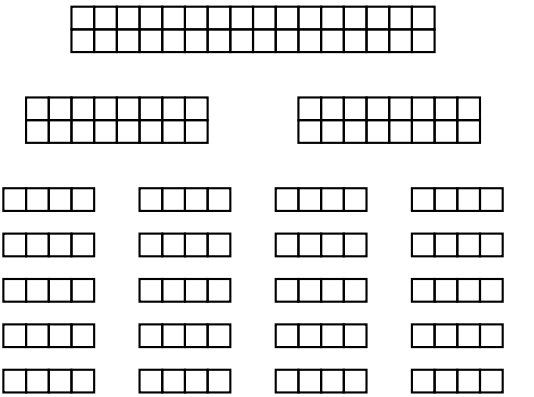








- Worst-case time complexity analysis
  - Assume C_0 is time spent in one iteration of merge loop
  - Assume C_2 is time spent in one iteration of isort inner loop



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

```
1  qsort_op( a, size )
2
3    if ( size <= 1 )
4      return a
5
6    pivot = a[size-1]
7
8    left       = []
9    left_size  = 0
10   right      = []
11   right_size = 0
12   for i in 0 to size-2 (inclusive)
13     if ( a[i] < pivot )
14       left.append(a[i])
15       left_size += 1
16     else:
17       right.append(a[i])
18       right_size += 1
19
20   left  = qsort_op( left,  left_size  )
21   right = qsort_op( right, right_size )
22   return left + [pivot] + right
```

- Time complexity analysis

4.2. In-Place Quick Sort

```
1  partition( A, lo, hi )
2    pivot = a[hi]
3    i = lo - 1
4    for j in lo to hi-1 (inclusive)
5      if ( a[j] < pivot )
6        i = i + 1
7        swap( a[j], a[i] )
8
9    if ( a[hi] < a[i+1] )
10     swap( a[hi], a[i+1] )
11
12   return i + 1
13
14 qsort_ip_h( a, lo, hi )
15   if ( lo < hi )
16     p = partition( a, lo, hi )
17     qsort_ip_h( a, lo, p - 1 )
18     qsort_ip_h( a, p + 1, hi )
19
20 qsort_ip( a, size )
21   qsort_ip_h( a, 0, size-1 )
22   return a
```

5. Radix Sort

5.1. Out-of-Place Radix Sort

5.2. Activity