ECE 2400 Computer Systems Programming Spring 2025

Topic 9: Sorting Algorithms

School of Electrical and Computer Engineering
Cornell University

revision: 2025-03-05-09-38

zyBooks logo indicates readings and coding labs in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

- 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
- 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

zyBooks The course zyBook also introduces *selection sort*, which is a very simple comparison sort, and *radix sort*, which is a non-comparison sort.

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 for correct location
- Once find correct location, insert value and push down rest of array

```
1 def sorted_insert_fwd( x, first, last, v ):
2   y = v
3   for i in first to last:
4     if y < x[i]:
5       swap( x[i], y )
6   x[last] = y
```

1.2. Sorted Insert (Reverse)

- Insert new element into sorted array such that array remains sorted
- Search array in the reverse direction
- Keep swapping until value is in the correct location

```
1 def sorted_insert_rev( x, first, last, v ):
2   x[last] = v
3   for i in last to first:
4     if x[i-1] > x[i]:
5       swap( x[i-1], x[i] )
6     else:
7       break
```

1.3. In-Place Insertion Sort

- Divide input array into sorted and unsorted partitions
- Use sorted insert to insert elements from unsorted to sorted partition
- Can use either the forward or reverse version of `sorted_insert`

```
1 def insertion_sort_ip( x, n ):
2   for i in 0 to n:
3     sorted_insert_rev( x, 0, i, x[i] )
```

zyBooks The course zyBook includes coding labs to implement insertion sort on an array of integers and an array of strings, and to parameterize insertion sort with a comparison function pointer.

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

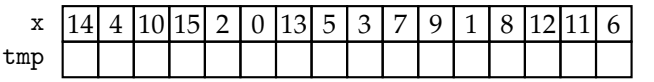
```
def merge( z, x, first0, last0, y, first1, last1 ):
  size  = ( last0 - first0 ) + ( last1 - first1 )
  assert len(z) == size

  idx0 = first0
  idx1 = first1

  for i in 0 to size:

    # done with array x
    if idx0 == last0:
      z[i] = y[idx1]
      idx1 += 1

    # done with array y
    elif idx1 == last1:
      z[i] = x[idx0]
      idx0 += 1

    # front of array a is less than front of array b
    elif x[idx0] < y[idx1]:
      z[i] = x[idx0]
      idx0 += 1

    # front of array b is less than front of array a
    else:
      z[i] = y[idx1]
      idx1 += 1
```

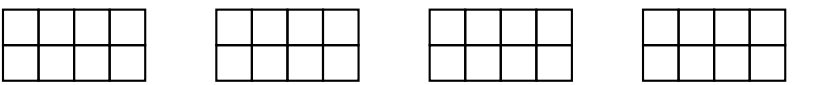
2.2. Merge Sort

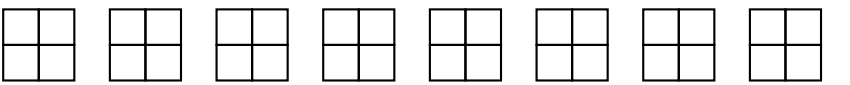
- Recursively partition input array into halves
- Base case is when a partition contains a single element
- After recursive calls return, use `merge` to merge sorted partitions

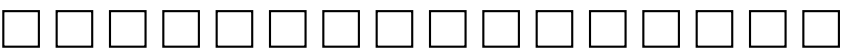
```
1  def merge_sort_h( x, first, last ):
2
3    size = last - first
4    if size == 1:
5      return
6
7    mid = ( begin + end ) / 2
8    merge_sort_h( x, first, mid  )
9    merge_sort_h( x, mid,   last )
10
11   set tmp to an empty array with size elements
12   merge( tmp, x, first, mid, x, mid, last )
13
14   # copy temporary array to input array
15   j = 0
16   for i in first to last:
17     x[i] = tmp[j]
18     j += 1
19
20 def merge_sort( x, n ):
21   merge_sort_h( x, 0, n )
```

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

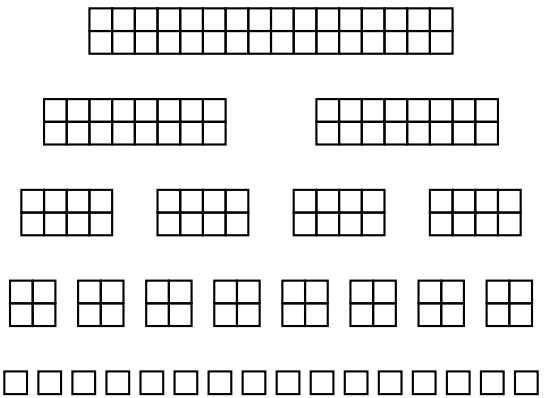




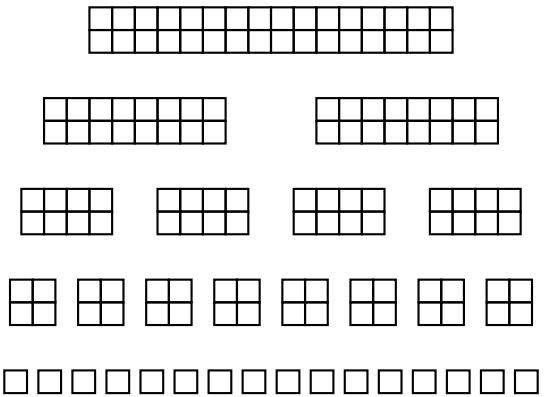




- Time complexity analysis



- Space complexity analysis



3. Quick Sort

- Use `partition` helper function to recursively partition array

3.1. Partition

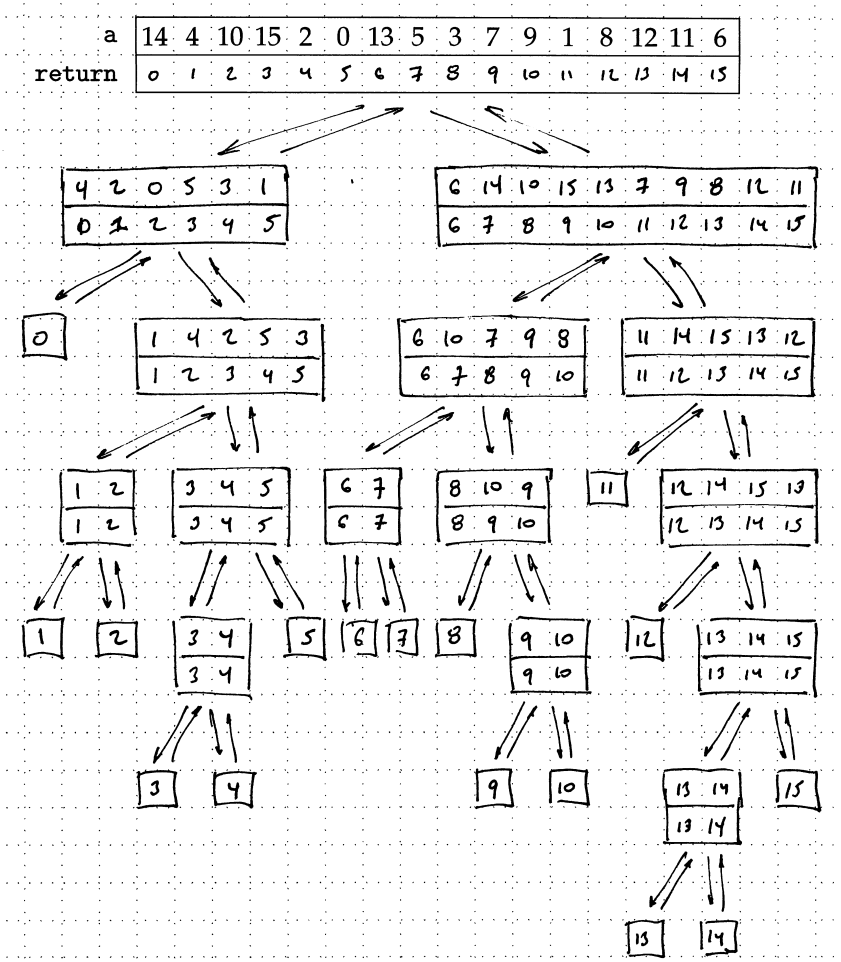
- Choose an element as the *pivot* and *partition* based on pivot
- Move all elements less than the pivot to front of the array
- Move all elements greater than the pivot to end of the array
- Pivot's final location is in between these two partitions

```
1 def partition( x, first, last ):
2   pivot = x[last-1]
3   idx = first
4   for i in first to last:
5     if x[i] <= pivot:
6       swap( x[i], x[idx] )
7       idx += 1
8   return idx-1
```

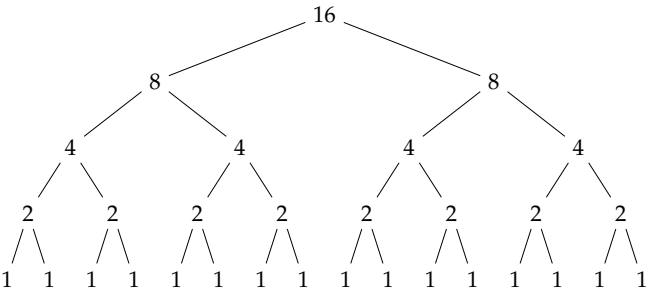
3.2. In-Place Quick Sort

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

```
1 def quick_sort_h( x, first, last ):
2   size = last - first
3   if size == 0 or size == 1:
4     return
5   p = partition( x, first, last )
6   quick_sort_h( x, first, p )
7   quick_sort_h( x, p, last  )
8
9 def quick_sort( x, n ):
10  quick_sort_h( x, 0, n )
```


a
14 4 10 15 2 0 13 5 3 7 9 1 8 12 11 6
return
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
4 2 0 5 3 1
0 1 2 3 4 5
6 14 10 15 13 7 9 8 12 11
6 7 8 9 10 11 12 13 14 15
0
1 4 2 5 3
1 2 3 4 5
6 10 7 9 8
6 7 8 9 10
11 14 15 13 12
11 12 13 14 15
1 2
1 2
3 4 5
3 4 5
6 7
6 7
8 10 9
8 9 10
11
12 14 15 13
12 13 14 15
1
2
3 4
3 4
5
6
7
8
9 10
9 10
12
13 14 15
13 14 15
3
4
9
10
13 14
13 14
15
13
14

- Best-case time complexity analysis



- Worst-case time complexity analysis

4. Comparing Sorting Algorithms

	Time Complexity			
Algorithm	**Best Case**	**Worst Case**	**Avg Case**	**Comparison Sort?**
insertion (fwd)				
insertion (rev)				
selection				
merge				
quick				
radix				

	Space Complexity	
Algorithm	**Avg Case Stack**	**Avg Case Aux Heap**
insertion (fwd)		
insertion (rev)		
selection		
merge		
quick		
radix		