ECE 2400 Computer Systems Programming
Fall 2021

Topic 9: Sorting Algorithms

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-29-22-43

zyBooks The zyBooks logo is used to indicate additional material included 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.

Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

- 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

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

Reminder about summation of finite series

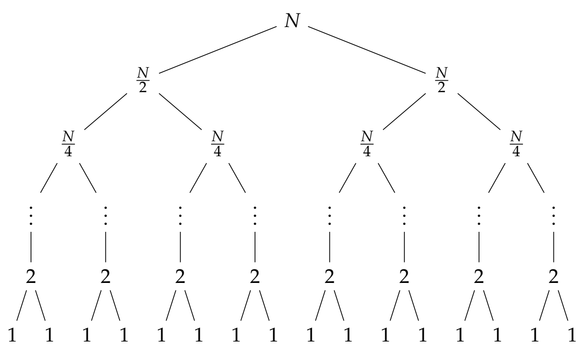
$$\sum_{i=0}^{N} i = 0+1+2 \ldots N-1+N = \frac{N+1}{2}(N) = \frac{1}{2}N^2 + \frac{1}{2}N$$

$$\sum_{i=0}^{N-1} i = 0+1+2 \ldots N-2+N-1 = \frac{N}{2}(N-1) = \frac{1}{2}N^2 - \frac{1}{2}N$$

$$\sum_{i=1}^{N} i = 1+2+3 \ldots N-1+N = \frac{N}{2}(N+1) = \frac{1}{2}N^2 + \frac{1}{2}N$$

$$\sum_{i=1}^{N-1} i = 1+2+3 \ldots N-2+N-1 = \frac{N-1}{2}(N) = \frac{1}{2}N^2 - \frac{1}{2}N$$

Reminder about binary trees



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

```
def sorted_insert_fwd( a, begin, end, v ):
  x = v
  for i in begin to end:
    if x < a[i]:
      swap( a[i], x )
  a[end] = x
```

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

```
def sorted_insert_rev( a, begin, end, v ):
  a[end] = v
  for i in begin to end:
    # ridx is the reverse index
    ridx = begin + end - i - 1
    if a[ridx+1] < a[ridx]:
      swap( a[ridx+1], a[ridx] )
    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
- Can use either the forward or reverse version of `sorted_insert`

```
def insertion_sort_op( a, size ):

  set tmp to an empty array with size elements
  for i in 0 to size:
    sorted_insert_fwd( tmp, 0, i, a[i] )

  for i in 0 to 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
- Can use either the forward or reverse version of `sorted_insert`

```
def insertion_sort_ip( a, size ):
  for i in 0 to size:
    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

```
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
```

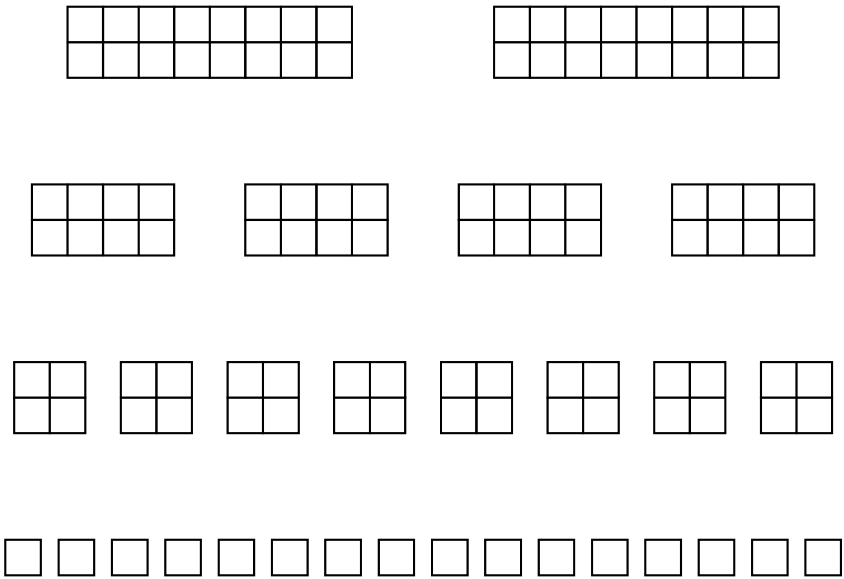
2.2. Merge Sort

- 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

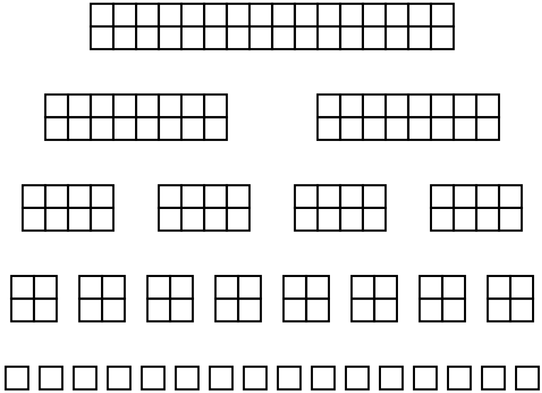
```
def merge_sort_h( a, begin, end ):

  size = end - begin
  if size == 1:
    return

  mid = ( begin + end ) / 2
  merge_sort_h( a, begin, mid )
  merge_sort_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 merge_sort( a, size ):
  merge_sort_h( a, 0, size )
```

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

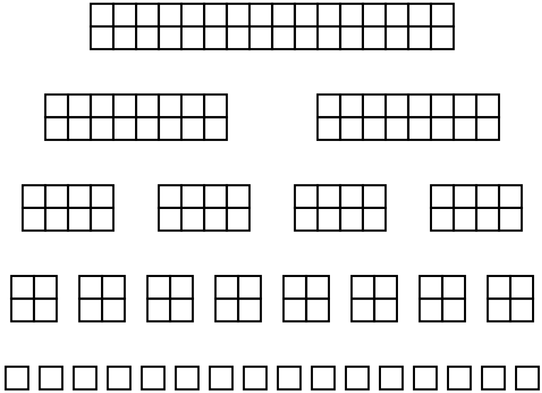
`a`	14	4	10	15	2	0	13	5	3	7	9	1	8	12	11	6
`tmp`																



- Time complexity analysis



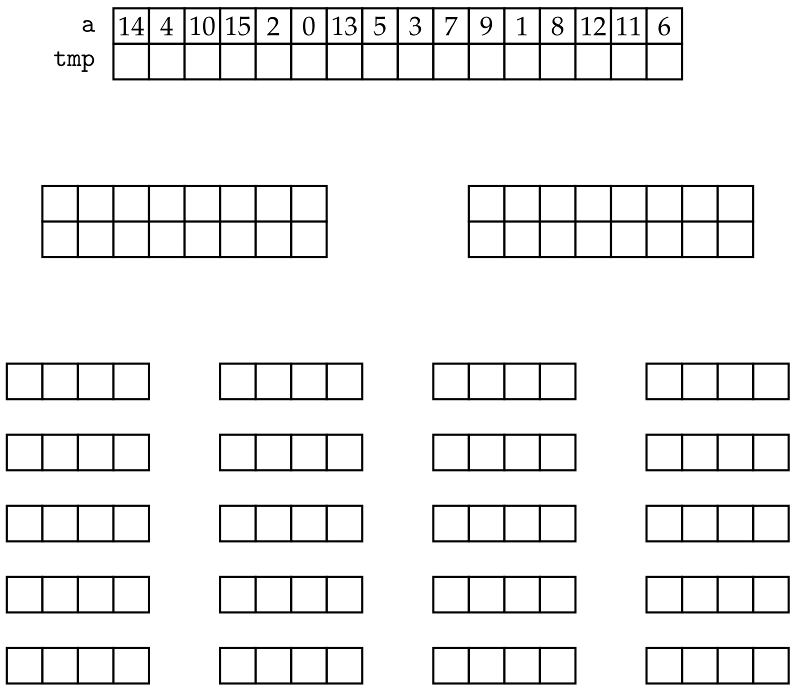
- Space complexity analysis



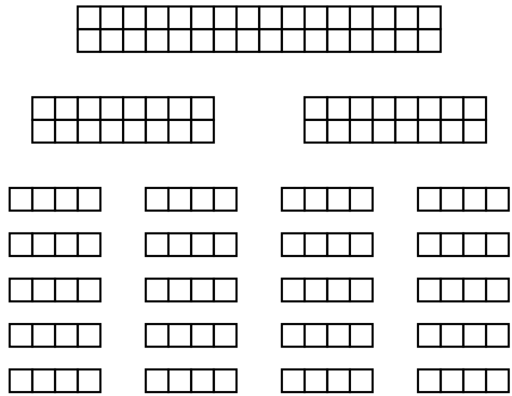
2.3. Hybrid Merge/Insertion Sort

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

```
merge_sort_hybrid_h( a, begin, end )
  size = end - begin
  if size <= 4:
    return insertion_sort_op( a, begin, end )
  ...
```



- Worst-case time complexity analysis



3. Quick Sort

- Use `partition` helper function to recursively partition array

3.1. Partition

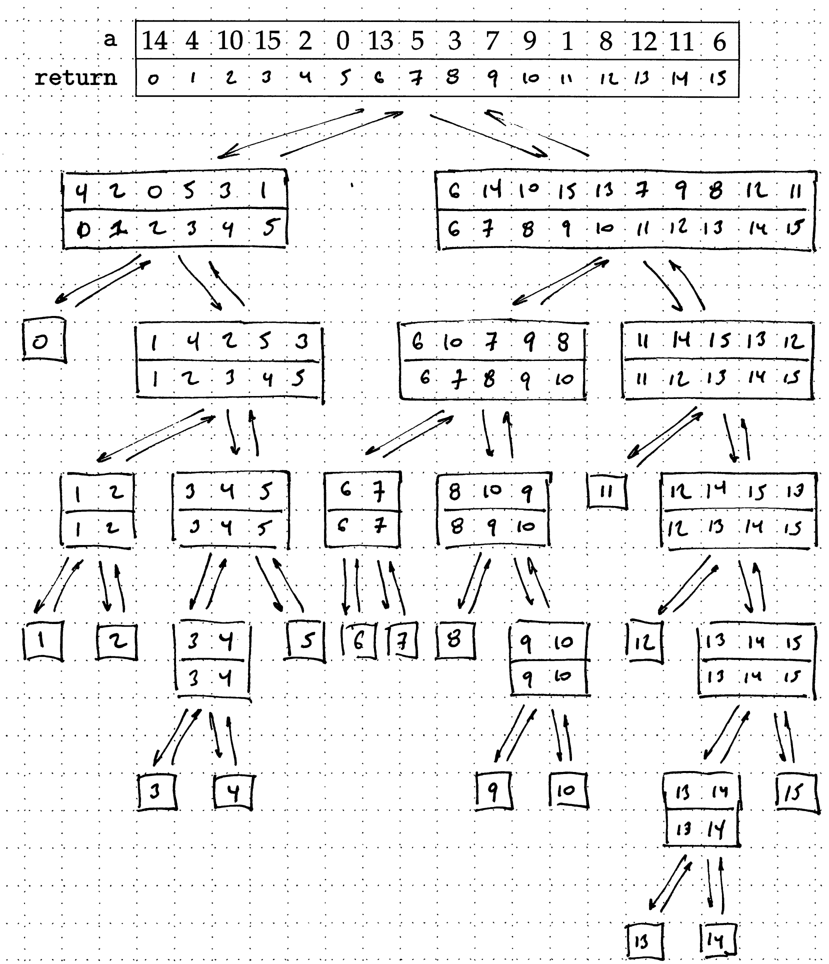
- Choose an element as the *pivot* and *partition* based on pivot
- Move all elements less than the pivot to front of the array
- Move all elements greater than the pivot to end of the array
- Pivot's final location is in between these two partitions

```
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
```

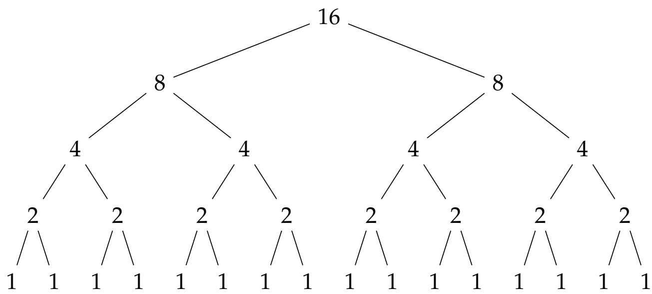
3.2. In-Place Quick Sort

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

```
def quick_sort_h( a, begin, end ):
  size = end - begin
  if size == 0 or size == 1:
    return
  p = partition( a, begin, end )
  quick_sort_h( a, begin, p   )
  quick_sort_h( a, p, end )

def quick_sort( a, size ):
  quick_sort_h( a, 0, size )
```


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

- Best-case time complexity analysis



- Worst-case time complexity analysis

4. Comparing Sorting Algorithms

	Time Complexity			Space Complexity		
	Best	Worst	Avg	Avg Case		
Algorithm	**Case**	**Case**	**Case**	**Stack**	**Heap**	**Cmp?**
insertion (fwd)						
insertion (rev)						
selection						
merge						
quick						
radix						