

CSCE 3110

Data Structures and Algorithms

Sorting

Reading: Weiss, chap. 7

Content

- Comparison-based sorting algorithms:
 - Insertion sort
 - Selection sort
 - Heapsort
 - Merge sort
 - Quick sort
- Integer sorting:
 - Bucket sort
 - Radix sort

Sorting

- Given a set (container) of n elements:
 - e.g., array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal: Arrange the elements in a certain order
 - e.g., ascending/descending orders

Sorting

- Given a set (container) of n elements:
 - e.g., array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal: Arrange the elements in a certain order
 - e.g., ascending/descending orders

Before sorting: 1 23 2 56 9 8 10 100

Sorting

- Given a set (container) of n elements:
 - e.g., array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal: Arrange the elements in a certain order
 - e.g., ascending/descending orders

Before sorting: 1 23 2 56 9 8 10 100

After sorting: 1 2 8 9 10 23 56 100

Importance of Sorting

- Why don't CS profs ever stop talking about sorting?
 - Computers spend a lot of time sorting, historically 25% on mainframes
 - Sorting is the best studied problem in computer science, with many different algorithms known
 - Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds

Stable Sorting

- A property of sorting
- If a sort guarantees the **relative order of equal items stays the same**, then it is a stable sort

Before sorting: $7_1, 6, 7_2, 5, 1, 2, 7_3, -5$ (subscripts added for clarity)

After sorting: $-5, 1, 2, 5, 6, 7_1, 7_2, 7_3$ (result of stable sort)

In Place Sorting

- Sorting of a data structure does **not** require any external data structure for storing the intermediate steps
- The amount of extra space required to sort the data is **constant** with the input size

Insertion Sort

- Insertion sort: orders a list of values by repetitively inserting a particular value into a sorted subset of the list
- More specifically:
 - 1) consider the first item to be a sorted sub list of length 1
 - 2) insert the second item into the sorted sub list, shifting the first item if needed
 - 3) insert the third item into the sorted sub list, shifting the other items as needed
 - 4) repeat until all values have been inserted into their proper positions

Insertion Sort

```
template <class Item>
void insertion_sort(Item data[ ], size_t n) {
    size_t i, j;
    Item temp;

    if(n < 2) return; // nothing to sort!!

    for(i = 1; i < n; ++i)
    {
        // take next item at front of unsorted part of array
        // and insert it in appropriate location in sorted part of array
        temp = data[i];
        for(j = i; data[j-1] > temp and j > 0; --j)
            data[j] = data[j-1]; // shift element forward

        data[j] = temp;
    }
}
```

Insertion Sort: Example

- Sorting: 3, 9, 6, 1, 2 using insertion sort

Insertion Sort: Example

- Sorting: 3, 9, 6, 1, 2 using insertion sort

3 is sorted.
Shift nothing. Insert 9.



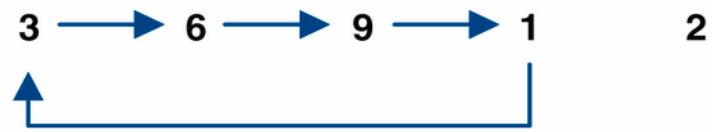
3 9 6 1 2

3 and 9 are sorted.
Shift 9 to the right. Insert 6.




3 9 6 1 2

3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.



3 6 9 1 2

1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.



1 3 6 9 2

1 2 3 6 9

Insertion Sort Time Analysis

- In O -notation, what is:
 - Worst case running time for n items?
 - Best case running time for n items?

Insertion Sort Time Analysis

- In O -notation, what is:
 - Worst case running time for n items?
 - Best case running time for n items?
- Steps of algorithm:

for $i = 1$ to $n-1$

 take next key from unsorted part of array

 insert in appropriate location in sorted part of array:

 for $j = i$ down to 0,

 shift sorted elements to the right if $\text{key} > \text{key}[i]$

 increase size of sorted array by 1

Outer loop:
 $O(n)$

Inner loop:
 $O(n)$

Selection Sort

- Basic idea:
 - 1) Find the smallest element in the array
 - 2) Exchange it with the element in the first position
 - 3) Find the second smallest element and exchange it with the element in the second position
 - 4) Continue until the array is sorted

Selection Sort

SELECTION-SORT(A):

$n \leftarrow \text{length}[A]$

 for $j \leftarrow 1$ to $n - 1$

 do $\text{smallest} \leftarrow j$

 for $i \leftarrow j + 1$ to n

 do if $A[i] < A[\text{smallest}]$

 then $\text{smallest} \leftarrow i$

 exchange $A[j] \leftrightarrow A[\text{smallest}]$

Selection Sort: Example

- Sorting: 8, 4, 6, 9, 2, 3, 1 using selection sort

Selection Sort: Example

- Sorting: 8, 4, 6, 9, 2, 3, 1 using insertion sort

8	4	6	9	2	3	1
---	---	---	---	---	---	---

1	2	3	4	9	6	8
---	---	---	---	---	---	---

1	4	6	9	2	3	8
---	---	---	---	---	---	---

1	2	3	4	6	9	8
---	---	---	---	---	---	---

1	2	6	9	4	3	8
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---

1	2	3	9	4	6	8
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---

Selection Sort Time Analysis

- It's clearly quadratic:
 - The first pass, we search through exactly $n - 1$ elements (no difference between average-case and worst-case), then swap (constant time)
 - Second time, $n - 2$ elements, then $n - 3$, etc.

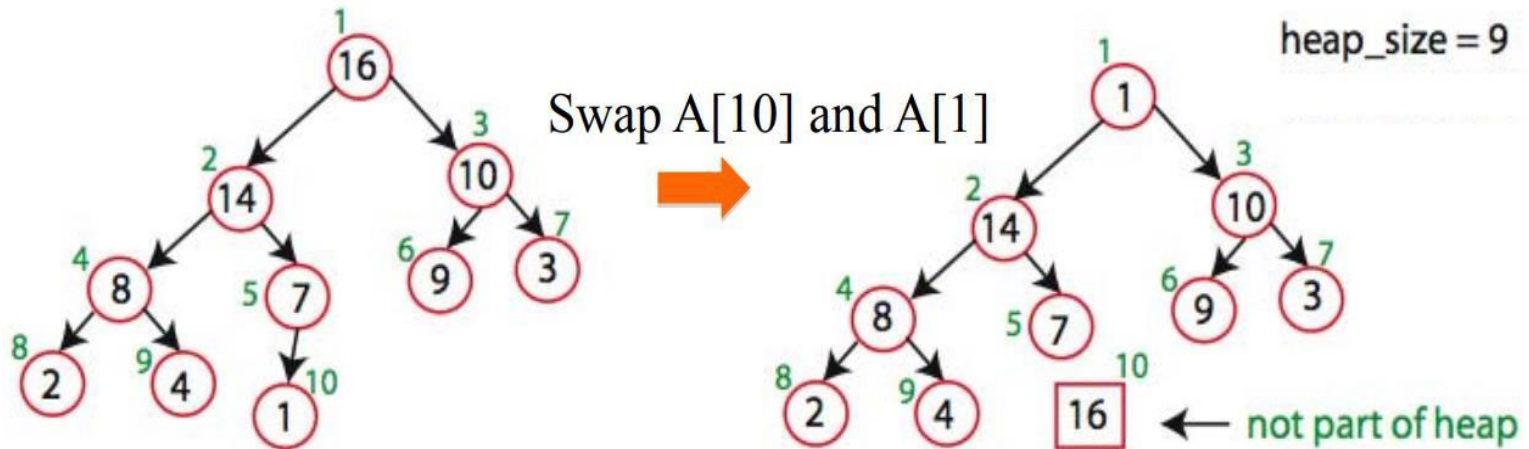
We get the arithmetic sum $(n-1) + (n-2) + (n-3) + \dots + 1 = \Theta(n^2)$

Heapsort

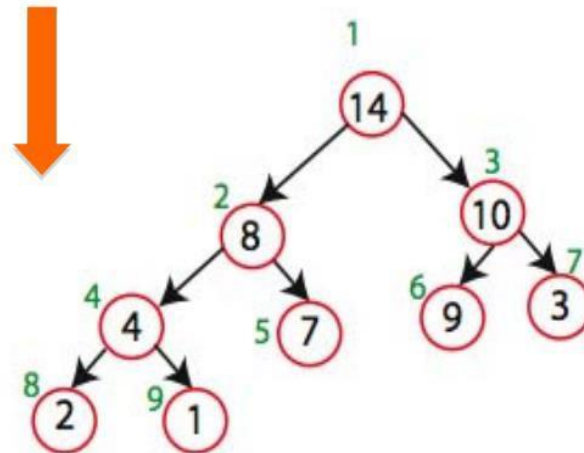
Sorting Strategy:

- Build Max Heap from unordered array;
- Find maximum element $A[1]$;
- Swap elements $A[n]$ and $A[1]$: now max element is at the end of the array!
- Discard node n from heap (by decrementing heap-size variable)
- New root may violate max heap property, but its children are max heaps. Run `max_heapify` to fix this.
- Go to Step 2 unless heap is empty.

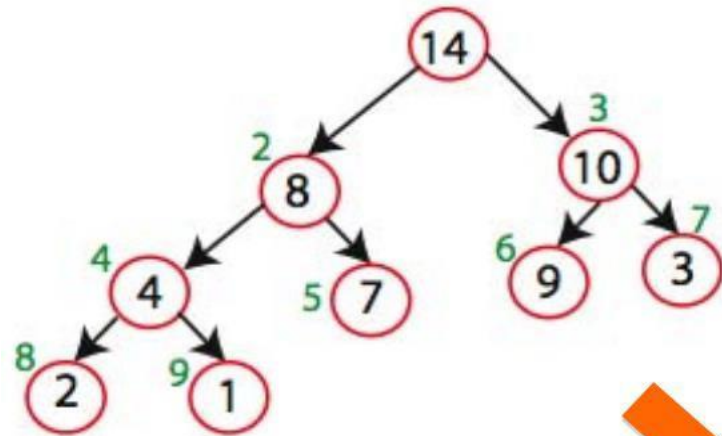
Heapsort Demo



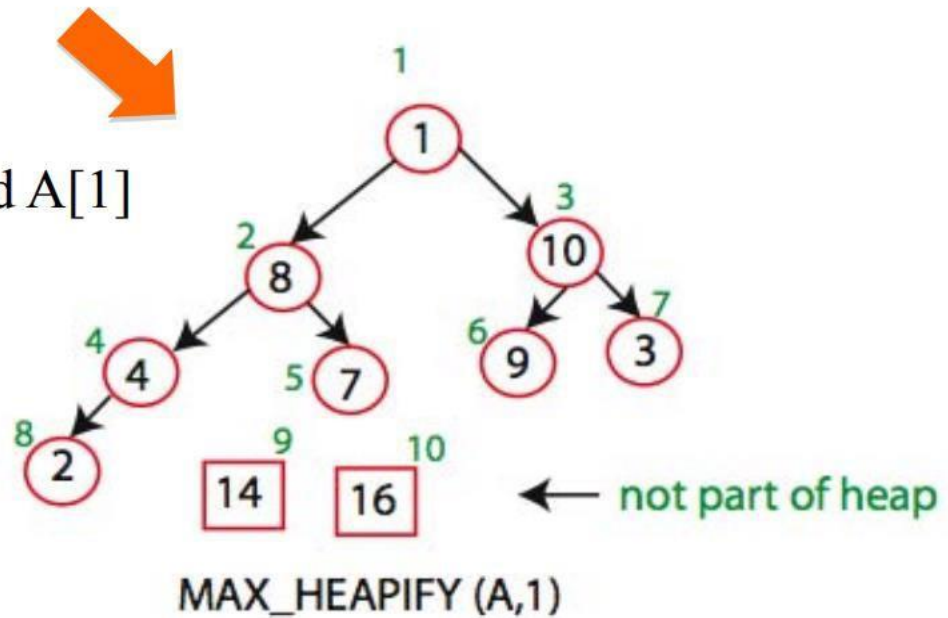
Max_heapify(A,1)



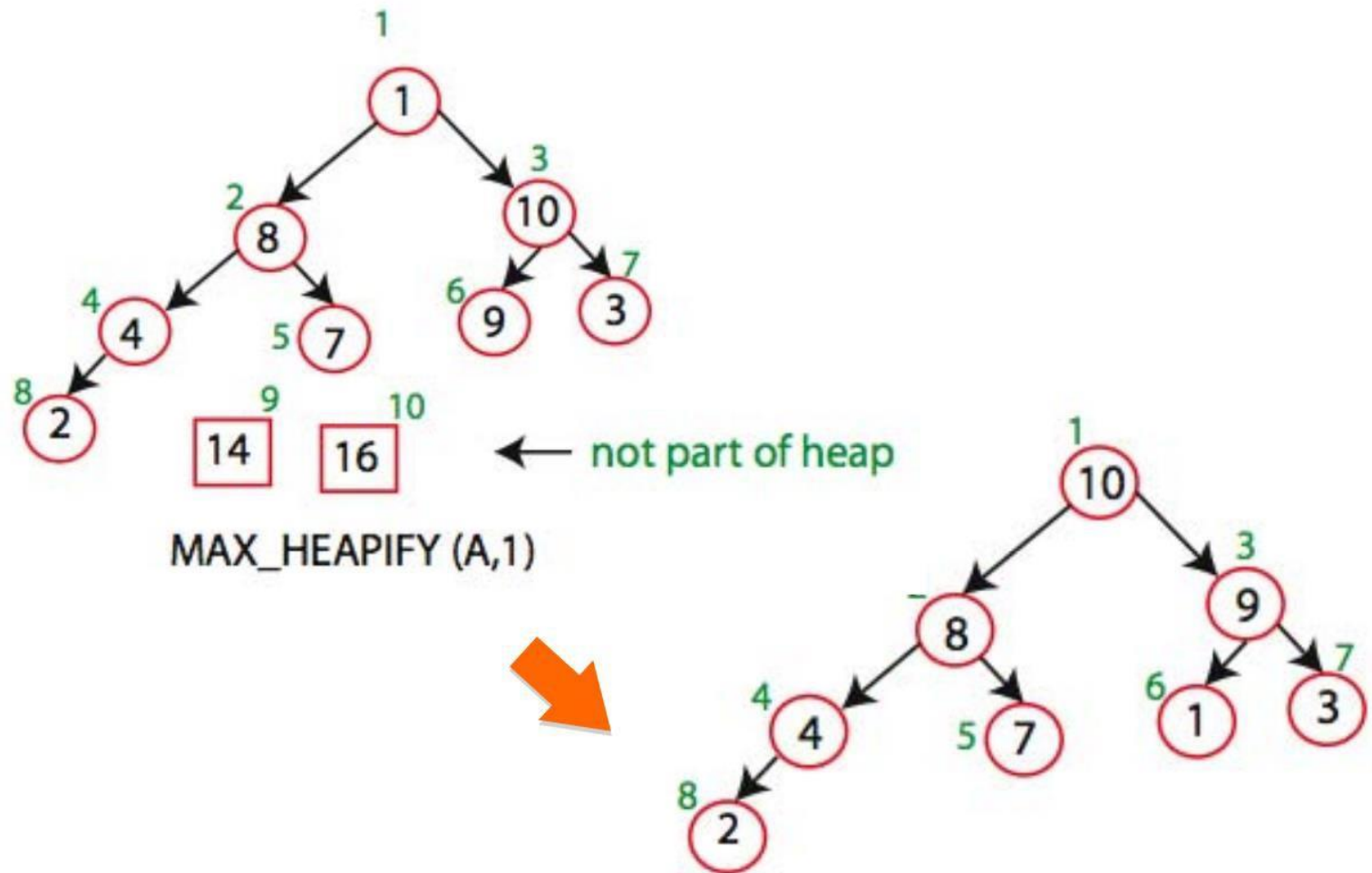
Heapsort Demo



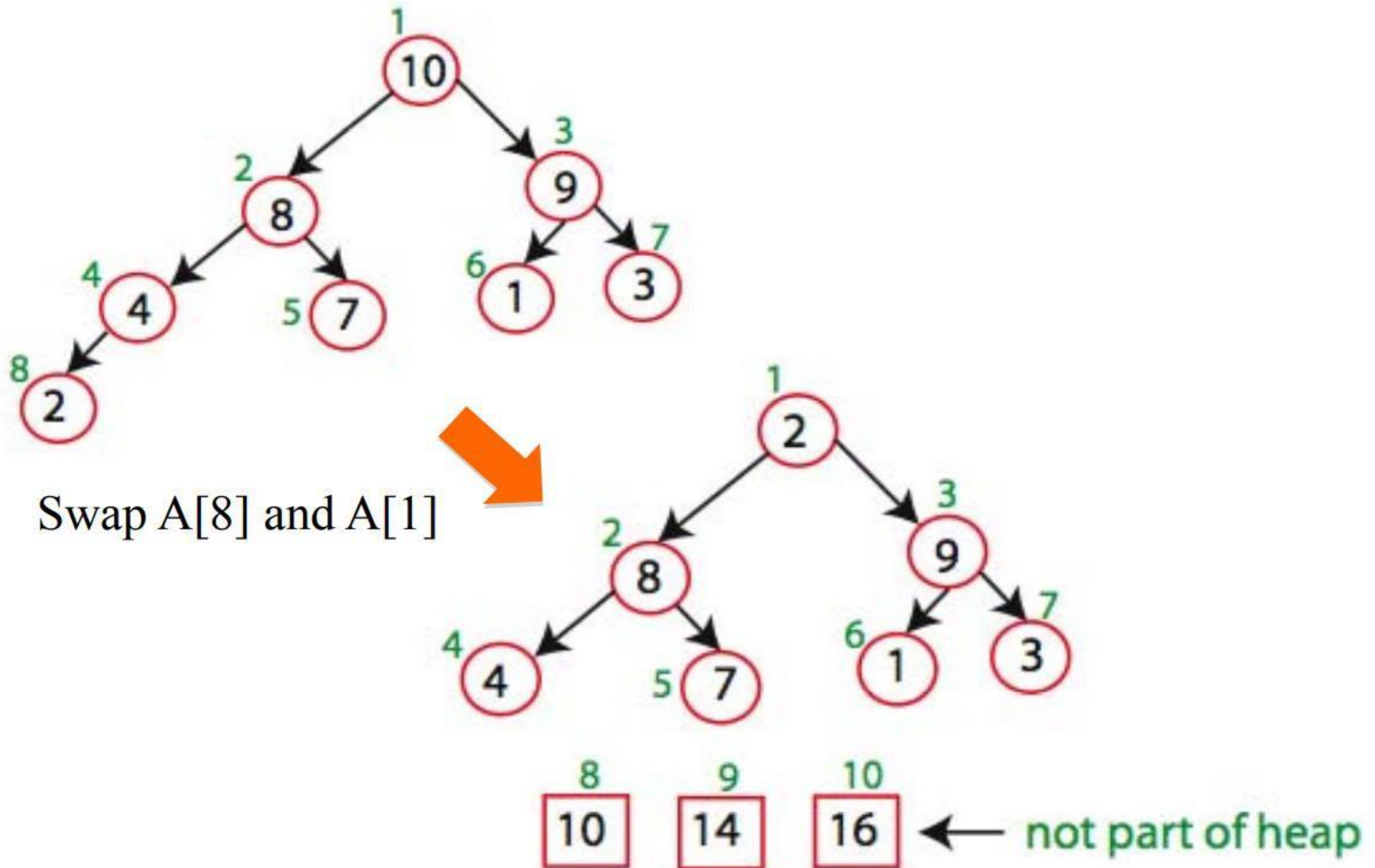
Swap A[9] and A[1]



Heapsort Demo



Heapsort Demo



Heapsort Time Analysis

- After n iterations the Heap is empty
- Every iteration involves a swap and a max_heapify operation;
- Hence it takes $O(n \log n)$ time overall

Divide and Conquer

- Very important technique in algorithm design
 - Divide problem into smaller parts
 - Independently solve the simpler parts
 - Think recursion
 - Or potential parallelism
 - Combine solution of parts to produce overall solution
- Two great sorting methods are fundamentally Divide-and-Conquer:
 - Merge Sort
 - Quick Sort

Merge Sort

- So simple — really, soooooo simple
- Split the array into two halves
 - Sort (using the same merge sort) the first half
 - Then, sort the second half
 - Then, merge them (since they are ordered sequence, it should be easy to merge them in linear time into a single ordered sequence)

Merge Sort

- Merging two sorted sequences into a single sorted sequence (in linear time):

How to merge?

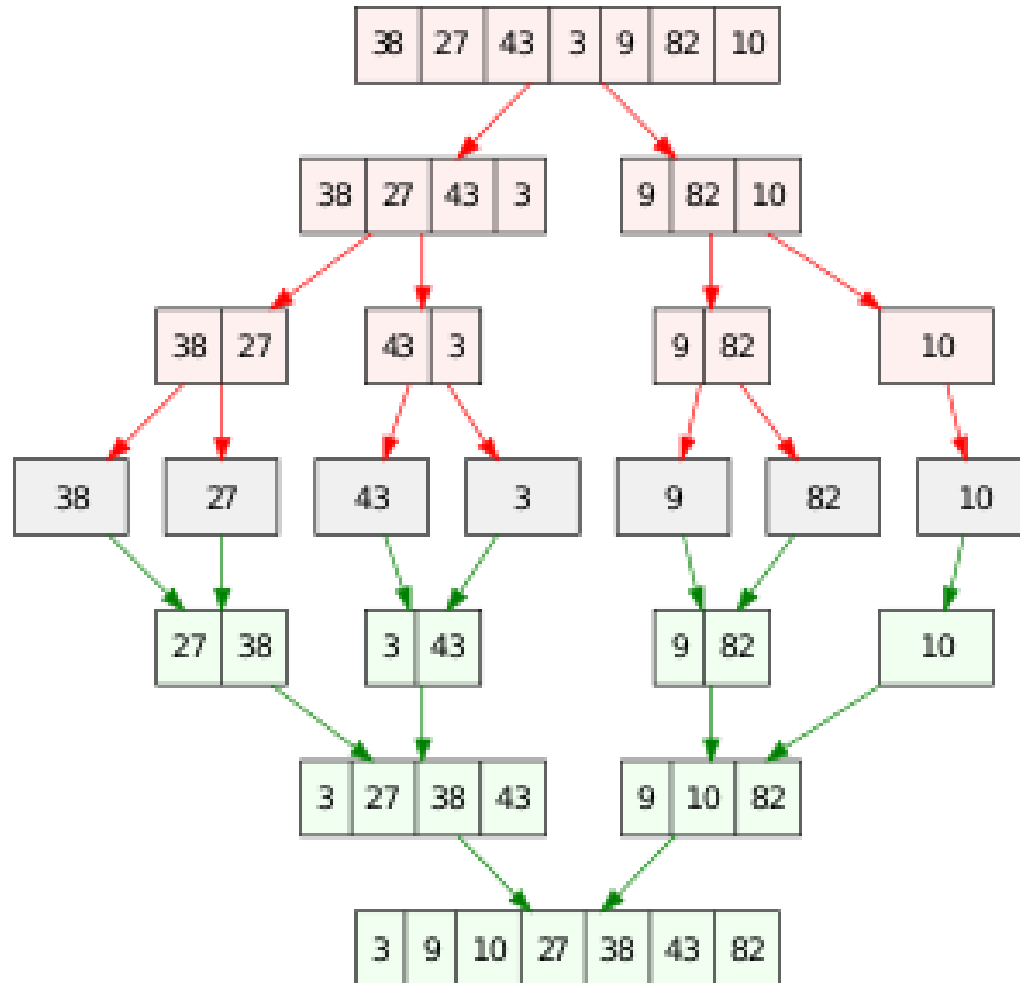
- Example:
 - Sequence A: 11, 23, 40, 57, 78, 93
 - Sequence B: 5, 9, 35, 36, 39, 63

Merge Sort

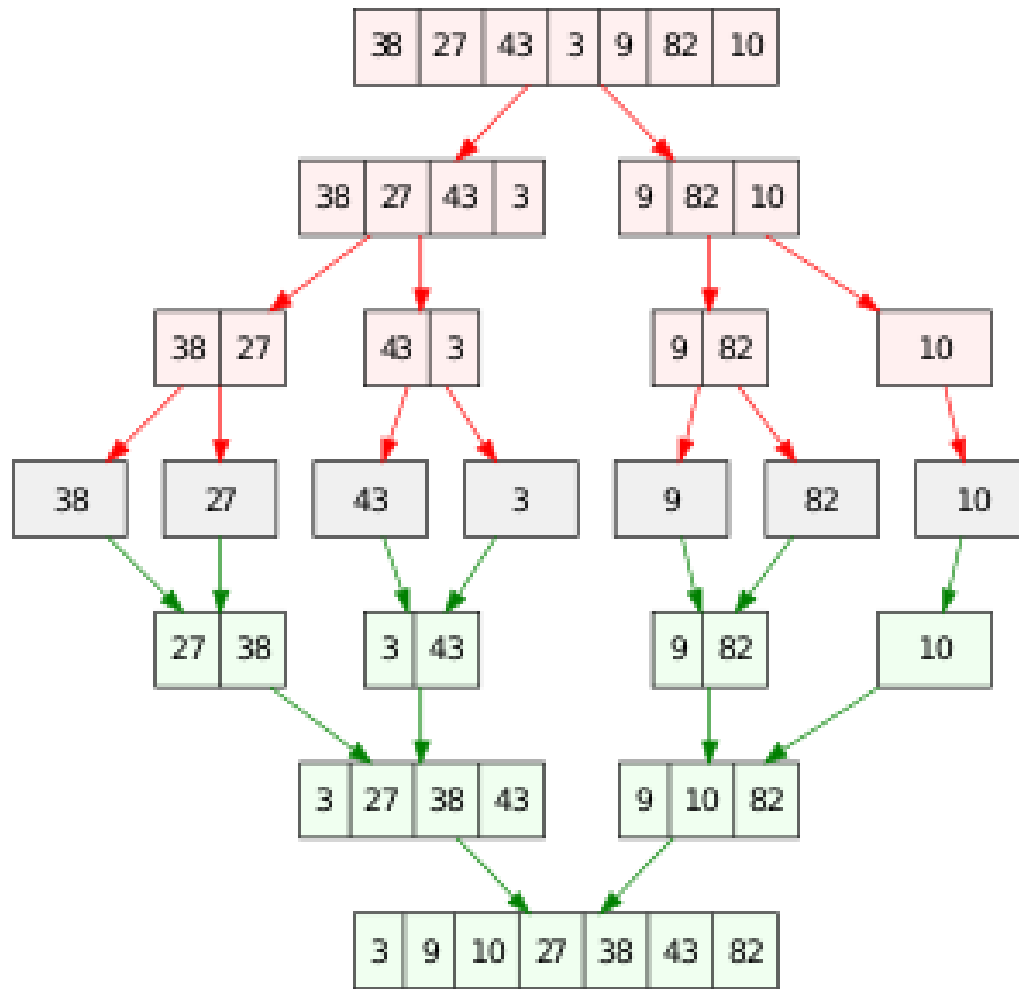
- Sorting 38, 27, 43, 3, 9, 82, 10 using merging sort

Merge Sort

- Sorting 38, 27, 43, 3, 9, 82, 10 using merging sort



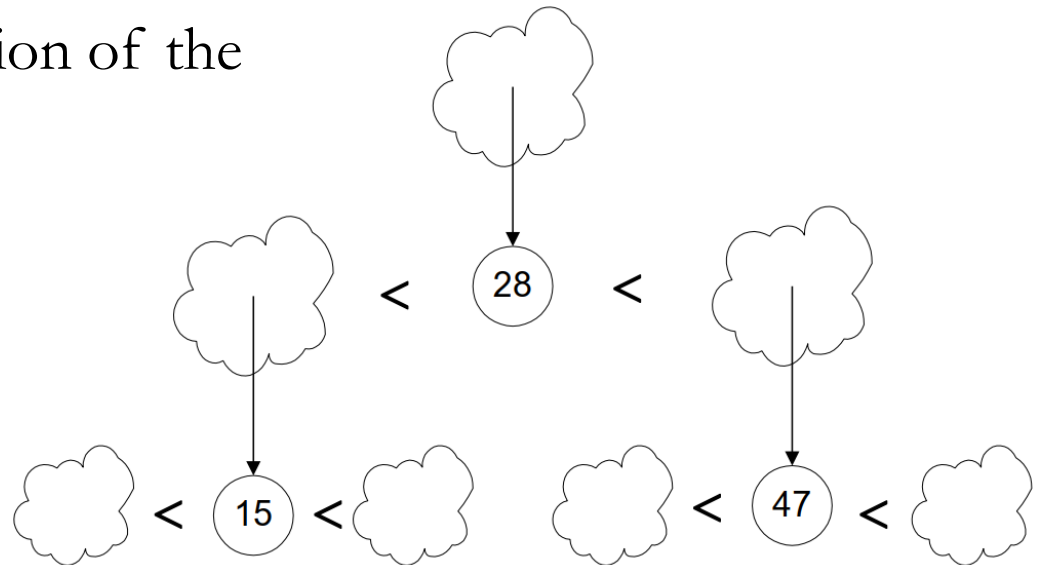
Merge Sort Time Analysis



- Recurrency relation of merge sorting?

Quick Sort

- Pick a “pivot”
- Divide list into two lists
 - One less-than-or-equal-to pivot value
 - One greater than pivot value
- Sort each sub-problem recursively
- Answer is the concatenation of the two solutions



Quick Sort Pseudocode

Sort(A)

 Quicksort(A,1,n)

 Quicksort(A, low, high)

 if (low < high)


 pivot-location = Partition(A,low,high)

 Quicksort(A,low, pivot-location - 1)

 Quicksort(A, pivot-location+1, high)

 Partition(A,low,high)

First element
is the pivot



 pivot = A[low]

 leftwall = low

 for $i = \text{low}+1$ to high

 if ($A[i] < \text{pivot}$) then

 leftwall = leftwall+1

 swap($A[i], A[\text{leftwall}]$)

 swap($A[\text{low}], A[\text{leftwall}]$)

Quick Sort: Example

- Sorting 38, 27, 43, 3, 9, 82, 10 using merging sort

Quick Sort: Example

- Sorting 38, 27, 43, 3, 9, 82, 10 using merging sort

Pick pivot:

7	2	8	3	5	9	6
---	---	---	---	---	---	---

Partition
with cursors

7	2	8	3	5	9	6
---	---	---	---	---	---	---



2 goes to
less-than

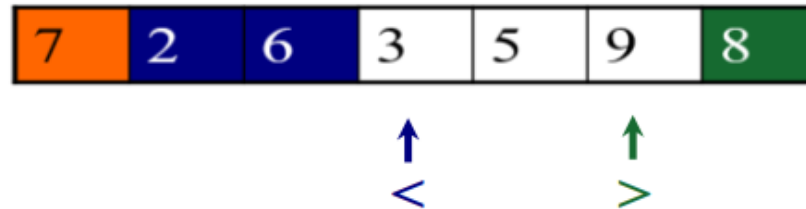
7	2	8	3	5	9	6
---	---	---	---	---	---	---



Quick Sort: Example

- Sorting 38, 27, 43, 3, 9, 82, 10 using merging sort

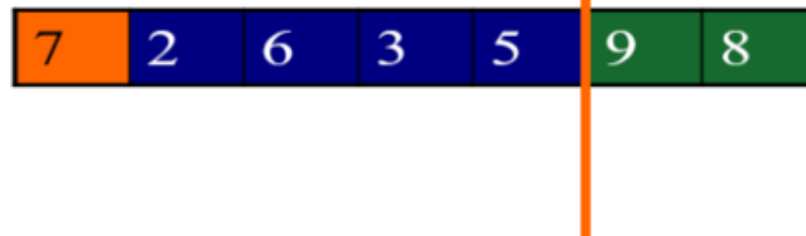
6, 8 swap
less/greater-than



3, 5 less-than
9 greater-than



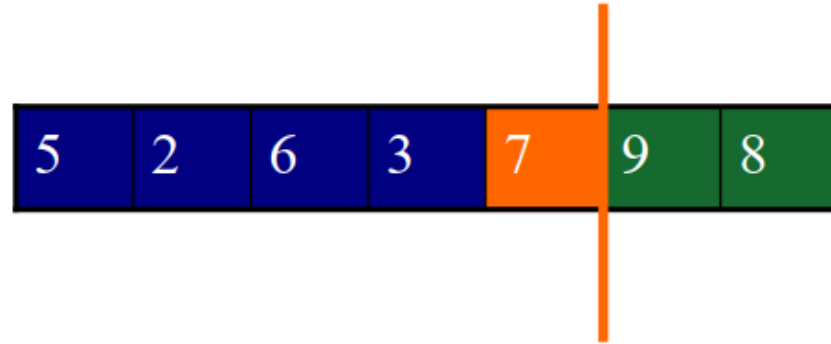
Partition done.



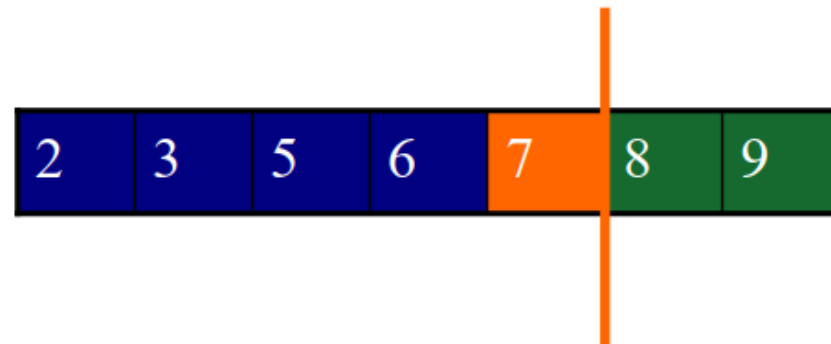
Quick Sort: Example

- Sorting 38, 27, 43, 3, 9, 82, 10 using merging sort

Put pivot
into final
position.



Recursively
sort each side.



Quick Sort Time Analysis

- Picking pivot: constant time
- Partitioning: linear time
- Recursion: time for sorting left partition (say of size i) + time for right (size $N - i - 1$) + time to combine solutions

$$T(N) = T(i) + T(N - i - 1) + cN$$

where i is the number of elements smaller than the pivot

Quick Sort Worst Case

- Quick Sort is fast in practice but has $\theta(N^2)$ worst-case complexity
- Pivot is always smallest element, so $i = 0$:

$$\begin{aligned}T(N) &= T(i) + T(N - i - 1) + cN \\&= T(N - 1) + cN \\&= T(N - 2) + c(N - 1) + cN \\&= T(N - k) + c \sum_{i=0}^{k-1} (N - i) \\&= O(N^2)\end{aligned}$$

Quick Sort Best Case

- Pivot is always middle element

$$T(N) = T(i) + T(N - i - 1) + cN$$

$$T(N) = 2T\left(\frac{N-1}{2}\right) + cN$$

$$< 2T\left(\frac{N}{2}\right) + cN$$

$$< 4T\left(\frac{N}{4}\right) + c\left(2\frac{N}{2} + N\right)$$

$$< 8T\left(\frac{N}{8}\right) + cN(1 + 1 + 1)$$

$$< kT\left(\frac{N}{k}\right) + cN \log k = O(N \log N)$$



Dealing with Slow Quick Sort

- Randomly choose pivot
 - Good theoretically and practically, but call to random number generator can be expensive
- Pick pivot cleverly
 - “Median-of-3” rule takes Median(first, middle, last element elements) as pivot. Also works well
 - e.g., Swap Median with either first or last element, then partition as usual

Integer sorting

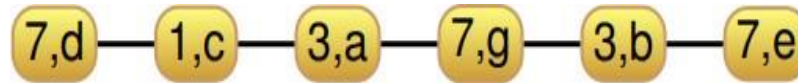
- We've already discussed that (under some more or less standard assumptions), no sort algorithm can have a run time better than $n \log n$
- However, there are algorithms that run in linear time (huh???)

Bucket Sort

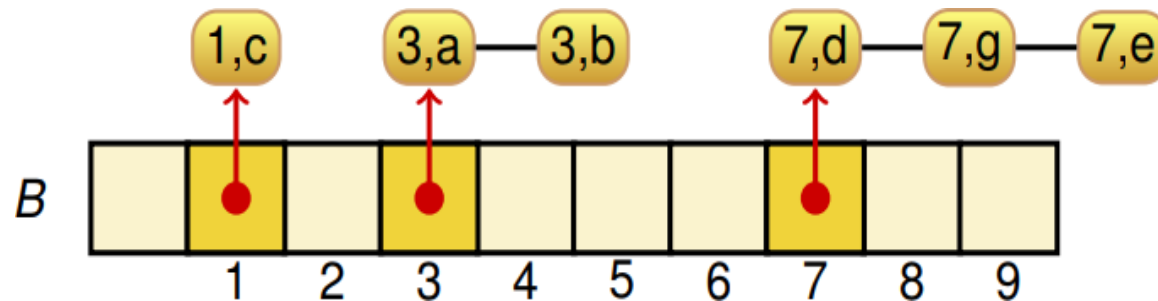
- If all keys are $0 \dots K - 1$
- Have an array of K buckets (linked lists)
- Put keys into correct bucket of array
 - linear time!
- Bucket Sort is a stable sorting algorithm:
 - Items in input with the same key end up in the same order as when they began
- Impractical for large K

Bucket Sort: Example

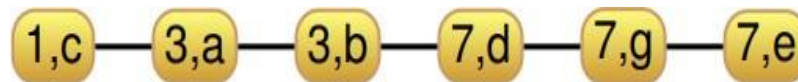
- Key range $[0, 9]$



- Phase 1: filling the buckets



- Phase 2: emptying the buckets into the list



Bucket Sort Time Analysis

- Phase 1 takes $O(n)$ time
- Phase 2 takes $O(n + K)$ time
 - Thus bucket-sort is $O(n + K)$
- Very efficient if keys come from a small interval $[0, K - 1]$

Radix Sort

- Radix = “The base of a number system” (Webster’s dictionary)
 - Alternate terminology: radix is number of bits needed to represent 0 to base 1; can say “base 8” or “radix 3”
- Idea: Bucket Sort on each digit, bottom up

The Magic of Radix Sort

- Input list:

126, 328, 636, 341, 416, 131, 328

- Bucket Sort on lower digit:

341, 131, 126, 636, 416, 328, 328

- Bucket Sort result on next-higher digit:

416, 126, 328, 328, 131, 636, 341

- Bucket Sort that result on highest digit:

126, 131, 328, 328, 341, 416, 636

Running time of Radix sort

- n items, d digit keys
- How many passes?
- How much work per pass?
- Total time?

Running time of Radix sort

- n items, d digit keys
- How many passes? d
- How much work per pass? n
- Total time? $O(dn)$

Summary

	Best	Average	Worst
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Bucket sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Radix sort	$O(dn)$	$O(dn)$	$O(dn)$

Next Class

Graph I

Reading: Weiss, chap. 9