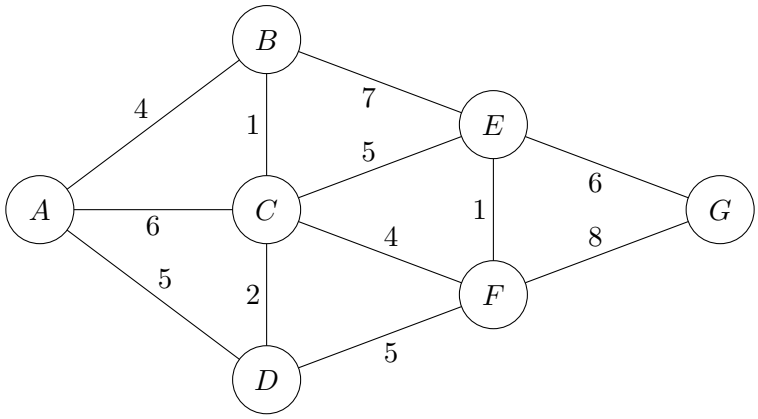
CSM 61B Fall 2020

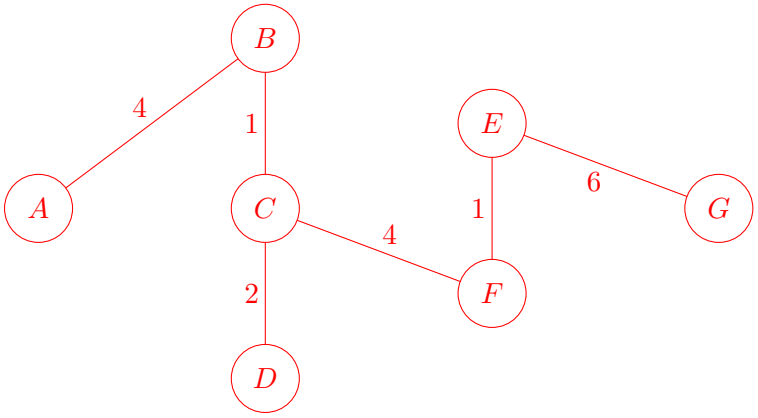
Graphs Sorting

Mentoring 11: November 2, 2020

1 Networking

1.1 Consider the telephone network from last week. Construct a minimum spanning tree by running Prim's Algorithm from node A.





Meta:

Intro: The cut property is introduced in the general statement for this question. Make sure to have a simple example to use as a visual when explaining it to students.

1.3: Your board work for how to run through Prim's should be very similar to how you ran through Djikstra's algorithm in the previous worksheet. Only add edges to your MST as you pop off the Fringe (PQ). Additionarlly, after running through the problem, explain how at a high level, for every edge added, we just added the shortest edge not in the MST into the MST. Use the cut property to explain this.

1.2 Run the quicksort algorithm. Assume we pick the middle element as the

pivot; if there is no exact middle, pick the element to the right of the middle.

```
{ 1, 3, 8, 2, 6, 4, 5, 9 }
```

```
{ 1, 3, 2, 4, 5 }, { 6 }, { 8, 9 }
{ 1 }, { 2 }, { 3, 4, 5 }, { 6 }, { 8 }, { 9 }
{ 1 }, { 2 }, { 3 }, { 4 }, { 5 }, { 6 }, { 8 }, { 9 }
```

2 Sorting Overview

So far, we've learned a few different types of basic sorting algorithms. While sorting might seem like a simple idea, there are many real-world applications of sorting, and several different algorithms that we can use depending on the situation.

In the table below, fill out the best and worst-case runtimes for each of the sorting algorithms provided.

Algorithm	Best-case	Worst-case
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$
Heapsort	$\Theta(N)$	$\Theta(N \log N)$
Quicksort	$\Theta(N \log N)$	$\Theta(N^2)$

In selection sort, we loop through the array to find the smallest element. Next, we swap the element at index-0 with the smallest element. Next, we repeat this procedure, but only looking at the array starting at index-1.

Selection Sort

- *Runtime, Best, Worst Case:* Since it takes $O(N)$ time to loop through the array, and we loop through the array N times, this algorithm has a runtime of $\Theta(N^2)$. Note that even if the array is already sorted, we need to iterate through it to find the minimum, and then iterate through it again, and again, N times.
- *Stability:* Consider an array `{ 3A, 2, 3B, 1 }`, where the 3s have been labeled to differentiate between them. The algorithm will find 1 to be the smallest, and will swap it with $3A$, pushing $3A$ after $3B$, making it not stable. However, it is also possible to make it stable if we implement Selection Sort in a different way, which

involves creating a new array instead of swapping the minimum elements.

Insertion Sort This is the way an adult would normally sort a pack of cards. Iterating through the array, swapping each element left-wards.

- *Best Case:* Given a sorted array, `{ 1, 2, 3, 4 }`, this algorithm would iterate through the array just once, and do 0 swaps, since all elements are already as left-wards as they can be.
- *Worst Case:* Given a fully unsorted array, `{ 4, 3, 2, 1 }`, this algorithm would first swap $(3, 4)$, then to move 2 left-wards, it needs to do 2 swaps. Finally to move 1 left-wards, it needs to do 3 swaps. This is of the ordering of $O(n^2)$ swaps.
- *Stability:* Consider an array `{ 3A, 2, 3B, 1 }`. We would get the following steps: `{ 2, 3A, 3B, 1 }`, `{ 1, 3, 3A, 3B, }`. In general, this algorithm is stable, because given a $3A, 3B$, we would never swap them with each other.

Merge Sort Given an array, divide it into two equal halves, and call merge-sort recursively on each half. Take the recursive leap of faith and assume that each half is now sorted. Merge the two sorted halves. Merging takes a single iteration through both arrays, and takes $O(N)$ time. The base case is if the input list is just 1 element long, in which case, we return the list itself.

- *Best case, Worst Case, Runtime:* Since the algorithm divides the array and recurses down, this takes $\Theta(N \log N)$ time, no matter what.
- *Stability:* Merge sort is made stable by being careful during the merging step of the algorithm. If deciding between 2 elements that are the same, one in the left half and one in the right half, pick the one from the left half first.

Heap Sort Place all elements into a heap. Remove elements one by one from the heap, and place them in an array.

- *Recall:* Creating a heap of N elements takes $N \log N$ time, because we have to bubble-up elements. Removing an element from a heap takes $\log N$ time, also because of bubbling and sinking.
- *Best Case:* Say that all the elements in the input array are equal. In this case, creating the heap only takes $O(N)$ time, since there is no bubbling-down to be done. Also, removing from the heap takes constant time for the same reason. Since we remove N elements,

and creating the heap takes $O(N)$ time, the overall runtime is $O(N)$.

- *Worst Case:* Any general array would require creating the heap with bubbling which itself takes $N \log N$ time.
- *Runtime:* HeapSort is not stable. Consider two elements(3a and 3b) that are considered equal. Based on the implementation, we pop the max element from the heap and add it to the end. This naturally puts 3a after 3b in the array after the two pops.

  For example, say the original array is already in heap structure: `{3(a),3(b),2,1}`. After the first pop of the heap and add it to the end, it will look like: `{3(b),1,2,3(a)}`. And the end result would be `{1,2,3(b),3(a)}`.

Quicksort Based on some pivot-picking strategy, pick a pivot. Divide the array up into 3 groups: elements smaller than the pivot, larger than the pivot and equal to the pivot. Recursively sort the first and second group.

- *Runtime:* Analyzed in detail in the next question.
- *Stability:* QuickSort is generally not implemented as a stable algorithm, assuming we are using Tony Hoare's in-place partitioning implementation. It is not stable because the algorithm swaps non-adjacent elements. However, if we use an extra space of $O(N)$, we can implement a stable QuickSort.

2.1 Give a best and worst case input for insertion sort.

Best case is a completely sorted array with 0 inversions while the worst case is a reverse-sorted array with $\Theta(N^2)$ inversions. Recall that the runtime for insertion sort is given by $\Theta(N + K)$ where K is the number of inversions.

2.2 Do you expect selection or insertion sort to run more quickly on a reverse list?

Asymptotically, both algorithms operate run in $\Theta(N^2)$ in this scenario where N is the length of the reversed list.

Selection sort might be better since it performs only $\Theta(N)$ swaps as opposed to $\Theta(N^2)$ swaps in insertion sort's case.

2.3 In Heapsort do we use a min-heap or max-heap? Why?

We use a max-heap because then we can fill in the array of sorted elements from the back to the front in the same array we use to represent our heap.

2.4 Sort the following array using Heap Sort. [3, 2, 1, 5, 6, 8, 7]

Heapify the array (may be easier to visualize with a tree structure)
[3, 2, 1, 5, 6, 8, 7]
[3, 6, 1, 5, 2, 8, 7]
[3, 6, 8, 5, 2, 1, 7]
[8, 6, 3, 5, 2, 1, 7]
[8, 6, 7, 5, 2, 1, 3]
Then delete the largest element and place it at the back of the array. Do this until the array is sorted. [1, 2, 3, 5, 6, 7, 8] **Meta**: Walkthrough heapification of the array when you go over solutions. No need to justify why/when we sink and swim nodes. Students should be comfortable with this already. Redirect students to the previous part when they answered why we use a max heap, and emphasize that we can do everything within one array.

2.5 Run the quicksort algorithm. Assume we pick the middle element as the pivot; if there is no exact middle, pick the element to the right of the middle.

```
{ 1, 3, 8, 2, 6, 4, 5, 9 }

{ 1, 3, 2, 4, 5 }, { 6 }, { 8, 9 }
{ 1 }, { 2 }, { 3, 4, 5 }, { 6 }, { 8 }, { 9 }
{ 1 }, { 2 }, { 3 }, { 4 }, { 5 }, { 6 }, { 8 }, { 9 }
```

3 Stability

Stability is a property of some sorting algorithms. Stability essentially means that if we have two elements that are equal, then their relative ordering in the sorted list is the same as the ordering in the unsorted list. For instance, let's say that we had an array of integers.

```
{ 1, 2, 1, 3, 1, 2, 4 }
```

Since we have multiple 1 and 2s, let's label these.

```
{ 1A, 2A, 1B, 3, 1C, 2B, 4 }
```

A stable sort would result in the final list being

```
{ 1A, 1B, 1C, 2A, 2B, 3, 4 }
```

Why is this desirable? Say that we have an Excel spreadsheet where we are recording the names of people who log in to CSM Scheduler. The first column contains the timestamps, and the second column contains their username. The timestamps are already ordered in increasing order. If we wanted to sort the username, so that we could group the list to see when each username logs in, we would want that the timestamps maintain their relative order. This is precisely what a stable sort ensures.

3.1 Why does Java's built-in `Array.sort` method use quicksort for **`int`**, **`long`**, **`char`**, or other primitive arrays, but merge sort for all `Object` arrays?

Fast, in-place solutions for quicksort are unstable, meaning that, for any two equivalent keys, their final order in the output is not guaranteed to be the same. Merge sort has good asymtotic behavior without sacrificing on stability.

4 *In'sort' Meme Here*

4.1 Each column below gives the contents of a list at some step during sorting. Match each column with its corresponding algorithm.

· Merge sort · Quicksort · Heap sort · LSD radix sort · MSD radix sort

For quicksort, choose the topmost element as the pivot. Use the recursive (top-down) implementation of merge sort.

	Start	A	B	C	D	E	Sorted
1	4873	1876	1874	1626	9573	2212	1626
2	1874	1874	1626	1874	7121	8917	1874
3	8917	2212	1876	1876	9132	7121	1876
4	1626	1626	1897	4873	6973	1626	1897
5	4982	3492	2212	4982	4982	9132	2212
6	9132	1897	3492	8917	8917	6152	3492
7	9573	4873	4873	9132	6152	4873	4873
8	1876	9573	4982	9573	1876	9573	4982
9	6973	6973	6973	1897	1626	6973	6152
10	1897	9132	6152	3492	1897	1874	6973
11	9587	9587	7121	6973	1874	1876	7121
12	3492	4982	8917	9587	3492	9877	8917
13	9877	9877	9132	2212	4873	4982	9132
14	2212	8917	9573	6152	2212	9587	9573
15	6152	6152	9587	7121	9587	3492	9587
16	7121	7121	9877	9877	9877	1897	9877

From left to right: unsorted list, quicksort, MSD radix sort, merge sort, heap sort, LSD radix sort, completely sorted.

MSD Look at the left-most digits. They should be sorted. Mark this immediately as MSD.

LSD One of the digits should be sorted. Start by looking at the right most digit of the remaining sorts. Then check the second from right digit of the remaining sorts and so on. As soon as you find one in which at

least something is sorted, mark that as LSD.

Heap Max-oriented heap so check that the bottom is in sorted order and that the top element is the next max element.

Merge Realize that the first pass of merge sort fixes items in groups of 2. Identify the passes and look for sorted runs.

Quick Run quicksort using the pivot strategy outlined above. Look for partitions and check that 4873 is in its correct final position.

5 Sorting Out My Head!

5.1 Web developers use many different sorts for the different types of lists that they might want to sort. For each of these, provide the best sorting algorithm amongst the following: Mergesort, Quicksort (with Hoare Partitioning), Insertion Sort, LSD Sort. Also, state the worst-case runtime.

(a) A list of N packets received by a server over time. Each packet has the timestamp at which the sender sent it. However, some packets may be dropped or arrive out-of-order due to the faulty network. Sort this list by that timestamp (sent time).

Since we expect the list to be largely sorted by time already, with a few packets out of place, we should use insertion sort. Worst case runtime is $O(N^2)$.

(b) A list of N websites. Each website has the number of total visitors. Sort this list by visitor count.

Quicksort, since it's generally pretty fast for sorting what is effectively a random list of numbers. Worst case runtime is $O(N^2)$. Could also argue for LSD sort since there might be some limit k on the total number of visitors, but less preferable.

(c) After sorting by visitor count, we now want to sort by webpage file size. If websites have the same file size, they should be ordered by visitor count.

Mergesort, since we want to sort stably. Worst case runtime is $O(N \log N)$.

(d) A list of 20 names. Sort in alphabetical order.

Insertion sort, since it has the least overhead and is fastest for small lists. Worst case runtime is $O(1)$, since 20 is a constant, and we assume that all names are shorter than some fixed constant as well.