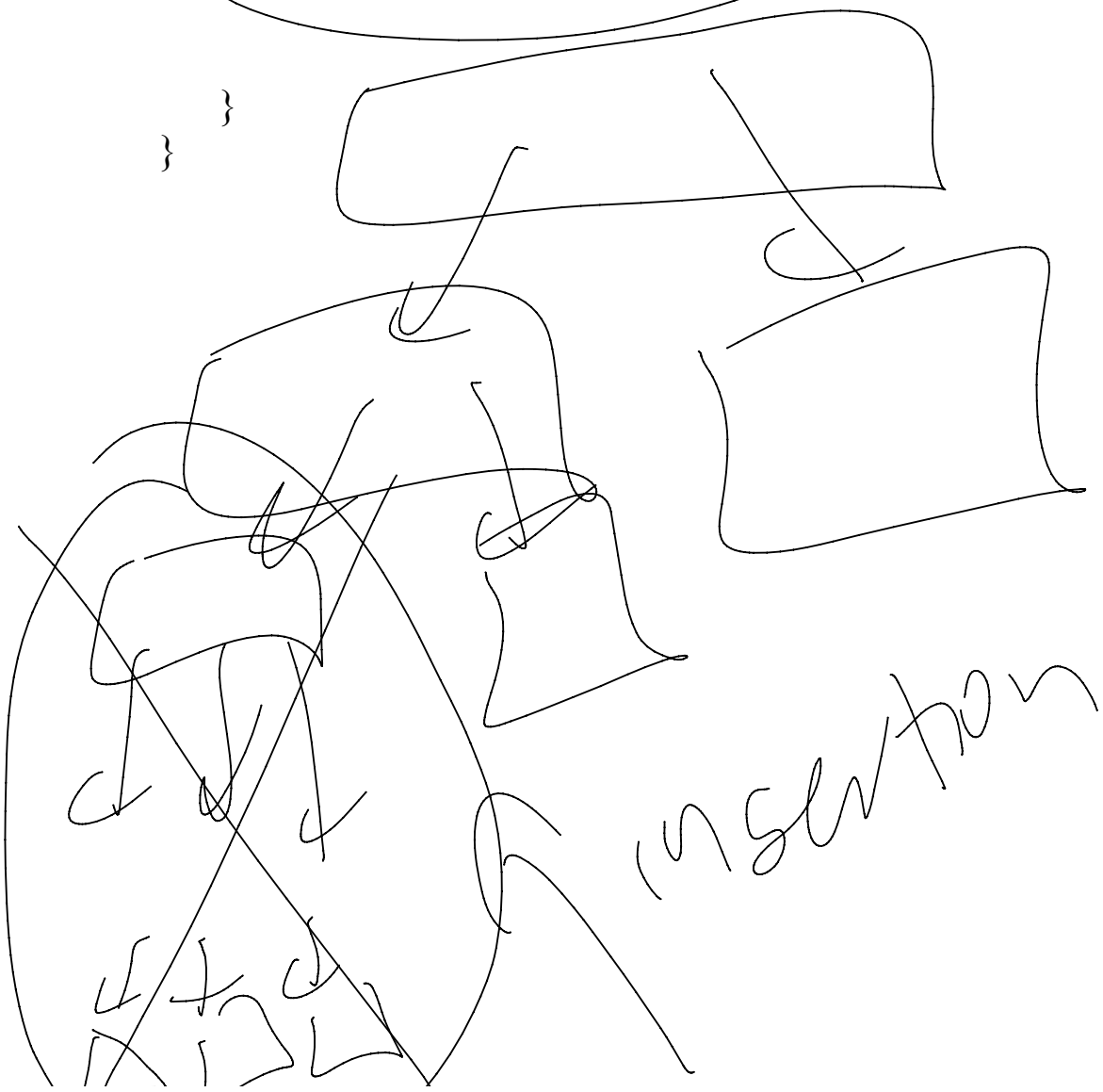
6/24/2020 - Sorting

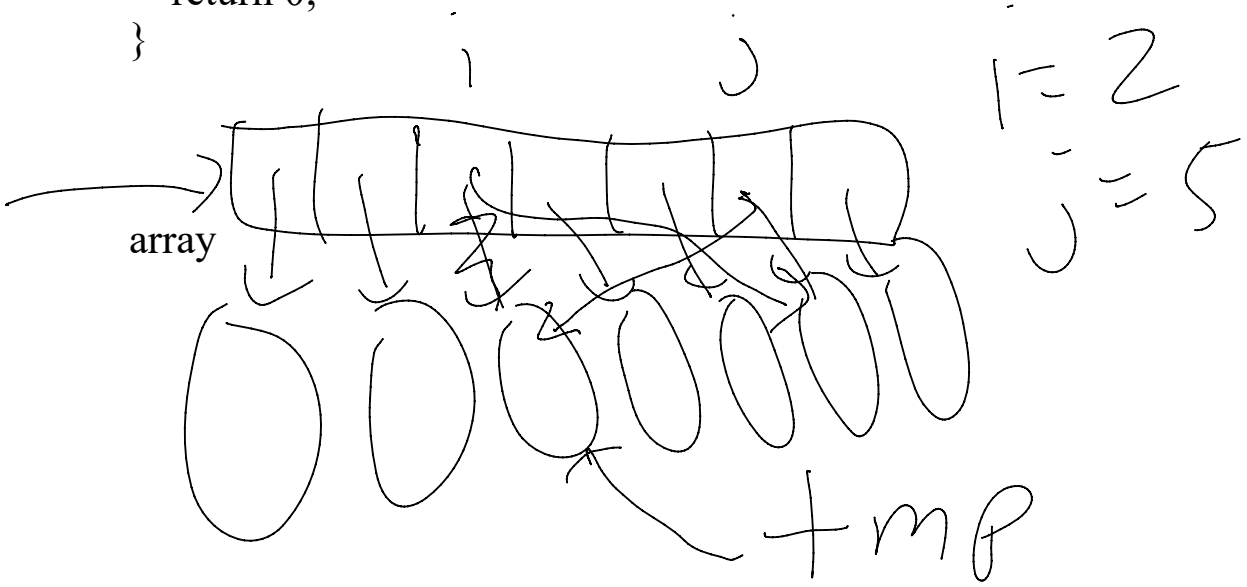
Wednesday, June 24, 2020 4:03 PM

Annoucements

```
void sort(pt** array, int len, int threshold) {
    mergesort(array, 0, len-1, threshold);
}

void mergesort(pt** array, int start, int end, int threshold) {
    if (end-start+1 <= threshold)
        insertionsort(array, start, end);
    else {

    }
}
```



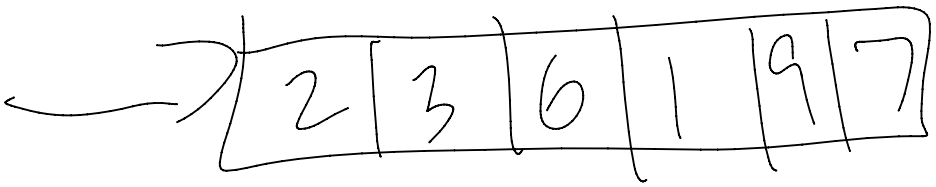
```
int mystrcmp(char* str1, char* str2) {

    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0') {
        if (str1[i] != str2[i])
            return str1[i] - str2[i];
        i++;
    }
    if (str1[i] != '\0') return 1;
    if (str2[i] != '\0') return -1;
    return 0;
}
```




```
pt* tmp = array[i];
array[i] = array[j];
array[j] = tmp;
```

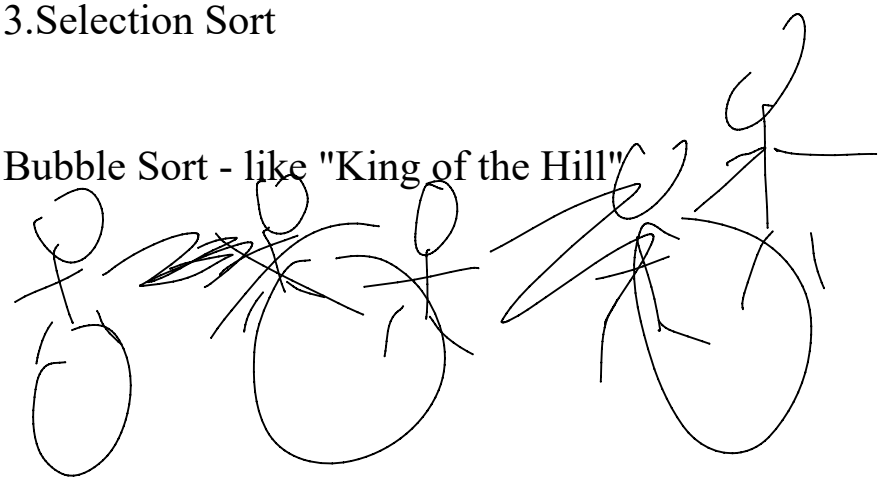
Storing pointers in the array instead of the structs themselves, makes swapping easier. It also makes passing the structs via pointer to functions a bit more natural.

But when we TEACH sorting, to keep things simple and to keep the focus on the algorithms, we just use an array of ints.



We cover three $O(n^2)$ sorts in this class
1.Bubble Sort
2.Insertion Sort (already learned)
3.Selection Sort

Bubble Sort - like "King of the Hill"



You start at the beginning and battle your way iteratively to the top spot, the so called "king of the hill".

8, 2, 9, 1, 3, 4, 7, 6, 5
2, 8, 9, 1, 3, 4, 7, 6, 5
2, 8, 9, 1, 3, 4, 7, 6, 5
2, 8, 1, 9, 3, 4, 7, 6, 5
this continues for a few more rounds
2, 8, 1, 3, 4, 7, 6, 5, 9

This is one iteration of bubble sort.

What is true now? Largest # is at the end.

Now, just repeat the process n-1 times, where n

is the # of values in the array.

First time around, go to end of the array. (upto slot n-1)
Second time, we go upto slot n-2
Third time we go upto slot n-3, etc.

Formal run-time analysis:

inner loop runs n-1 times.
then n-2.
then n-3.
…
downto 1

runtime = (n-1) + (n-2) + (n-3) + … + 1 =

$(n-1)n/2 = O(n^2)$

Bubble sort of 50,000 items took 14 sec
How long will a bubble sort of 100,000 items take?

$c50000^2 = 14$ sec
$c = 14/50000^2$ sec

$T(100000) = c100000^2 = (14 \text{ sec}) * 100000^2/50000^2$
$= (14 \text{ sec})*(100000/50000)^2 = (14 \text{ sec})(2^2) = 56$ sec

Selection Sort

The idea here, is to not swap stuff all of the time.
Why not figure out who is "king of the hill" and just
swap that value into the correct place?

8, 2, 9, 3, 5, 6, 1, 7, 4

First pass, keep a variable called maxIndex = 0
I have a loop variable, let's call it j. And I compare
which value is bigger, the one in index j or index
maxIndex. I update maxIndex anytime I find a bigger value.

maxIndex = 0, j = 1, no change
maxIndex = 0, j = 2, change maxIndex = 2
maxIndex = 2, j = 3, no change
maxIndex = 2, j = 4, no change
…more no change
maxIndex = 2, end of the array is index 8
We swap index 2 with index 8:

8, 2, 4, 3, 5, 6, 1, 7, 9

Run Time Chart

	Best	Avg	Worst
Bubble	n^2	n^2	n^2
Insertion	n	n^2	n^2 (reverse ordre)
Selection	n^2	n^2	n^2

Avg Insertion: On average, we would hop
1/2 way through the array, so when I am inserting
index I, I do about i/2 swaps. So the run time
on average would be
$1/2 + 2/2 + 3/2 + .. + (n-1)/2 = (n-1)n/4$
$= O(n^2)$

Wikipedia's bubble sort is different than the one I showed you
for theirs their best case is O(n). (This is
because they break out of the outer loop if there
ere no swaps in an iteration…)

Clearly, for large data, $O(n^2)$ sorts seem slow.
Can we do better?

YES!!!!

1. Merge Sort (it's best case, average case and worst case are all O(nlgn) - the recurrence relation analysis is accurate for ALL cases.

2. Quick Sort(turns out to be O(nlgn) in best and average case, but $O(n^2)$ in the worst case.

Why would we use quick sort, if its worst case is worse Merge Sort?

Answer: On AVERAGE, on most data, Quick Sort is FASTER.
Reason it's faster: It sorts in place and never has to copy values into a temporary array like merge sort does.

Merge Sort works because of the merge.
Quick Sort works because of the partition.

Partition - to split apart into smaller pieces.
Mathematical definition: To split a set into multiple subsets, where each element in the original set belongs to precisely one of the multiple subsets.

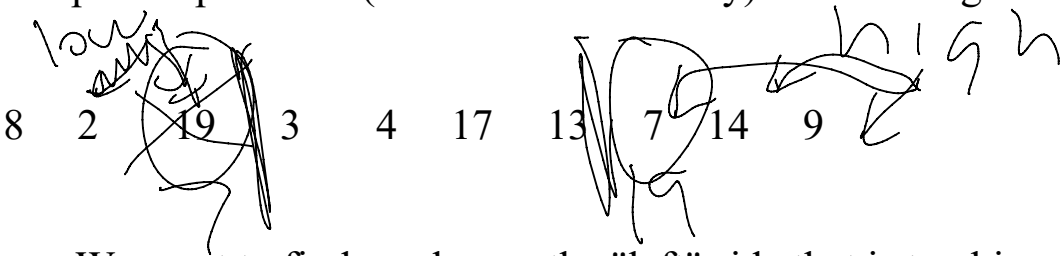
The Partition for Quick Sort splits an array of values into two groups - one group where each item is less than or equal to a "partition value" and the other group where each item is greater than the partition value.

Partition Trace:

8, 2, 19, 3, 4, 17, 13, 7, 14, 9

Let 8, the leftmost element, be the partition value.

Keep two "pointers" (indexes into the array): low and high:

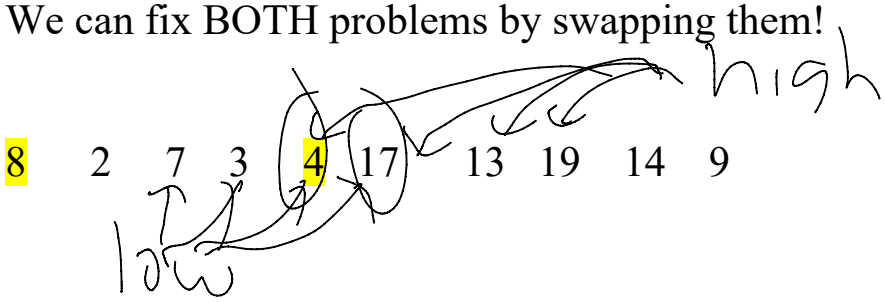


We want to find a value on the "left" side that is too big.
And we want to find a value on the "right" side that is too small.

So, we iterate low until we find a value greater than the partition.
and we iterate high (down) until we find a value less than or equal to

the partition.

We can fix BOTH problems by swapping them!



low goes forward until it hits 17
high goes backwards until it hits 4
So, the array has been partitioned…almost
To finish the job, we swap 8 with 4:

4 2 7 3 8 17 13 19 14 9

Now, imagine if we wanted to sort the array!

1. Sort(4 2 7 3) - left of partition
2. Sort(17 13 19 14 9) - right of partition

Worst Case of the Partition is something like this:

1 17 13 19 22 3 15 12 8 2 11
Why is this bad?
We do O(n) work to partition…and we STILL have n-1 items left to sort
We don't achieve any efficiency by having two smaller lists.

What is the best case split? --- Half and Half!
Worst case is 0 and n-1…

So we choose the partition element randomly, to reduce the probability of the worst case.

Worst case, we partition and get one array of size n-1.
Then the next partition we get one array of size n-2.
And so forth.
Run time = n + n-1 + n-2 + = $O(n^2)$

1.Way to try to prevent it is by choosing a random partition element.

2. Pick three random items, and of those, choose the median to be the partition
3. Pick five random items, and of those choose the median to be the partition.

Upside to #2, #3 - more likely to get a "middle" value
Downside to #2, #3 - slows down process of picking a partition.
If the array is super big it make sense to do 2 or 3, but the smaller it gets, 1 becomes a better option, and below 30 elements or so, it's better to do insertion sort…

Quick Sort on 1,000,000 items was 1 sec
Quick Sort on 500,000 items was 0 sec (actually a fraction of a sec)
Insertion Sort on only 100,000 items was 58 seconds.
HUGE DIFFERENCE!!!

Best Case Run Time - occurs when split is half and half.
Since partition takes O(n), if we let T(n) be the best case run time
f a quick sort of n items, then
$T(n) = T(n/2) + T(n/2) + O(n)$
(left) (right) (partition)
We saw this exact recurrence with Merge Sort…
The answer was $T(n) = O(n \lg n)$.

Proof of the Average Case Run Time of Quick Sort

We want to consider each possible input, we assume each is equally likely and add up all of those times and then divide by the total number of possible inputs.

An equivalent way to think about average (expected value) is to multiply each run time by the probability of getting that run time and add these values up.

Let T(n) = average case run time of a quick sort of n items.

When we run partition, we have many different outcomes:

1. Left = 0 items, Right = n-1 items
2. Left = 1 item, Right = n-2 items

3. Left = 2 items, Right = n-3 items
4. …
5. Left = n-1 items, Right = 0 items

There are n possible ways in which the array could split.
We will assume that each occurs with probability 1/n:

$T(n) = 1/n*(T(0) + T(n-1) + T(1) + T(n-2) + T(2) + T(n-3) + \ldots. + T(n-1) + T(0)\ ) + O(n)$ (last term is time for partition, with probability 1)

$T(n) = 1/n*(2T(0) + 2T(1) + 2T(2) + \ldots 2T(n-1)) + cn$

Multiply both sides by n:

$nT(n) = 2(T(0) + T(1) + T(2) + \ldots + T(n-2) + T(n-1)\ ) + cn^2$

This is sort of creative, but let's plug in n-1 where we see n above:

$(n-1)T(n-1) = 2(T(0) + T(1) + \ldots + T(n-2)\ ) + c(n-1)^2$

We do this, so that we can subtract and get rid of the sum.

Subtract the bottom from the top to yield:

$nT(n) - (n-1)T(n-1) = 2T(n-1) + c(2n - 1)$

$nT(n) = (n+1)T(n-1) + c(2n-1)$
---- ---------- --------
n*(n+1) n*(n+1) n*(n+1), divide equation by n(n+1)

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + O(1/n)$$

Let S(n) = T(n)/(n+1) and substitute:

$S(n) = S(n-1) + c/n$

$= S(n-2) + c/(n-1) + c/n$
$= S(n-3) + c/(n-2) + c/(n-1) + c/n$
$= \ldots$
$= S(0) + c/1 + c/2 + c/3 + \ldots. + c/n$
$= c(1 + 1/2 + 1/3 + \ldots + 1/n)$

The thing in blue is a famous number, it is known as the nth Harmonic number. We can use integration to approximate its value to be very close to ln n.

$\sim c \ln n = O(\lg n)$

$T(n) = (n+1)S(n) = O(n \lg n)$