CS 315 - Day 19, O(N) Sorts

Consider a list `a[]` of three numeric elements, `a[1]`, `a[2]`, and `a[3]`.

Write code which outputs these elements in sorted (ascending) order, ***WITHOUT*** using any loops or recursion.

If statements are allowed (and required).

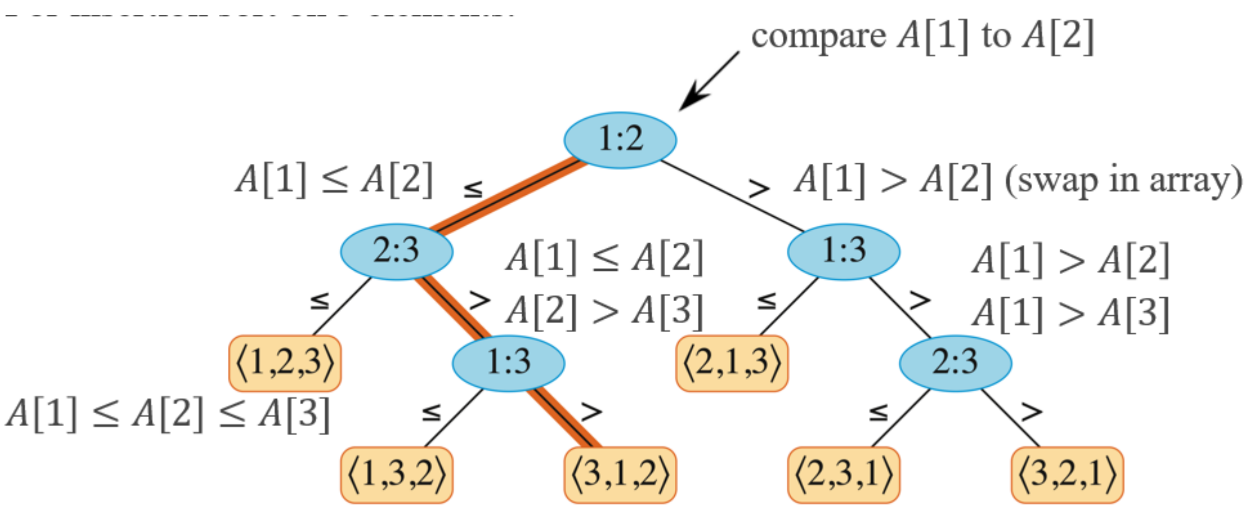
Do not reorder the values of elements, ie. no sorting in-place.

Once you have determined the ordering, you can print it, `print(a[2],a[1],a[3])`

How many print statements will you need? How many if statements?

```
if a[1] <= a[2]:
        if a[2] <= a[3]:
            print(a[1],a[2],a[3])
        else:
            if a[1] <= a[3]:
                print(a[1],a[3],a[2])
            else:
                print(a[3],a[1],a[2])
else:
        if a[1] <= a[3]:
            print(a[2],a[1],a[3])
        else:
            if a[2] <= a[3]:
                print(a[2],a[3],a[1])
            else:
                print(a[3],a[2],a[1])
```

Let's consider a Decision Tree. Oval nodes are the if statements, rectangular nodes are the print statements.



This tree models all possible execution traces.

What is the length of the longest path from the root to a leaf? → Depends on the algorithm.

So, can we generalize this to find a lower bound on comparison based sorts?

How many print statements were there? $3! = 6$, which means that for three numeric inputs, we had six possible orderings.

In general, if we have n elements, how many orderings would we have? $n!$

So the question is, what is the height of a decision tree with $n!$ orderings?

The decision tree, based on if statements, is binary, and we know $\#\, leaves \le 2^{ht}$ → $n! \le 2^{ht}$

Taking the logarithm of both sides gives $\lg n! \le ht$

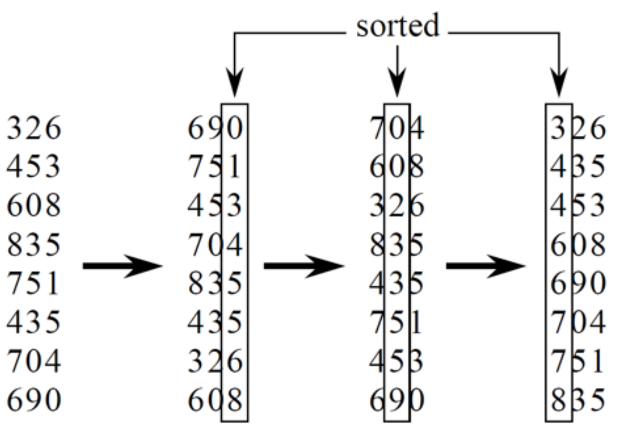
It turns out, there are some useful asymptotic identities we can use from our textbook.

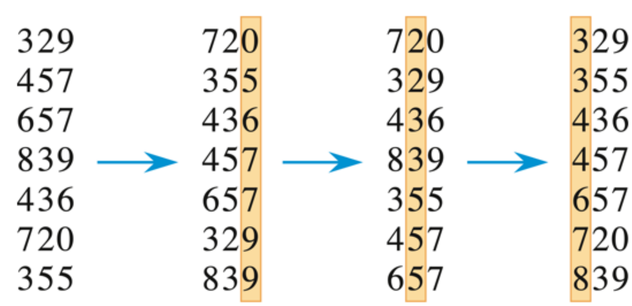
$$n \cdot \lg\left(\frac{n}{2}\right) \le \lg n! \le ht$$

So, from this, we can conclude that a lower bound on any sort requiring comparisons is $\Omega(n \lg n)$!

Next, consider how IBM (International Business Machines) made its money. Initially it was Tabulating Business Machines, and they made punch card readers for census tabulation. These card sorters worked on one column at a time. Herman Hollerith was the Census employee who proposed this and went on to found what is now IBM.

It's the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!





What is the asymptotic running time?

Hollerith leveraged the data format (columns of digits) for a $O(N)$ sort!

But first, let's consider sorting just one column of integers without using ANY comparisons. Remember, the fastest possible sort using comparisons will be $N \cdot \lg N$. ***Without comparisons / no if statements***, but using some other information, we could be faster.

Counting Sort assumes each input is in the range: $[0, k]$, for some integer k.

```
Counting-Sort(A,B,k)
let C[0..k] be a new array
for i=0 to k
   C[i]=0
for j=1 to A.length
   C[ A[j] ] = C[ A]j] ] + 1
// C[i] now contains # of elements equal to i
for i=1 to k
   C[i] = C[i] + C[i-1]
// C[i] now contains # of elements less than or equal to i
for j=A.length downto 1
   B[ C[ A[j] ] ] = A[j]
   C[ A[j] ] = C[ A[j] ] - 1
```

For an example, let's start with:

A:	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8

`C[i]` will contain the number of elements equal to `i`:

C:	2	0	2	3	0	1
	0	1	2	3	4	5

Next, `C[i]` will contain the number of elements $\leq$ `i`:

C:	2	2	4	7	7	8
	0	1	2	3	4	5

Now, fill the output array:

```
B[ C[ A[8] ] ] = A[8]
B[ C[0] ] = 0
B[ 7 ] = 0
```

B:	0	0	0	0	0	0	3	0
	1	2	3	4	5	6	7	8

C:	2	2	4	6	7	8
	0	1	2	3	4	5

```
B[ C[ A[7] ] ] = A[7]
B[ C[0] ] = 0
B[ 2 ] = 0
```

B:	0	0	0	0	0	0	3	0
	1	2	3	4	5	6	7	8

C:	1	2	4	6	7	8
	0	1	2	3	4	5

```
B[ C[ A[6] ] ] = A[6]
B[ C[3] ] = 3
B[ 6 ] = 3
```

B:	0	0	0	0	0	3	3	0
	1	2	3	4	5	6	7	8

C:	1	2	4	5	7	8
	0	1	2	3	4	5

And so on…

How expensive?

$\theta(k)$ to initialize `C[]` to 0.
$\theta(n)$ to fill `C[]`.
$\theta(k)$ to update `C[]`.
$\theta(n)$ to fill `B[]`.

So we have $\theta(k + n)$ total time, which equals $\theta(n)$ if k is $O(n)$.

The key here is that there are no comparisons, but the values of the elements are used as indices.

Counting sort is ***stable***. Key values with the same value appear in the output in the same order they appear in the input.

Now, back to Herman Hollerith's card sorting. It uses a stable (see above!) $O(N)$ sort, like Counting Sort, on each column.

How would we apply Counting Sort to the following list? We need to apply Counting Sort twice. Which means the question is "which column should we apply Counting Sort to first?"

33
15
25
12
23
17
32
22

If we do the left, or most significant digit first, our list becomes:

15
12
17
25
23
22
33
23

Then, we'd have to sort the 10's, 20's, and 30's separately, which could get messy. We already went counter-intuitive and sorted the right, or least significant digit first, and then moved left:

33	→	12	→	12
15	→	32	→	15
25	→	22	→	17
12	→	33	→	22
23	→	23	→	23
17	→	15	→	25
32	→	25	→	32
22	→	17	→	33

So, when Herman Hollerith sorted cards, he just set it to go from right (least) to left (most), and after a column was sorted put the ***whole*** pile back in the sorter for the next column.

Here's the sort, called Radix-Sort:

```
Radix-Sort(A,d)
for i=1 to d
   use a stable sort to sort array A on digit i
```

This is $\theta(d \cdot (k+n))$, and since k is $O(n)$, this is gives $\theta(d \cdot n)$. In practice, Radix-Sort is sometimes used on data with multiple fields, like YY-MM-DD.