04. Lower bounds for sorting. Radix sort. Sorting networks.

Lower Bound of Sorting

We've learned

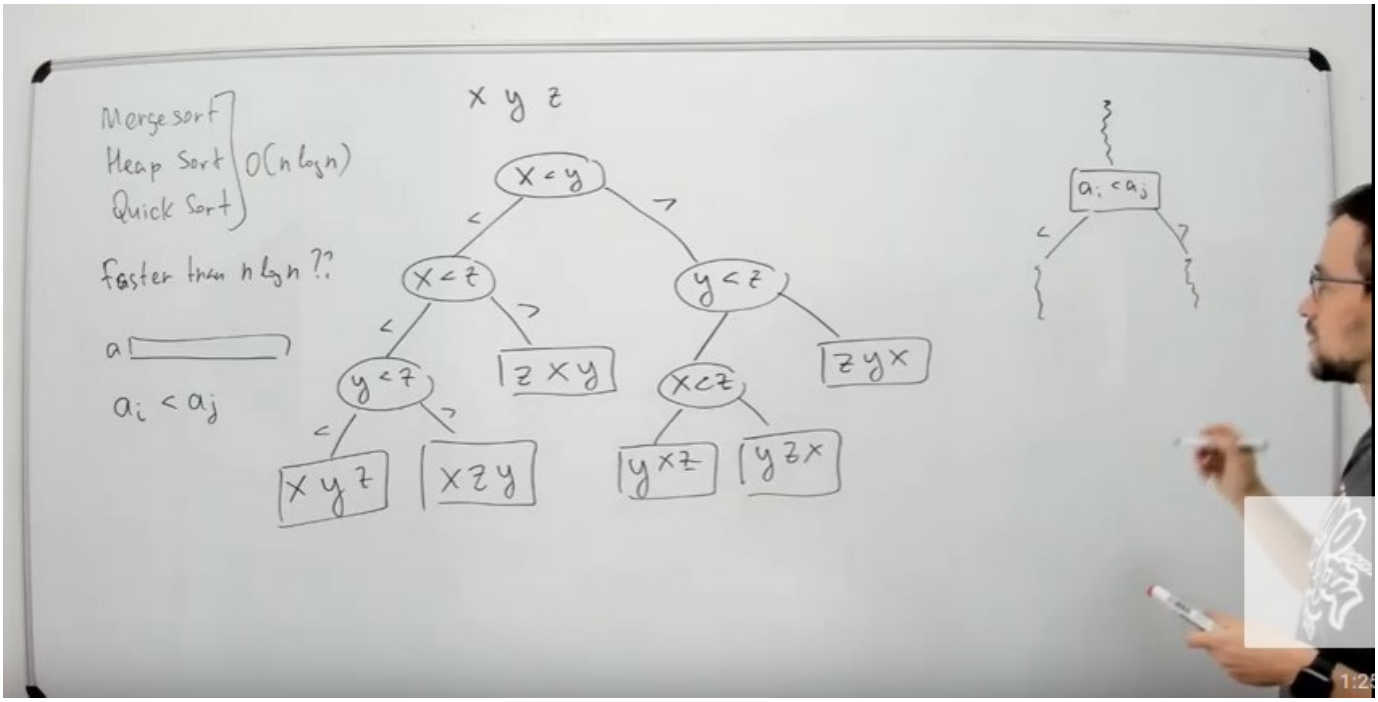
1. Merge Sort
2. Heap Sort
3. Quick Sort

All three algorithms have different techniques, but they all work in O(nlogn).

Q: Is it possible to have a sorting algorithm that runs faster than nlogn?
A: No.

Consider the set of operations we can do on an array. In the above sorting algorithms we only need to **compare** two elements.

To prove: If the only operation we do is comparison, it's not possible to get a faster time than O(nlogn)
Proof:
Consider three elements x, y, z, that need to be sorted. By comparing elements, we get the following tree:



Any comparison of two elements will create two branches.
Height of this tree is number of comparisons you need, equal to the time of your algorithm.

H = T(n)
The leaf nodes are different outcomes of the algorithm.
Number of leaf nodes = Number of permutations of all elements in the array = n!

Minimum possible height of tree:
H = T(n) >= log(number of leaves)
T(n) >= log2(n!)
= log2(123..*n)
= Σ(logi) = Ω(nlogn) (lower bound)

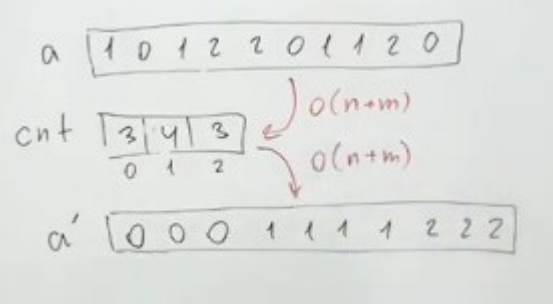
To sort the array, we need to basically guess one of the permutations. For this, you need at least nlogn operations.

This proof works only when you have abstract elements you can only compare.
Sometimes it's possible to get a sorting algorithm that is faster. This happens when comparison is not the only operation you can perform.

Counting Sort

Consider an array a.
a[i] - Integers from 0..m-1, where m is a small enough number.
eg. m=3 ; a = [1,0,1,2,2,0,1,1,2,0]

Just traverse the array and keep a count of how many times an element occurs in a different array.



O(n+m) = O(max(n,m))
n and m should be close enough.

What if we have a bigger number?

Radix Sort

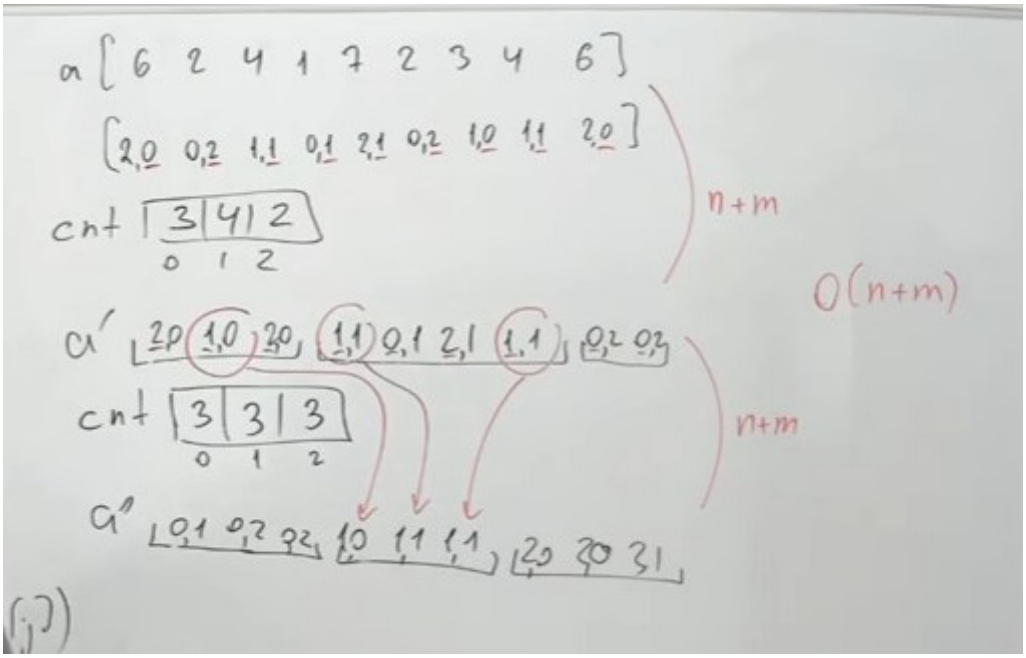
Consider an array a.
a[i] = Integers from 0..(m^2)-1, where m is a small enough number.
We can split each element into two parts.
If x belongs to [0..(m^2)-1],
x = y.m + z,
where y, z belong to [0..m-1]
eg. m=10 ; [0..99]
`a[i] = b[i].m + c[i]`

Comparing a[i] and a[j] is the same as comparing pairs (b[i],c[i]) and (b[j], c[j]).
For instance, if m=10, comparing 42 and 69 is the same as comparing (4,2) and (6,9) as 42 = 4*10 + 2 and 69 = 6*10 + 9

eg. m = 3 ; a = [6,2,4,1,7,2,3,4,6]
6 = 2*3 + 0 -> (2,0)
2 = 0*3 + 2 -> (0,2)
4 = 1*3 + 1 -> (1,1)

So a = [(2,0),(0,2),(1,1),(0,1),(2,1),(0,2),(1,0),(1,1),(2,0)]

Do counting sort twice. First sort by second element of pair, then by first element of pair.



In practice, an algorithm with poor asymptotics might perform better, and vice versa. Usually, this algorithm isn't the best to use in practice. It depends on given data. For n, m upto a million, it's a little faster than standard sorting algorithms.
But in theory it's good to know that sometimes you can sort an array in linear time.

Also, we got a time complexity of O(n+m) in both cases when a[i] was from [0..m-1] and [0..m^2-1].

Generalizing it to [0..m^k-1], we'll do the same thing.
O(k(n+m))

Sorting Networks

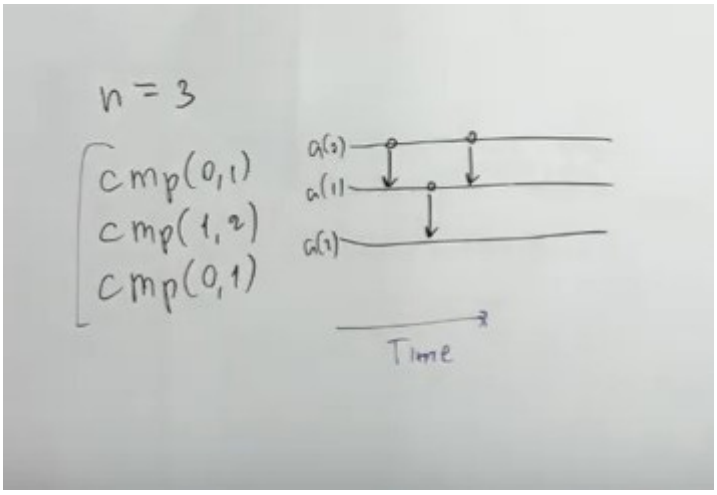
Suppose we need to sort an array, but we have a different computational model where the only operation allowed on an array is comparing two elements and swapping them.

```
comp(i,j)
    if a[i]>a[j]
        swap(a[i], a[j])
```

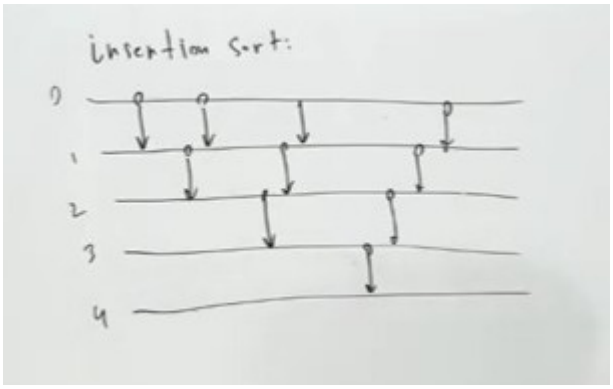
Suppose this ^ is the only operation available. And we can't use anything except the above operation. We need to sort a given array.

eg. n=3; array having three elements. Suppose we implement insertion sort.
For array indices [0, 1, 2], we will call
comp(0,1), comp(1,2), comp(0,1)

A more visual way of representing it is to think of every element as a horizontal line, and the comparator as arrows between lines

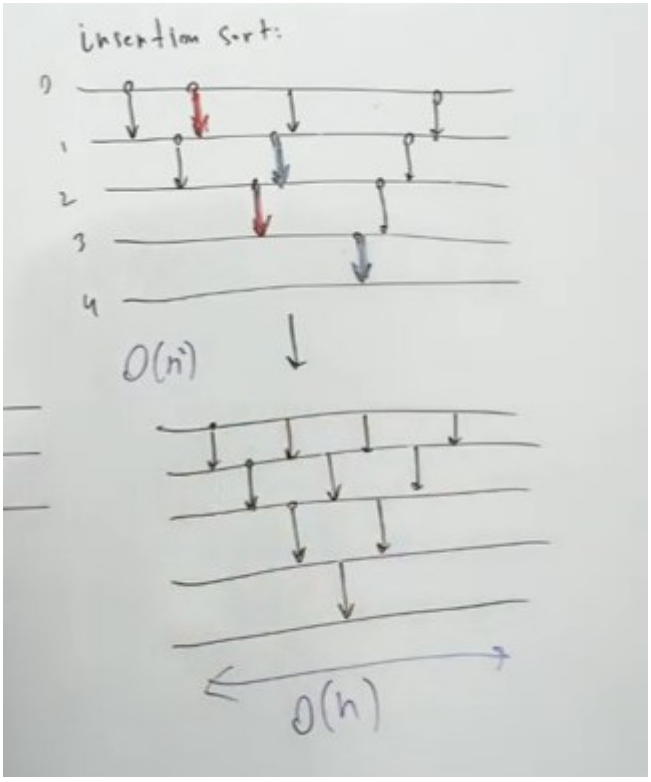


Insertion sort for five elements:



There are two parameters we want to optimize:

1. No. of operators: Here, no. of operators = nlogn
2. Width of this network: If two calls of a comparator don't interact with each other, we can call them together. eg. cmp(0,1), cmp(2,3) can be called together so number of operations can be less than n.

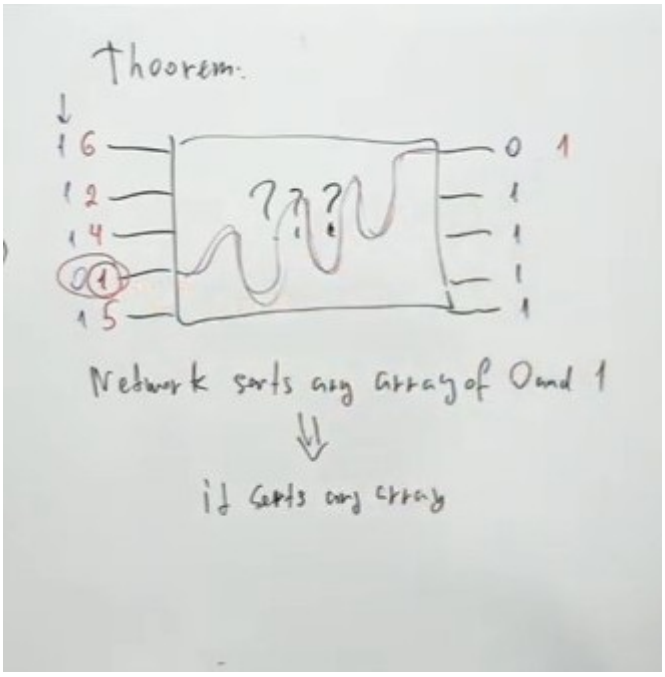


Complexity of concurrent calls went from O(n^2) to O(n). This is not time complexity exactly.

Note: The RAM model allows only one operation at a time. Here, a different computational model is assumed.

It is possible to make a sorting network that uses nlogn comparators and has depth upto logn. But that's complex.
We'll discuss a different algorithm.

Theorem. If we have a sorting network that sorts any array of 0s and 1s, then it sorts any array.
Proof:

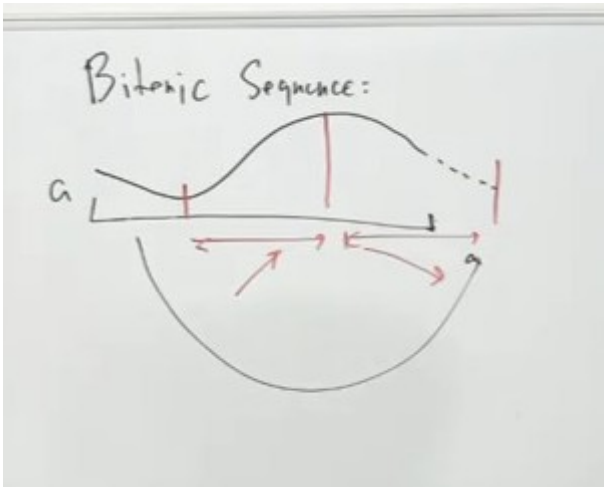


Assign all values of the array 1, and minimum value of 0. After first pass, the minimum element ends up on the top.

Repeat for smaller array.

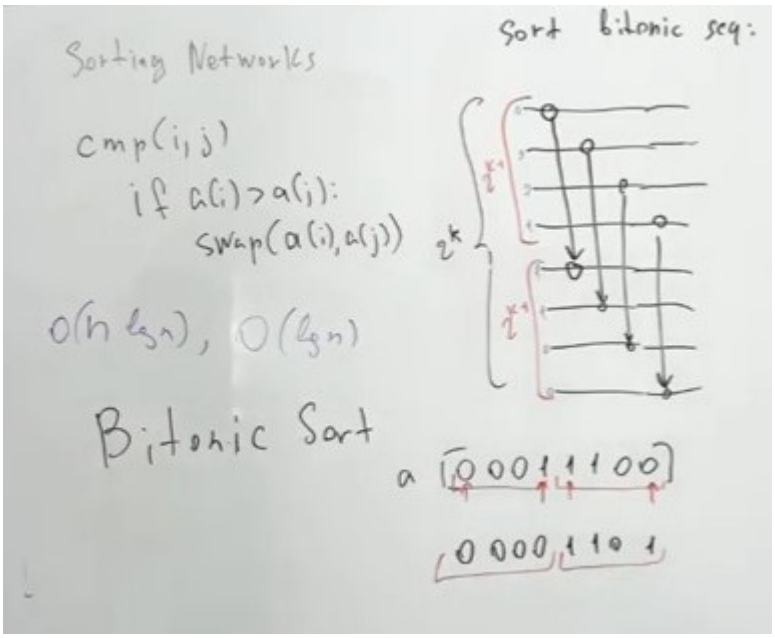
To build the sorting network, we use **Bitonic Sort.**
Bitonic sort is used for parallel algorithms and stuff.

Bitonic sequence: First strictly increasing, then strictly decreasing, or vice versa.
We can view the array as a cyclic array.

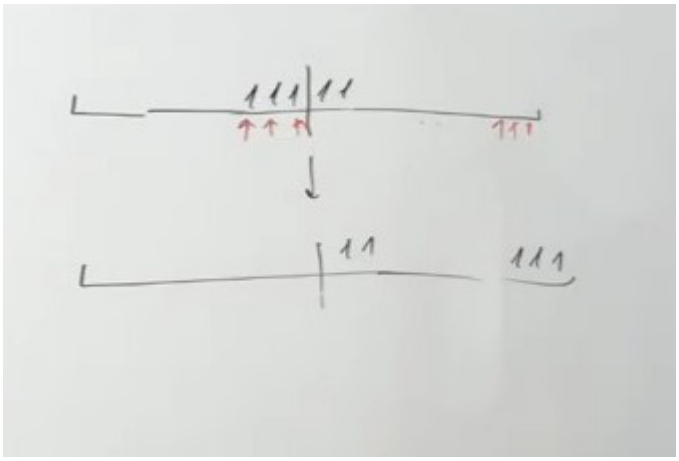


eg. a = [3, 2, 4, 6, 10, 25, 11, 8, 6]
Increases from 2 to 25 and decreases from 25 to 2.
Bitonic array can be sorted using bitonic sort.

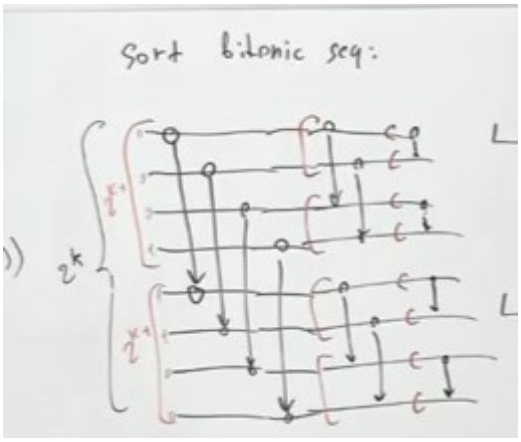
A bitonic array of 0s and 1s can be considered.
Consider a bitonic array of size 2^k. Split it into two arrays of size 2^(k-1).
We run the comparator as follows:



Doing the above splits the bitonic array into two bitonic arrays like so



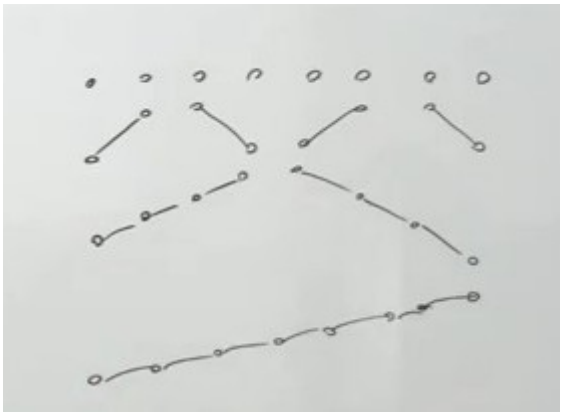
Split each half into two halves, and recursively repeat the same.



In every recursive call, all comparators can run simultaneously. So, number of comparators = nlogn.
Total number of concurrent calls = logn

Now,
to sort an arbitrary array (not bitonic), we can use the following technique:

1. An array of two elements is bitonic
2. So we group elements in groups of two in the array, and call bitonic sort for each of these groups.



No. of calls = logn
No. of comparators = n(logn)^2

Total depth is (logn)^2 since we have logn depth per call and logn calls.