Order Statistics (aka Selection Problems)

Part I: Intro to ideas around Min, Max, and Median

Order Statistics

Simply stated: “Given a list of *n* unique values, find the i^{th} smallest.”

<u>Common Examples</u>

1^{st} smallest (Minimum)

n^{th} smallest (Maximum)

$n/2^{th}$ smallest (Median)

Solving Selection Problems

How can we approach solving such problems?

Trivial Way: Sort the list and then return the i^{th} position.

This is clearly not a good approach for things such as minimum and maximum.

This may or may not be a good approach for other problems such as median finding.

Minimum

Finding the minimum can easily be done using at worst n-1 comparisons:

- Call the first item in the list the smallest.
- For each item remaining, compare it to the item currently considered smallest and if it is smaller than that item, set this new item as the smallest.

Do other algorithms exist? Sure, but are they any better? We already saw the runtime of the following recursive algorithm…

- Split the list in half.
- Find the minimum of each half.
- Take the minimum of the two "local" minimums returned.

Maximum

Is there any practical difference between algorithms for finding the maximum as opposed to finding the minimum value in a list?

- No!

Minimum AND Maximum

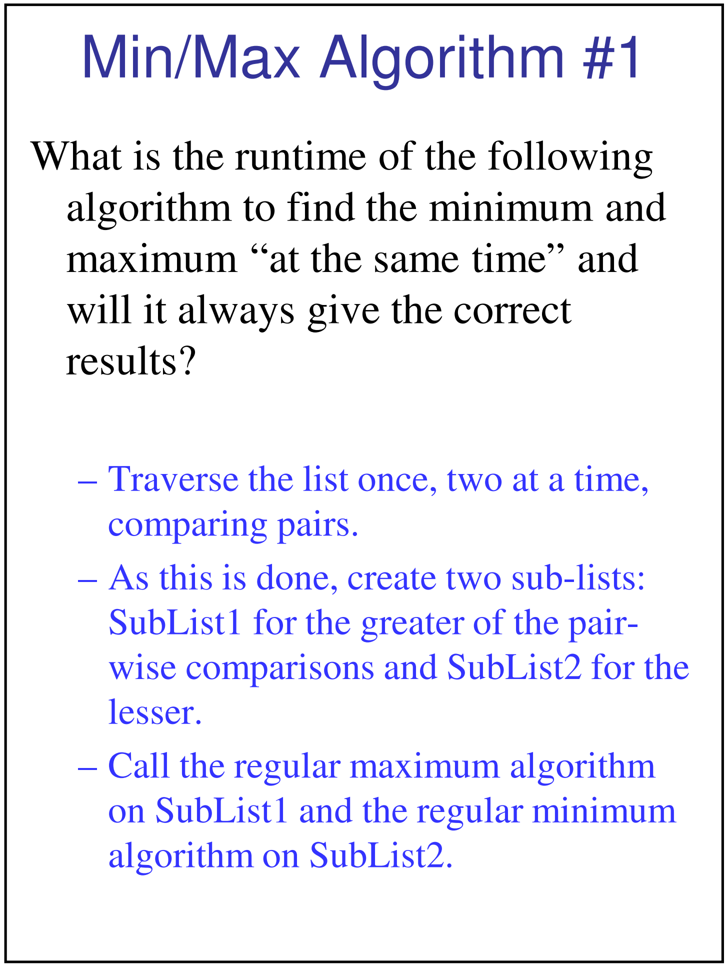
Consider the following scenario:

> You are given a list of coordinates and are asking to return a bounding box for these points.

Your getBoundingBox() method would need to find both the minimum x-coordinate as well as the maximum x-coordinate (and then do the same for the y-coordinates).

In general, given a list of items, it is easy to find the minimum and the maximum using 2(n-1) comparisons.

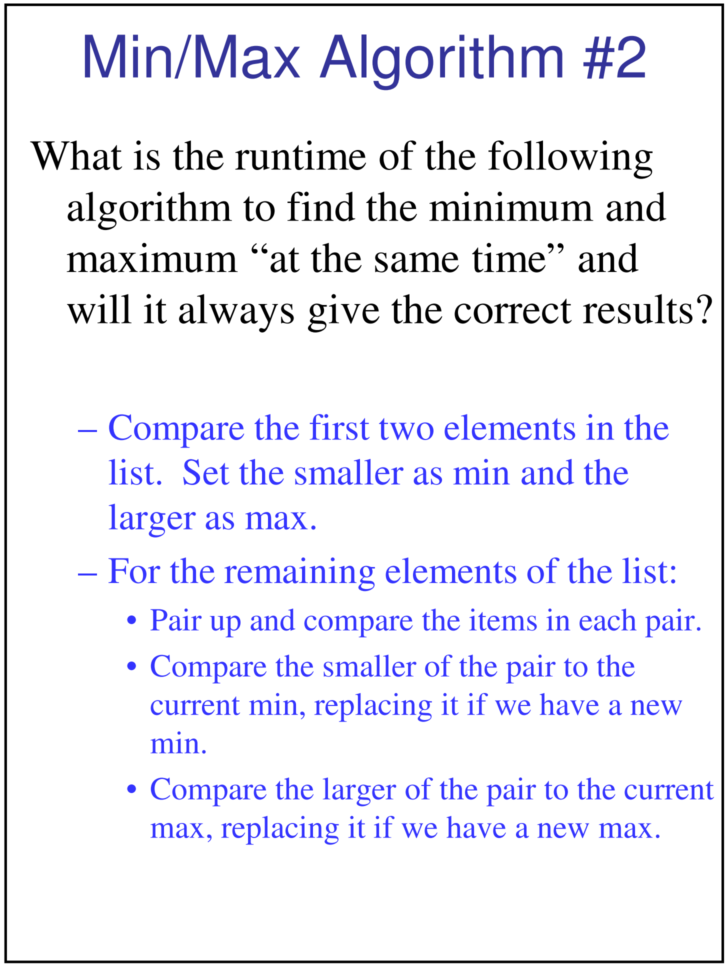
Can we do better?



Min/Max Algorithm #1

What is the runtime of the following algorithm to find the minimum and maximum "at the same time" and will it always give the correct results?

- Traverse the list once, two at a time, comparing pairs.
- As this is done, create two sub-lists: SubList1 for the greater of the pair-wise comparisons and SubList2 for the lesser.
- Call the regular maximum algorithm on SubList1 and the regular minimum algorithm on SubList2.



Min/Max Algorithm #2

What is the runtime of the following algorithm to find the minimum and maximum "at the same time" and will it always give the correct results?

- Compare the first two elements in the list. Set the smaller as min and the larger as max.
- For the remaining elements of the list:
  - Pair up and compare the items in each pair.
  - Compare the smaller of the pair to the current min, replacing it if we have a new min.
  - Compare the larger of the pair to the current max, replacing it if we have a new max.

Median

Here we are given a list of *n* values and are asked to determine which would be in the "middle" of the sorted version of the list.

This can also be stated as "there are the same number of values lower than the median as greater than it" if you'd like.

Finding the median can easily be done using at worst order *n*log*n* comparisons:

– Sort the list using MergeSort.

– Return the n/2th element.

Can we do better?

Inspiration from Min? Max?

If we looked for the minimum value, discarded it, looked for the next minimum value, etc. n/2 times we would get the Median, but it would take $\Theta(n^2)$ time, so we might as well sort.

Using a divide and conquer approach *did not* lead to any improvements in Min or Max but it *did* lead to some improvement for Min&Max…

My First Thought

My thought process when I saw this problem was:

- Hmmmmm. Finding the Median of a 3-element list would be a matter of finding the Min and the Max and discarding them to find the Median…
- I wonder if I could do something where I group things on clusters of 3 values and take advantage of this…

The First
Golub Median-3 Algorithm

- Group values into blocks of 3.
- Find the median of each of those 3-groups.
- Recursively find the median of those medians.

First question: Does this work?

Second question (if it does): What’s the worst-case runtime?

An Algorithm That Works?

While the “median of median-3s” is not guaranteed to be the median of the original list, it turns out that it looks like it might be a nice pivot value.

If we partition around this pivot value:

- We get two lists.
- The median value is in ONE of these two lists, and by looking at the ***size*** of the lists, we know which one.

Challenge: We won’t be looking for the median anymore in the next round.

Select(list, pos)

Place the *n* elements of the list into groups of 3 and find the median of those groups and create Med3List.

MoM3=Select(Med3List, *n*/6);

Partition the original list around MoM3 into LeftList and RightList and figure out the position of MoM3.

```
if pos==MoM3pos then
  DONE!
elseif pos<MoM3pos then
  Select(LeftList, pos);
else
  Select(RightList, pos-MoM3pos)
```

How bad is that last call?

After partitioning around the MoM3, in the worst case possible, how many elements are there in the sub-list that we are going to call Select( ) on recursively?

What's The Worst Runtime?

Find the Med3s:	$\Theta(n)$
Find the MoM3:	$T(n/3)$
Partition around MoM3:	$\Theta(n)$
Worst Case Recursion:	$T(2n/3)$

Use recursion tree to figure out the runtime of this…