Recap: Finding median of unsorted elements

```
RandomizedSelect(A, i)
    pick j randomly from {1,2,..., len(A)}   // pivot=A[j]
    k = Partition(A, j)   // pivot is now at A[k]
    If k=i
        Return A[k]
    ElseIf k>i
        Return RandomizedSelect(A[1,...,k-1], i)
    Else   // k<i
        Return RandomizedSelect(A[k+1,...,n], i-k)
    EndIf
```

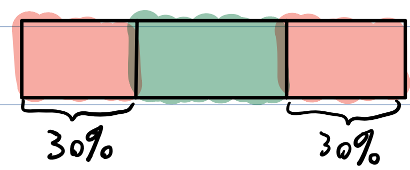
Expected Runtime: $T(n) = \Theta(n)$

To turn into deterministic algorithm, we pick the "approx-median" deterministically.

```
DeterministicSelect(A, i)
    Compute pivot = A[j] that is an "approx-median"
    k = Partition(A, j)   // pivot is now at A[k]
    If k = i
        Return A[k]
    ElseIf k > i
        Return DeterministicSelect(A[1,...,k-1], i)
    Else   // k < i
        Return DeterministicSelect(A[k+1,...,n], i-k)
    EndIf
```

First Attempt to find approx-median:

Take <u>any</u> $\frac{3}{5}n$ elements and find their median. It's guaranteed to be a good "approx-median"!

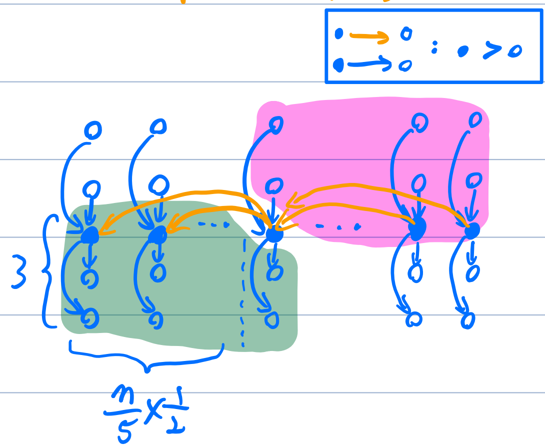


$$T(n) = \underline{T\left(\tfrac{3}{5}n\right)} + T\left(\tfrac{7}{10}n\right) + \Theta(n)$$

$$\Rightarrow T(n) = \Theta(n^{1.616})$$

1. Median and order statistics (cont'd)

Actual Algorithm to find "approx-median" ("median of medians"):



① Partition A into $\frac{n}{5}$ sets of size 5 each.

② Compute median of each set in $O(1)$.

③ Compute median of these $\frac{n}{5}$ medians, that'll be our "approx-median" x.

– How many elements smaller than x? # elements in green area $\geq (\frac{n}{5} \times \frac{1}{2}) \times 3 = \frac{3}{10} n$

– How many elements greater than x? # elems in pink area: $\geq \frac{3}{10} n$

Runtime is now:

$$T(n) = \underline{T(\tfrac{n}{5})} + T(\tfrac{7}{10} n) + \Theta(n)$$

$$\Rightarrow T(n) = \Theta(n)$$

Same as finding min/max!!

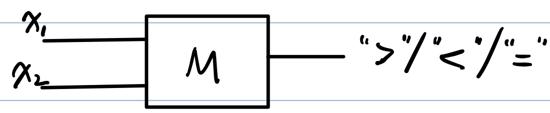
Also deterministic!!

💡: **Reduce the size of subproblems**

2. Lower bound on comparison sort

All the selecting/sorting algorithms we have seen so far are in the comparison model: we don't care about the actual values in the array, we only care about how they compare to each other (relative order).

- Comparison Model:



Only allowed operation is comparison using this "black-box" M.

✓ Transitivity ($a>b, b>c \Rightarrow a>c$) Needed for sorting!

- Time cost: # of comparisons (calls to M)

e.g. Finding Max

```
currMax = bigger(A[1], A[2])
For i = 3 to n
      currMax = bigger(A[i], currMax)
Return currMax
```

# of comparisons: $n-1$

Can we compute with fewer comparisons?

$x, y, z \geq 2$ Comparisons

$w, x, y, z \geq 3$ Comparisons

Claim: Computing maximum of n elements requires $\geq n-1$ comparisons.

Proof: Consider *any* algorithm that outputs the max:

There can be at most one element that has never lost a comparison. Otherwise, each of the two elements can potentially be the max, and the algorithm has no way of telling which one is the max.

Therefore, $n-1$ elements must lose at least one comparison.

But since there is only one loser per comparison, # of comparisons $\geq n-1$.

$\#$

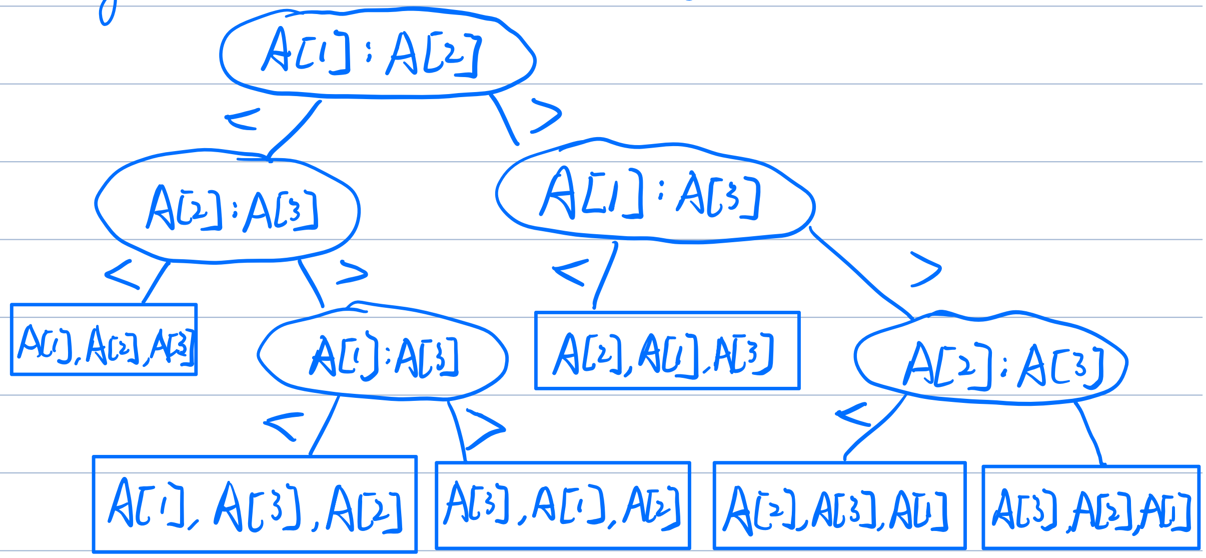
Alternative proof: Consider running the algorithm on input $1, 2, \ldots, n$;

we claim that each of $1, 2, \ldots, n-1$ must be compared at least one to a bigger number, i.e. "lose" a comparison. If, say $k \in \{1, 2, \ldots, n-1\}$ never loses, then we can replace k with $n+1$, and the algorithm wouldn't notice, and still output n (incorrectly).

What about sorting?

We can model any comparison-based algorithm as a *decision tree*:

e.g. Insertion Sort on 3 elements



\# of leaves = \# of permutations of n elements $= n!$

When sorting n elements, any correct algorithm (in the comparison model) must have at least $n!$ leaves, since all permutations are possible.

What's the number of comparisons on the worst case input?

The <u>depth</u> of the tree!

For a tree of depth k, it has at most 2^k leaves,

so we get: $2^k \geq n! \Rightarrow \log(2^k) \geq \log(n!)$

$$\Rightarrow k \geq \log(n!) = \Omega(n \log n)$$

since $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

Therefore, any algorithm for sorting n elements using comparisons must use at least $\Omega(n \log n)$, and in particular run in time $\Omega(n \log n)$ in the worst case.

<u>Cor</u>: MergeSort, QuickSort (with median pivot) are optimal up to a constant.