Sorting out Sorting

- http://www.youtube.com/watch?feature=player_detailpage&v=SJwEwA5gOkM
- List all sorting algorithms from movie
- What are worst case, best case complexity? Space complexity? Other noteworthy properties?

	Type of Sorting Algorithm	Worst Case # comparisons	Worst Case # swaps	Worst Case Time	Best Case Performance	Average Case Performance	Properties
Insertion Sort	Comparison Based Sorting						
Bubblesort	Comparison Based Sorting						
Selection Sort	Comparison Based Sorting						
Binary Insertion	Comparison Based Sorting						
Shakersort	Comparison Based Sorting						
Tree Selection	Comparison Based Sorting						
Shellsort	Comparison Based Sorting						
Quicksort	Comparison Based Sorting						
Heapsort	Comparison Based Sorting						
Mergesort	Comparison Based Sorting						
Bucketsort	Integer Sorting						
Radixsort	Integer Sorting						

	Type of Sorting Algorithm	Worst Case Time	Best Case Performance	Average Case Performance	Properties
Insertion Sort	Comparison Based Sorting	**O(**	O(n)	O(	adaptive, in place, stable, online
Bubblesort	Comparison Based Sorting	**O(**	O(n)	O(	in place
Selection Sort	Comparison Based Sorting	**O(**	O(	O(	in place
Binary Insertion	Comparison Based Sorting	**O(**	O(n)	O(	adaptive, in place
Shakersort	Comparison Based Sorting	**O(**	O(n)	O(	stable, in place
Tree Selection	Comparison Based Sorting	**O(n log n)**	O(n)	O(n log n)	adaptive not in place
Shellsort	Comparison Based Sorting	**O(**	O(n log n)		in place
Quicksort	Comparison Based Sorting	**O(**	O(n)	O(n log n)	in place with O(log n) extra space
Heapsort	Comparison Based Sorting	**O(n log n)**	O(n log n)	O(n log n)	in place
Mergesort	Comparison Based Sorting	**O(n log n)**	O(n log n)	O(n log n)	not in place
Bucketsort	Integer Sorting	**O(n+k)**		O(n+k)	can be implemented such that stable
Radixsort	Integer Sorting	**O(dn)**			stable

Comparison Based Sorting

- sorting algorithm that sorts based only on comparisons
- elements to be sorted must satisfy total order properties

Reminder of a fast comparison sort: Quicksort(S)

if *S*.length() > 1 **then**
  let L, E, G be empty sequences
  $x \leftarrow$pickPivot(S)
  split(L, E, G, S, x) \\ L contains the elements smaller
    \\ than, E contains the elements equal to,
    \\ and G contains the elements greater than x
  QuickSort(L)
  QuickSort(G)
  Concatenate(L, E, G, S)
return S

Properties of Quicksort

- Worst case time complexity: $O(n^2)$
- Expected: $O(n \log(n))$
  - since, when picked at random, the pivot is likely good and splits up the sequence pretty evenly …
- Mergesort has $O(n \log(n))$ worst case time complexity, but quicksort is better in practice

How fast can we sort? A lower bound for comparison based sorting

- We prove: using a comparison based sorting algorithm, we cannot do better than $\mathrm{O}(n \log(n))$ worst case time complexity
- Therefore, $\boldsymbol{\Omega(n \log(n))}$ denotes the **lower bound** for comparison based sorting

Reminder: Big-Omega

Let $f: \mathbb{N} \to \mathbb{R}$ and $g: \mathbb{N} \to \mathbb{R}$.

$f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$.

Theorem: No comparison based sorting algorithm for n distinct elements has a worst case running time that is better than $\mathrm{O}(n \log n)$

Theorem: No comparison based sorting algorithm for n distinct elements has a worst case running time that is better than $\mathrm{O}(n \log n)$

Proof:

Theorem: No comparison based sorting algorithm for n distinct elements has a worst case running time that is better than $\mathrm{O}(n \log n)$

Proof:

- Consider a sequence S containing n distinct elements, say

  $x_0, x_1, x_2, \ldots, x_{n-1}$

Theorem: No comparison based sorting algorithm for n distinct elements has a worst case running time that is better than $O(n \log n)$

Proof:

- Consider a sequence S containing n distinct elements, say $x_0, x_1, x_2, \ldots, x_{n-1}$
- To decide the order of elements, a comparison-based algorithm compares elements pairwise—a sufficient number of times

Theorem: No comparison based sorting algorithm for n distinct elements has a worst case running time that is better than $O(n \log n)$

Proof:

- Consider a sequence S containing n distinct elements, say $x_0, x_1, x_2, \ldots, x_{n-1}$
- To decide the order of elements, a comparison-based algorithm compares elements pairwise—a sufficient number of times
- In particular, to decide which element of x_i and x_j is smaller, it answers "is $x_i < x_j$?"

Theorem: No comparison based sorting algorithm for n distinct elements has a worst case running time that is better than $O(n \log n)$

Proof:

- Consider a sequence S containing n distinct elements, say $x_0, x_1, x_2, \ldots, x_{n-1}$
- To decide the order of elements, a comparison-based algorithm compares elements pairwise—a sufficient number of times
- In particular, to decide which element of x_i and x_j is smaller, it answers "is $x_i < x_j$?"
- Depending on the outcome—i.e., yes or no—the algorithm performs either no further comparisons or it continues with more comparisons

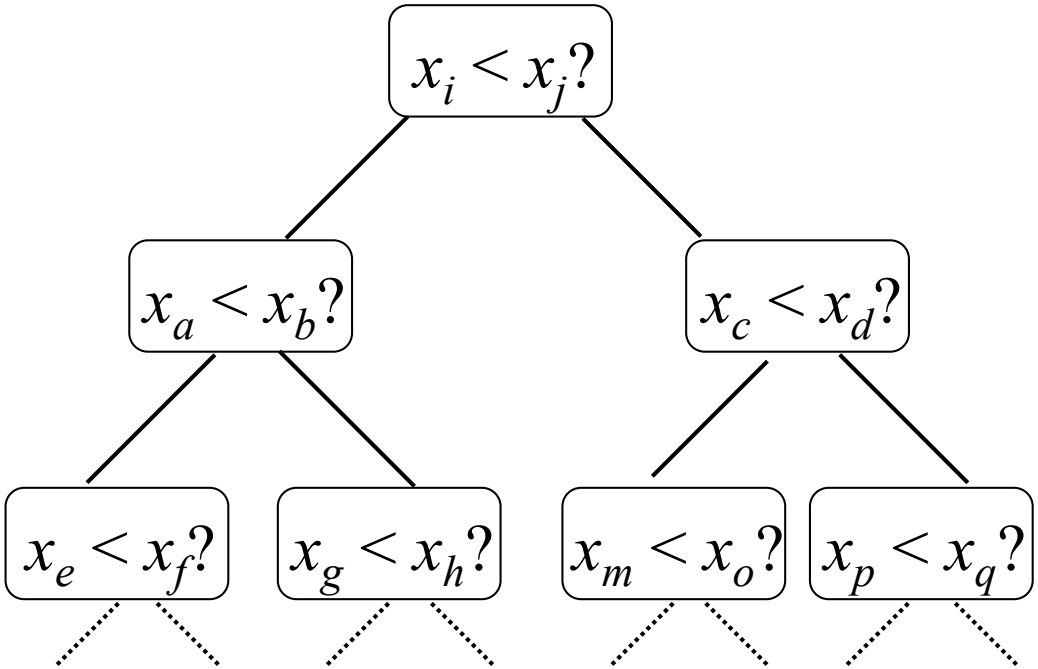
Proof (continued)

- We want to know: how good is the best of all comparison-based sorting algorithms? (Let's call it the *optimal* algorithm)
- This optimal sorting algorithm requires a certain number of comparisons (at least) to sort *any* sequence (not just the easiest input)
- We ask: How many comparisons are required for an optimal sorting algorithm to sort n elements?

Proof (continued)

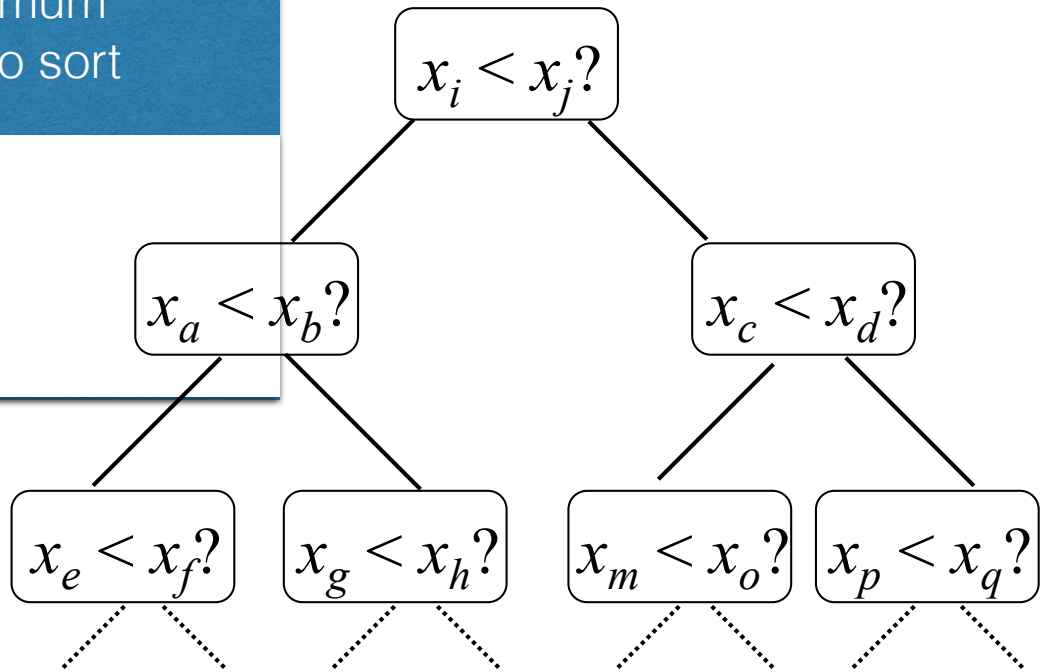
- How many comparisons are required for an optimal sorting algorithm to sort n elements?
- We consider our sequence S: $x_0, x_1, x_2, \ldots, x_{n-1}$
- The optimal algorithm will pick two elements for a first comparison, and, based on the outcome choose a second, etc.
- This can be depicted in a decision tree

Decision tree of an optimal sorting algorithm that is sorting a general sequence of elements



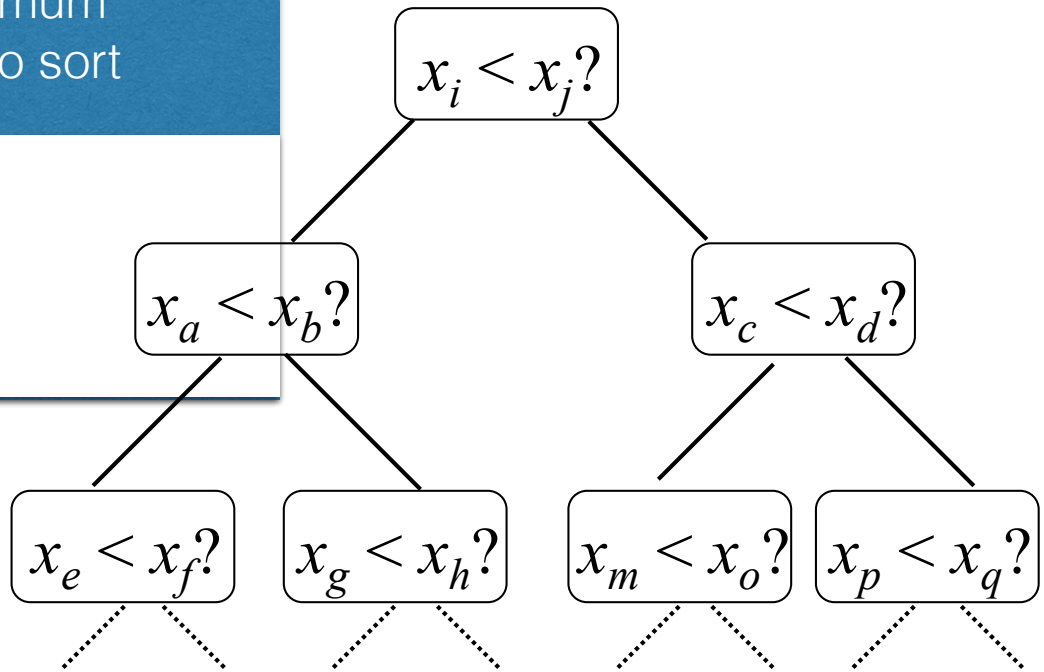
Decision tree of an optimal sorting algorithm that is sorting a general sequence of elements

- The decision tree contains every possible path the optimum algorithm might take to sort sequence S.
- How many leaves does the tree have at least?



Decision tree of an optimal sorting algorithm that is sorting a general sequence of elements

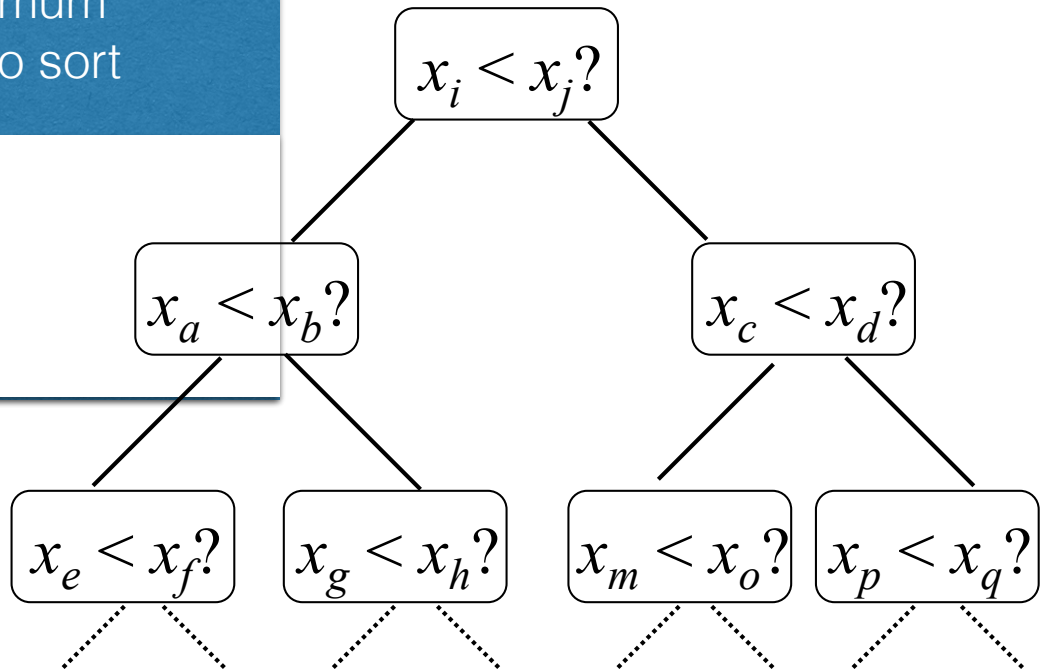
- The decision tree contains every possible path the optimum algorithm might take to sort sequence S.
- How many leaves does the tree have at least?



Since we don't know what S looks like, any permutation of S could be the sorted one. Thus, every permutation of S has to be represented by a path from the root to a leaf in the decision tree. Therefore:

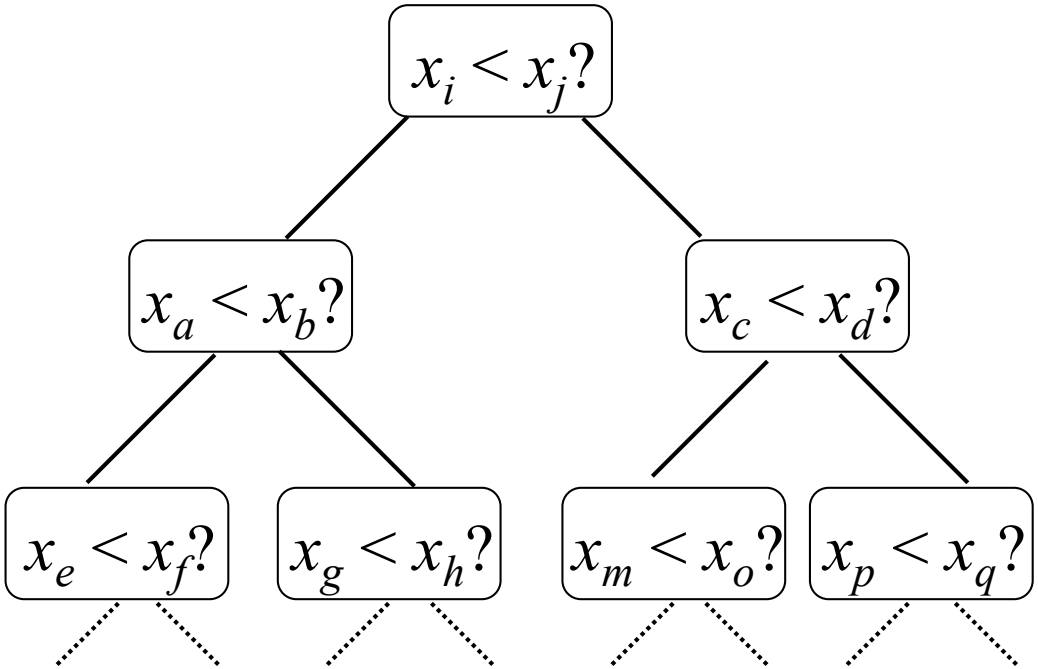
Decision tree of an optimal sorting algorithm that is sorting a general sequence of elements

- The decision tree contains every possible path the optimum algorithm might take to sort sequence S.
- How many leaves does the tree have at least?

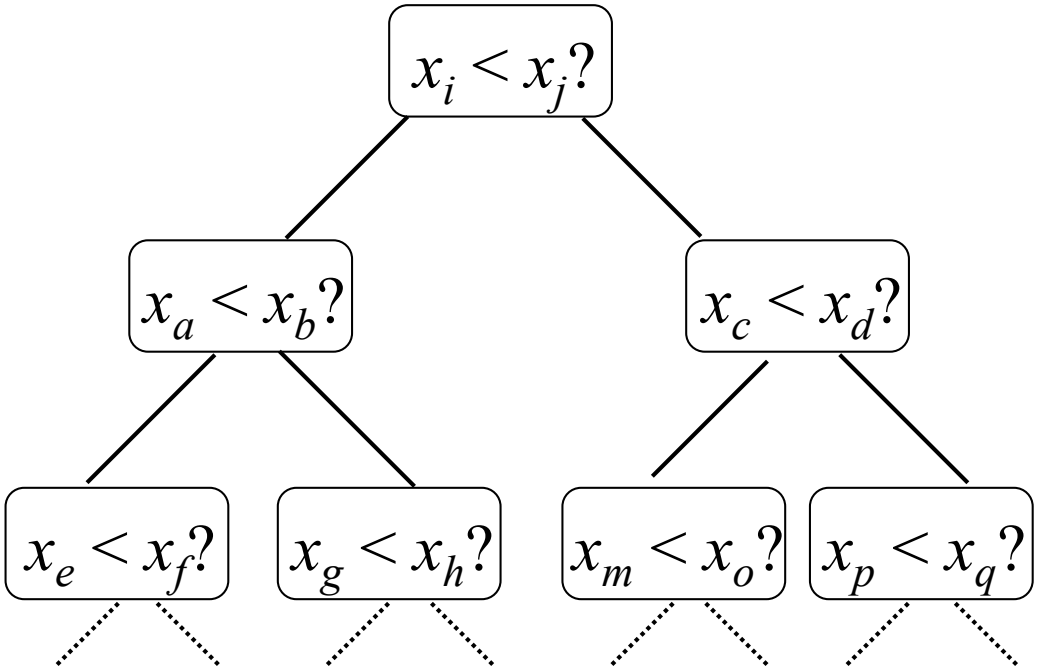


leaves ≧# possible permutations of the elements in S

Decision tree of an optimal sorting algorithm sorting a general sequence of elements



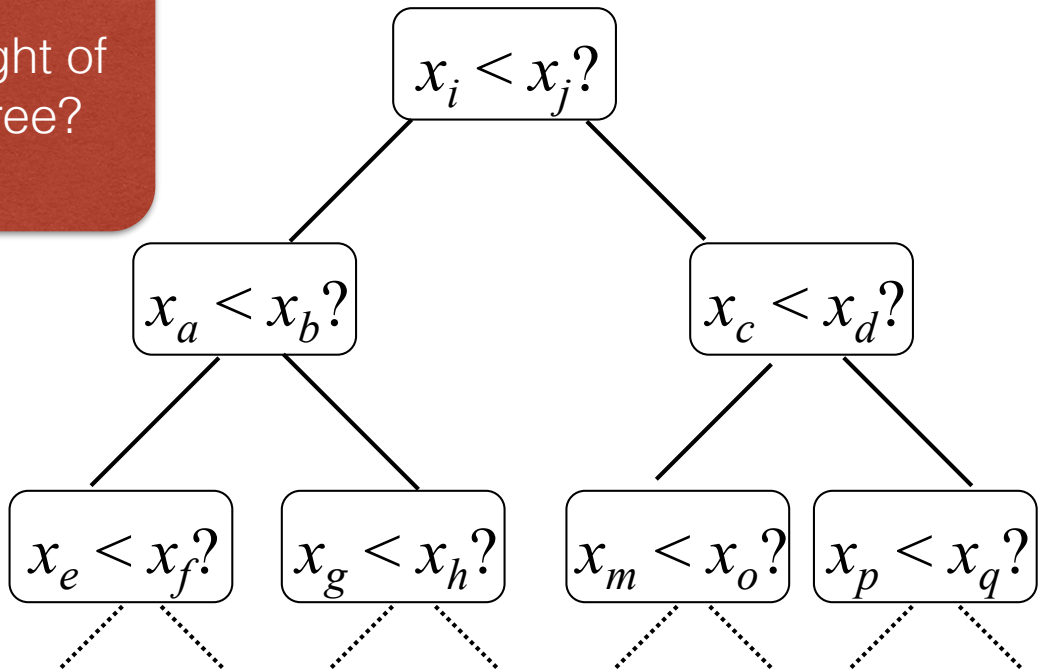
Decision tree of an optimal sorting algorithm sorting a general sequence of elements



leaves $\geq n!$

Decision tree of an optimal sorting algorithm sorting a general sequence of elements

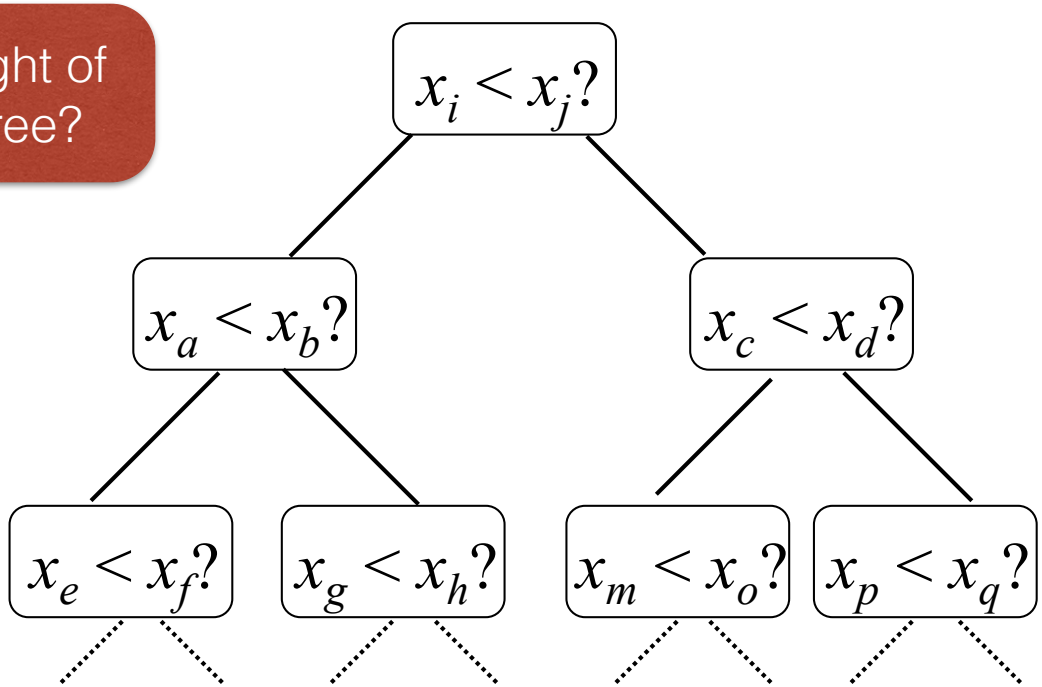
What is the height of this decision tree?



leaves $\geq n!$

Decision tree of an optimal sorting algorithm sorting a general sequence of elements

What is the height of this decision tree?



- # leaves ≧ $n!$
- tree is binary
- height is ≧ $\log(n!)$; can't be less, since the shortest binary tree that has $n!$ leaves has a height of $\log(n!)$ [definition of log]

Proof (continued)

- Since the height of the tree is at least $\log(n!)$, we know that
  - at least $\log(n!)$ worst case comparisons are required by an optimal comparison based sorting algorithm
  - at least $\log(n!)$ worst case comparisons are required by any comparison based algorithm

Proof (continued)

- What is $\Omega(\log(n!))$?
- $\log(n!) \geqq \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1)$
  $\geqq \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n/2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1)$
  $\geqq \log((n/2) \cdot \ldots \cdot (n/2))$
  $\geqq \log((n/2)^{(n/2)}) = (n/2) \log (n/2)$
- and therefore $\log(n!) \in \Omega(n \log(n))$
- We conclude: there is no comparison-based sorting algorithm that has a worst-case time complexity that is better than $O(n \log(n))$

(Using sorting to) determine the k^{th}-smallest element

- **Given**: a sequence S of n (pairwise comparable, pairwise distinct) elements, an integer k, $0 < k < n+1$
- **Task**: determine the k^{th} smallest element in S
- Solutions
  - NaiveSelect
  - QuickSelect
  - LinearSelect

Naive Select

- Sort the sequence in increasing order
- Pick the k^{th} element (which is the k^{th}-smallest one)

QuickSelect

- a variant of quicksort
- after splitting the sequence using the pivot, only focus on the (sub)sequence that must contain the k^{th}-smallest one, ignore the other

QuickSelect(S, k)

if *S*.length() $= 1$ **then return** element of S
else let L, E, G be empty sequences
    $x \leftarrow$pickPivot(S)
    split(L,G, S, x) \\ L contains the elements smaller
            \\ than and G the elements greater than x
    if $k \leq L$.length() **then return** QuickSelect(L,k)
    else if $k \leq L$.length()+1 **then return** x
    else return QuickSelect(G,k-L.length()-1)

QuickSelect is quick

- If the pivot is picked at random in algorithm QuickSelect, then the *expected* worst case time complexity of QuickSelect is O(n)
- Reason: we expect sufficiently often to pick a good pivot, which guarantees the sequences (L and G) to contain at least n/4 elements

Linear Selection

- We can do better than QuickSelect
- Choosing the pivot smart allows us to guarantee a linear time complexity in the worst case

LinearSelect(S, k)

if *S*.length() $= 1$ **then return** element of S
else let L, G be empty sequences
  $x \leftarrow$pickGoodPivot(S)
  split(L,G, S, x) \\ L contains the elements smaller
    \\ than and G the elements greater than x
  if $k \leq L$.length() **then return** LinearSelect(L,k)
  else if $k \leq L$.length()+1 **then return** x
  else return LinearSelect(G,k-L.length()-1)

PickGoodPivot(S)

- Divide S into small sequences of size 5 each
- Sort each small sequence
- Determine the median of each small sequence
- Use LinearSelect to determine the median of the medians, which is our good pivot x

Linear Selection: What is going on?

- We determine the pivot for LinearSelect using a recursive call of LinearSelect
- We have to show that x, found as described, is indeed a good pivot
- We have to show that the running time of LinearSelect is indeed linear

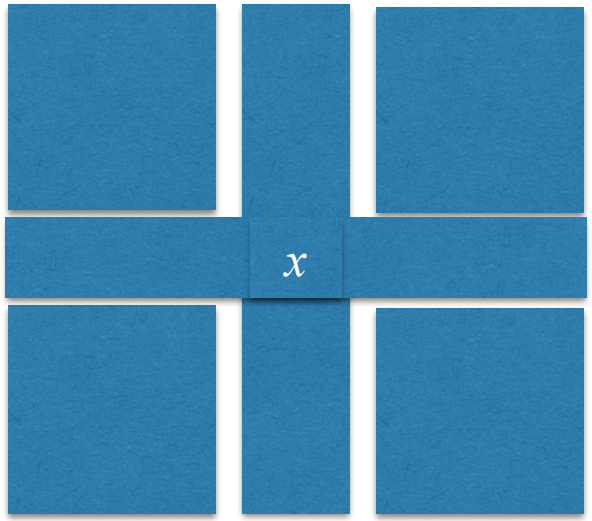
LinearSelect(S, k)

if S.length() = 1 **then return** element of S
else let L, G be empty sequences
    \\ Pick Good Pivot
    Divide S into sequences S_i of size 5 each
    for each S_i do
        sort(S_i)
        let $X[i]$ be the sequence of all medians of the S_i
    $x \leftarrow$ LinearSelect(X, $\lfloor X.\text{length}()/2 \rfloor$)
    split(L, G, S, x)
    if $k \leq L$.length() **then return** LinearSelect(L,k)
    else if $k \leq L$.length()+1 **then return** x
    else return LinearSelect(G,k-L.length()-1)

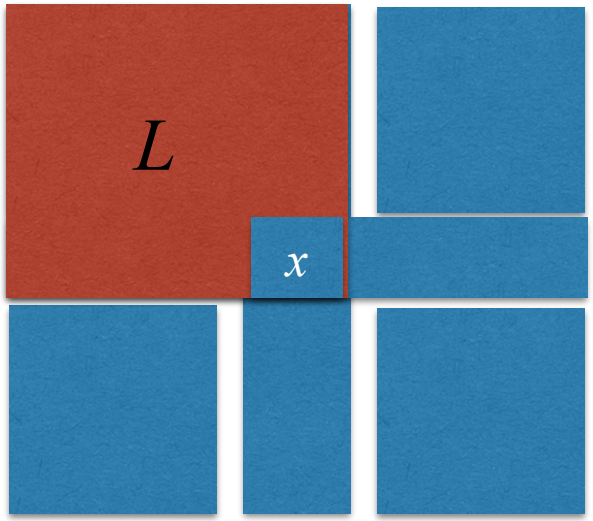
x is a good pivot

- x is the median of the medians of the $\lceil n/5 \rceil$ small sequences
- To show that x is good, we need to show lower bounds of the size of L and G after split(L, G, S, x).
- How many of the $\lceil n/5 \rceil$ medians are smaller than x, how many larger?

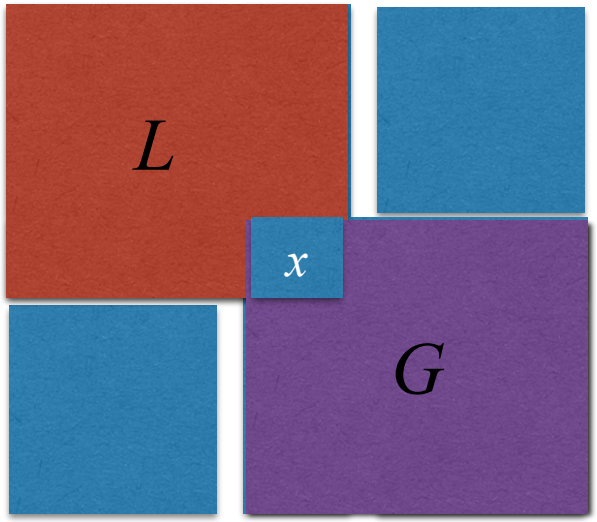
x is a good pivot



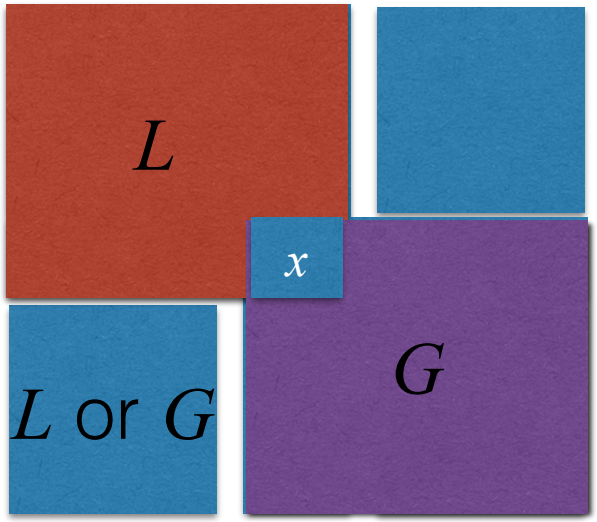
x is a good pivot



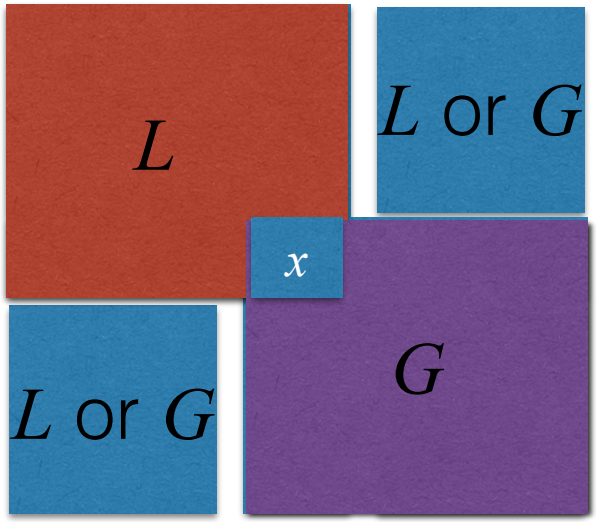
x is a good pivot



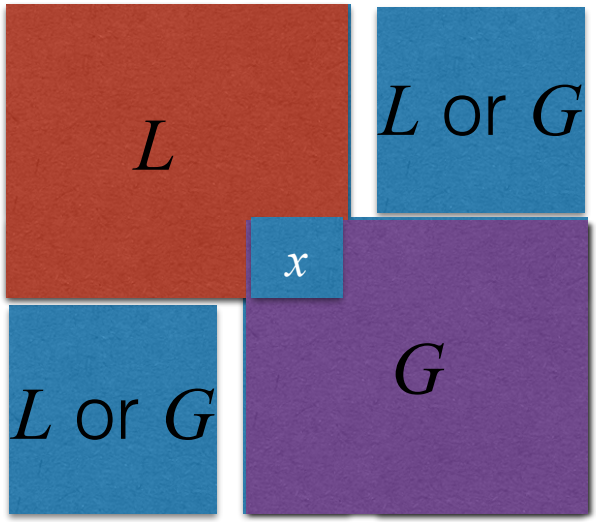
x is a good pivot



x is a good pivot

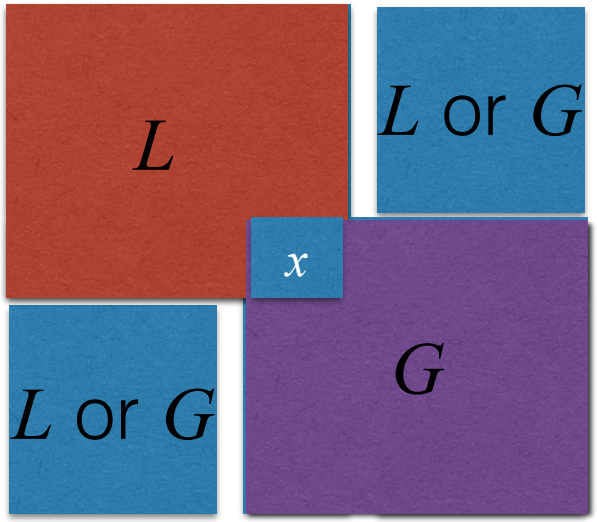


x is a good pivot



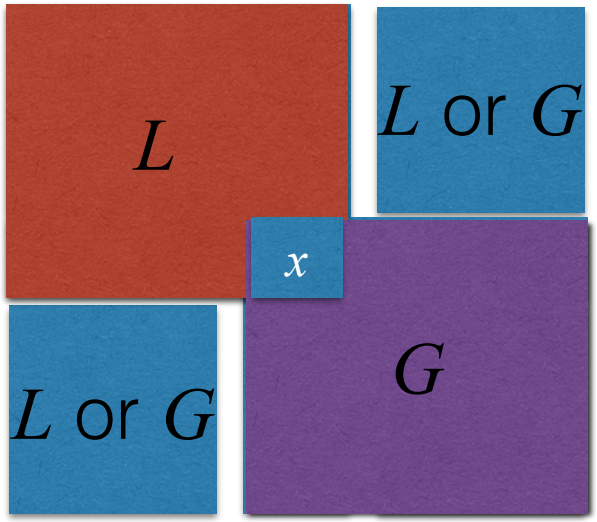
- How many of the $\lceil n/5 \rceil$ medians are smaller than x, how many larger?

x is a good pivot



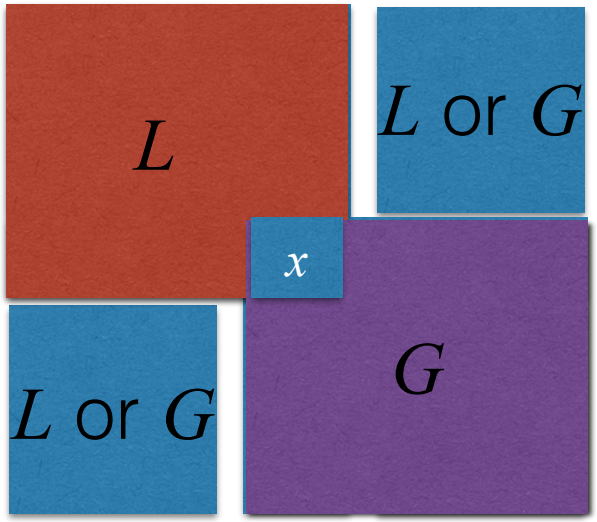
- How many of the $\lceil n/5 \rceil$ medians are smaller than x, how many larger?
- about half of medians are smaller than x, about half medians are larger

x is a good pivot



- How many of the $\lceil n/5 \rceil$ medians are smaller than x, how many larger?
- about half of medians are smaller than x, about half medians are larger
- In L and G are about $n/4$ elements each (at least)

x is a good pivot



- How many of the $\lceil n/5 \rceil$ medians are smaller than x, how many larger?
- about half of medians are smaller than x, about half medians are larger
- In L and G are about $n/4$ elements each (at least)
- Therefore we can conclude that x is a good pivot