Quicksort
Lower Bounds for Comparison Sorts

CS 4102, Algorithms

Prof. Floryan and Prof. Horton

Spring 2021

Quicksort and Partition

Readings:
CLRS Chapter 7 (not 7.4.2)

Quicksort: Introduction

- Developed by C.A.R. (Tony) Hoare (a Turing Award winner)
  http://www.wikipedia.org/wiki/C._A._R._Hoare
  - Published in 1962
- Classic divide and conquer, but…
  - Mergesort does no comparisons to divide, but a lot to combine results (i.e. the merge) at each step
  - Quicksort does a lot of work to divide, but has nothing to do after the recursive calls. No work to combine.
    - If we're using arrays. Linked lists? Interesting to think about this!
- Dividing done with algorithm often called ***partition***
  - Sometimes called *split*. Several variations.

Quicksort's Strategy

- Called on subsection of array from *first* to *last*
  - Like mergesort
- First, choose some element in the array to be the ***pivot*** element
  - Any element! Doesn't matter for correctness.
  - Often the first item. For us, the last. Or, we often move some element into the last position (to get better efficiency)
- Second, call ***partition***, which does two things:
  - Puts the pivot in its proper place, i.e. where it will be in the correctly sorted sequence
  - All elements below the pivot are less-than the pivot, and all elements above the pivot are greater-than
- Third, use quicksort recursively on both sub-lists

Quicksort is Divide and Conquer

- Divide: select pivot element p, Partition(p)
- Conquer: recursively sort left and right sublists
- Combine: Nothing!

Contrast to mergesort,
where divide is simple and combine is work

Quicksort's Strategy (a picture)

- Use last element as pivot (or pick one and move it there)

	pivot
first	***last***

- After call to partition…

<= pivot (unsorted)	**pivot**	**> pivot (unsorted)**
first	***split point***	***last***

- Now sort two parts recursively and we're done!

<= pivot (sorted)	**pivot**	**> pivot (sorted)**
first	***split point***	***last***

- Note that splitPoint may be anywhere in *first..last*
- Note our assumption that all keys are distinct

Quicksort Code

Input Parameters: *list, first, last*

Output Parameters: *list*

```
def quicksort(list, first, last):
    if first < last:
        q = partition(list, first, last)
        quicksort(list, first, q-1)
        quicksort(list, q+1, last)
    return
```

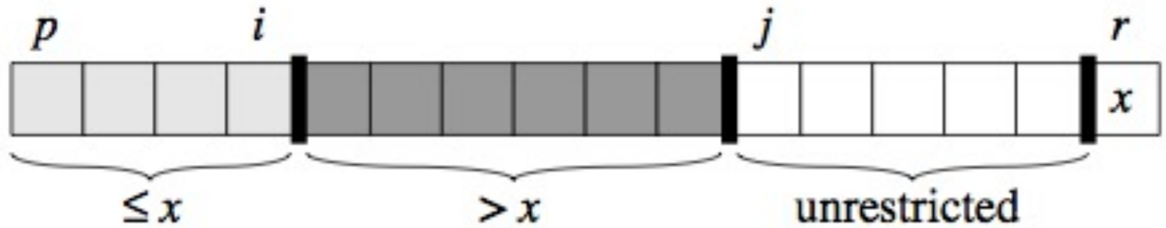
Partition Does the Dirty Work

- Partition rearranges elements
  - How? How many comparisons? How many swaps?
- How? Two well-known algorithms
  - In this chapter of CLRS, Lomuto’s algorithm
  - In the exercises, the original: Hoare’s algorithm. (Page 185. Look at on your own.)
  - Important:
    - Both are in-place!
    - Both are linear.

Strategy for Lomuto's Partition

- Invariant: At any point:
  - *i* indexes the right-most element <= *pivot*
  - *j-1* indexes the right-most element > *pivot*

Note: book uses *p* and *r* for what we called *first* and *last* in earlier slides



- Strategy:
  - Look at next item *a[j]*
  - If that item > pivot, all is well!
  - If that item < pivot, increment *i* and then swap items at positions *i* and *j*
  - When done, swap pivot with item at position *i+1*
- Number of comparisons: n-1

Efficiency of Quicksort

- Partition divides into two sub-lists, perhaps unequal size
  - Depends on value of pivot element
- Recurrence for Quicksort

  T(n) = partition-cost +

  T(size of 1st section) + T(size of 2nd section)
- If divides equally, $T(n) = 2\,T(n/2) + n-1$
  - Just like mergesort
  - Solve by substitution or master theorem

    $T(n) \in \Theta(n \lg n)$
- This is the best-case. But…

Worst Case of Quicksort

- What if divides in most unequal fashion possible?
  - One subsection has size 0, other has size n-1
  - T(n) = T(0) + T(n-1) + n-1
  - What if this happens every time we call partition recursively?

$$W(n) = \sum_{k=2}^{n}(k-1) \in \Theta(n^2)$$

  - Uh oh. Same as insertion sort.
    - "Sorry Prof. Hoare – we have to take back that Turing Award now!"

Quicksort's Average Case

- Good if it divides equally, bad if most unequal.
  - Remember: when subproblems size 0 and n-1
  - Can worst-case happen?
    Sure! Many cases. One is when elements already sorted. Last element is max, pivot around that. Next pivot is 2nd max…
- What's the average?
  - Much closer to the best case
  - A bad-split then a good-split is closer to best-case (pp. 176-178)
  - To prove A(n), fun with recurrences!
  - The result: If all permutations are equal, then
    $A(n) \cong 1.386\ n \lg n$ (for large n)
- So very fast on average.
- And, we can take simple steps to avoid the worst case!

Avoiding Quicksort's Worst Case

- Make sure we don't pivot around max or min
  - Find a better choice and swap it with last element
  - Then partition as before
- Recall we get best case if divides equally
  - Could find median. But this costs $\Theta(n)$. Instead…
  - Choose a **random element** between first and last and swap it with the last element
  - Or, estimate the median by using the "median-of-three" method
    - Pick 3 elements (say, first, middle and last)
    - Choose median of these and swap with last. (Cost?)
    - If sorted, then this chooses real median. Best case!

Tuning Quicksort's Performance

- In practice quicksort runs fast
  - A(n) is log-linear, and the "constants" are smaller than mergesort and heapsort
  - Often used in software libraries
  - So worth tuning it to squeeze the most out of it
  - <u>Always</u> do something to avoid worst-case
- Sort small sub-lists with (say) insertion sort
  - For small inputs, insertion sort is fine
    - No recursion, function calls
  - Variation: don't sort small sections at all.
    After quicksort is done, sort entire array with **insertion sort**
    - It's efficient on almost-sorted arrays!

Quicksort's Space Complexity

- Looks like it's in-place, but there's a *recursion stack*
  - Depends on your definition: some people define *in-place* to **<u>not</u>** include stack space used by recursion
    - E.g. our CLRS algorithms textbook
    - Other books and people do "count" this
  - How much goes on the stack?
    - If most uneven splits, then $\Theta(n)$.
    - If splits evenly every time, then $\Theta(\lg n)$.
- Ways to reduce stack-space used due to recursion
  - Various books cover the details (not ours, though)
  - First, remove 2nd recursive call (tail-recursion)
  - Second, always do recursive call on smaller section

Summary: Quicksort

- Divide and conquer where divide does the heavy-lifting
- In worst-case, efficiency is $\Theta(n^2)$
  - But it's practical to avoid the worst-case
- On average, efficiency is $\Theta(n \lg n)$
- Better space-complexity than mergesort.
- In practice, runs fast and widely used
  - Many ways to tune its performance
- Various strategies for Partition
  - Some work better if duplicate keys
- More details? See Sedgewick's algorithms textbook
  - He's the expert! PhD on this under Donald Knuth

Lower Bounds Proof for Comparison Sorts

Readings: CLRS Section 8.1

Mental Stretch

Show $\log(n!) \in \Theta(n \log n)$

Hint: show $n! \leq n^n$

Hint 2: show $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

$\log n! \in O(n \log n)$

$$\begin{array}{cccccccccc} n! = & n & \cdot & (n-1) & \cdot & (n-2) & \cdot \ldots \cdot & 2 & \cdot & 1 \\ & \parallel & & \wedge & & \wedge & & \wedge & & \wedge \\ n^n = & n & \cdot & n & \cdot & n & \cdot \ldots \cdot & n & \cdot & n \end{array}$$

$n! \leq n^n$

$\Rightarrow \log(n!) \leq \log(n^n)$

$\Rightarrow \log(n!) \leq n \log n$

$\Rightarrow \log(n!) \in O(n \log n)$

$\log n! \in \Omega(n \log n)$

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2}-1\right) \cdot \ldots \cdot 2 \cdot 1$$

$$\vee \qquad \vee \qquad \vee \qquad \parallel \qquad \vee \qquad \vee \quad \parallel$$

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2} \cdot 1 \cdot \ldots \cdot 1 \cdot 1$$

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

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

$$\Rightarrow \log(n!) \geq \frac{n}{2} \log \frac{n}{2}$$

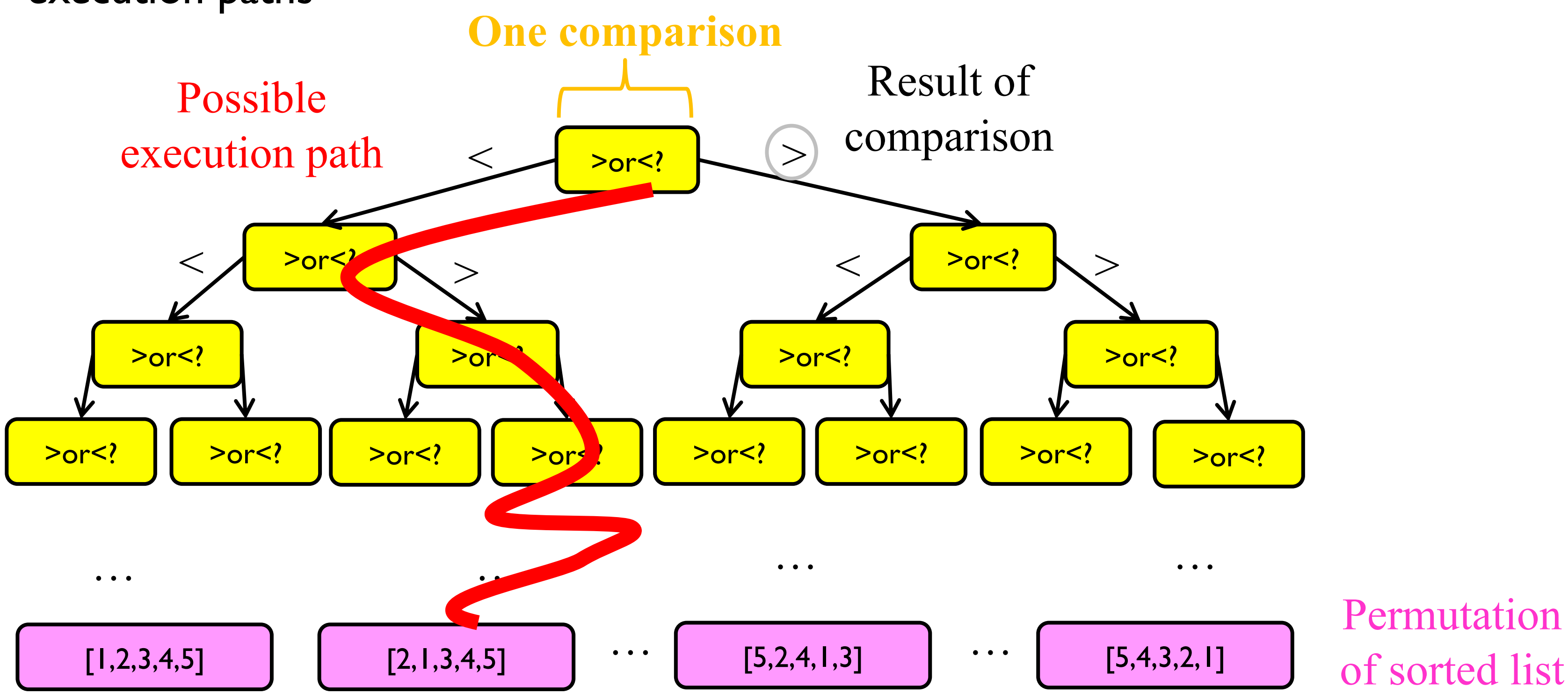
$$\Rightarrow \log(n!) \in \Omega(n \log n)$$

Worst Case Lower Bounds

- Prove that there is no algorithm which can sort faster than $O(n \log n)$
- Non-existence proof!
  - Seems like maybe it would be very hard to do… (?)

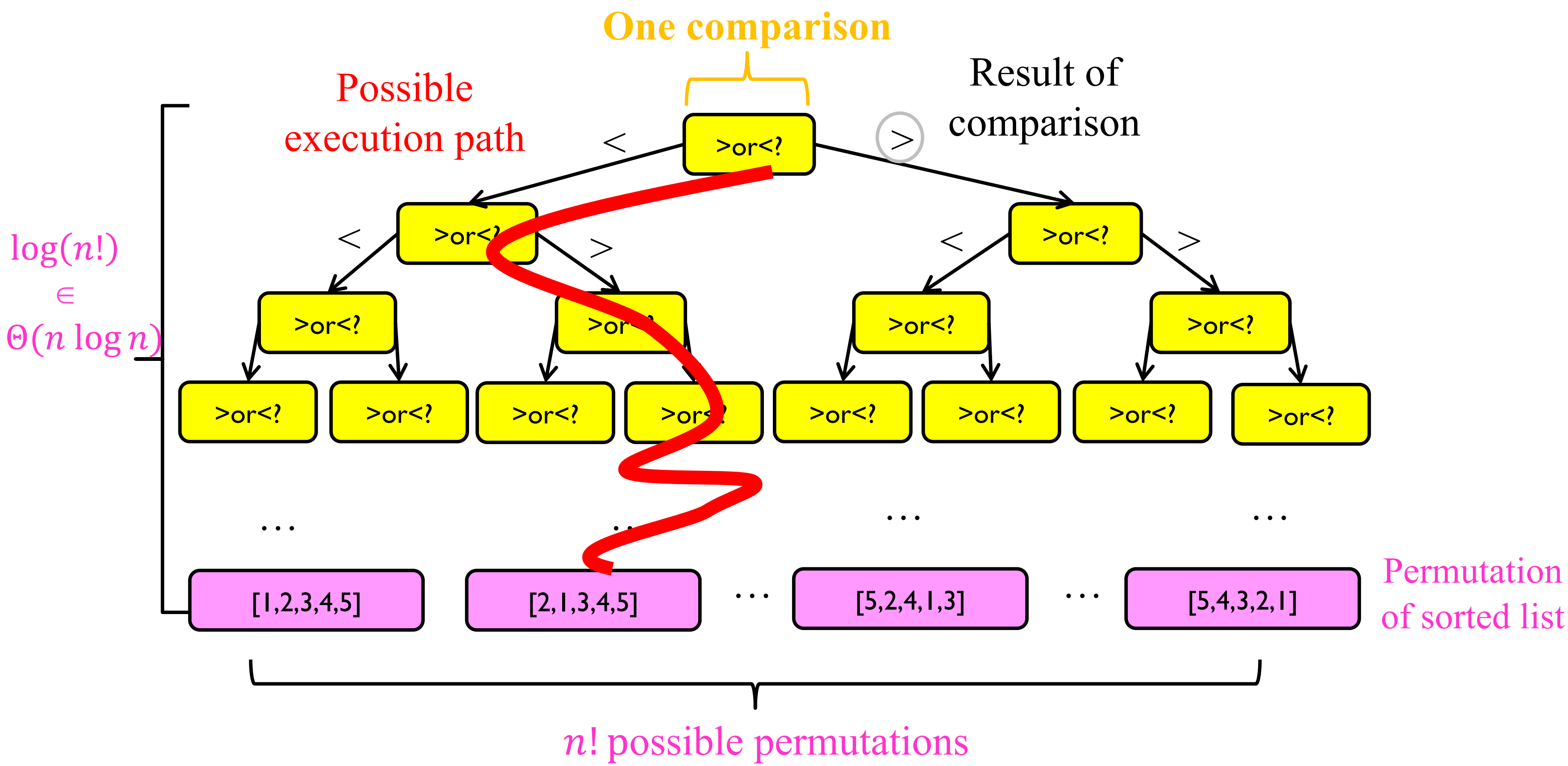
Strategy: Decision Tree

- Sorting algorithms use comparisons to determine the order of input elements
- Conceptually possible to draw a tree to illustrate all possible execution paths



Strategy: Decision Tree

- Worst case run time is the longest execution path
- i.e., "height" of the decision tree



Lower Bound for Worst Case

- Binary tree property: At level d in a binary tree, there are at most 2^d nodes (where level of root is 0)
- Also, let's say a tree's height is number of levels minus one
  - Height of our decision tree is the W(n) number of comparisons
- Theorem 8.1 (p. 193):
  - Let L be the number of leaves in a binary tree and let h be its height. (Book uses lower-case l, not L like we do here.)
  - Then $L \le 2^h$. (Number of leaves is no more than 2^h.)
  - Therefore $h \ge \lceil \lg L \rceil$ (Height is not less than…)
  - For a correct sorting algorithm, L >= n!
  - Therefore
$$h \ge \lceil \lg L \rceil \ge \lceil \lg n! \rceil$$
- Thus, for any algorithm that sorts by comparison of keys W(n) is at least $\lceil \lg n! \rceil$

Formula for the Lower Bound

- Earlier we showed this was $\Theta(n \lg n)$
- Or, we can we lose that factorial in other ways
  - Stirling's formula: $(n/e)^n \text{ sqrt}(2\pi n)$
    - Take the log of this approximation of n! and you'll see that it's $\Theta(n \lg n)$
  - Better to re-write, use integrals, and…
    - See a textbook for details (but not ours)
- If you were to do all this, you'd see:

$$W(n) \geq \lceil \lg n! \rceil \geq \lceil n \lg n - 1.443n \rceil$$

  which is of course $\Theta(n \lg n)$
- FYI Mergesort is very close to optimal
  - But not for all values of n

Summary

- Our lower-bound proof shows any algorithm must be $\Omega(n \lg n)$ in the worst-case **<u>if</u>** it works by comparing keys
  - Algorithms that only do key-comparisons can sort any data type
  - Algorithms that can calculate on their keys can do better
    - E.g. counting sort and radix sort for numbers (Ch. 8 of CLRS)
  - In the same way that binary search is optimal, but hashing can be faster
- Mergesort and Quicksort are in this order-class
  - Mergesort is very close to the L.B. (but not in-place)
  - But quicksort will run faster generally
    - Why? Constants and lower-order terms are smaller.
      In other words, the overhead per comparison is less.
  - But Quicksort really could be $\Theta(n^2)$ at its worst