CS 2100: Data Structures & Algorithms 1

Advanced Sorts (Part II)

Quicksort; Discussion on Hybrid Algorithms

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

{Reminder} How to Sort?

- Some “straightforward” sorting algorithms
  - Insertion Sort, Selection Sort, Bubble Sort
  - **Each is O(n^2)**

- More efficient sorting algorithms
  - Quicksort, Mergesort, Heapsort
  - **Each is O(n log n)**

Best Sorts are O(n log n)

Quicksort

Another divide-and-conquer style algorithm

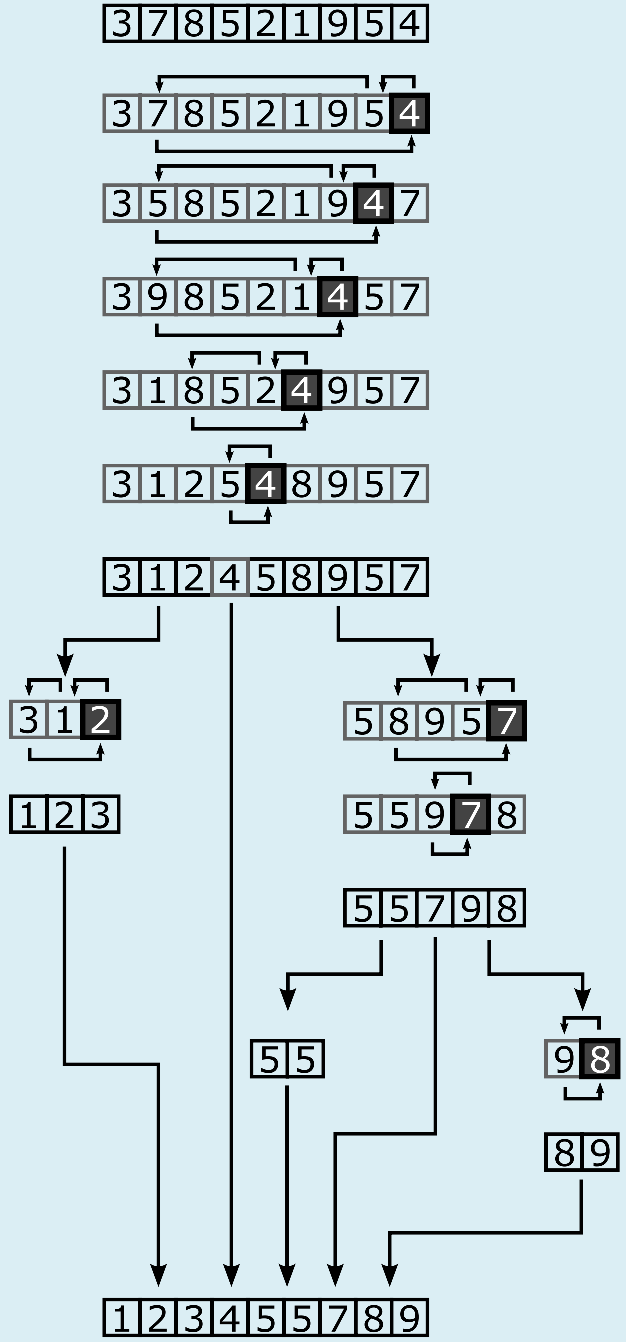
Quicksort Introduction

- Similar to **Mergesort**, except the "work" is done during the dividing instead of in the merging
- Is ***recursive***
- Another example of a **divide-and-conquer** algorithm

Quicksort: Overall Idea

- **Idea**: Select an item in the list to be a **pivot** value.
- Divide the list into two halves
  1. Items **less** than pivot and recursively sort
  2. Items **greater** than pivot and recursively sort
- "**merge**" by concatenating `lessList,pivot,greaterList`
- return

Quicksort



Quicksort Pseudo-Code:

```
quickSort(list, i, j)
  /* BASE CASE GOES HERE */

  //partition list and return index of pivot
  int pivot = partition(list, i, j);
  quickSort(list, i, pivot-1)
  quickSort(list, pivot+1, j)
```

```
i <-- low index in array
j <-- high index in array
```

Example call (assume “list” is array):

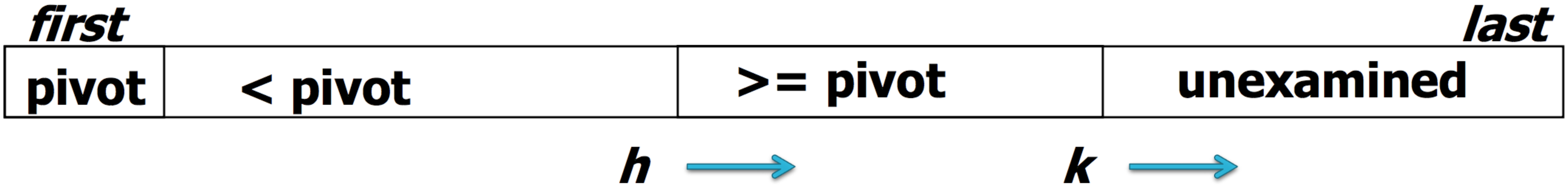
quickSort(list, 0, size - 1);

Quicksort: Partition

- **Partition** is responsible for:
  - Selecting a pivot value
  - re-arranging list so that
    - pivot in correct place
    - items **less** than pivot are below
    - items **greater** than pivot are above

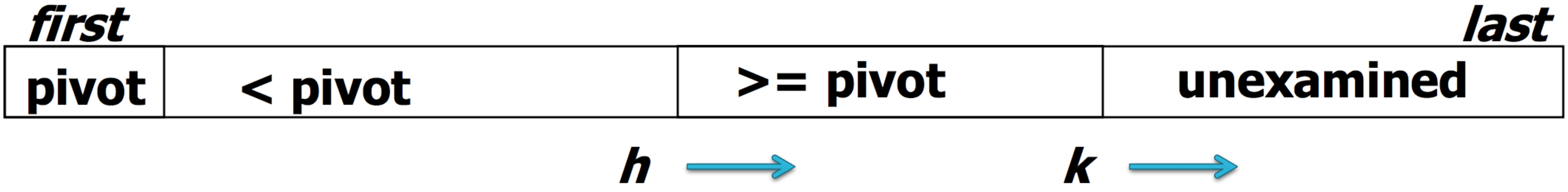
- **Two approaches:**
  - Hoare's Partition
  - Lomuto's Partition

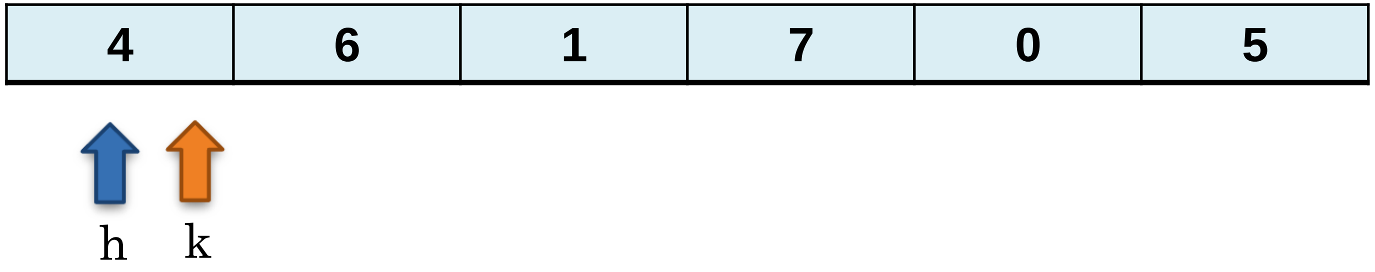
Quicksort: Lomuto's Partition



- **Strategy:**
  - Increment `k`, look at `A[k]`
  - If `A[k] > pivot`, all is well
  - Otherwise, `h++` and swap `k` and `h`
  - When done, swap `h` and `pivot` to place pivot in correct spot
    - **Done?** Unexamined portion disappears (k gets to end),
    - and h divides items < pivot and items > pivot

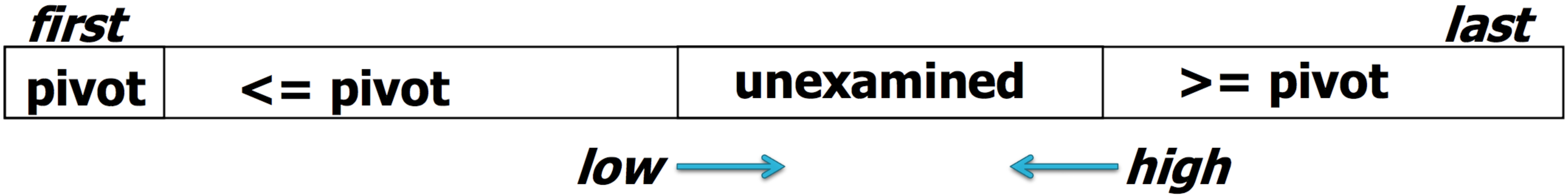
Quicksort: Lomuto's Partition





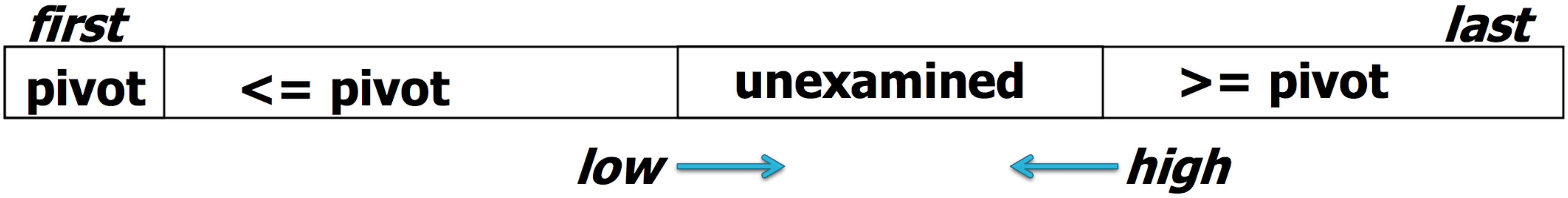
- **Strategy:**
  - Increment `k`, look at `A[k]`
  - If `A[k] > pivot`, all is well
  - Otherwise, `h++` and swap `k` and `h`
  - When done, swap `h` and `pivot` to place pivot in correct spot
    - **Done?** Unexamined portion disappears (k gets to end),
    - and h divides items < pivot and items > pivot

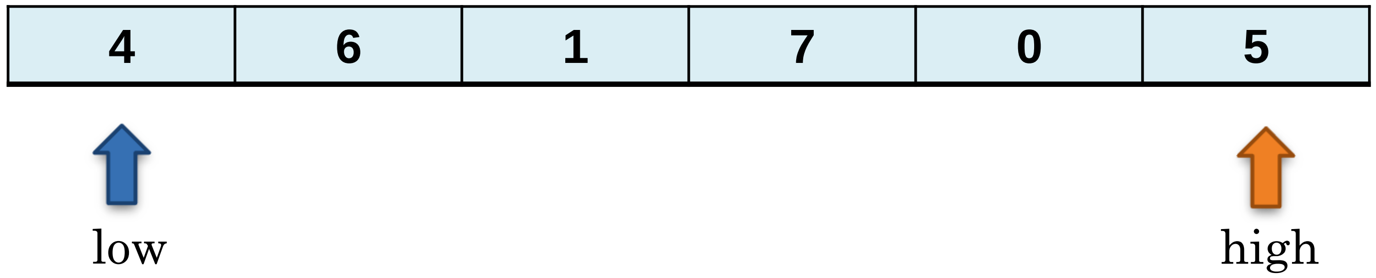
Quicksort: Hoare's Partition



- **Strategy:**
  - Move `low` up until `something > pivot` found
  - Move `high` down until `something <= pivot` found
  - Swap items at `low` and `high`
  - When done, swap items at `high` and `pivot` to put pivot in place

Quicksort: Hoare's Partition





- **Strategy:**
  - Move `low` up until `something > pivot` found
  - Move `high` down until `something <= pivot` found
  - Swap items at `low` and `high`
  - When done, swap items at `high` and `pivot` to put pivot in place

Analysis of Quicksort

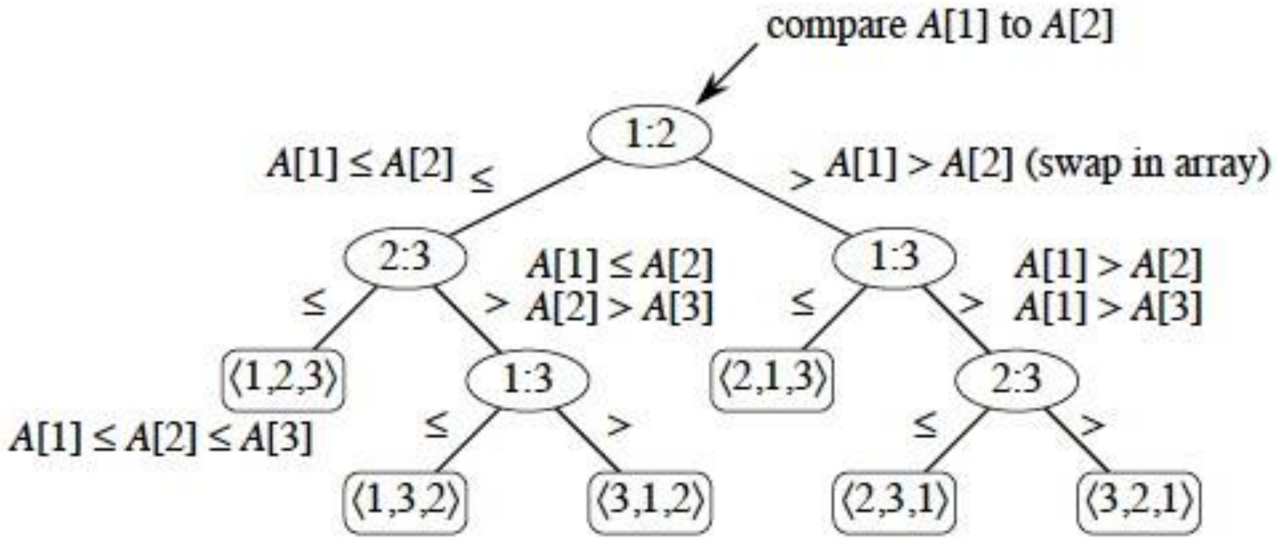
- It is **in-place** (if you don't count the recursive bookkeeping)
  - It doesn't use scratch array like mergesort usually does
- Same runtime analysis as mergesort
  - T(n) = 2T(n/2)+n = **Θ(nlog(n))**
  - Caveat to this: See next slide

Analysis of Quicksort: Worst Case

- Technically, we could pick a very bad pivot every time.
  - A bad pivot means the **list isnot split in half**. Worst case split into sizes 0 and n-1
- So $T(n) = T(n-1) + n =$ $\mathbf{\Theta(n^2)}$
- This is NOT VERY LIKELY
  - In addition, *some advanced techniques can be used to ensure it never happens.*

Lower Bound Proof

Discussion:
Best Sorting Algorithm: Decision Tree



Discussion:
Best Sorting Algorithm: Decision Tree

- The "best" decision tree must exist (i.e., there is **SOME** best algorithm)
- The number of leaves `L >= n!`
  - Because list has **n!** permutations
- So, the height of the "best" decision tree is the best possible runtime for a sorting algorithm.

- For a binary tree, $L \leq 2^h$
  - **L** is number of leaves
  - **h** is height of tree
- Solve for h:
  - $h \geq \log_2(n!)$

Discussion:
Best Sorting Algorithm: Decision Tree

- For a binary tree, `L <= 2^h` ($L \le 2^h$)
  - **L** is number of leaves
  - **h** is height of tree
- Solve for h:
  - $h \geq \log_2(n!)$
- For now, just trust me...but:
  - `log(n!) = Θ(n*log(n))`
- Thus, any algorithm that sorts by comparing keys must be **Ω(n*log(n))**

Hybrid Sorts &
Other Sorting Algorithms

Hybrid Sorts

- Some sorting algorithms (like Java's internal one) will look at **properties** of the list and **call different algorithms** depending on the situation.
- For example:
  - **Insertion sort** is faster than merge/quick on smaller lists
  - **Insertion sort** is faster on almost sorted lists

- Strategy:
  - **Switch to insertion sort once recursive calls get small** (small could be ~100-150 elements; or even down to 30-50 elements) or on an almost sorted list → **speedup!**
  - You could start with **quicksort** or **mergesort** which is **log-linear** time, and stop when the size of the list is small (e.g. 30-40) then switch to ***insertion sort*** (although **quadratic**, it is *faster on smaller lists!)* In the base case, check if size < **threshold** (instead of 1) if so, call ***insertion sort!***

Other Sorting Algorithms

- There are MANY more… but to name a few…

- **Heap Sort:** We haven't seen this data structure, so we will study this a little later
- **Radix Sort:** Uses values of digits to sort numbers very quickly.
- **TimSort:** What Java `Collections.sort()` uses
- ...and many others.