# CS 209
# Data Structures and Mathematical Foundations

04 / 08 / 2024

Instructor: Michael Eckmann

# Today's Topics

- Questions?/Comments?
- Heapsort
- Quicksort

# Heapsort

- The algorithm is simply:
  - start with n unsorted data items
  - create a maxHeap (of size n) out of these items --- store it as a list
  - set i = n -1
    - swap the root (index 0) and last node (index i)
    - reheapify (downward reheapification) the tree that starts at the root (index 0) and goes to i-1 (do not include the nodes at i and higher in the new heap)
    - i = i-1
  - do the above 3 steps until the size of the tree we are heapifying is one (i=0)
  - The list is now sorted from low to high.
- Let's go through an example.

# Next assignment

- Your next assignment (to be officially assigned within the next couple of days) --- will involve implementing heapSort among other tasks

# Quicksort

- Does anyone remember MergeSort?  What kind of algorithm was that?

# Quicksort

- Does anyone remember MergeSort? What kind of algorithm was that?
- Divide and Conquer
- It divided the list into equal sized halves and did MergeSort on each half then it combined the two halves.
  - Any recollection on what it did to combine/conquer the two halves?

# Quicksort

- Does anyone remember MergeSort? What kind of algorithm was that?
- Divide and Conquer
- It divided the list into equal sized halves and did MergeSort on each half then it combined the two halves.
  - Any recollection on what it did to combine/conquer the two halves?
    - created a new list of same size and got the original halves merged into this new list so that the new list was sorted. Then the new list was copied back.

# Quicksort

- Quicksort is a Divide and Conquer sorting algorithm as well.
- It has a few distinctions from MergeSort though.
- Quicksort
  - divides the list into 2 portions at each step, but these two portions aren't necessarily the same size
  - a pivot value is chosen and the elements are divided into two sublists – one sublist containing elements less than the pivot and the other sublist containing elements greater than or equal to the pivot
  - Also, it doesn’t have the extra space requirement that MergeSort has.  QS space: O(1), MergeSort space: O(n)

# Quicksort

- Quicksort algorithm
  - 1) if size of list, L is 0 or 1, return
  - 2) pick some element in list as a pivot element
  - 3) divide the remaining elements (minus the pivot) of L into two groups, $L_1$, those with elements less than the pivot, and $L_2$, those with elements greater than or equal to the pivot
  - 4) sorted list is: (Quicksort($L_1$) followed by pivot, followed by Quicksort($L_2$))
- Depending on which is the pivot element, the sizes of the two sides could differ greatly.
- Unlike MergeSort, Quicksort does not guarantee equal size portions to sort (which is bad.) But, the divide stage can be done in-place (without any additional space, which is good.)

# Quicksort

- Choosing the pivot
  - to pick some element in list as a pivot element we can
    - pick the first (bad if list is almost sorted, why?)
    - pick a random one (random # generation could be time consuming – high constant factor)
    - pick the pivot that is the median of 3 elements (say the median of the first, middle and last element)
      - not much extra work
      - the almost sorted case isn't a problem for this

# Quicksort

- Divide strategy – how to divide our list into two sublists of less than pivot and greater than pivot (assume all elements distinct for now)
- The following strategy gives good results.

# Quicksort

- Divide strategy
  - 1) swap the pivot with the last in the list
  - 2) start index i pointing to first in list and index j to next to last element
  - 3) while (element at i < pivot)
    increment i
  - 4) while (element at j >= pivot)
    decrement j
  - 5) if (i < j)  element at i is > pivot and element at j is < pivot so, we swap them and repeat from step 3.
  - 6) when i > j, we swap the pivot that is in the last place with the element at i.

Let's see an example: 4,20,5,7,16,18,22,3,1,8,25,30,40,15 (and let's always choose the last element as the pivot, first time it is 15)

# Quicksort

- Let's write Quicksort
  - We can make quicksort be a recursive method that takes in
    - a list
    - the starting index of the data to be sorted
    - the ending index of the data to be sorted
  - quicksort can partition the elements
    - find a pivot and divide a (portion of a) list into elements less than pivot, followed by pivot, followed by elements greater than pivot
    - This can be a function that will take in
      - a list
      - the starting index of the data to be sorted
      - the ending index of the data to be sorted
    - and it will return the pivot index and alter the order of the elements of a subset of the list passed in

# Quicksort

- A typical speedup for Quicksort is to do the following:
  - when we get down to some small number of elements (say 10) in our list, instead of using quicksort on them, we do insertion sort.

- How would we alter the code we just wrote to do insertion sort when the number of elements to sort is small?