Lesson #5

September 17, 2019

Warm up #1 (5 min)

- Description
  - You are a junior developer with GoDaddy's Infrastructure team. You need to implement a modulo operator (% in javascript), which will be used to generate public/private keys.
- Task
  - Implement a function `hashData(base, input)`. hashData returns the remainder of a number after division by base.
    - `hashData(10, 12)` returns 2; `hashData(11, 15)` returns 4; and so on.
  - Don't forget guard clauses!
- Constraint
  - This is bare metal programming so you don't have access to Math library (no division or modulo operator). You can increment and compare numbers.

Warm up #2 (5 min)

- Description
  - You're a software developer with Dropbox.. Your team is responsible for reducing storage costs by reducing the number of duplicate files (aka "deduplication") across thousands of machines.
  - Each machine holds a collection of files, eg Box_A holds {A, G, M, P}, Box_B holds {A, B, N, Q}, Box_C holds {B, F, X}.
- Task
  - Implement a function `findDuplicates(data)` that returns a list of duplicate files.
  - For example, if we applied `findDuplicates` to Box_A, Box_B, and Box_C, it should return {A, B}, which are the files that show up more than once.
- Bonus:
  - How would you implement findUniques(data) in constant time?

Warm up #3 (5 min)

- Description
  - You work for Cloudera, a big data company, that develops tools for large scale data processing
- Task
  - Implement function `splitData(data)`
    - Given `arr = [10, 15, 23, 24, 76, 1, 5, 7],` then `splitData(data)` returns `[10], [15], [23], [24], [76], [1], [5], [7]`
- Constraints
  - You can distribute work across as many machines as you'd like (assume no I/O cost), but we'd like to reduce wall clock time

Warm up #4 (5 min)

- Description
    - You work for Cloudera, a big data company, that develops tools for large scale data processing
- Task
    - Implement function `reduce(dict1, dict2)`
    - Inputs are dictionaries eg {“jessica”: 2, “daniel”: 3} and {“jessica”: 6, “rinat”: 3}
    - Output is a single, merged dictionary {“jessica”: 8, “rinat”: 3, “daniel”: 3}

Congrats, you just implemented map reduce

Warm up #6 (2 min)

- Description
  - You work for Cloudera, a big data company, that develops tools for large scale data processing
- Task
  - Implement function `reduce(data_1, data_2)` that combines two sorted lists into a single one.
    - Inputs are lists of sorted numbers, eg data_1 = [2, 5, 15] and data_2 = [3, 4, 20]
    - Output is a single sorted list, eg [2, 3, 4, 5, 15, 20]

In the previous episode...

- Reviewed recursion, one of the most fundamental concepts in computer science
    - Explored the connection between recursion, memoization, and dynamic programming
    - Most interesting algorithms (sorting, tree traversals, search) are recursive so make sure you understand it

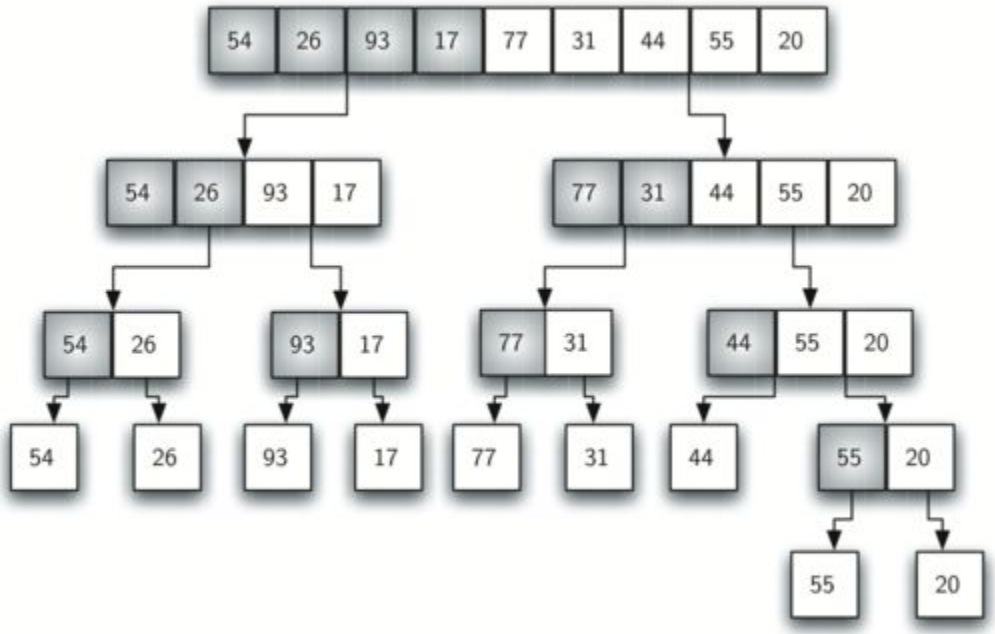
Agenda for tonight

- Sorting: merge sort, quick sort, insertion sort (again)

Merge Sort

- It’s straightforward to disassemble a single list into constituents (we just did it)
- It’s straightforward to combine/reduce two sorted lists (we just did it)

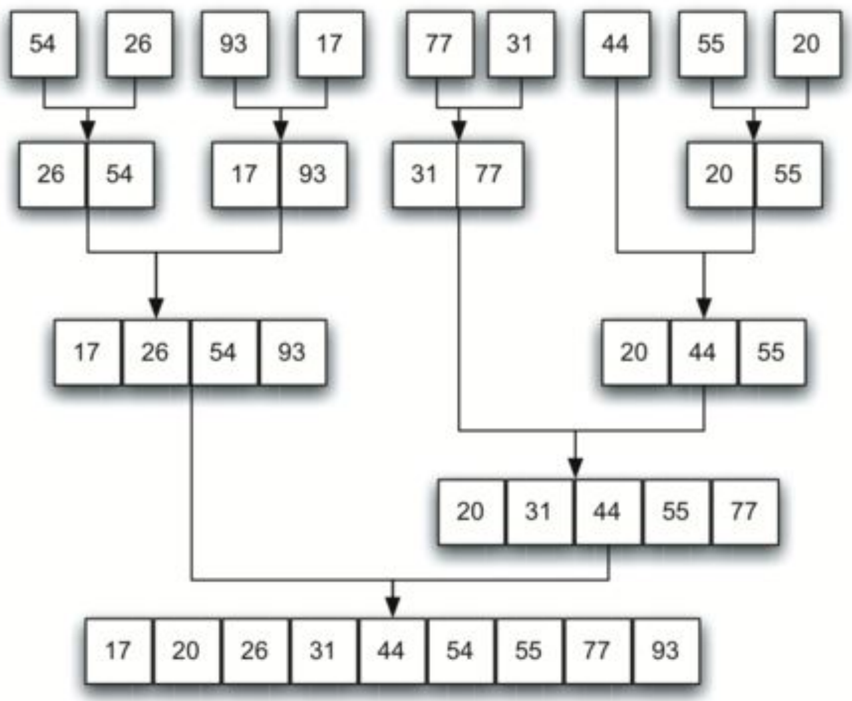
First we split...



Why sort recursively, instead iteratively?

(Hint: we talked about this earlier)

Then we sort and merge

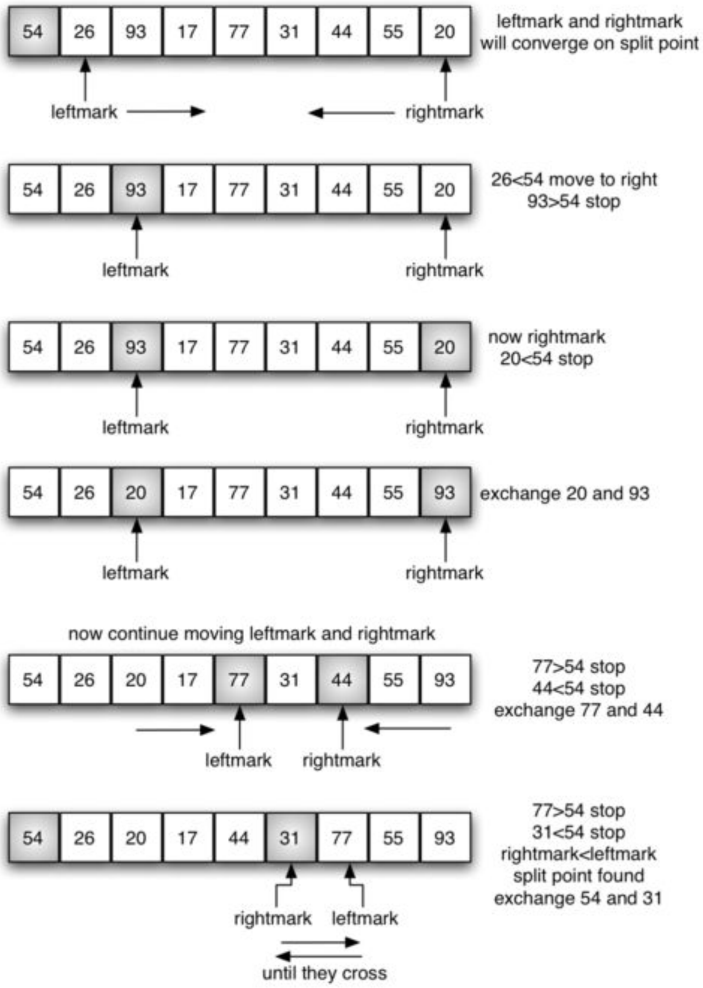


Why is Merge Sort O(n log n)?

- What is the runtime for the merge operation?
- How many times do we execute merge?

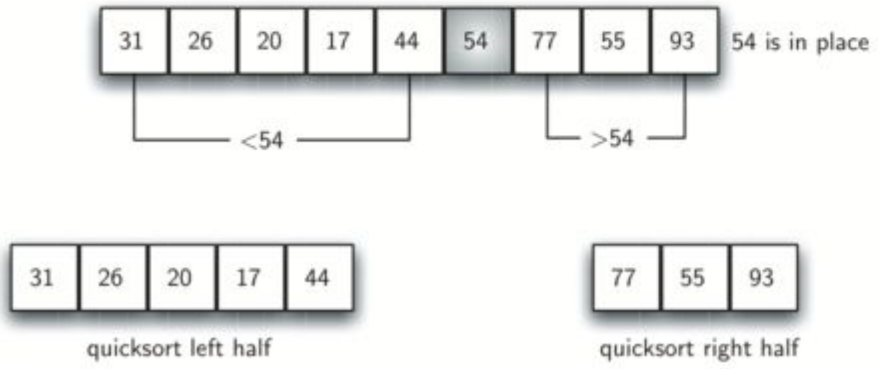
Quick Sort

- Find the median (aka split point)



54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark
54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross

Recursively apply quicksort



Why is O(nlogn) the best we can do?

Why is O(nlogn) the best we can do?

- Let's assume we have a deck of cards (numbered 1 through 52)
  - How many possible combinations are there? (You should know this from our discussions on time complexity!)

Why is O(nlogn) the best we can do?

- There are N! possible combinations of N cards
  - $5! = 5 \times 4 \times 3 \times 2 \times 1$
- In binary, we would need $\log_2 N!$ bits to represent N! possible states
  - For example, we could encode a system of eight states with only three bits
  - Alternatively, we interpret this there are 3 bits of information in a system with 8 possible states, all equally likely

Why is O(nlogn) the best we can do?

- $\log_2 N!$ can be approximated as N^N (when N approaches infinity)
  - See Stirling's formula
- Rewrite as $N \log_2 N$
  - There is $N \log_2 N$ "bits" of information in a deck of cards

Why is O(nlogn) the best we can do?

- N Log N bits of information can be interpreted as: “what is the expected number of decisions/questions I need to ask, in order to determine the state of the system”
- A randomized deck of cards has N Log N bits of information; a sorted deck has 0 bits of information since there is no variation
- Δ entropy = (N Log N) - 0
    - We can interpret sorting as driving entropy to zero

What just happened?

- We borrowed the concept of entropy from statistical physics
- Used it to measure amount of “information” in a system (eg deck of cards)
- Discretized this measure from real number, continuous systems to binary
-