Lecture 30: Quicksort

11/4/2020

Backstory, Partitioning

Sorting So Far

- Core ideas:
  - Selection sort: Find the smallest item and put it at the front
    - Heapsort variant: Use MaxPQ to find max element and put at the back
  - Merge sort: Merge two sorted halves into one sorted whole
  - Insertion sort: Figure out where to insert the current item
- Quicksort:
  - Much stranger core idea: Partitioning

Context for Quicksort's Invention

- 1960: Tony Hoare was working on a crude automated translation program for Russian and English
- How would you do this?
  - Binary search for each word
    - Find each word in log D time
  - Total time to find N words: N log D
- Algorithm: N binary searches of D length dictionary
  - Total runtime: N log D
  - ASSUMES log time binary search
- Limitation at the time:
  - Dictionary stored on long piece of tape, sentence is an array in RAM
    - Binary search of tape is not log time (requires physical movement!)
  - Better: **Sort the sentence** and scan dictionary tape once. Dictionary is sorted. Takes N log N + D time
    - But Tony had to figure out how to sort an array

The Core Idea of Tony's Sort: Partitioning

- To partition an array a[] on element x = a[i] is to rearrange a[] so that:
  - x moves to position j (may be the same as i)
  - All entries to the left of x are <= x
  - All entries to the right of x are >= x

Interview Question

- Given an array of colors where the 0th element is white, and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, and all blue squares are to the right. Your algorithm must complete in Θ(N) time (no space restriction).
  - Relative order of red and blues does NOT need to stay the same

Simplest (but not fastest) Answer: 3 Scan Approach

- Algorithm: Create another array. Scan and copy all the red items to the first R spaces. Then scan for and copy the white item. Then scan and copy the blue items to the last B spaces.

Quicksort

Partition Sort, a.k.a. Quicksort

- Observations:
  - The partitioning object is "in its place". Exactly where it'd be if the array were sorted
  - Can sort two halves separately, e.g. through recursive use of partitioning
- Quicksorting N items:
  - Partition on leftmost item
  - Quicksort left half
  - Quicksort right half

Quicksort

- Quicksort was the name chosen by Tony Hoare for partition sort
  - For most common situations, it is empirically the fastest sort
- How fast is Quicksort? Need to count number and difficulty of partition operations
- Theoretical analysis:
  - Partitioning costs Theta(K) time, where Theta(K) is the number of elements being partitioned
  - The interesting twist: Overall runtime will depend crucially on where pivot ends up

Quicksort Runtime

Best Case: Pivot Always Lands in the Middle

- What is the best case runtime?
  - Total work at each "level": N
  - Number of "levels": H = Theta(log N)
  - So overall runtime is Theta(HN) = Theta(N log N)

Worst Case: Pivot Always Lands at Beginning of Array

- What is the worst case runtime?
  - Total work at each "level": N
  - Number of "levels": N
  - So overall runtime is N^2

Quicksort Performance

- Theoretical analysis:
  - Best case: Theta(N log N)
  - Worst case: Theta(N^2)
- Compare this to Mergesort
  - Best case: Theta(N log N)

    - Worst case: Theta(N log N)
- Recall that Theta(N log N) vs. Theta(N^2) is a really big deal. So how can Quicksort be the fastest sort empirically? Because on average it is Theta(N log N) (proof requires probability theory and calculus)

(Intuitive) Argument #1 for Average Runtime: 10% Case

- Suppose pivot always ends up at least 10% from either edge
- Work at each level: O(N)
    - Runtime is O(NH) (H = number of levels)
        - H is approximately log_{10/9}N = O(log N)
    - Overall: O(N log N)
- Even if you're unlucky enough to have a pivot that never lands in the middle, but at least always 10% from the edge, runtime is still O(N log N)

(Intuitive) Argument #2: Quicksort is BST Sort

- Key idea: compareTo calls are same for BST insert and Quicksort
    - Every number gets compared to the partitioning number in both
- Reminder: Random insertion into a BST takes O(N log N) time

Empirical Quicksort Runtimes

- For N items:
    - Mean number of compares to complete Quicksort: ~2N ln N
    - Standard deviation is about 0.6482776N
    - Very few arrays take a ridiculous number of comparisons

Quicksort Performance

- Theoretical analysis:
    - Best case: Theta(N log N)
    - Worst case: Theta(N^2)
    - Randomly chosen array case: Theta(N log N) expected
- Compare this to Mergesort
    - Best case: Theta(N log N)
    - Worst case: Theta(N log N)
- Why is it faster than mergesort?
    - Requires empirical analysis. No obvious reason why

Sorting Summary (so far)

- Listed by mechanism:
    - Selection sort: Find the smallest item and put it at the front
    - Insertion sort: Figure out where to insert the current item
    - Merge sort: Merge two halves into one sorted whole
    - Partition (quick) sort: Partition items around a pivot

Avoiding the Quicksort Worst Case

Quicksort Performance

- The performance of Quicksort (both order of growth and constant factors) depend critically on:
  - How you select your pivot
  - How you partition around that pivot
- Bad choices can be ver bad indeed, resulting in Theta(N^2) runtimes

Avoiding the Worst Case

- If pivot always lands somewhere "good", Quicksort is Theta(N log N). However, the very rare Theta(N^2) cases do happen in practice, e.g.
  - Bad ordering: Array already in sorted order (or almost sorted order)
  - Bad elements: Array with all duplicates
- Recall our version of Quicksort has the following properties:
  - Leftmost item is always chosen as the pivot
  - Out partitioning algorithm preserves the relative order of <= and >= items
- How can we avoid worst case behavior?
  - Always use the median as the pivot
  - Shuffle before quicksorting