CSE 373: Sorting II

Pete Morcos
University of Washington
4/18/00

http://www.cs.washington.edu/education/courses/cse373/00sp

Analyzing Mergesort

- Recall: mergesort splits array in half and recursively calls itself on the halves
- Work done per step is O(N) for the merge
- Each recursive call has half as much work to do
- We write a *recurrence relation* for the time T as a function of N:
  - T(1) = O(1)
  - T(N) = 2 * T(N/2) + O(N)

Recurrences

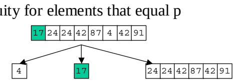
- One way to see the value of this recurrence is to keep expanding the terms
  - T(N) = 2 * T(N/2) + N
  - T(N) = 2 * [2 * T(N/4) + N/2] + N
  - = 4 * T(N/4) + 2 * N — or, 2^2 * T(N/2^2) + 2 * N
  - T(N) = 4 * [2 * T(N/8) + N/4] + 2 * N
  - = 8 * T(N/8) + 3 * N — or, 2^3 * T(N/2^3) + 3 * N
  - ...
  - T(N) = $2^{\log N}$ * T(1) + (log N) * N — or, $2^{\log N}$ * T(N/$2^{\log N}$) + log N * N
  - = N * O(1) + N log N
  - = O(N log N)

Quicksort

- Mergesort operated by arbitrarily splitting the array in half, sorting the halves, and merging
  - Splitting was easy
  - Merging took O(N) work
- Quicksort is slightly different
  - *Partition* array into halves
    - Elements in left half all smaller than elements in right half
  - Recurse on halves
  - Concatenate halves – O(1) work

Partitioning

- Choose a value p (the *pivot*) from the list
- Move all elements ≤ p into left half, elements ≥ p into right half
  - Note ambiguity for elements that equal p



- How long does this take?
- Can partition array in place by swapping elements in the wrong half

Partitioning In-place

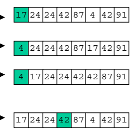
- Important not to use an extra array
- Algorithm:
  - Swap pivot with last element, leaving N-1
  - Set pointers i, j at beginning, end
  - Move i up array until hit an element > p
  - Move j down array until hit an element < p
  - Swap elements pointed at by i and j
  - Repeat until i and j meet
  - Swap pivot with element at meeting point
- Keys equal to p are annoying—see book

This inner loop is very simple—just repeated compare and increment. This is why quicksort is so fast.

Choosing Pivot

BAD!

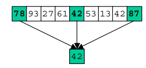
- Naive: pick first element
  - What if it's the smallest or largest?
  - What happens if array was sorted?
- Optimal: pick the median
  - How much work?



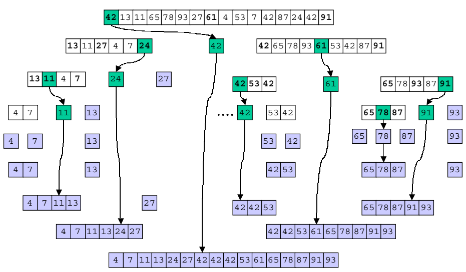
- Randomize: pick a random element
  - Makes naive problem very unlikely
  - Requires random number generator
- Standard trick is *median-of-three...*

Median-of-three pivot selection

- Instead of finding median of whole list, find median of first, middle, and last elements
- Constant time
- Reduces chance of poor behavior compared to just looking at one element, and doesn't have troubles with sorted inputs



Quicksort example



Quicksort Analysis

- Best case: we choose the ideal pivot every time, and split the list evenly
  - $T(0) = T(1) = O(1)$
  - $T(N) = 2 * T(N/2) + O(N)$
  - same as mergesort discussion: $O(N \log N)$
- Worse case: we choose the worst pivot, leaving one of the halves empty
  - $T(0) = T(1) = O(1)$
  - $T(N) = T(N-1) + O(N)$
    - $T(N) = [T(N-2) + O(N-1)] + O(N)$
    - ...
    - $= O(N^2)$

Sorting Choices

- $O(N^2)$
  - Bubble
  - Selection
  - Insertion: may be easiest to remember
- $O(N \log N)$
  - Heapsort
  - Mergesort: simple, easy to remember
  - Quicksort: fastest in practice, danger of $O(N^2)$
- For small N (e.g. < 20), the N log N sorts are slower due to extra complexity
  - Test N and use simpler sort if small

Quickselect

- Recall that to select the kth smallest item in a list, we have a few choices:
  - k linear scans, removing smallest each time: O(k N)
  - make a heap, do k DeleteMin's: O(N + k log N)
- For the median, k = N/2, these are O(N2), O(N log N)
- Quickselect uses a similar divide-and-conquer strategy to quicksort, and runs in O(N) average time, but $O(N^2)$ worst case

Quickselect algorithm

- If array is size 1, we're done
- Otherwise, partition array as with quicksort
  - Left half size L, right half size N – L – 1
  - If k <= L, the kth smallest is in the left half
  - if k = L + 1, the pivot is the kth smallest, stop!
  - if k > L + 1, the kth smallest is in the right half
- Recurse on the side chosen above
- Only one recursive call, unlike quicksort
- T(N) = T(N/2) + O(N)
  - = [T(N/4) + O(N/2)] + O(N)
  - = [T(N/8) + O(N/4)] + O(N/2) + O(N)
  - ... = O(N)

Sorting large structures

- Sorting involves a lot of swapping, as you've seen
- What if each item is a big data record, e.g. 2000 bytes of info about a student?
  - Don't want to repeatedly copy that much data
- Instead, sort *pointers* to each record
  - Obviously, you don't sort the pointer values themselves—they're just addresses
  - When you want to compare two items, dereference the pointers to get at the sort key in the record

Bucket sort

- What if set of values have a relatively small range of values? (e.g. 1 to 10,000)
- Allocate array `count[10000]`
- Scan list, for an element `p`, increment `count[p]`
- When done, go through `count` array and output value `v` as many times as `count[v]`
- O(N)...doesn't this violate our lower bound proof?
- No. We are no longer limited to comparing one element to another. When we jump directly to the right element of the count array, we are effectively comparing against all N-1 other elements.

Stable Sorting

- A stable sort is one that does not change the order of items with the same sort key
- Matters if there was some other ordering already present

Joe	32
Ann	17
Bob	21
Ann	31
Gina	32
Gina	27
Ann	27

sort by age

Ann	17
Bob	21
Gina	27
Ann	27
Ann	31
Joe	32
Gina	32

sort by name

Ann	17
Ann	27
Ann	31
Bob	21
Gina	27
Gina	32
Joe	32

Radix sort

- Bucket sort + stable sorting
- Sometimes bucket sort is unusable because there are too many buckets
- But, if you can divide sort key into "slices", you can bucket sort on each slice
  - e.g. digits in a number, characters in a string
- Trick is to sort on *least* significant slice first, not most significant

Radix sort example

Original	sort on ones digit	sort on tens digit	sort on hundreds digit
812	111	102	102
348	821	907	111
111	812	208	208
208	762	111	247
762	102	812	348
427	422	821	366
102	366	422	422
366	427	427	427
247	247	247	762
907	907	348	812
821	348	762	821
422	208	366	907

External Sorting

- (We won't cover in detail, but just so you know it exists...)
- Often in the real world, the data to be sorted is so big it can't fit in memory.
- So, read in a chunk at a time and sort the bits.
- Then, merge two sorted chunk together
  - Once sorted, we only need first element in each to merge
- Repeat until we only have one "chunk"