CSE 373: Selection and Sorting

Pete Morcos
University of Washington
4/17/00

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

The Selection Problem

- Given a set of N integers, which one is the k^{th} largest?
- Common to ask about $k = 1$, $k = N$, $k = N/2$ (the *median*)
- Also typical to want multiple, e.g. top ten
- Seems clear that it will be at least O(N) since we have to look at every element
  - Obviously O(N) for $k = 1$ or N
- Several of the data structures we've talked about should jump to mind

Sorting

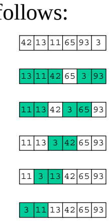
- In principle, if we do N selections, we know the sorted order of the data
  - $O(N^2)$ if selection is O(N), O(N log N) if it's O(log N)
- This is actually how some sorting algorithms work
- Sorting is valuable in many situations
  - Allows binary search of an array
  - Once sorted, selections are O(1) [if set is contiguous]
  - Detecting duplicates becomes easy
  - Makes it easier to hand homework back to students

Assumptions

- Data starts in an array, in any order
  - A pointer-based structure might make rearrangement easier; we'll talk about that if it matters
- Large range of possible values
  - e.g. all integers, all strings, etc
- We can compare any two items with < , > , ==
  - Known as a *total ordering* of the set of possible values
  - Some data isn't totally ordered—is CSE 373 < BIOL 401?
- Relaxing these assumptions enables other techniques

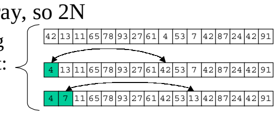
Using Lists – Bubblesort

- We can define that a set A_i is "sorted" as follows:
  - For any *i* and *j*, if $i < j$ then $A_i <= A_j$
- Suppose we just consider *i* and *i* + 1
- Repeat the following until sorted:
  - Scan list; for each pair out of order, swap
- Time?
  - Obviously each step does N-1 comparisons
  - Items can move left at most once per step
  - So up to N-1 steps needed => $O(N^2)$
- Let's try moving items more than 1 space per step

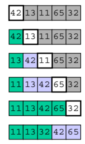


Using Lists – Selection Sort

- Naive selection ($k = 1$)*:* scan for smallest, O(N)
- Sort then becomes N iterations of
  - Select smallest remaining
  - Remove it and add to end of separate array
- Time? N steps taking N, N-1, N-2, ... , 3, 2, 1
- Space? Need extra array, so 2N
  - Can avoid by swapping next item with smallest:



Using Lists – Insertion Sort

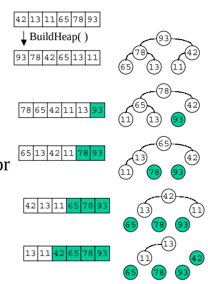


- Avoid expensive selections
- N steps, sorts in place:
  - get first remaining item
  - swap left past larger items
- Time? each step = select + swap
  - Let *s* = # already sorted, *k* = # sorted and larger than new
  - time = 1 + *k*
    - but k is on average *s*/2
    - *s* goes from 1 to N
    - grand total is $O(N^2)$

Using Trees

- We want to beat $O(N^2)$
- Suppose we use a BST
  - N steps, in each we do an Insert operation
  - Then, an inorder tree traversal will give us the sorted result
- Time? Each insert is a log N operation, so this is an O(N log N) algorithm
- Downside is that we need to separately allocate the tree (and use pointers), so roughly 3N space
- If we knew the tree was *complete*, then we could use an array representation and sort in-place, which leads to...

Using Heaps - Heapsort



- Basic idea:
  - Build a maxheap
  - Do N DeleteMax steps
    - Put value in unused end of array
- Time? O(N) + N*O(log N)
  - O(N log N) is as good as it gets for sorting, but in practice heapsort is a bit slower than competing sorts (larger constant factors)

Lower Bounds on Sorting

- Algorithms like bubblesort that only compare and swap adjacent elements can do no better than $O(N^2)$
  - An *inversion* is any pair of elements that are in the wrong order
  - There are N(N-1)/2 possible pairings of elements
  - On average, half of those will be out of order (consider the reversed array to see why)
    - Average and worst cases are both $O(N^2)$
  - An adjacent swap **only** fixes one inversion
- To do better, your algorithm must move things more than one space at a time

Lower Bound on Comparison Sorting

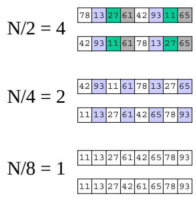
- We assumed at the beginning that the only thing we can do to elements is compare them two at a time
- Any comparison-only sort is $\Omega(N \log N)$
  - There are N! possible orderings of a list
    - Only one of them is sorted (if no duplicates)
  - A single comparison gives us information to cut the number of possible orderings in half
  - Thus, we need log (N!) comparisons
  - Book shows that $\log (N!) = \Omega(N \log N)$

Shellsort

- Named after its inventor, shellsort tries to get items in rough position during early passes, then refines that by doing more specific passes
  - For some *increment sequence* $k_1, k_2, k_3, ..., k_i, ...$
  - Sort all k_i subsequences of elements separated by k_i
  - Go to the next smaller increment k_{i-1} and repeat
- Proofs have been difficult since there are so many possible increment sequences
- Turns out that shellsort is N^x, where *x* might be 3/2, 5/4, 4/3, etc
  - This is asymptotically worse than N log N for *any* $x > 1$
  - In practice, works well up to moderate sizes of N

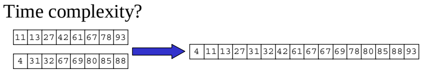
Shellsort Example

- Example uses the increment sequence Shell originally proposed: N/2, N/4, N/8, ..., 2, 1
  - Seems natural, but turns out to be quite bad! $O(N^2)$
  - Hibbard's sequence, 2^k-1, ..., 15, 7, 3, 1 is $O(N^{3/2})$. Adjacent increments have no common factors
- Note that within each color, we are doing an insertion sort, so *h* = 1 is a plain old insertion sort
  - h = 1 as last increment ensures final list is completely sorted



Mergesort

- Our first recursive algorithm, mergesort uses the *divide-and-conquer* strategy
  - Slice the problem into smaller parts
  - Independently solve the parts, then combine
  - Very powerful concept in computer science
- Heart of the algorithm is the merge( ) function
  - Given two sorted arrays, make one big sorted array
  - Time complexity?



Mergesort cont'd

- To sort, recurse:
  - If N = 1, array is sorted already
  - If N > 1
    - Divide array in half
    - Recursively sort halves
    - Merge halves
- Time: log N subdivision levels
  - Total of all subdivisions at one level is O(N)
  - O(N log N) total time

Mergesort example