Mergesort/Quicksort

Review

* 3 algorithms for sorting so far
* selection sort
* bubble sort
* insertion sort
* all are O(n^2) quadratic time.
* But what if I told you there was a way to sort FASTER?
* We are going to spend a few days looking at algorithms that can sort FASTER than quadratic time.

Treesort

* Recall the SORTING PROBLEM:
  + Suppose we have an array of length n: A[0]...A[n-1] of integers, and we’d like to rearrange them within the array so they’re in sorted order.
* Ask: How can we use a binary search tree to sort numbers?
  + Algorithm:
  + Insert each item in the array into the BST.
  + Run an inorder traversal over the tree.
  + As you see each item in the traversal, copy each item back into the the array.
* Write this pseudocode:
* Treesort:
  + for i = 0 to n:
    - bst-insert(A[i])
  + end for
  + for i = 0 to n:
    - A[i] = next item from BST inorder traversal
* Let’s do a big-oh analysis.
  + bst-insert = O(log n)
  + inside a loop that runs n times, so O(n log n)
  + How long does it take to do an inorder traversal O(n).
  + Total = nlogn + n = O(n log n).
* What goes wrong if the tree is unbalanced? bst-insert goes from logn to n. So performance degrades to O(n^2). So average case is nlogn, worse case is n^2.
* good and bad points of treesort
  + faster than the O(n^2) sorts.=🡺 good thing!
  + still degrades to O(n^2) in worste case -> not great.
  + uses separate data structure to do the sorting (the BST). Not an **in-place** sort. Uses O(n) extra space besides original array.
  + NEW IDEA: **STABILITY**.
  + Discuss with example of sorting a deck of cards. 7S 5H 2H 5S
  + Binary tree sort is STABLE if we design our binary tree to be able to hold duplicate elements, and we insert them as right children.
* So we still don’t have a guarantee of that we will run faster than O(n^2) in the worst case. Let’s look an algorithm that DOES make than guarantee.

MERGE SORT

* We’re going to look at a number of algorithms that fall into the category of divide and conquer algorithms.
* **Divide and conquer algorithm**= algorithm that breaks a problem down into two or more SMALLER sub- problems of the same type, until the sub-problems can be solved directly. Then the solutions to the sub-problems are combined to give a solution to the larger problem.
* Very simple algorithm using D&Conquer for doing a sort:
* Algorithm:
  + MergeSort(A[0]...A[n-1]).
    - if length of A == 1
      * return A.
    - else
      * // split A in half as equally as possible into 2 subarrays, and call
      * // merge sort on each one.
      * B = Mergesort(A[0]...A[n/2])
      * C = Mergesort(A[n/2+1]...A[n-1])
      * Merge B and C together into one sorted Array called D.
      * Return D.
* Run this in class on 8 3 4 7 6 2 1 5
* Do a big-oh analysis:
  + if length...return A O(1).
  + else part:
    - B and C are each T(n/2).
    - How long does merge take? let’s see.
* Design algorithm as class:
  + abstract algorithm:
  + maintain two indices in each array. Compare the elements at each pointer. Whichever one is smaller, take that element and copy it into the new array. Advance that pointer.

Diagram

Description automatically generated

* What is the big-oh of this sub-algorithm, MERGE?
  + O(n).
* Return to big-oh analysis.
* T(1) = 1.
* T(n) = T(n/2) + T(n/2) + n = 2T(n/2) + n.
* Works out to O(n log n).
* Hand out code for merge sort.

GOOD/BAD THINGS ABOUT MERGESORT

* Avg & worst case are n log n.
* Because we always divide the list in half (+/- 1 element), there’s no way we can degrade into O(n^2) time.
* However, big downside it the sort is not in-place, it uses additional arrays to do the MERGING and copying.
* It is usually stable.

LOWER-BOUND ON COMPARISON SORTING

* So we’ve seen 2 algorithms so far for sorting in nlogn time: tree sort and merge sort.
* People often ask, can we do better?
* Answer is in the most general case, NO.
* Here’s why.
* All of the sorts we’ve looked at so far are called COMPARISON SORTS.
* This means that the way we figure out if element A comes before element B is to run the test A < B. That’s the only kind of test we ever do, checking if one element is less than the other.
* There is a proof that for any sorting algorithm which relies only on comparisons, it cannot run faster than n log n.
* Here’s why.
* Suppose we have an array with n things in it.
* How many different possible orderings of this array are there?
  + n!
  + Each of these n! orderings corresponds to a correctly-sorted array for some initial mixed-up version of the array.
  + In other words, say we have 3 numbers, call them a, b, c. The six permutations of abc are
    - abc
    - acb
    - bac
    - bca
    - cab
    - cba
  + There’s a way I can assign numbers to a/b/c such that the initial ordering should be re-arranged into each of those six re-orderings.
* So any algorithm which relies only on comparisons has to be able to eventually distinguish between all n! possible orderings.
* We know that every comparison the algorithm does has 2 outcomes—if I test A<B, the outcome is either TRUE OR FALSE.
* Let’s say I have an algorithm that finishes after C comparisons. How many possible outcomes can I distinguish between if I run C comparisons? 2^C.
* I need 2^C to be greater than or equal to n!.
* or equivalently, C >= log(n!).
* We don’t have a nice formula for what log(n!) is, but we can estimate it.
* 
* So any comparison-based sorting algorithm CANNOT run faster than n log n.
  + That doesn’t mean that for a particular input, the algorithm runs faster than n log n. We saw for bubble sort that for already sorted input, it runs in time O(n).
  + It means that for any algorithm, there must be a case than runs in n log n time. They all can’t be faster.

DAY 2---Quicksort

mergesort(A):

split A in half O(1)  
 mergesort(A’s left half) T(n/2)  
 mergesort(A’s right half) T(n/2)

merge(A’s left half, A’s right half) O(n)

Today we’re going to look at a new algorithm, quicksort, that is another divide and conquer algorithm.

It has a similar sequence of steps to mergesort, but it has an interesting twist, in that it:

splits A in half

calls itself recursively on left half

calls itself recursively on the right half

But doesn’t need to merge the two halves together at the end. There’s a little bit of magic in quick sort that eliminates the merging. Here’s the magic:

* Suppose I have an array that contains the numbers 1-10, all mixed up, and I want to put them in sorted order.
  + Write on board: 5 9 2 8 3 1 7 4 10 6
* When I divide the array in half down the middle, each side is going to have 5 numbers, and those 5 numbers in each half are just as mixed up as the original array.
* That’s why, even after I sort each half, I still have to merge them.
  + Write on board each sorted left and right half:
  + 2 3 5 8 9 1 4 6 7 10
* What if there were a way to divide the original 10-element list into 2 5-element sublists but as I’m doing it, all the numbers 1-5 end up on the left and all the numbers 6-10 end up in the right list.
* They don’t need to end up in sorted order as I’m doing it, but as long as all the small numbers go into the left list and all the large numbers go into the right list, watch what happens.  
  + Suppose the left/right lists list looks like:
  + 2 3 5 1 4 9 8 7 10 6
  + Then I sort them recursively.
  + 1 2 3 4 5 6 7 8 9 10
* After I do this, I don’t need to merge the lists!
* Does this seem to good to be true? It should! But it’s not! This can be done! It’s called quicksort.

Here’s the basic algorithm:

Quicksort(A[])

If the list has 0 or 1 element, do nothing.

else:

pick an element from the array to use as the PIVOT element.

**Partition** the elements in the array so that it consists of 3 sections:

[ all elements < pivot | pivot | all elements > pivot ]

Quicksort(smaller-than-pivot section of A)

Quicksort(larger-than-pivot section of A)

In a perfect world, the two sections of the array less than and greater than the pivot are roughly equal in size. That way the two recursive calls are operating on roughly half of the array each, just like merge sort does. Unfortunately, getting those two halves to be equal depends entirely on the first step of the algorithm, choosing a pivot. There are a lot of different ways to choose the pivot. One common way is to pick the pivot element to be the first element in the array. Let’s see how the algorithm works when we do this.

The partition step of the algorithm is where the magic happens. Partition’s job is to take the pivot and somehow move all the things greater than the pivot to the end of the array, and all the things less than pivot to the beginning of the array. Here’s how we do it.

* Set up 2 index variables, one at the beginning of the array and one at the end.
* Move the lower pointer to the right until we find an element that is BIGGER than the pivot.
* Move the higher pointer to the left until we find an element that is SMALLER than the pivot.
* Swap the elements at these positions.
* Continue until the index variables are the same (pointing at the same item in the array).

Do first book example, illustrating partition.

Hand out source code.

Do second book example, following code.

Compute big-oh.

worst

average

choice of pivot---median of 1st, middle, last elements.

small sublists <=20, use insertion sort.

Extra class Fall 2020 to finish quicksort

My example:

Calling partition with [5, 8, 4, 2, 6, 1, 7, 3] 0 7 partition chosen is 5

**Swap pos 1 7**

**Swap pos 4 5**

Move partition into place. **Swap first element= 0 with with rightpos= 4**

After partition, x is [1, 3, 4, 2, 5, 6, 7, 8]

Calling partition with [1, 3, 4, 2, 5, 6, 7, 8] 0 3 partition chosen is 1

Move partition into place. **Swap first element= 0 with with rightpos= 0**

After partition, x is [1, 3, 4, 2, 5, 6, 7, 8]

Calling partition with [1, 3, 4, 2, 5, 6, 7, 8] 1 3 partition chosen is 3

**Swap pos 2 3**

Move partition into place. **Swap first element= 1 with with rightpos= 2**

After partition, x is [1, 2, 3, 4, 5, 6, 7, 8]

Calling partition with [1, 2, 3, 4, 5, 6, 7, 8] 5 7 partition chosen is 6

Move partition into place. **Swap first element= 5 with with rightpos= 5**

After partition, x is [1, 2, 3, 4, 5, 6, 7, 8]

Calling partition with [1, 2, 3, 4, 5, 6, 7, 8] 6 7 partition chosen is 7

Move partition into place. **Swap first element= 6 with with rightpos= 6**

After partition, x is [1, 2, 3, 4, 5, 6, 7, 8]

final array [1, 2, 3, 4, 5, 6, 7, 8]

Students do:

Calling partition with [3, 4, 5, 2, 6, 1] 0 5 partition chosen is 3

Swap pos 1 5

Swap pos 2 3

Move partition into place. Swap first element= 0 with with rightpos= 2

After partition, x is [2, 1, 3, 5, 6, 4]

Calling partition with [2, 1, 3, 5, 6, 4] 0 1 partition chosen is 2

Move partition into place. Swap first element= 0 with with rightpos= 1

After partition, x is [1, 2, 3, 5, 6, 4]

Calling partition with [1, 2, 3, 5, 6, 4] 3 5 partition chosen is 5

Swap pos 4 5

Move partition into place. Swap first element= 3 with with rightpos= 4

After partition, x is [1, 2, 3, 4, 5, 6]

final array [1, 2, 3, 4, 5, 6]