Sorting and searching notes

Prof Bill - Mar 2020

Sections are:

A. Basic sorting: Bubble sort, Selection sort

B. Advanced sorting: Quicksort, Heapsort

C. Searching: linear search, binary search

D. Bonus: NP-complete problem

Reading:

- Morin, Open Data Structures Ch 11 Sorting, opendatastructures.org/ods-java/11_Sorting_Algorithms.html
- Sedgewick, Algorithms Ch 2 Sorting, algs4.cs.princeton.edu/20sorting

The **best way** to learn these various sorting algorithms (imho) is to use our favorite animation site. He does an outstanding job showing how each sort works, www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

thanks… yow, bill

A. Basic sorting: Bubble sort, Selection sort

The two most basic sorting algorithms are Bubble sort and Selection sort.

It's easy to remember which is which:

- ➔ **Bubble sort** - lots of swaps
- ➔ **Selection sort** - one swap per pass

Structure: These basic algorithms use a nested for loop:

```
for( each element in the array)
   for( every other element in the array or so)
      do work: compare, swap, etc
```

Analysis: Bubble sort and Selection sort are both **O(n^2)**. (yuk)

So, sorting 100 items means about 100*100 = 10,000 time units.

What's the hit to sort 10,000 items? (gulp)

Note: Insertion sort is another basic sort with **O(n^2)** performance.

It sure would be nice if we had a faster sorting algorithm than this.

Next page.

B. Advanced sorting: Quicksort, Heapsort

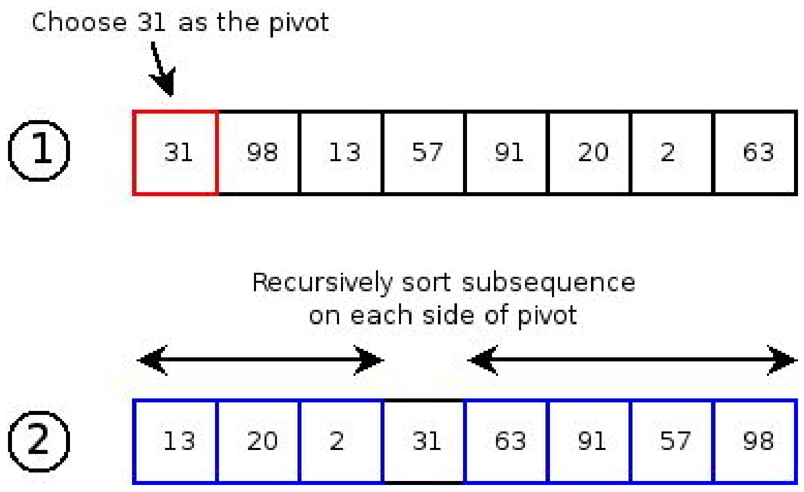
Quicksort is the $$$. Some details:

- ❏ The Quicksort algorithm is recursive and has 3 steps:

```
quicksort( data) {
   partition data into two halves // not equal halves
   quicksort( the first half)
   quicksort( the second half)
}
```

- ❏ Average performance is **O(N log N)**.
- ❏ “The quicksort algorithm was developed in 1959 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University.”, en.wikipedia.org/wiki/Quicksort
- ❏ Sedgewick says: “2.3 Quicksort describes quicksort, which is used more widely than any other sorting algorithm.”

Partition, pivot - Quicksort places pivot in spot so that all values below pivot in the array are <= the pivot value, and all values above are > pivot value.

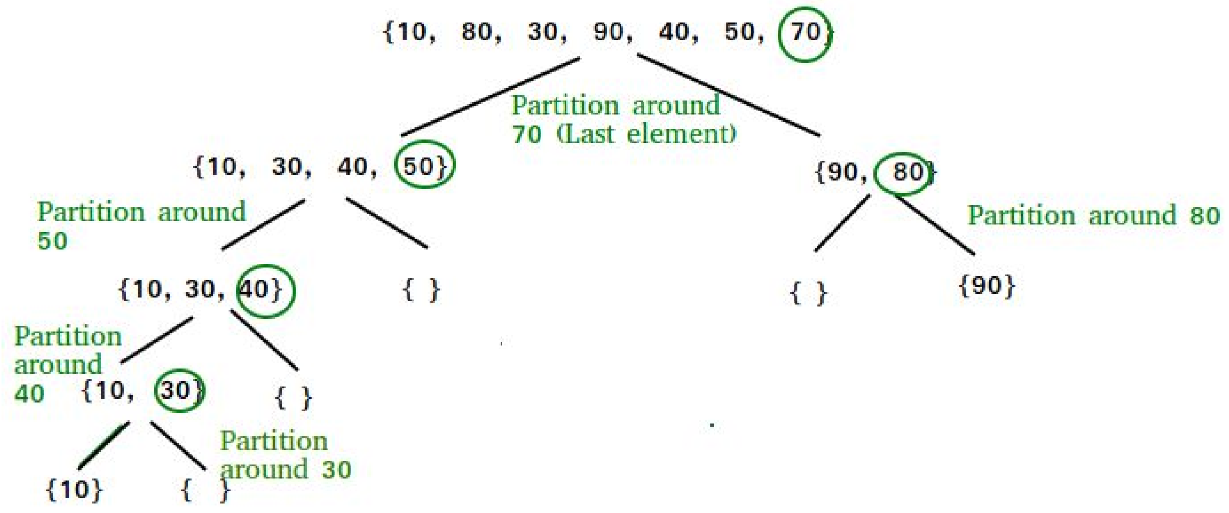


Source: faculty.ycp.edu/~dhovemey/fall2005/cs102/lecture/11-1-2005.html

Quicksort pseudocode, source: www.geeksforgeeks.org/quick-sort/

```
// recursive quicksort algorithm
quickSort(arr[], low, high)
    if (low < high)
        // pi is partitioning index, arr[p] is now correct
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi


// swap within array range so everything below pivot is <= and above is >
// pivot index is returned
int partition (arr[], low, high)
    pivot = arr[high];    // pivot, last item
    i = (low - 1)    // index of smaller element

    for (j = low; j <= high- 1; j++)
        if (arr[j] <= pivot)    // if current is <= pivot
        i++;
        swap arr[i] and arr[j]

    swap arr[i + 1] and arr[high])
    return (i + 1)
```

Quicksort example



One problem with Quicksort. It’s not **stable**. A stable sort is one where two equal objects are left in the same order in sorted output as they appear in the input array, www.geeksforgeeks.org/stability-in-sorting-algorithms/

The **Heapsort** algorithm is very simple. Build a heap with your data, then use removeMin().

We know that heap insert and search are O(log N).

Do that N times and you get O(N log N) performance for heapsort.

Mergesort is another O(N log N) sorting algorithm.

C. Searching

Two methods for searching an array of data: sequential search, binary search.

Sequential search: is the obvious search solution.

- Look at each item in the array, from 1 to N
- Performance is **O(N)**

Binary search is a little fancier:

- ❑ Array must already be sorted!
- ❑ Divide problem size in half with each iteration, hence the **O(log N)**

Binary search pseudocode (source: www.codecodex.com/wiki/Binary_search):

```
// recursive binary search, returns item if found
binarySearch(a, value, left, right)
    if right < left, then return not found
    mid = (right + left)/2
    if a[mid] == value
        return mid
    if value < a[mid]   // smaller value means search left
        return binarySearch(a, value, left, mid-1)
    else   // larger value means search right
        return binarySearch(a, value, mid+1, right)
```

Since binary search is tail recursion, we can (pretty easily) make it iterative (faster!).

```
// iterative binary search
binarySearch(a, value, left, right)
    while left ≤ right
        mid = (right + left)/2
        if a[mid] == value
            return mid
        if value < a[mid]   // smaller value means search left
            right := mid-1
        else    // larger value means search right
            left  := mid+1
    return not found
```

D. NP-Complete problem

Finally…

Big-O is a part of one of the most significant unsolved problems in math/computer science today. It's known as the **P vs. NP Problem**. If you can publish a solution, these guys will give you a million bucks.

> www.claymath.org/millennium-problems/p-vs-np-problem
>
> *In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.*

P problems are those where solutions are easily found (polynomial time, hence the "P").

NP problems are those that are easy to verify (in polynomial time), but hard to solve.

A large set of problems are called **NP-complete** as they can be easily verified, but no one has proven that they are hard to solve yet.

Wikipedia does a nice summary as well, en.wikipedia.org/wiki/NP-completeness.

www.claymath.org