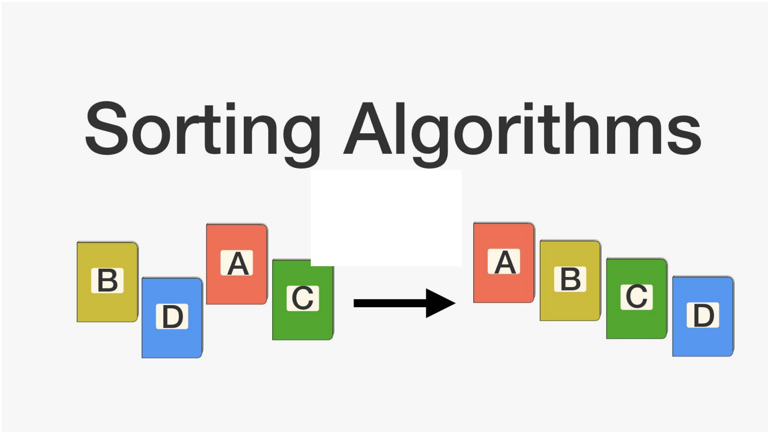
Sorting Algorithms

A **sorting algorithm** is an algorithm made up of a series of instructions that takes an array as input, performs specified c on the array, sometimes called a list, and outputs a sorted array. Sorting algorithms are often taught early in computer sc classes as they provide a straightforward way to introduce other key computer science topics like Big-O notation, divide-a methods, and data structures such as binary trees, and heaps. There are many factors to consider when choosing a sort algorithm to use.



Contents

Sorting Algorithms
Properties of Sorting Algorithms
Common Sorting Algorithms
Choosing a Sorting Algorithm
See Also
References

Sorting Algorithms

In other words, a sorted array is an array that is in a particular order. For example, $[a, b, c, d]$ is sorted alphabetically, $[1,$ a list of integers sorted in increasing order, and $[5, 4, 3, 2, 1]$ is a list of integers sorted in decreasing order.

A sorting algorithm takes an array as input and outputs a permutation of that array that is sorted.

There are two broad types of sorting algorithms: **integer sorts** and **comparison sorts**.

Comparison Sorts

Comparison sorts compare elements at each step of the algorithm to determine if one element should be to the left or rig another element.

Comparison sorts are usually more straightforward to implement than integer sorts, but comparison sorts are limited by a bound of $O(n \log n)$, meaning that, on average, comparison sorts cannot be faster than $O(n \log n)$. A lower bound for algorithm is the *worst-case* running time of the *best* possible algorithm for a given problem. The "on average" part here is there are many algorithms that run in very fast time if the inputted list is *already* sorted, or has some very particular (and unlikely) property. There is only one permutation of a list that is sorted, but $n!$ possible lists, so the chances that the input sorted is very unlikely, and on average, the list will not be very sorted.

PROOF

The running time of comparison-based sorting algorithms is bounded by $\Omega(n \log n)$.

A comparison sort can be modeled as a large binary tree called a decision tree where each node represents a single comparison. Because the sorted list is some permutation of the input list, for an input list of length n, there are $n!$ poss permutations of that list. This is a decision tree because each of the $n!$ is represented by a leaf, and the path the algori take to get to each leaf is the series of comparisons and outcomes that yield that particular ordering.

At each level of the tree, a comparison is made. Comparisons happen, and we keep traveling down the tree; until the a reaches the leaves of the tree, there will be a leaf for each permutation, so there are $n!$ leaves.

Each comparison halves the number of future comparisons the algorithm must do (since if the algorithm selects the rig out of a node at a given step, it will not search the nodes and paths connected to the left edge). Therefore, the algorith performs $O(\log n!)$ comparisons. Any binary tree, with height h, has a number of leaves that is less than or equal to 2

From this,

$$2^h \geq n!.$$

Taking the logarithm results in

$$h \geq \log(n!).$$

From Stirling's approximation,

$$n! > \left(\frac{n}{e}\right)^n.$$

Therefore,

$$\begin{aligned} h &\geq \log\left(\frac{n}{e}\right)^n \\ &= n \log\left(\frac{n}{e}\right) \\ &= n \log n - n \log e \\ &= \Omega(n \log n). \end{aligned}$$

Integer Sorts

Integer sorts are sometimes called counting sorts (though there is a specific integer sort algorithm called counting sort). I do not make comparisons, so they are not bounded by $\Omega(n \log n)$. Integer sorts determine for each element x how mar are less than x. If there are 14 elements that are less than x, then x will be placed in the 15^{th} slot. This information is u each element into the correct slot immediately—no need to rearrange lists.

Properties of Sorting Algorithms

All sorting algorithms share the goal of outputting a sorted list, but the way that each algorithm goes about this task can v working with any kind of algorithm, it is important to know how fast it runs and in how much space it operates—in other w time complexity and space complexity. As shown in the section above, comparison-based sorting algorithms have a time of $\Omega(n \log n)$, meaning the algorithm can't be faster than $n \log n$. However, usually, the running time of algorithms is dis terms of big O, and not Omega. For example, if an algorithm had a worst-case running time of $O(n \log n)$, then it is gua the algorithm will never be slower than $O(n \log n)$, and if an algorithm has an average-case running time of $O(n^2)$, the average, it will not be slower than $O(n^2)$.

The running time describes how many operations an algorithm must carry out before it completes. The space complexity how much space must be allocated to run a particular algorithm. For example, if an algorithm takes in a list of size n, and reason makes a new list of size n for each element in n, the algorithm needs n^2 space.

TRY IT YOURSELF

Find the big-O running time of a sorting program that does the following:

- It takes in a list of integers.
- It iterates once through the list to find the largest element, and moves that element to the end.
- It repeatedly finds the largest element in the unsorted portion by iterating once through, and moves that element to the end of the unsorted portion.

At the end, the list is sorted low to high.

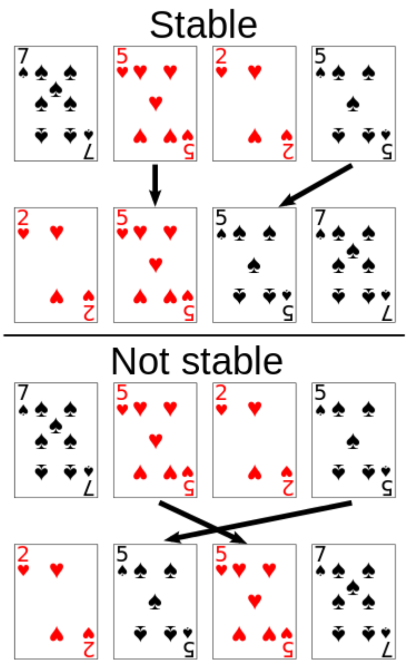
(Also, try implementing this program in your language of choice.)

- $O(1)$
- $O(n)$
- $O(n^2)$
- $O(n \log n)$

Additionally, for sorting algorithms, it is sometimes useful to know if a sorting algorithm is stable.

Stability

A sorting algorithm is stable if it preserves the original order of elements with equal key values (where the key is the va algorithm sorts by). For example,



[1]

When the cards are sorted by value with a stable sort, the two 5s must remain in the same order in the sorted output th were originally in. When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted ou

Common Sorting Algorithms

There are many different sorting algorithms, with various pros and cons. Here are a few examples of common sorting alg

Merge Sort

Mergesort is a comparison-based algorithm that focuses on how to merge together two pre-sorted arrays such that the re array is also sorted.

6 5 3 1 8 7 2 4

Insertion Sort

Insertion sort is a comparison-based algorithm that builds a final sorted array one element at a time. It iterates through ar and removes one element per iteration, finds the place the element belongs in the array, and then places it there.

6 5 3 1 8 7 2 4

Bubble Sort

Bubble sort is a comparison-based algorithm that compares each pair of elements in an array and swaps them if they are order until the entire array is sorted. For each element in the list, the algorithm compares every pair of elements.

6 5 3 1 8 7 2 4

Quicksort

Quicksort is a comparison-based algorithm that uses divide-and-conquer to sort an array. The algorithm picks a pivot ele and then rearranges the array into two subarrays $A[p \ldots q-1]$, such that all elements are less than $A[q]$, and $A[q+$ such that all elements are greater than or equal to $A[q]$.

6 5 3 1 8 7 2 4

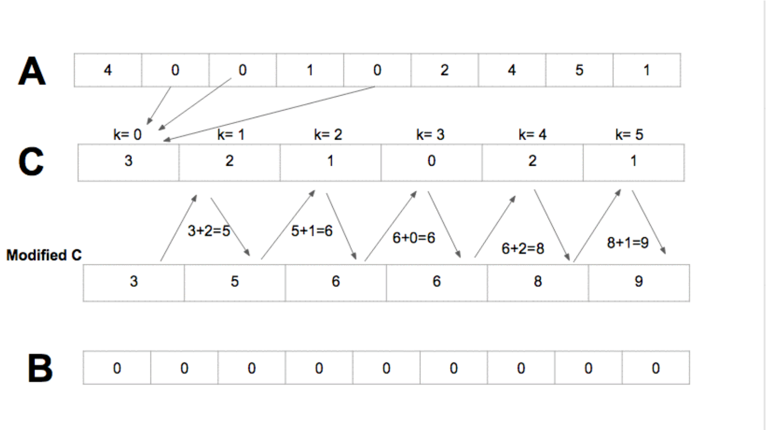
Heapsort

Heapsort is a comparison-based algorithm that uses a binary heap data structure to sort elements. It divides its input intc and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that tc region.

6 5 3 1 8 7 2 4

Counting Sort

Counting sort is an integer sorting algorithm that assumes that each of the n input elements in a list has a key value rang to k, for some integer k. For each element in the list, counting sort determines the number of elements that are less than sort can use this information to place the element directly into the correct slot of the output array.



Choosing a Sorting Algorithm

To choose a sorting algorithm for a particular problem, consider the running time, space complexity, and the expected for input list.

Algorithm	Best-case	Worst-case	Average-case	Space Complexity	Stable?
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$\log n$ best, n avg	Usually not*
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting Sort	$O(k+n)$	$O(k+n)$	$O(k+n)$	$O(k+n)$	Yes

*Most quicksort implementations are not stable, though stable implementations do exist.

When choosing a sorting algorithm to use, weigh these factors. For example, quicksort is a very fast algorithm but can be tricky to implement; bubble sort is a slow algorithm but is very easy to implement. To sort small sets of data, bubble sort r better option since it can be implemented quickly, but for larger datasets, the speedup from quicksort might be worth the implementing the algorithm.

See Also

- Quick Sort
- Insertion Sort
- Radix Sort
- Heap Sort
- Bubble Sort
- Merge Sort
- Counting Sort

References

1. , D., & , W. *Sorting stability playing cards*. Retrieved May 18, 2016, from https://en.wikipedia.org/wiki/File:Sorting_stability_playing_cards.svg