Sorting Algorithms

Bubble Sort

Bubble sortisa sorting algorithmthat compares two adjacent elements and swaps them until they are in the intended order. Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort. Suppose we are trying to sort the elements inascending order. 1. First Iteration (Compare and Swap) 2. Remaining Iteration The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end. In each iteration, the comparison takes place up to the last unsorted element. The array is sorted when all the unsorted elements are placed at their correct positions. In the above algorithm, all the comparisons are made even if the array is already sorted. This increases the execution time. To solve this, we can introduce an extra variableswapped. The value ofswappedis set true if there occurs swapping of elements. Otherwise, it is setfalse. After an iteration, if there is no swapping, the value ofswappedwill befalse. This means elements are already sorted and there is no need to perform further iterations. This will reduce the execution time and helps to optimize the bubble sort. Algorithm for optimized bubble sort is Bubble Sort compares the adjacent elements. Hence, the number of comparisons is nearly equals ton2 Hence,Complexity:O(n2) Also, if we observe the code, bubble sort requires two loops. Hence, the complexity isn*n = n2 Bubble sort is used if

Selection Sort

Selection sort isa sorting algorithmthat selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. Number of comparisons:(n - 1) + (n - 2) + (n - 3) + ... + 1 = n(n - 1) / 2nearly equals ton2. Complexity=O(n2) Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity isn*n = n2. Time Complexities: The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached. Space Complexity: Space complexity isO(1)because an extra variablemin_idxis used. The selection sort is used when

Insertion Sort

Insertion sort isa sorting algorithmthat places an unsorted element at its suitable place in each iteration. Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place. A similar approach is used by insertion sort. Suppose we need to sort the following array. Time Complexities Space Complexity Space complexity isO(1)because an extra variablekeyis used. The insertion sort is used when:

Merge Sort

Merge Sort is one of the most popularsorting algorithmsthat is based on the principle ofDivide and Conquer Algorithm. Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution. Using theDivide

and Conquertechnique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem. Suppose we had to sort an arrayA. A subproblem would be to sort a sub-section of this array starting at indexpand ending at indexr, denoted asA[p..r]. Divide If q is the half-way point between p and r, then we can split the subarrayA[p..r]into two arraysA[p..q]andA[q+1, r]. Conquer In the conquer step, we try to sort both the subarraysA[p..q]andA[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them. Combine When the conquer step reaches the base step and we get two sorted subarraysA[p..q]andA[q+1, r]for arrayA[p..r], we combine the results by creating a sorted arrayA[p..r]from two sorted subarraysA[p..q]andA[q+1, r]. The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e.p == r. After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged. To sort an entire array, we need to callMergeSort(A, 0, length(A)-1). As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array. Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it,mergestep. The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array). The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array. A noticeable difference between the merging step we described above and the one we use for merge sort is that we only perform the merge function on consecutive sub-arrays. This is why we only need the array, the first position, the last index of the first subarray(we can calculate the first index of the second subarray) and the last index of the second subarray. Our task is to merge two subarraysA[p..q]andA[q+1..r]to create a sorted arrayA[p..r]. So the inputs to the function are A, p, q and r The merge function works as follows: In code, this would look like: A lot is happening in this function, so let's take an example to see how this would work. As usual, a picture speaks a thousand words. The arrayA[0..5]contains two sorted subarraysA[0..3]andA[4..5]. Let us see how the merge function will merge the two arrays. This step would have been needed if the size of M was greater than L. At the end of the merge function, the subarrayA[p..r]is sorted. Best Case Complexity:O(n*log n) Worst Case Complexity:O(n*log n) Average Case Complexity:O(n*log n) The space complexity of merge sort isO(n).

Quick Sort

Quicksort isa sorting algorithmbased on thedivide and conquer approachwhere 1. Select the Pivot Element There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element. 2. Rearrange the Array Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right. Here's how we rearrange the array: 3. Divide Subarrays Pivot elements are again chosen for the left and the right sub-parts separately. And,step 2is repeated. The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted. You can understand the working of quicksort algorithm with the help of the illustrations below. The space complexity for quicksort isO(log n). Quicksort algorithm is used when

Heap Sort

Heap Sort is a popular and efficientsorting algorithmin computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees. The initial set of numbers that we want to sort is stored in an array e.g.[10, 3, 76, 34, 23, 32]and after sorting, we get a sorted array[3,10,23,32,34,76]. Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap. Note: As a prerequisite, you

must know abouta complete binary treeandheap data structure. A complete binary tree has an interesting property that we can use to find the children and parents of any node. If the index of any element in the array isi, the element in the index2i+1will become the left child and element in2i+2index will become the right child. Also, the parent of any element at indexiis given by the lower bound of(i-1)/2. Let's test it out, Let us also confirm that the rules hold for finding parent of any node Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort. Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if The following example diagram shows Max-Heap and Min-Heap. To learn more about it, please visitHeap Data Structure. Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap. Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements. The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property. If you're worked with recursive algorithms before, you've probably identified that this must be the base case. Now let's think of another scenario in which there is more than one level. The top element isn't a max-heap but all the sub-trees are max-heaps. To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position. Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node. We can combine both these conditions in one heapify function as This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps. To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element. In the case of a complete tree, the first index of a non-leaf node is given byn/2 - 1. All other nodes after that are leaf-nodes and thus don't need to be heapified. So, we can build a maximum heap as As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element. If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort. The code below shows the operation. Heap Sort hasO(nlog n)time complexities for all the cases (best case, average case, and worst case). Let us understand the reason why. The height of a complete binary tree containing n elements islog n As we have seen earlier, to fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it. In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple oflog(n)comparisons and swaps. During the build_max_heap stage, we do that forn/2elements so the worst case complexity of the build_heap step isn/2*log n ~ nlog n. During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takeslog nworst time because we might have to bring the element all the way from the root to the leaf. Since we repeat thisntimes, the heap_sort step is alsonlog n. Also since thebuild_max_heapandheap_sortsteps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order ofnlog n. Also it performs sorting inO(1)space complexity. Compared with Quick Sort, it has a better worst case(O(nlog n)). Quick Sort has complexityO(n^2)for worst case. But in other cases, Quick Sort is fast. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort. Systems concerned with security and embedded systems such as Linux Kernel use Heap Sort because of theO(n log n)upper bound on Heapsort's running time and constantO(1)upper bound on its auxiliary storage. Although Heap Sort hasO(n log n)time complexity even for the worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort). However, its underlying data structure, heap, can be efficiently used if we want to extract the smallest (or largest) from the list of items without the overhead of keeping the remaining items in the sorted order. For e.g Priority Queues.

Radix Sort

Radix sort isa sorting algorithmthat sorts the elements by first grouping the individual digits of the sameplace value. Then, sort the elements according to their increasing/decreasing order. Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place. Let the initial array be[121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below. Please go throughthe counting sortbefore reading this article because counting sort is used as an intermediate sort in radix sort. Since radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms. For the radix sort that uses counting sort as an intermediate stable sort, the time complexity isO(d(n+k)). Here,dis the number cycle andO(n+k)is the time complexity of counting sort. Thus, radix sort has linear time complexity which is better thanO(nlog n)of comparative sorting algorithms. If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers then it can perform in linear time however the intermediate sort takes large space. This makes radix sort space inefficient. This is the reason why this sort is not used in software libraries. Radix sort is implemented in