Lists

Thursday, 25 January 2024 2:52 AM

Sorting Algorithms

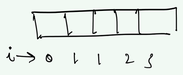
Selection Sort

simple and efficient sorting algorithm that work by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to sorted portion of the list.

```
def selectionSort(self,arr,n):
    for i in range(n-1,-1,-1): /range(n-1):
        self.select(arr,i)
    return arr

def select(arr,i):
    max_index = i
    for j in range(i):
        if arr[j] > arr[max_index]:
            max_index = j
    arr[i],arr[max_index]
```

Same logic for increasing way



Then swap these two nodes

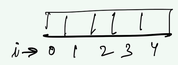
Time: $O(n^2)$

Space: $O(1)$

Not stable

Bubble Sort (Stable Algo)

→ simple sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.



start from i → 0 to (n-i);
and swap repeatedly the largest number

```
for i → 0,n
    for j → 0, n-i-1;
        check for j and j+1
```

Takes $O(n^2)$

Insertion Sort

→ works similar to the way you sort playing cards in your hand.
Array is virtually split into a sorted and an unsorted part.
Values from unsorted part are picked and placed at the correct position in sorted part.

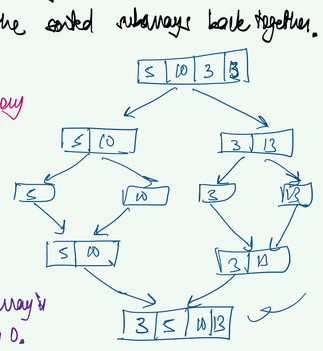
Stable

```
for each i:
    keep the before part sorted
    j = i-1 , key = arr[i]
    while j>=0 and arr[j] > key :
            shift j element
            arr[j+1] = arr[j]
    arr[j+1] = key
```

Merge Sort O(N log N)

→ sorting algo that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together.

→ stable — maintains relative ordering
→ Not in place → requires extra memory



Problem directly related

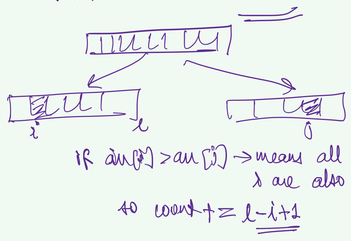
① Count Inversions

Inversion count indicates how far (or close) the array is from being sorted. If the array is already sorted then the inversion count is 0.

Two elements i, j form inversion if a[i] > a[j] and i < j.

while merging the array →

count += mid+1-i if arr[left] > arr[right]



if arr[i] > arr[j] → means all after i are also greater

so count += l-i+1

② Reverse pairs

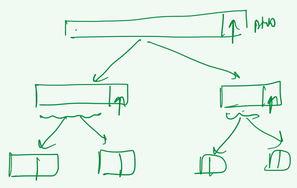
* reverse pair is (i,j) where 0 <= i < j < nums.length and nums[i] > 2*nums[j]

Before merging — count how many reverse pairs exist

Then merge the sorted array and return count

Quick Sort O(n log n)

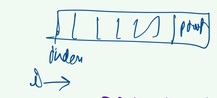
→ based on Divide and Conquer algo that pick an element as a pivot and partitions the given array around the pivot by placing in its correct position in sorted array




```
def quicksort(arr,low,high):
    if low >= high:
        return arr
    else:
        partition = self.partition(arr,low,high)
        self.quicksort(arr, low, partition-1)
        self.quicksort(arr, partition+1, high)
        return arr
```

```
def partition(arr, low, high):
    pivot = arr[low]
    index = high
    for i in range(high, low-1, -1):
        if arr[i] > pivot:
            arr[index], arr[i] = arr[i], arr[index]
            index -= 1
    arr[index], arr[low] = arr[low], arr[index]
    return index
```

```
pivot = arr[high]
```



i →
if arr[i] is less than increase index and swap arr[index] and arr[i]

At last swap pivot with element at index.

Quick Select

selecting algorithm to find the k-th smallest element in an unsorted list. It is related to quicksort.

Similar to quicksort. Only difference is that, instead of recurring for both direction (after finding pivot point), it recurs only for the part that contains the kth smallest element.

Logic — if index of partitioned element is more than k, then we recur for left part. If index = k, we found the k'th smallest element.

This reduces time complexity to $O(n)$

Sort an array using Heap (Heapsort)

— comparison based technique based on Binary Heap (similar to selection sort)

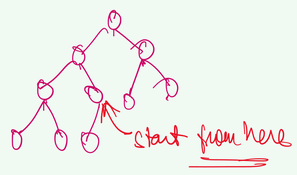
Convert array into heap data structure using heapify, then one by one delete the root node of Max-Heap and replace it with last node in heap and heapify the root. Repeat this until size of heap is greater than 1.

$O(n)$ time
$O(n)$ (recursive stack space)

```
class solution:
    def parent(i):
        return (i-1)//2
    def lchild(i):
        return (2*i+1)
    def rchild(i):
        return (2*i+2)
    def heapify(arr,n,i):
        left = lchild(i)
        right = rchild(i)
        largest = i
        if left < n and arr[largest] < arr[left]:
            largest = left
        if right < n and arr[largest] < arr[right]:
            largest = right
        if largest != i:
            arr[largest], arr[i] = arr[i], arr[largest]
            heapify(arr,n,largest)
```

```
def buildHeap(arr,n):
    for i in range(parent(i),-1,-1):
        heapify(arr,n,i)

def HeapSort(arr,n):
    buildHeap(arr,n)
    for i in range(n-1,-1,-1):
        arr[i],arr[0] = arr[0],arr[i]
        heapify(arr, i, 0)
    return arr
```



---

① Kadane's Algo

② Minimize the height —
Idea → to increase first i towers by k and decrease the rest towers by k after sorting the heights, then calculate max height difference.

③ Two repeated elements
① add largest number to each element
② modify array by multiplying by -1.

④ Next Permutation

⑤ Count Inversions / Reverse Pairs

⑥ Find common elements in three sorted array
— three pointer technique but take care of duplicates

⑦ Rearrange array alternately
i → odd arr[i] >= arr[i] + (arr[max_index] % max_value) * max_value
i → even arr[i] >= arr[i] + (arr[min_index] % max_value) * max_value

⑧ Maximum product subarray (Kadane's Approach)

⑨ Find factorial of large number

⑩ Longest consecutive subsequence — hashing problem