12/12/2021 Radhy Radhy

→ SORTING :- process of arranging items systematically.

- Bubble sort
- Quick sort
- Insertion sort
- count sort
- Selection sort
- Cyclic sort
- merge sort

* BUBBLE SORT : in every step, you are comparing adjacent elements.
This works by swapping adjacent elements if they are in wrong order. Also known as sinking and exchange sort

Example explanation

first pass) 3, 1, 5, 4, 2
(comparing)
→ 3 is larger than 1, it'll swap.

[if on comparing, one number is larger than the adjacent one, nos. will swap (for ascending order)]

1, 3, 5, 4, 2
(comparing)
→ 3 is smaller than 5, it'll not swap

1, 3, 5, 4, 2
→ 5 is larger than 4, swap

1, 3, 4, 5, 2
→ 5 is larger than 2, swap

1, 3, 4, 2, (5) largest

After the every pass the last element will be sorted.
→ exclude sorted part from next pass

Second pass)

1, 3, 4, 2, 5

No swap

1, 3, 4, 2, 5

No swap

1, 3, 4, 2, 5

swap, 2 < 4

1, 3, 2, 4, 5

since 5 (last) element is sorted, in the second pass we will not include it in sorting process.

After second pass, last element of array taken in this pass will be sorted.

Third pass)

1, 3, 2, 4, 5

No swap

1, 3, 2, 4, 5

2 < 3, swap

1, 2, 3, 4, 5

Since, 4 & 5 is sorted in previous passes, we'll not include it in third pass

1, 2, 3, 4, 5 SORTED

→ Complexity

* Space complexity : O(1) //constant

→ since here no extra space is required, i.e., like copying the array etc. is not required.

Also known as inplace sorting algo.

* Time complexity :

1) Best case → Array sorted → O(N) → No. of comparisons

2) Worst case → O(N²) → sorted inopposite i.e. sorting descending order array to ascending.

As the size of array is growing, the no. of comparisons growing

STABLE SORTING ALGO	UNSTABLE SORTING ALGO
→ original order maintained for the values equal to each other after the sorting process.	→ original order not maintained for the values equal to each other after sorting process

* Algorithm for Bubble Sort

```
bubbleSort(array)
  swapped = false
  for i = 0 to length-1          → for j = 1 to length-i
    if array[i-1] > array[i].
      swap(array[i-1], array[i])
      swapped = true
    end if
  end for
end bubbleSort
```

* Pseudocode

```
void bubbleSort(int[] arr){
  boolean swapped ;
  for (int i =0; i<arr.length ; i++){ // counter
    swapped false
    for (int j = 1; j <arr.length-i ; j++){ // for each step, max item will come at the last respective index.
// swap if a<b   if (arr[j] < arr[j-1]){
// swap          int temp = arr[j];
                 arr[j] = arr[j-1];
                 arr[j-1] = temp;
                 swapped = true;
    }} if (!swapped){ break ; // if you didn't swap for a particular value of i, it means the array is sorted, so stop
    }}
```

!swapped = !false = true

Bubble Sort using Recursive fun

Recursive implementation

Pseudocode

```
void bubble(int[]arr, int r, int c){
  if (r==0){
     return ;}
  if (c<r){
        if (arr[c] > arr[c+1]){
           //swap
           int temp = arr[c];
           arr[c] = arr[c+1];
           arr[c+1] = temp;
           }
           bubble(arr, r, c+1);}
        else{
           bubble(arr, r-1, 0);}
        }
}
```

BUBBLE SORT

WORST/AVG CASE :: $O(n^2)$ when array is reverse sorted

BEST CASE :: $O(n)$ Already sorted

Auxiliary Space :: $O(1)$

Sorting in place :: Yes

Stable :: Yes

Selection Sort using Recursive fn

Recursive Implementation

fn call → main()
arr = {4, 5, 3, 1}
selection(arr, arr.length, 0, 0);

Pseudocode

```
void selection(int[] arr, int r, int c, int index){
    if (r==0){
        return;}
    if (c<r){   // r = no. of iteration, c=0
        if (arr[c] > arr[index]){   // if element on c > element on index
            selection(arr, r, c+1, c);   // make this
        }else{ // if they are equal or smaller,
            selection(arr, r, c+1, index);} // do this
    }else{
        int temp = arr[index]; // swap
        arr[index] = arr[r-1];
        arr[r-1] = temp;
        selection(arr, r-1, 0, 0);}}
```

from this you'll get highest no. in array

after getting highest no. in array swap it to last index of array.

SELECTION SORT

worst case → $O(n^2)$

Avg case → $O(n^2)$

Best case → $O(n^2)$

Space complexity → $O(1)$

Stable → NO.

→ **INSERTION SORT** :- Array virtually split into a sorted and unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Example - 5, 3, 4, 1, 2

* Outer loop will run from 0 to (n-2) (n = length of array)

1st iteration $i<n-2$, $j>0$

$i=0, j=1$

[5, 3], 4, 1, 2

$3<5$ → swap

3, 5, 4, 1, 2

2nd iteration

$i=1, j=2$

[3, 5, 4], 1, 2

$4<5$ → swap & $4>3$ → break inner loop

3, 4, 5, 1, 2

3rd iteration

$i=2, j=3$

[3, 4, 5, 1], 2

$1<5, 4, 3$ → swap all

1, 3, 4, 5, 2

4th iteration $i=3, j=4$

[1, 3, 4, 5, 2]

$2<5, 4, 3$ → swap

$2>1$ → break loop

1, 2, 3, 4, 5 Sorted.

ALGORITHM

Step 1 - Assume first element is already sorted.
Step 2 - Pick next element.
Step 3 - compare key with all elements in the sorted part.
Step 4 - If element in the sorted part is smaller than current element, break inner loop
If element in sorted part > current element, swap with them one by one.
Step 5 - Repeat until the array is sorted.

TIME COMPLEXITY

(1) Worst case (descending order) → $O(n^2)$
(2) Best case (already sorted) → $O(n)$

* Why to use Insertion Sort?

→ Adaptive: steps get reduced if array is sorted (i.e., no. of swaps are reduced as compared to bubble sort)
→ Stable sorting Algorithm
→ used for smaller values of n: works good when array is partially sorted.

Tails part in hybrid sorting algorithm.

PSUEDOCODE

```
void InsertionSort(int[] arr){
  for(int i=0; i<arr.length-1 ; i++) {
    for(int j = 1+i ; j>0 ; j--){
      if(arr[j] < arr[j-1]){
        swap(arr, j, j-1);
      }else{
        break;
      }
    }
  }
}
```

```
// void swap(int[] arr, int first, int second)
{ int temp = arr[first];
  arr[first] = arr[second];
  arr[second] = temp;
}
```

INSERTION SORT

Time complexity -

→ worst case → $O(n^2)$ Reverse order

→ Best case → $O(n)$ sorted

→ space complexity → $O(1)$

→ stable → Yes

→ Sorting in place → Yes

08-01-2022 Amazon asr questions

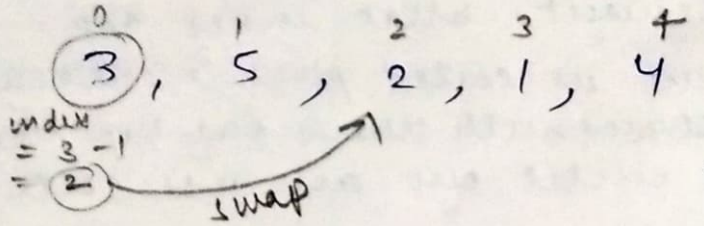
→ CYCLIC SORT :- in-place sorting algo.

→ when nos. from 1 to N given → use cyclic sort in sequence

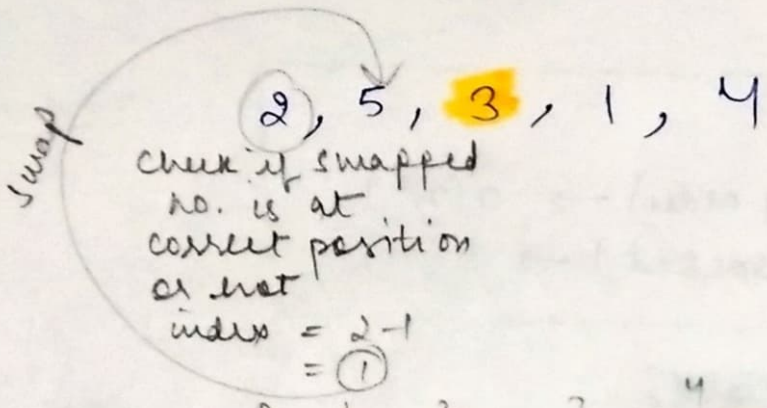
WORKING - check, swap, move

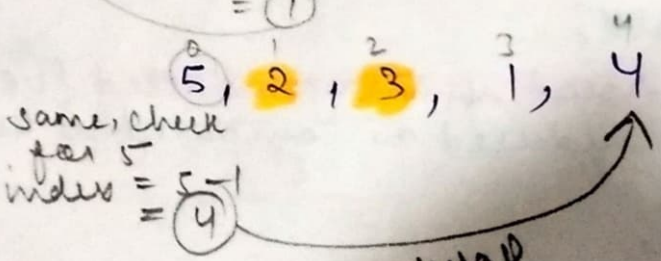
Example

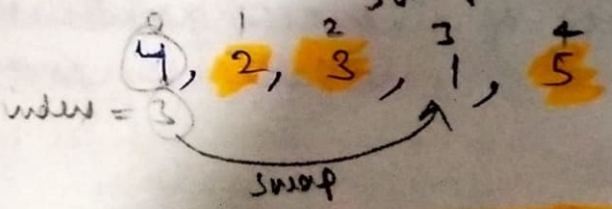
worst case example →

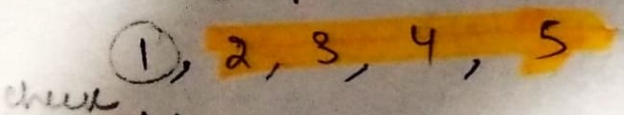


index = value - 1









check index = 1 - 1 = 0 ✓ then move forward to check next item.

1, 2, 3, 4, 5 sorted

In worst case
= n-1 swaps will be done. After it, we will check every index value if it is on correct index or not
∴ (n-1) + n
= (2n+1) swaps
O(N) linear

* every unique no. will be swapped once.
* Not incrementing i if value on current index is not on sorted index.

PSEUDOCODE

```
void cyclicSort(int[] arr){
    int i=0;
    while(i<arr.length){
      int correct_index = arr[i]-1;
      if(arr[i] != arr[correct_index]){
        swap(arr, i, correct_index]);
      }else{
        i++;
      }
    }
}
```

```
// swap(int[]arr, from, to){
    int temp = arr[from];
    arr[from] = arr[to];
    arr[to] = temp; }
```

* Hence, in sorted version, element == index

i.e. if arr = 0,1,2,3,4

so indexes will be 0,1,2,3,4 (0 1 2 3 4)

which is value of element == index

Note if range = [0,N]

index = value

if range = [1,N)

index = value + 1

Questions using index

→ Missing no. question, in array list too

→ duplicate no. question, in array list too

→

* MERGE SORT :- based on divide & conquer.

It divides array into sub-arrays and do recursion calls to sort them and then merge all the sub-arrays to one main array. Fully based on divide & conquer.

→ **steps :-**

1. Divide array into sub-arrays
2. Get all sub-arrays sorted via recursion
3. When you get sorted parts/sub-arrays, merge them.
   How will you merge two sorted sub-arrays

→ **explanation →**

example → (i) arr1 = [3, 5, 9, 19, 32]
arr2 = [4, 6, 8]

compare 3 and 4, elements of both array.
3 < 4, so 3 will be placed at 0th index.

(ii) arr1 = [3, 5, 9, 19, 32]
arr2 = [4, 6, 8]
compare 5 and 4
5 > 4, so place 4 at 1st index.

(iii) arr1 = [3, 5, 9, 19, 32]
arr2 = [4, 6, 8]
compare 5 and 6, 5 < 6, so place 5 at 2nd index.

(iv) arr1 = [3, 5, 9, 19, 32]
arr2 = [4, 6, 8]
compare 6 and 9, and 6 < 9, so 6 → 3rd index

(v) arr1 = [3, 5, 9, 19, 32]
arr2 = [4, 6, 8]
compare 8 and 9, 8 < 9,
8 → 4th index

(vi) arr1 = [3, 5, 9, 19, 32]
arr2 = [4, 6, 8]
Now, if one of the comparing array get empty, place rest of the elements of another array concurrently.
9 → 5th index, 19 → 6th, 32 → 7th

sorted → merged array →

[4, 5, 6, 8, 9, 19, 32]

Pseudocode

```
int[] mergeSort(int[] arr){
    if(arr.length == 1){
      return arr;}
    int mid = arr.length/2;
    int[] first = mergeSort(Array.copyOfRange(arr, 0, mid));
    int[] second = mergeSort(Array.copyOfRange(arr, mid, arr.length));
    return sort(first, second);
    }
int[] sort(int[] first, int[] second){
    int[] result = new int[first.length + second.length];
    int i=0;
    int j=0;
    int k=0;
    while(i<first.length && j<second.length){
      if(first[i]<second[j]){
          result[k] = first[i];
          i++;}
      else{
          result[k] = second[j];
          j++;}
      k++;}
    while(i<first.length){
        result[k] = first[i];
        i++;
        k++;}
    while(j<second.length){
        result[k] = second[j];
        j++;
        k++;
    }
    return result;
}
```

→ why merge sort preferred in linked list?

Sol). Unlike array linked list is not continuous memory allocation.

PROS

→ Large size list
→ Suitable for Linked list.
→ supports external sorting
→ stable

slower for small size list

CONS

→ extra space → no small problem

Time complexity of merge sort :- $O(N * \log N)$

Space complexity :- $O(N)$

Recurrence relation :-

$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + (N-1)$$

$$\boxed{T(N) = 2T\left(\frac{N}{2}\right) + (N-1)}$$

$\log_2 2^1 = 1$, $K = 1$.

So, it is $O(N \log N)$

According to master's theorem

Best & worst case

In-place Merge Sort ⟷ it is a ptr based virtual sort

s → start
e → end
m → mid

Pseudocode

```
void mergeSort(int[] arr, int s, int e) {
    if(e - s == 1){
      return; }
    int mid = (s+e)/2;
    mergeSort(arr, s, mid);
    mergeSort(arr, mid, e);
    sort(arr, s, mid, e); }
```

in this sorting takes place in original array.

```
void sort(int[] arr, int s, int m, int e) {
    int[] mix = new int[e-s];
    int i = s; int j = m; int k = 0;
    while(i<m && j<e){
      if (arr[i] < arr[j]){
        mix[k] = arr[i];
        i++; }
      else{
        mix[k] = arr[j];
        j++; }
        k++; }
    while (j<e){
      mix[k] = arr[j];
      j++; k++; }
    while(i<m){
      mix[k] = arr[i];
      i++;
      k++; }
    for(int l = 0; l < mix.length; l++)
    { arr[s+l] = mix[l];
    }}
```

[26-01-22]

QUICK SORT :-

→ Pivot → choose any element from array → after first pass
  ↳ All the elements smaller than pivot will be on LHS of pivot and greater elements will be on RHS of pivot.

→ we'll take last element of array as pivot.

→ we can take any element
  ↳ Random element
  ↳ corner elements
  ↳ middle element

Example –

{10, 80, 30, 90, 40, 50, (70)} → Pivot

Now we'll partition around 70

elements < 70 | elements > 70

{10, 30, 40, (50)} Pivot | {90, (80)} Pivot

Running call

{10, 30, (40)} pivot | { } | { } | {90}

{10, (30)} pivot | { }

{10} | { }

Recurrence Relation (imp)

Total elements = N

K | 1 | N-K-1

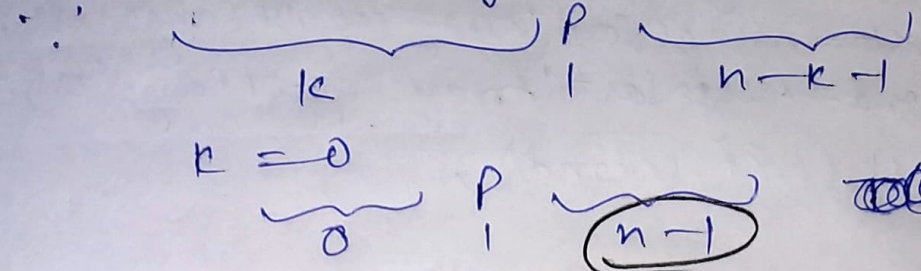
This is K no. of elements | one element

$$T(N) = \underbrace{T(K)}_{\text{sort K elements}} + \underbrace{T(N-K-1)}_{\text{sort these no. of elements}} + \underbrace{O(n)}_{\text{to put pivot on correct position}}$$

worst case :- if the pivot element is the largest or smallest no. of the array/list.

Time complexity

$k = 0$, which means one side of the array, like either right or left side is empty.



$$T(n) = T(0) + T(n-1) + O(n)$$

$$\boxed{T(n) = T(n-1) + O(n)}$$

* **worst case complexity** $= O(n^2)$

Best case :- if the pivot is middle and have its correct position and itself dividing array according to LHS & RHS.

$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + O(N)$$

$$\boxed{T(N) = 2T\left(\frac{N}{2}\right) + O(N)}$$

* **Best case complexity** $= O(N \log N)$

→ **Not stable** – if two elements have same value in same order maybe they will not appear as they were originally, after sorting.

→ **In place** – sorting originally. No extra space is taken.

→ **Hybrid sort** :- Merge sort + insertion sort

works well in partially sorted array.

Pseudo code of Quick Sort

```
void sort(int[] nums, int low, int high){
base condition { if (low >= high){
                  return; }
    int s = low;
    int e = high;
    int mid = s + (e-s)/2;
    int pivot = nums[m];
    while(s<=e){
        while (num[s] < pivot){
            s++; }
        while(num[e] > pivot){
            e--; }
        if(s<=e){
            int temp = num[s];
            num[s] = num[e];
            num[e] = temp;
            s++;
            e--; }
    } // now my pivot is at correct index, please sort two halves now.
    sort(nums, low, e);
    sort(nums, s, high);
}
```

s = start index
e = end index
pivot = m (middle)

placing pivot at its right index

→ Revise thoroughly again

→ Another approach of implementing (gfg) taking adjacent element

* inbuilt sort → use hybrid sorting.
Syntax → Arrays.sort(array name);