Code of Academic Integrity

I affirm that

- This code is my own original code and is not a borrowed code, either from other students or from assignments for other courses.

- I have not given or received any unathorize help on this assignment.

- This submission is free from
  - Plagiarism
  - fabrication of facts.
  - Unauthorized assistance.
  - Collusion

This submission gives proper credit to sources and references, acknowledges the contributions and ideas of others relevant to this academic work.

- This submission was prepared by me fully adhering to the rules that govern this assignment regarding resource material, electronic aids, copying, collaborating with others, or engaging in any other behavior that subverts the purpose of the assignment and the directions of the teacher.

21BRS1529

Design and Analysis of Algorithm

Practice Problem Sheet 1

1) Input: An n-element array A
Output: The Array A with its elements rearranged into increasing order.

Pseudocode

```
Algorithm (A)
1  for i=1 to A.length-1
2      minIndex = i
3      for j = i+1 to A.length
4      if A[j] < A[minIndex] and j ≠ minIndex
5          minIndex = j
6      swap A[i] with A[minIndex]
```

~~Re~~

The Algorithm needs to run for only n-1 elements rather than all n elements because the last iteration will compare A[n] with the minimum element in A[1 . . n-1] in line 4 and swap them if necessary.

Running Time

For both the cases ~~(sort~~ → Best case (Sorted Array) and worst case (reverse sorted array), the algorithm will always take one element at a time and compare it with all other elements. So the running time for both scenario will be $\Theta(n^2)$

Best case, worst case, Average case is $\Theta(n^2)$

Proof of correctness

Loop Invariant

At the start of each iteration of the outer for loop of lines 1-6, the subarray $A[1..i-1]$ consists of $i-1$ smallest elements of A, sorted in increasing order.

Run Time

Let's assume the inner ~~loop~~ for loop in line 3-5 executed for t_j times for $j = 2, 3, \ldots, n$ when $n = A.length$. Now that line 5 will be executed less than $t_j - 1$ times in the average case.

line	cost	Times
1	c_1	n
2	c_2	$n-1$
3	c_3	$\sum_{j=2}^{n} t_j$
4	c_4	$\sum_{j=2}^{n} (t_j - 1)$
5	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6	c_6	$n-1$

$$\sum_{j=2}^{n} t_j = (n-1) + (n-2) + \ldots + 1$$

$$= \frac{n(n-1)}{2}$$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} t_j - \sum_{j=2}^{n} 1$$

$$= \frac{n(n-1)}{2} - (n-1) = \frac{(n-3)(n-2)}{n}$$

$\therefore$ we can calculate running time as

$$T(n) = c_1(n-1) + (c_2+c_6)n + c_3\frac{n(n-1)}{2} + (c_4+c_5)\frac{(n-1)(n-2)}{2}$$

for Best case: Array Sorted $\rightarrow c_5 = 0$.

Worst case: $c_5 = 0$.

$\rightarrow$ reduced to form $T(n) = an^2 + bn + c$

i.e the algorithm will run at $\Theta(n^2)$ time.

This type ~~than~~ (Selection sort) Vs Insertion Sort.

Selection sort

$\rightarrow$ has time complexity $O(n^2)$ in all cases

Insertion Sort

has time complexity $O(n^2)$ in worst case.

Best case: $O(n)$

For Sorted types of Inputs

Insertion Sort works better than Selection Sort as its Best case time complexity is $O(n)$

For Unsorted types of Inputs

Both works same as both take $O(n^2)$ time for completion.

2.) This type of approach described called as Bubble Sorting

Bubble sort:

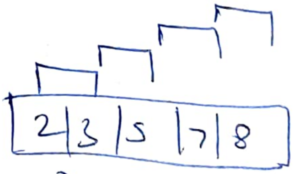
```
begin BubbleSort(A)
    for all elements of A
        if A[i] > A[i+1]
            swap (A[i], A[i+1])
        end if
    end for.

    return A.
end Bubble sort
```

In Bubble sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total no of comparison will be,

$$(n-1)+(n-2)+ \dots 3+2+1$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

Adaptive bubble sort




```
~~void~~ begin Bubble sort (A, n)
    declare flag.
    for i = 0 do i = A.length.
        flag = 0
        for j = 0 to j = A.length.
            if ~~B~~ A[j] > A[j+1]
                swap (A[j], A[j+1])
```

flag = 1

if flag = 0

break

end Bubble Sort

Total comparisons : n-1 O(n)

Proof of correctness

Input arr[] = {6, 3, 0, 5}

First Pass :

- Bubble sort starts with very first two elements comparing them to check which one is greater.

(6, 3, 0, 5) → (3, 6, 0, 5) → 1st 2 elements are compared and swaps since 6>3

- (3, 6, 0, 5) → (3, 0, 6, 5) since 6>0
- (3, 0, 6, 5) → (3, 0, 5, 6) since 6>5.

Second pass

- (3, 0, 5, 6) → (0, 3, 5, 6)
- (0, 3, 5, 6) → (0, 3, 5, 6)
- (0, 3, 5, 6) no change.

Array now sorted and no more pass will happen

Run Time Analysis

Worst and Average Case Time Complexity.
O(N²). The worst case occurs when an array is reverse sorted.

Best Case Time Complexity : O(N). The best case occurs when an array is already Sorted.

Auxilary Space : O(1)

On comparing Bubble sort algorithm to the insertion sort algorithm we came to conclusion that the bubble sort performs better as it has a better best case time complexity.

3.) Input: An n-element array of numbers.

Output: Inversion count of array.

Logic

Inversion Count for an array indicates how close the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, inversion count is maximum.

Example

Input: A[] = [3,2,1]

Output : 3

Explaination :

The three pair of inversions are –
(3,2), (3,1), (2,1).

Approach 1: Brute Force

A Simple approach to consider every possible pair of the array and check if the ~~given condition~~ pair satisfies the given condition. If true, increment count.

Pseudocode

```
begin getInversions (A, n)
    ~~init~~    count = 0
        for i = 0 to i = n-1
            for j = i+1 to j = A.length().
                if A[i] > A[j]
                    ++count

        Return count.
END getInversions.
```

Time Complexity: $O(N^2)$

Space Complexity: $O(1)$

Approach 2: Merge Sort.

Divide the array into two parts. For each left and right half, count the inversions

Pseudocode

```
COUNT-INVERSIONS (A, p, r)
   if p ≥ r
        return 0
   q = ⌊(p+r)/2⌋
   left = Count-INVERSIONS (A, p, q)
   right = Count-Inversions (A, q+1, r)
   Inversions = left + right + Merge (A, p, q, r)
   return Inversions

MERGE (A, P, P, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1...n1] and R[1...n2] be new
   arrays.
  for i=1 to n1
       L[i] = A[p+i-1]
  for j=1 to n2
       R[j] = A[q+j]
  L[n1+1] = ∞
  R[n2+1] = ∞
  i=1
  j=1
  inversions = 0
  for k=p to r
       if L[i] ≤ R[j]
            A[k] = L[i]
            i = i+1
       else
```

inversions = inversions + $(n_1 - i + 1)$

$A[K] = R[j]$

$j = j + 1$

return inversions

Time complexity $O(N \log N)$

Space complexity: $O(N)$

The more the number of inversions in an array, the more times the inner loop will run.

Worst case - Scenario

The numbers will be totally inverted, we would have list as [6, 5, 4, 3, 2, 1]

The total number of inversion will be:

$$\frac{n(n-1)}{2} = O(n^2).$$

Best case

$O(n \log n)$

Relationship with Insertion Sort

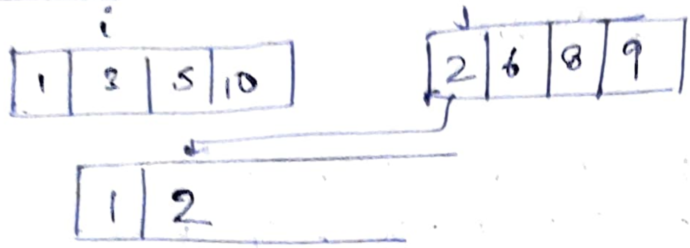
The more the number of inversions in an array, the more times the inner loop will run

Thus maximum number of inversions are possible when array is sorted.

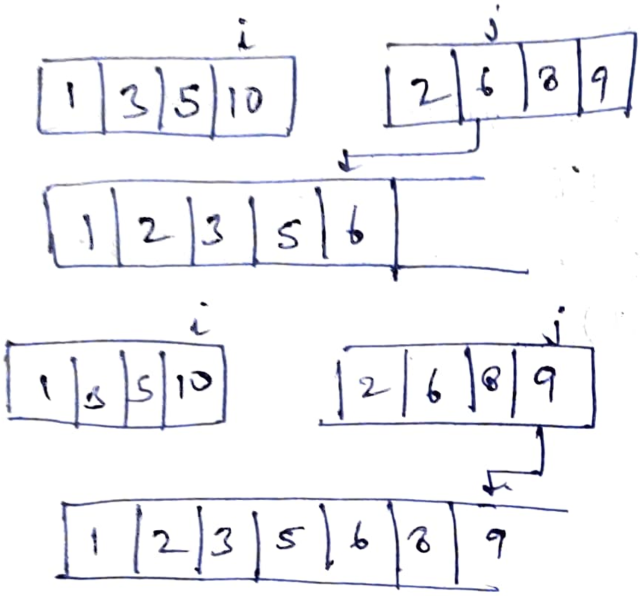
So, the higher the number of inversions in an array, the longer insertion sort will take to sort the array

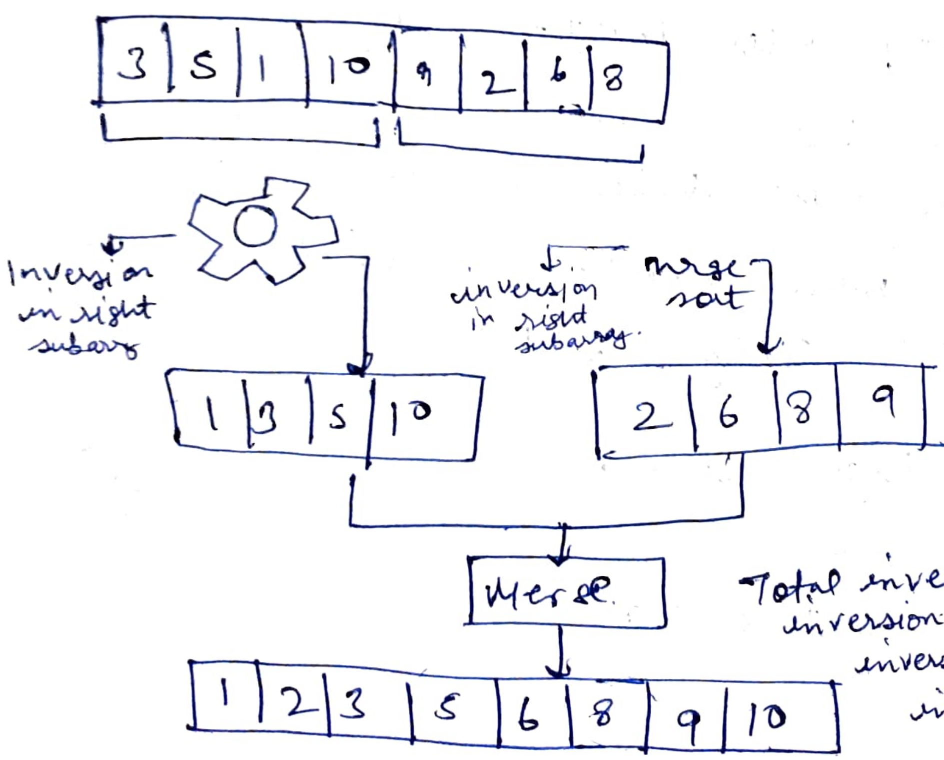
Similarity between Insertion Sort and Inversion Algorithm: Yes, there is a direct relation between the complexity of Insertion Sort and the number of inversions

Illustration



Since A[i] > A[j]





Total inversion = inversion in left + inversion in right + inversion in merge

4) Input: a positive number n

Output: distinct Non-Negative Integer pairs (x, y) that satisfy the inequality $x^2 + y^2 < n$

Logic

A simple solution is to run two loops. The outer loop goes for all possible values of x (from 0 to $\sqrt{n}$). The inner loops pick all possible values of y for the current value of x (picked by outer loop).

PseudoCode

begin countSolutions(n)
  res = 0
  for x = 0 to ~~xx~~ sqrt(n) -1 do
    for y from 0 to sqrt(n - x*x) -1 do
      res = res + 1
  return res.

An upper bound for the time complexity is $O(n)$. Outer loop runs $\sqrt{n}$ times and inner loop runs $\sqrt{n}$ times.

Using an efficient solution, we can find the count in $O(\sqrt{n})$ time.

Pseudocode

```
countSolutions (n):
    x = 0
    ycount = floor (sqrt(n))
    res = 0
    while ycount > 0 do
        res += count
        x += 1
        while ycount > 0 and x*x + (ycount-1) * (ycount-1) >= n do
        ycount -= 1
    return res
```

This algorithm is more efficient than the previous version, as it only calculates the number of solutions without actually iterating over all possible pairs.

Time complexity : $O(\sqrt{n})$

In every step inside the inner loop, the value of ycount is decrement by 1. The value ycount can decrement at most $O(\sqrt{n})$ times as ycount is counted y values for x=0. In the outer loop, the value of x is incremented. The value of x can also increment at most $O(\sqrt{n})$ times as the last x is for ycount equals to 1.