NAME-SHISHU

REG-2020CA089

(4) We can express insertion sort as a recursive procedures as follows. In order to sort $A[1 \dots n]$, we recursively sort $A[1 \dots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \dots n \dots 1]$. Write a recurrence for the running time recursive version of insertion sort.

There are two steps in this recursive sorting algorithm:

step1: Sort the sub-array $A[1..n-1]$

step2: $A[n]$ into the sorted sub-array from step 1 in proper position.

For $n=1$, step 1 doesn't take any time as the sub-array is an empty array and step 2 takes containt time, i.e, the algorithm runs in $O(1)$ time.

For $n>1$, step 1 again calls for the recursion for $n-1$ and step2 run in $O(n)$ time.

So, we can write the recurrence as:

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ T(n-1)+O(n) & \text{if } n>1 \end{cases}$$

Let us first write the pseudo code for auxiliary procedure required to insert $A[n]$ into the sorted array as folloco.

	COST	TIMES
recursive-insertion (A, n) {		
if n< 1:	c1	1
return A		
recursive-insertion (A, n-1)	T(n-1)	1
Key = A[n]	c2	1
While n>0 and Key< A[n-1]	c3	t(n)+1
A [n] = A[n-1]	c4	t(n)
A -= 1	c5	t(n)

$A[n] = key$ c_6 1

return A c_7 1

}

Here $t(n)$ is the number of times that the while loop needs to be iterated when the function is called with the value n. This depends on the input A. So we migth need to make separate worst -case and best - case analysis.

So we have

$$T(n) = T(n-1) + c_1 + c_2 + c_3 + c_6 + c_7 + (c_3 + c_4 + c_5)\,t(n)$$

Now the best best case, the input is already sorted, and so the while loop is never executed and so $t(n) = 0$, hence

$$T(n) \geq T(n-1) + c_1 + c_2 + c_3 + c_6 + c_7$$

So $T(n) = \Omega(n)$ // In Best case

In the worst case, the input is sorted in reverse order, and so that $t(n) = n-1$

hence,

$$T(n) \leq T(n-1) + c_1 + c_2 + c_3 + c_6 + c_7 + \underbrace{(c_3 + c_4 + c_5)(n-1)}_{\simeq n}$$

$$T(n) \leq \underbrace{T(n-1) + c_1 + c_2 + c_3 + c_6 + c_7}_{\simeq n}$$

$$T(n) \simeq O(n \cdot n) \simeq O(n^2).$$

⑤ Describe a O(n log n)-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exits two elements in S whose sum is exactly x.

So, the problem asks for a Search algorithm and we already know binary search is an efficient one at that which run at O(lgn) time for a sorted array

So, we can sort the array with merge sort (O(nlgn)) and then for each element S[i] in the array, we can do a binary search for x - S[i] on the sorted array (O(nlgn)). So, the overall algorithm will run at O(nlgn)) time.

But this is not better approch. Solving for above problem we can extends some ~~extra~~ another search that is two-way-search. Simuntaneous search from both end of the array, to chech if two elements sums up to expected sum x.

```
Sum_Search (S, x)
   Merge-sort(S, 1, S.length)
   left = 1
   right = S.length
   while (left < right)
      if S[left] + S[right] == x
            return true;
      else if S[left] + S[right] < x
            left = left + 1
      else
            right = right - 1
   return false.
```

(6)

Solution (a) Sorting sublists.

for input of size k, insertion sort runs on $O(k^2)$ worst-case time. So, worst-case to sort n/k sublists, each of length k will be n/k.

$O(k^2) = O(nk)$

-means, Sorting each list takes $ak^2 + bk + c$ for some constants a, b and c. We have n/k. ~~have~~ of those, thus;

$$\frac{n}{k}(ak^2 + bk + c) = ank + bn + \frac{cn}{k} = O(nk).$$

(b) Merging sublist

We have n elements divided into n/k sorted sublists each of length k. To merge these n/k sorted sublists to get a single sorted list of length n, we have to take 2 sublists at a time and continue to merge them.

Sorting a sublists of length k each takes

$$T(a) = \begin{cases} 0 & \text{if } a = 1 \\ 2T(a/2) + ak & \text{if } a = 2^p, \text{ if } p > 0. \end{cases}$$

Since we have two arrays each of length $\frac{a}{2}k$

Proof by induction

Base: Simple as ever:

$$T(1) = 1k \lg 1 = k \cdot 0 = 0$$

Step : Assume that $T(a) = ak \lg a$ and we calculate $T(2a)$

$$T(2a) = 2T(a) + 2ak = 2(T(a) + ak) = 2(ak \lg a + ak)$$
$$= 2ak(\lg a + 1) = 2ak(\lg a + \lg 2) + 2ak \lg(2a)$$
$$= 8$$

This proves it. Now if we substitute the number of sublists n/k for a

$$T(n/k) = \frac{n}{k} k \lg \frac{n}{k} = n \lg(n/k)$$

while this is exact only when n/k is a power of 2, it tells us that overall time complexity of merge is $O(n \lg(n/k))$.

(c) Largest value of k

For the modified algorithm to have the same asymptotic running time as standard merge sort, $O(nk + n\lg(n/k))$ must be same as $O(n \lg n)$.

To satisfy this condition, k cannot grow faster than $\lg n$ asymptotically. If it does then because of the nk term, the algorithm will run at worse asymptotic time than $\Theta(n \lg n)$.

So, let's assume, $k = O(\lg n)$ and see if we can meet the criteria..

$$O(nk + n\lg(n/k)) = O(nk + n\lg n - n\lg k)$$
$$= O(n\lg n + n\lg n - n\lg(\lg n))$$
$$= O(2n\lg n - n\lg(\lg n))$$
$$= \Theta(n \lg n).$$

$\lg(\lg n)$ is very small compared to $\lg n$ for sufficiently larger value of n.

(d) The Value of K in practice

In practice, K should be the largest list length on which insertion sort is faster than merge sort.

(7) (a) List of Inversions

Inversions in the given array are: (1,5), (2,5), (3,4), (3,5), and (4,5). (Note: Inversions are specified by indices of the array, not by value.)

(b) Array with most Inversions

The array with elements from the set 1, 2 .... n. With the most inversions will have the elements in reverse sorted order i.e. (n, n-1, ... 2, 1)

As the array has n unique elements in reverse sorted order, for every unique pair of (i, j), there will be an inversion. So, total number of inversion = number of ways to choose.

(c) Relation with Insertion Sort

If we take at the pseoudocode for insertion sort with the definition of inversion in mind, we will realize that more the number of inversions in an array, the more times the inner while loop will run.

This is also in line with our finding in sub-~~program~~ problem b. Maximum number of inversions are possible when the array is reversed sorted.

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

(d) Alogrithm to Calculate Inversions

Although a hint to modify merge sort is already given, without that also we shoud think of divide-and-conquer dlogrithms whenever we see running time with lg n term.

As was done in merge sort, we need to recursively divide the array into halves and count number of inversions in the sub-arrays. This will result in lgn step and O(n) operations in each step to count the inversions. All in all a O (nlgn) algorithm.

The problem did not specifically asked to write pseudocode, but we can do that as well for the sake of completion.

We can rewrite Merge-Sort as follows to repeatedly subdivide the array and count number of inversions in each half.

```
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)
    inversing = left + Right + Merge (A, p, q, r)
    return inversions.
}
```

And here is modified Merge-Sort psedo code that actually counts the number of inversion in linear time.

```
Merge (A, P, q, r) {
    n1 = q - P + 1
    n2 = r - q
    Let L[1...n1] and R[1....n2] be new array
    for i = 1 to n1
        L[i] = A[p + i - 1]

    for j = 1 to n2
        R[j] = A [q + j]

    L[n1 + 1] = ∞
    L[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 + (n1 - i + 1)
            A[k] = R[j]
            j = j + 1
    return inversions
}
```

⑧ solution

- **Insertion sort** is stable because the comparison in the inner while will move elements to right only when they greater than key. If they are equal they don't move to right.

- **Merge sort** is stable the comparision in the merge comapare L and R array only check if L is less than R. If it is equal it uses the element from L array which is in order with the input.

- **Heap Sort** is not stable because the comparison is between left and right child and picks the greatest element. ex: 5, 3, 3, 4. element 3 is not inserted in the correct order.

- **Quick sort** is not stable because we balanced split random procedure to get similar element which can easily pick an make it pivot element in the beginning and

Q Give a simple sheme that makes any sorting algorithm stable. How much additional time and spase does your scheme entail?

All we need to make sure is save the inserted order in another array and use it to break the tie for similar element values. We need to store indices from 1 to n each will need log n space (since an element has to written in the form 2^x and x will be no. of bits ex: 8 need 3 bits!) we pick log n based on the maximum no. All the elements need $\theta(n \lg n)$ space.

Additional time: the worst case occurs when all elements are same and we need compare indices for every element. But the comparision occurs in constant time.

(10) Sol:

first run through the first of integer and converted each one to base n, then Redix sort then, each number will have at most $\log_n n^3 = 3$ digits so, there will only need to be 3 passes.

For each pass, there are n possible values which can be taken on, so we can use Counting sort to sort each digit in $O(n)$ time