DSA Project

Algorithms Time and Space Complexities:

20K-0153 Syed Ahmed Mehmood

20K-0157 M Qasim Fuzail

Insertion Sort:

Time Complexities:

Best Case O(n): When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

Average And Worst-Case O(n^2):

$T(N) = 1 + 2 + 2N + 2(N-1) + (N-1) + (N-1) + (X+1) + X + 2X + 3X + X + X + 2(N-1) + 1$

$T(N) = 8N + 9X - 1$

$X = 1 + 2 + 3 + \ldots\ldots\ldots. + (N-1)$

IN A.P:

$$1 + 2 + 3 + \ldots\ldots\ldots k = \frac{k(k+1)}{2}$$

$$X = \frac{(N-1)(N-1+1)}{2}$$

$$X = \frac{N(N-1)}{2}$$

$$T(N) = 8N + \frac{9}{2}N^2 - \frac{9}{2}N - 1$$

$$T(N) = \left(8 - \frac{9}{2}\right)N + \frac{9}{2}N^2 - 1$$

$$T(N) = O(N^2)$$

Space Complexity: Space complexity is O(1) because an extra variable key is used.

Bubble Sort:

Time Complexities:

Best Case O(n): If the array is already sorted, then there is no need for sorting.

Average And Worst-Case $O(n^2)$:

T(N) = 1 + 1 + 2N + (N-1) + (X+1) + X + 3X + 2X + 2X + 2X + 1

T(N) = 11X + 3N + 3

X= 1 + 2 + 3 + …………. + (N-1)

IN A.P:

$$1 + 2 + 3 + \dots\dots\; k = \frac{k(k+1)}{2}$$

$$X = \frac{(N-1)(N-1+1)}{2}$$

$$X = \frac{N(N-1)}{2}$$

$$T(N) = \frac{11}{2} N^2 - \frac{11}{2} N + 3N + 3$$

$T(N) = O(N^2)$

Space Complexity: Space complexity is O(1) because an extra variable is used for swapping.

Merge Sort:

Time Complexities:

Best, Average And Worst-Case O(n*logn):

T(N) = $\{B \quad N = 1\ T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + CN$

T(N) = 2T$(\frac{N}{2})$ + CN --------EQ.1

T(N) = 2T$(\frac{N}{4})$ + $\frac{CN}{2}$ --------EQ.2

Substituting eq.2 in eq.1:

T(N) = 4T$(\frac{N}{4})$ + 2CN --------EQ.3

T$(\frac{N}{4})$ = 2T$(\frac{N}{8})$ + $\frac{CN}{4}$ --------EQ.4

Substituting eq.4 in eq.3:

T(N) = 8T$(\frac{N}{8})$ + 3CN --------EQ.5

Pattern Identified:

T(N) = 2^i T(N/2^i) + i CN

At some point:

T(N/2^i) = 1

T(N) = 2^i T(N/2^i) + i CN -------EQ.6

T(N/2^i) = 1 = B

N/2^i = 1

N = 2^i

Taking log on both sides:

$Log_2 N = i$ -------------EQ.7

Substituting eq.7 in eq.6:

$$T(N) = 2^{\log_2 N}\, T(N/2^{\log_2 N}) + Log_2 N(CN)$$

$$T(N) = 2^{\log_2 N}\, T(1) + (CN)Log_2 N$$

$$T(N) = B\ 2^{\log_2 N} + (CN)Log_2 N$$

$$T(N) = B(N) + C(NLog_2 N)$$

$$T(N) = 0(N\ Log_2 N)$$

Space Complexity: Space complexity is O(n) because to sort the unsorted array. Merge sort is splitting and creating subarrays but the sum of sizes of all the subarray will be n.

Heap Sort:

Time Complexities:

Best, Average And Worst-Case O(n*logn):

Phase 1: Construction of Heap

Phase 2: Delete root repeatedly

1:

For n elements

1. Add element--------- c
2. Rearrange ------------ log n

Total = nc + n logn

2:

Delete repeatedly

For n elements

1. Remove------------------------------ c
2. Down heap bubbling ------------ log n

Total = nc + n logn

Adding phase 1 and 2:

T(N) = 2NC + 2N Log N

T(N) = O(N LogN)

Space Complexity: Space complexity is O(1)

Quick Sort:

Time Complexities:

Best And Average-Case O(n Logn):

Let’s T(n) be the time complexity for best cases

n = total number of elements

then

T(n) = 2*T(n/2) + constant*n

2*T(n/2) is because we are dividing array into two arrays of equal size

constant*n is because we will be traversing elements of array in each level of tree

therefore,

T(n) = 2*T(n/2) + constant*n

further we will divide array in to array of equal size so

T(n) = 2*(2*T(n/4) + constant*n/2) + constant*n == 4*T(n/4) + 2*constant*n for this we can say that

T(n) = 2^k * T(n/(2^k)) + k*constant*n

then n = 2^k

k = log2(n)

therefore,

T(n) = n * T(1) + n*logn

T(N) = O(n*Log_2(n))

Space Complexity: Space complexity is O(Log n).

Radix Sort:

Time Complexities:

Best, Average And Worst-Case O(n+k):

RADIX_SORT (A, d)

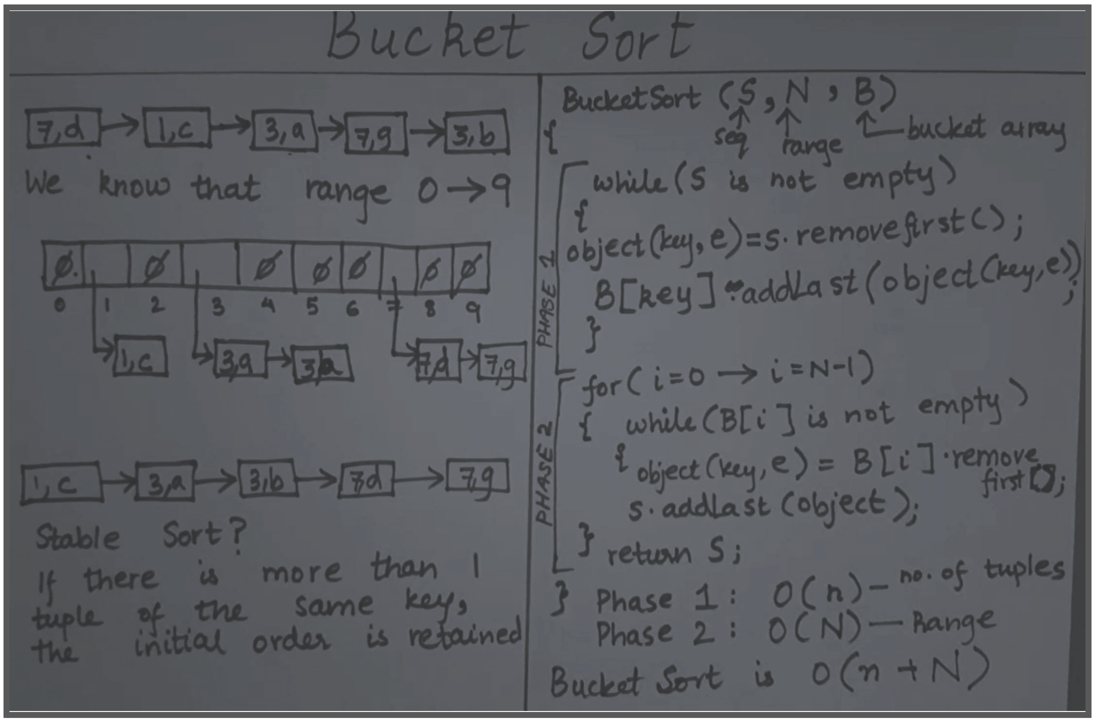
```
for i ← 1 to d do
    use a stable sort to sort A on digit i
    // counting sort will do the job
```

The running time depends on the stable used as an intermediate sorting algorithm. When each digit is in the range 1 to k, and k is not too large, COUNTING_SORT is the obvious choice. In case of counting sort, each pass over n d-digit numbers takes O(n + k) time. There are d passes, so the total time for Radix sort is (n+k) time. There are d passes, so the total time for Radix sort is (dn+kd). When d is constant and k = (n), the Radix sort runs in linear time.

Space Complexity: Space complexity is O(max).

Bucket Sort:

Time Complexities:



T(N) = O(n+N)

Space Complexity: Space complexity is O(n+k)

Counting Sort:

Time Complexities:

Best, Average And Worst-Case O(n+k):

COUNTING-SORT(A,B,k)

step 1 : let C [0...k] be a new array

step 2 : for i = 0 to k :

C[i] = 0

```
step 3 : for j = 1 to A.length :
             C[A[j]] =C[A[j]] + 1
          // C[i] now contains the number of elements
equal to i.
step 4 : for i = 1  to k :
             C[i] = C[i] + C[i - 1]
          // C[i] now contains the number of elements
less than or equal to i.
step 5 : for j = A.length down to 1 :
             B[C[A[j]]] = A[j]
             C[A[j]] = C[A[j]] – 1
```

- step 1 takes constant time.
- In step 2, for loop is executed for k times and hence it takes O(k) time.
- In step 3, for loop is executed for n times and hence it takes O(n) time.
- In step 4, for loop is executed for k times and hence it takes O(k) time.
- In step 5, for loop is executed for n times and hence it takes O(n) time.

Thus the overall time complexity is O(n+k)

where:

- N is the number of elements
- K is the range of elements (K = largest element - smallest element)

Space Complexity: The space complexity of Counting Sort is O(max). Larger the range of elements, the larger the space complexity.

7.4.5:

Time Complexity:

A leaf has an equal probability to be of size between 1 to k.
So the expected size of a leaf is k/2.
If the expected size of a leaf is k/2 then we expect n/(k/2)=(2n)/k such leaves.
For simplicity lets say that we expect n/k such leaves and that the expected size of each leaf is k.
The expected running time of INSERTION-SORT is O(n^2).
We found that in exercise 5.2-5 and problem 2-4c.
So the expected running time of INSERTION-SORT usage is O((n/k)*(k^2))=O(nk).
If we expect our partition groups to be of size k then the height of the recursion tree is expected to be Logn-Logk=Log(n/k) since we expect to stop Logk earlier.
There are O(n) operations on each level of the recursion tree.
That leads us to O(nLog(n/k)).
We conclude that the expected running time is O(nk+nLog(n/k)).

Space Complexity: Space complexity is O(n*Logn).