DAA ASSIGNMENT-1 GROUP-3
Tracing longest sorted partition in each row and diagonal of 10x10 matrix

Sai Charan IIT2016039 | Mantek Singh IIT2016007 | Shruti Reddy IIT2016019 | Rakesh Lakra IIT2016052

***Abstract*—Here, we have tried to design an algorithm to find the maximum sorted partition in each row and each diagonal of the matrix.An $O(n^2)$, $\Omega(n^2)$ and $\theta(n^2)$ algorithm has been designed to solve this problem.**

***Index Terms*—Sorting, Longest Sorted Subarray, time complexity, graphs**

I. Introduction and Literature Survey

Sorting is any process of arranging items in a sequence ordered by some criterion. Here we are sorting the alphabets in order of their ASCII values.

The first part of the problem is to find the maximum sorted partition in each row.

The second part of the problem is to find the maximum sorted partition along every diagonal from left to right and from right to left.

The approach is to traverse through each row and diagonal and store the length of the maximum sorted partition till the current element as well as the starting index of the maximum sorted partition.

II. Algorithm Design

The algorithm has a simple approach which extends the basic algorithm for an array of size N. To understand the algorithm designed, let's first understand how to find a the longest non decreasing or non increasing sub-array in an array of length N.

1. Consider an array of size N filled with random characters of the English alphabet.
2. Let l1 store the length of the current non increasing partition of the array. Similarly, let l2 store the length of current non decreasing partition of the array,both initialized to 1.
3. Let curlen store the maximum length of sorted partition till current element, which is also initialized to 1.
4. If the current element is greater than or equal to the next element, we increment l1, else l1 is again set to 1.
5. If the current element is lesser than or equal to the next element, we increment l2, else l2 is again set to 1.
6. Concretely, if arr[i] $\geq$ arr[i+1](i being the index of current element), l1 is incremented, else l1 $=$ 1, and if arr[i] $\leq$ arr[i+1], l2 is incremented, else l2 $=$ 1.
7. Let ma store the maximum of l1 and l2 and let start store starting point of maximum sorted partition obtained till current element.
8. If ma is greater than curlen, it implies that the length of the current sorted partition is greater than the length of any other sorted partition we obtained while traversing till the current element of the array.
9. Hence, if ma is greater than curlen, we update curlen and the start point of the maximum sorted partition of the array.
10. Concretely, if ma $>$ curlen, then curlen $=$ ma and start will be stored accordingly.
11. Now we are extending the above algorithm for each row and each diagonal of the matrix as stated in the given problem.
12. In each row only the column index gets incremented every time, in each diagonal from left to right the row and the column index get incremented and in each diagonal from right to left the row index gets incremented while the column index gets decremented.

The pseudo-code for the above explained algorithm is as follows:

Algorithm 1 Longest-sorted-partition

```
1:  procedure ROWMAX(mat, n)
2:      l1 ← 1                                          ▷ t+1
3:      l2 ← 1                                          ▷ t+1
4:      curlen ← 1                                      ▷ t+1
5:      start                                           ▷ t+1
6:      ma                                              ▷ t+1
7:      for i = 0 to n-1 do                             ▷ t+2*n
8:          l1 ← 1                                      ▷ t+1
9:          l2 ← 1                                      ▷ t+1
10:         curlen ← 1                                  ▷ t+1
11:         ma ← 1                                      ▷ t+1
12:         for j = 0 to n-2 do                         ▷ t+2*(n-1)
13:             if m[i][j]≥m[i][j+1] then               ▷ t+3
14:                 l1 ← l1 + 1                         ▷ t+1
15:             else
16:                 l1 ← 1                              ▷ t+1
17:             if m[i][j]≤m[i][j+1] then               ▷ t+3
18:                 l2 ← l2 + 1                         ▷ t+1
19:             else
20:                 l2 ← 1                              ▷ t+1
21:             if l1≥l2 then                           ▷ t+1
22:                 ma ← l1                             ▷ t+1
```

```
23:             else
24:                 ma ← l2                                   ▷ t+1
25:             if ma>curlen then                             ▷ t+1
26:                 curlen ← ma                               ▷ t+1
27:                 start ← j − ma + 2                        ▷ t+1

28: procedure RIGHTDIAGONAL(mat, n)
29:     l1 ← 1
30:     l2 ← 1
31:     curlen ← 1
32:     start
33:     ma
34:     for i = 0 to n-1 do                                   ▷ t+2*n
35:         l1 ← 1                                            ▷ t+1
36:         l2 ← 1                                            ▷ t+1
37:         curlen ← 1                                        ▷ t+1
38:         ma ← 1                                            ▷ t+1
39:         j ← 0                                             ▷ t+1
40:         while i+j<n-1 do                                  ▷ t+n-1+1
41:             if m[i+j][j]≥m[i+j+1][j+1] then               ▷ t+3
42:                 l1 ← l1 + 1                               ▷ t+1
43:             else
44:                 l1 ← 1                                    ▷ t+1
45:             if m[i+j][j]≤m[i+j+1][j+1] then               ▷ t+3
46:                 l2 ← l2 + 1                               ▷ t+1
47:             else
48:                 l2 ← 1                                    ▷ t+1
49:             if l1≥l2 then                                 ▷ t+1
50:                 ma ← l1                                   ▷ t+1
51:             else
52:                 ma ← l2                                   ▷ t+1
53:             if ma>curlen then                             ▷ t+1
54:                 curlen ← ma                               ▷ t+1
55:                 start ← j − ma + 2                        ▷ t+1
56:             j ← j + 1                                     ▷ t+1
57:     for i = 1 to n-1 do                                   ▷ t+2
58:         l1 ← 1                                            ▷ t+1
59:         l2 ← 1                                            ▷ t+1
60:         curlen ← 1                                        ▷ t+1
61:         ma ← 1                                            ▷ t+1
62:         j ← 0                                             ▷ t+1
63:         while i+j<n-1 do                                  ▷ t+n-i+1
64:             if m[j][j+i]≥m[j][j+i+1] then                 ▷ t+3
65:                 l1 ← l1 + 1                               ▷ t+1
66:             else
67:                 l1 ← 1                                    ▷ t+1
68:             if m[j][j+i]≤m[j][j+i+1] then                 ▷ t+3
69:                 l2 ← l2 + 1                               ▷ t+1
70:             else
71:                 l2 ← 1                                    ▷ t+1
72:             if l1≥l2 then                                 ▷ t+1
73:                 ma ← l1                                   ▷ t+1
74:             else
75:                 ma ← l2                                   ▷ t+1
76:             if ma>curlen then                             ▷ t+1
77:                 curlen ← ma                               ▷ t+1
78:                 start ← j − ma + 2                        ▷ t+1
79:             j ← j + 1                                     ▷ t+1

80: procedure LEFTDIAGONAL(mat, n)
81:     l1 ← 1                                                ▷ t+1
82:     l2 ← 1                                                ▷ t+1
83:     curlen ← 1                                            ▷ t+1
84:     for i = 0 to n-1 do                                   ▷ t+2*n
85:         l1 ← 1                                            ▷ t+1
86:         l2 ← 1                                            ▷ t+1
87:         curlen ← 1                                        ▷ t+1
88:         ma ← 1                                            ▷ t+1
89:         j ← 0                                             ▷ t+1
90:         while i+j<n-1 do                                  ▷ t+n-i+1
91:             if m[i+j][n-j-1]≥m[i+j+1][n-j-2] then         ▷ t+3
92:                 l1 ← l1 + 1                               ▷ t+1
93:             else
94:                 l1 ← 1                                    ▷ t+1
95:             if m[i+j][n-j-1]≤m[i+j][n-j-2] then           ▷ t+3
96:                 l2 ← l2 + 1                               ▷ t+1
97:             else
98:                 l2 ← 1                                    ▷ t+1
99:             if l1≥l2 then                                 ▷ t+1
100:                ma ← l1                                   ▷ t+1
101:            else
102:                ma ← l2                                   ▷ t+1
103:            if ma>curlen then                             ▷ t+1
104:                curlen ← ma                               ▷ t+1
105:                start ← j − ma + 2                        ▷ t+1
106:            j ← j + 1                                     ▷ t+1
107:    for i = 1 to n-1 do                                   ▷ t+2*n
108:        l1 ← 1                                            ▷ t+1
109:        l2 ← 1                                            ▷ t+1
110:        curlen ← 1                                        ▷ t+1
111:        ma ← 1                                            ▷ t+1
112:        j ← 0                                             ▷ t+1
113:        while i+j<n-1 do                                  ▷ t+n-i-1
114:            if m[j][n-j-i-1]≥m[j+1][n-j-i-2] then         ▷ t+3
115:                l1 ← l1 + 1                               ▷ t+1
116:            else
117:                l1 ← 1                                    ▷ t+1
118:            if m[j][n-j-i-1]≤m[j][n-j-i-2] then           ▷ t+3
119:                l2 ← l2 + 1                               ▷ t+1
120:            else
121:                l2 ← 1                                    ▷ t+1
122:            if l1≥l2 then                                 ▷ t+1
123:                ma ← l1                                   ▷ t+1
124:            else
125:                ma ← l2                                   ▷ t+1
126:            if ma>curlen then                             ▷ t+1
127:                curlen ← ma                               ▷ t+1
128:                start ← j − ma + 2                        ▷ t+1
129:            j ← j + 1                                     ▷ t+1
```

III. Time Complexity Analysis

The time complexity for the above mentioned algorithm for an array of size n is $O(n)$, because we are traversing through the row only once and we are able to get the maximum length and starting index of the maximum sorted partition.

For each row it takes time which is proportional to n and there are n rows therefore it is proportional to $O(n^2)$.

For each diagonal the maximum time complexity is proportional to n as the principal diagonal is of size n ,remaining diagonals are smaller than the principal diagonal. There are 2*(2*n-1) diagonals, so the complexity is proportional to 2*n*(2*n-1) which is approximately proportional n^2

By these we can say that the total time complexity to compute for each row and diagonal of a matrix is proportional to $O(n^2)$

Worst Case :In the worst case the length of current sorted partition will always be larger than the length of the maximum sorted partition we have obtained till now. i.e the condition ma>curlen is always true and hence the start and curlen variables will get updated for every new element traversed.This happens when all the elements in each row and diagonal are sorted and one of such cases is when all the elements in the matrix are equal.

Concretely the expression obtained for time complexity is $17*n^2-13*n+7$ for rows and $32*n^2+8*n+22$ for diagonal, from these two the total complexity is $49*n^2+5*n+29$ which serves as the upper bound i.e the worst case time complexity..

Best Case :In the best case the length of current sorted partition will always be less than or equal to the length of the maximum sorted partition we have obtained till now. i.e the condition ma$\leq$curlen is always true and hence the start and curlen variables will get updated only for the first time.This happens when the elements are in minimum sorted length i.e of length 2 and one of such cases is when a and b occur alternatively in each row and diagonal.

Concretely the expression obtained for time complexity is $16*n^2-10*n+7$ for rows and $30*n^2+20*n+38$ for diagonal, from these two the total complexity is $46*n^2+10*n+45$ which serves as the lower bound i.e the best case time complexity..

The expression for time complexity in the average case is bounded below by the expression for time complexity obtained in the best case and bounded above by the expression for time complexity obtained in the worst case.

g(x) $<$ f(x) $<$ h(x) where g(x) is the tightest lower bound of the function and h(x) is the least upper bound of the function. Since the order of polynomials is n^2, the time complexity is $\theta(n^2)$. Note, we have used $\theta(n^2)$ because the time complexity in all the case is the same. This problem does not have worst case and best case time complexities of differing orders as both are of the the order of n^2.

IV. Experimental study

We have found the time complexity of the algorithm by calculating it in each step. Now to will verify it with the experimental results.

We have plotted T VS N graph where is T is the total number of computations performed for a given N. The T and N values for calculating in rows (Table 1 and figure 1), diagonals (Table 2 and figure 2)and for calculating both (Table 3 and figure 3) are mentioned below in tables and N vs T graphs for them are plotted accordingly.

Table I
N vs T graph for Rows

n	t_{best}	$t_{average}$	t_{worst}
2	53	53	53
3	122	128	128
4	221	231	237
5	350	366	380
6	509	521	557
7	698	716	768
8	917	935	1013
9	1166	1198	1292
10	1445	1477	1605

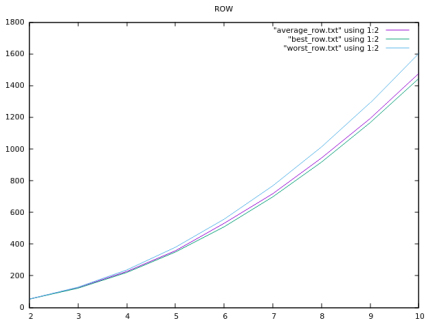

Figure 1. Rows T(y-axis) VS N(x-axis)

Table II
N vs T graph for Diagonals

n	t_{best}	$t_{average}$	t_{worst}
2	84	84	84
3	205	209	209
4	385	389	401
5	624	630	660
6	922	942	986
7	1279	1305	1379
8	1695	1729	1839
9	2170	2212	2366
10	2704	2770	2960

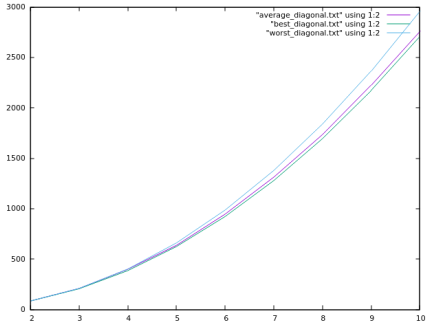

Figure 2. Diagonals T(y-axis) VS N(x-axis)

Table III
N VS T GRAPH FOR BOTH ROWS AND DIAGONALS

n	t_{best}	$t_{average}$	t_{worst}
2	137	137	137
3	327	337	337
4	606	620	638
5	974	996	1040
6	1431	1463	1543
7	1977	2021	2147
8	2612	2664	2852
9	3336	3410	3658
10	4149	4247	4565

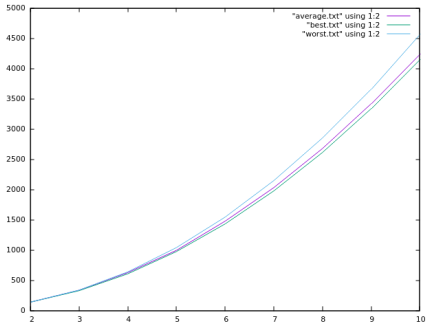

Figure 3. Both Rows and Diagonals T(y-axis) VS N(x-axis)

V. DISCUSSIONS

Thus from time complexity analysis and experimental study we can see the overall complexity of the algorithm is always proportional to n^2 i.e always $O(n^2)$.

So we can see that the curve is a semi parabola with axis of symmetry as Y-axis , but the co-efficient of the n^2 term varies in best and the worst cases.

Even we can observe that the curve for random case is very much closer to the best case than the worst case. This is because is more likely to have unsorted elements more than the sorted elements when we generate inputs randomly.

VI. CONCLUSION

Here, we have tried to design an efficient algorithm that finds out the maximum sorted path in every row and diagonal of an $N * N$ matrix. A simple approach of traversing each row and diagonal of the matrix is used, the details of implementation and pseudo code along with relevant calculations of time complexities and experimental details is done above to the best of our abilities. Hence, it can be concluded that the given question can be solved in $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Moreover, all the possible cases have been discussed for the formation of as efficient algorithm as possible.