IDAA432C Assignment 1

Group - 41

- Aditya Vallabh
- Kiran Velumuri
- Neil Syiemlieh
- Tauhid Alam

Abstract

A sorted partition of a list is a sequence of elements arranged in either ascending or descending order. The purpose of this project is to determine the presence and location of such sorted partitions in:

- Each column of a matrix
- A child of the matrix

Algorithm Design

A. Longest Sorted Partition in a column

- Iterate through each cell
- Keep track of the current partition’s length
- Keep track of the maximum length found so far
- When length is greater than the maximum length, update maximum length and the start of the longest sorted partition so far

Pseudocode

Algorithm 1 Longest Sorted Partition algorithm

```
for j ← 0 to n − 1 do
    start ← 0
    len ← 1
    max ← 1
    for i ← 1 to n − 1 do
        if matrix[i − 1][j] ≥ matrix[i][j] then
            len ← len + 1
        else
            len ← 1
        if len > max then
            max ← len
            start ← i − len + 1
```

B. Longest Sorted Path in a matrix

- Treat the matrix as a graph
- Traverse through the matrix using depth first search and backtracking
- Find the longest sorted path starting from every cell
- Record the length of the longest path from every cell onto that cell’s location

Pseudocode

Algorithm 3 Driver function

$max \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
        $len \leftarrow recursiveFind(matrix, n, i, j, 0)$
        $init[i][j] \leftarrow len$
        if $len > max$ **then**
            $max \leftarrow len$
$maxLen \leftarrow max$

Algorithm 2 Recursive Algorithm

```
procedure RECURSIVEFIND(matrix, n, x, y, len)
    visited[x][y] ← matrix[x][y]
    if len + 1 = maxLen then
        printMatrix(visited)
    l ← len + 1
    r ← len + 1
    u ← len + 1
    d ← len + 1
    if visited[x][y − 1] == −1 AND matrix[x][y − 1] ≥ matrix[x][y]] then
        l ← recursiveFind(matrix, n, x, y − 1, len + 1)
    if visited[x][y + 1] == −1 AND matrix[x][y + 1] ≥ matrix[x][y]] then
        r ← recursiveFind(matrix, n, x, y + 1, len + 1)
    if visited[x − 1][y] == −1 AND matrix[x − 1][y] ≥ matrix[x][y]] then
        u ← recursiveFind(matrix, n, x + 1, y, len + 1)
    if visited[x + 1][y] == −1 AND matrix[x + 1][y] ≥ matrix[x][y]] then
        d ← recursiveFind(matrix, n, x, y, len + 1)
    visited[x][y] ← −1
    return MAX(l, r, u, d)
```

Traversing the longest sorted path

Algorithm 4 Print Path

for $i \leftarrow 0$ to $n-1$ **do**
  for $j \leftarrow 0$ to $n-1$ **do**
    if $init[i][j] == maxLen$ **then**
      $recursiveFind(matrix, n, i, j, 0)$

Complexity Analysis

A. Longest Sorted Partition - All cases

For a single column of size *n*, complexity:

$$\theta(n)$$

For an n x n matrix, final complexity:

$$\theta(n^2)$$

Worst Case:

$$t \; \alpha \; 20n^2 - 8n + 1$$

Best Case:

$$t \; \alpha \; 15n^2 + 2n + 1$$

B. Longest Sorted Path - Best Case

```
The Random Matrix:
00 01 00 01 00 01 00 01 00 01
01 00 01 00 01 00 01 00 01 00
00 01 00 01 00 01 00 01 00 01
01 00 01 00 01 00 01 00 01 00
00 01 00 01 00 01 00 01 00 01
01 00 01 00 01 00 01 00 01 00
00 01 00 01 00 01 00 01 00 01
01 00 01 00 01 00 01 00 01 00
00 01 00 01 00 01 00 01 00 01
01 00 01 00 01 00 01 00 01 00
```

Time complexity: $\Omega(n^2)$

B. Longest Sorted Path - Worst Case

For DFS starting from a particular cell, complexity:

$$O(4^{n^2})$$

For an n x n matrix, final complexity: $$O(n^2 * 4^{n^2})$$

```
The Random Matrix:
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
```

However,

In practice, in the *recursiveFind()* DFS algorithm, only 1 or 2 neighbouring cells are valid or not yet visited. Therefore, worst case time complexity:

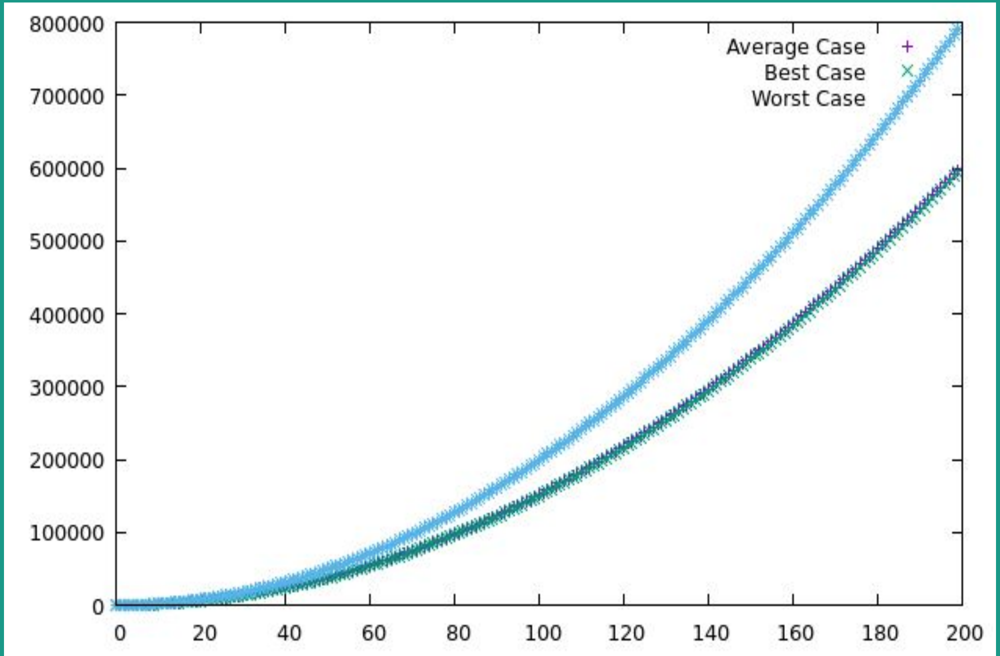
$$O(n^2 * k^{n^2}) \text{ where } 1 < k < 2$$

Experimental Study

Longest Sorted Partition

Size (n)	50	75	100	125	150	175	200
t_{max}	49601	111901	199201	311501	448801	611101	790429
t_{avg}	38216	85601	151646	236631	340216	462591	597819
t_{min}	37601	84526	150201	234626	337801	459726	594414

Time complexity - Longest Sorted Partition

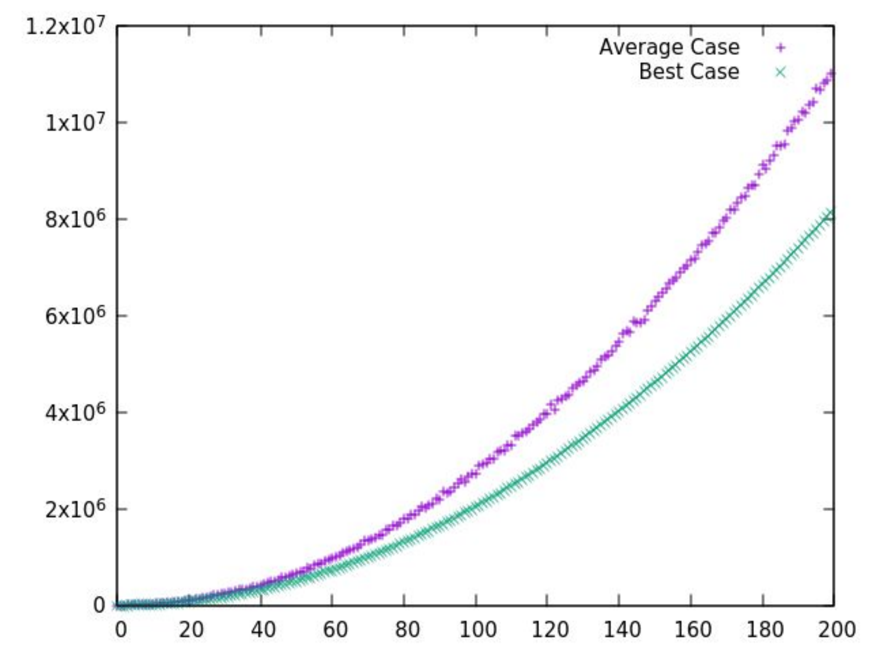


Longest Sorted Path

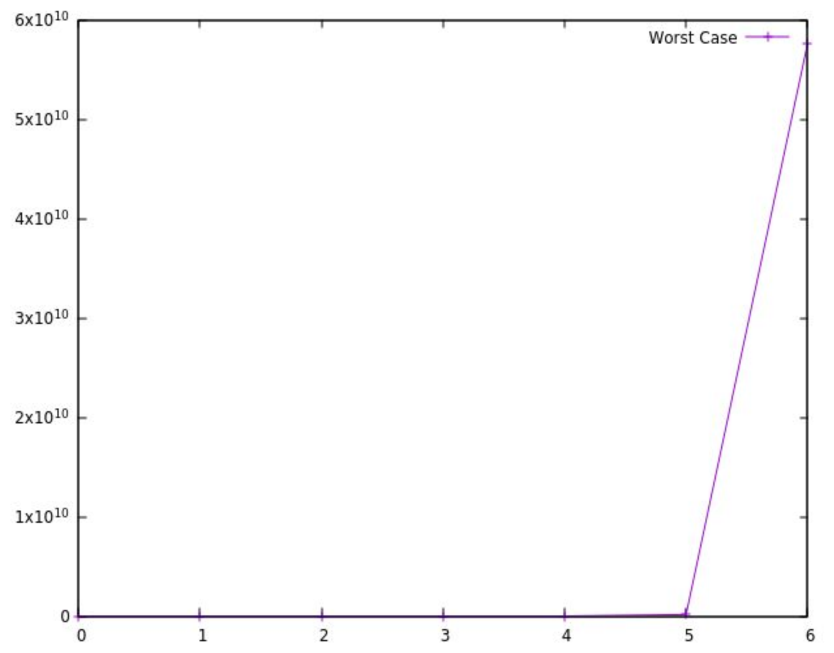
n	35	70	105	140	175	200
t_{avg}	309086	1311148	2893500	5229024	8279143	10753165
t_{min}	225249	913612	2065024	3679487	5756999	7449203

n	1	2	3	4	5	6
t_{max}	82	1892	34101	1882452	154667081	5.7×10^{10}

Time complexity - Longest Sorted Path



Best and average cases



Worst case

Discussions

Generation of random numbers

- Dynamic allocation of memory for creating and storing matrices
- Used rand() and srand() from stdlib
- srand() sets the seed which is used by rand to generate “random” numbers
- Setting the seed as current time to produce different pseudo-random numbers on each run

Complexity of Longest Sorted Path

- This problem has a non-polynomial worst case

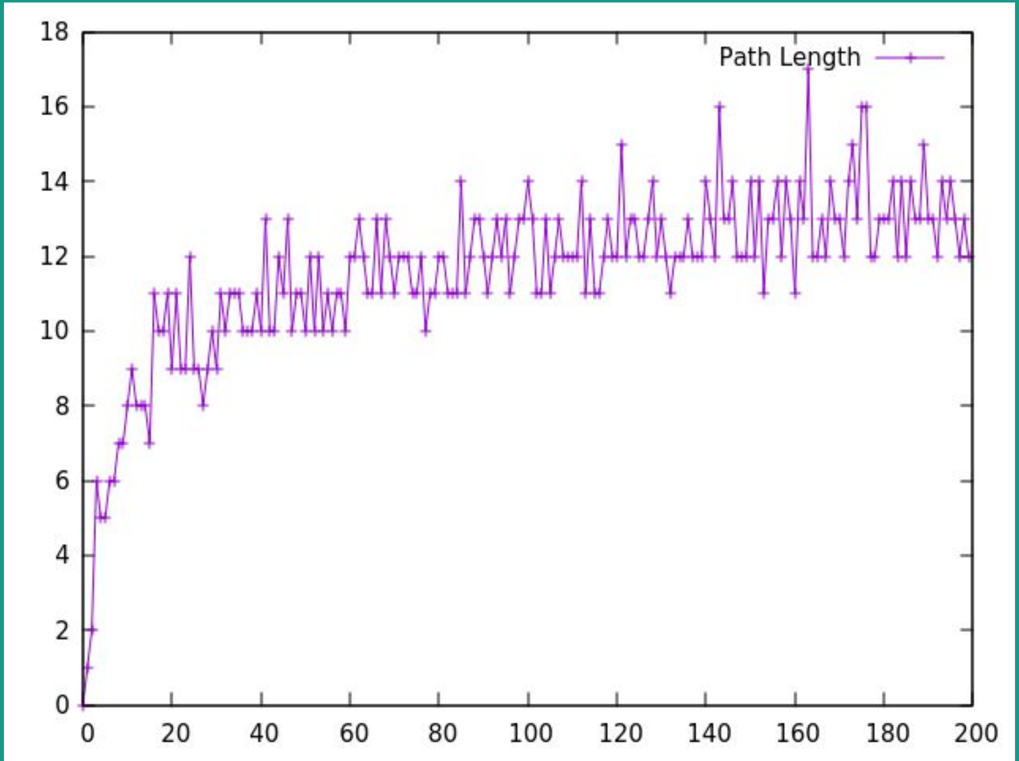
  complexity ≈ $O(n^2 * k^{n^2})$

- However the average case is good owing to the random numbers implying the probability to get a matrix close to the best case seems to be relatively greater

Path Length vs n

As we increase n, the rate of increase in the length of the longest sorted path appears to be decreasing; meaning the larger the matrix, the lesser the chances of getting a longer path

Path Length vs N



Tracing all possible paths

- In the algorithm all the possible paths are being explored while backtracking
- So instead of breaking after finding a possibility we modified the algorithm to trace all paths of the maximum length

Demo

```
dedsec@ubuntron:$./a.out
Enter matrix dimensions (n x n): 10

The Random Matrix:
00 21 12 05 28 64 44 08 34 39
78 39 88 96 13 34 54 66 94 81
18 66 94 95 75 30 98 55 80 14
60 32 35 24 37 15 88 34 23 22
73 54 61 13 02 75 99 56 93 93
90 63 59 36 10 34 67 09 41 99
23 02 83 10 78 20 78 67 06 01
41 31 07 55 44 09 82 43 18 75
88 08 91 47 96 01 34 63 62 75
14 37 29 97 00 08 70 78 27 28
```

```
The Partition Matrix
 * 21 12  *  *  * 44  *  *  *
 * 39 88 96  *  * 54 66 94  *
18 66 94 95 75  * 98 55 80 14
60  *  * 24 37  *  * 34 23 22
73  *  * 13 02 75  *  *  * 93
90  *  *  *  * 34  *  *  * 99
 *  *  *  *  * 20  *  *  *  *
 *  *  *  *  * 09  *  *  *  *
 *  *  *  *  * 01  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
```

Demo (contd.)

```
The Longest Path Matrix
Initial Point: (0,3)
 *  21 12 05  *  *  *  *  *  *
 *  39 88 96  *  *  *  *  *  *
 *  *  94 95  *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
```

```
 *  21 12 05  *  *  *  *  *  *
 *  39 *  96  *  *  *  *  *  *
 *  66 94 95  *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
 *  *  *  *   *  *  *  *  *  *
```

```
Initial Point: (2,9)
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  14
 *  *  *  *  *  *  *  34 23 22
 *  *  *  *  *  *  *  56 93 93
 *  *  *  *  *  *  *  *  *  99
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *

Longest path length: 8
Total number of paths found: 3
dedsec@ubuntron:$
```

Conclusion

In the first part, we have implemented a $\Theta(n^2)$ algorithm

For the second part, an algorithm using backtracking has been developed whose average case very close to the best case which has $\Omega(n^2)$ complexity.

Graphs have also been plotted and agree with the theoretically calculated complexities.


Thank you