To find length of longest sorted component diagonally in a matrix

Indian Institute of Information Technology, Allahabad

Karan Kunwar
iit2019001@iiita.ac.in

Anirudh Arora
iit2019003@iiita.ac.in

Naina Kumari
iit2019004@iiita.ac.in

***Abstract*—In this paper, we have devised two algorithms based on concepts of dynamic programming and binary search to find the longest sorted component in a given (50x50) matrix diagonally. Also, we have discussed and compared time and space complexities of both the algorithms using Apriori Analysis as well as A Posteriori Analysis.**

***Index Terms*—length; matrix; diagonally; sorted; increasing (key words)**

I. Introduction

Dynamic Programming is an optimized form of recursion technique. In simpler words, it is recursion with memorization. In recursion, a function for lower values is called in order to get the value of function at a particular value. In dynamic programming, we define the base cases. As we move to upper values, we keep on storing the function values, so that we can retrieve these values in constant time whenever they are needed. The values are stored in a data structure which can later be called to get the desired value.
For example:
For Fibonacci sequence,
fib[0]=1 //base case
fib[1]=1 //base case
fib[n]=fib[n-1]+fib[n-2] //function
Now, when we need to find the fibonacci number at index x, it calls the number at index x-1, x-2. Subsequently index at x-1 calls number at x-2,x-3 and so on. So the same values are being called multiple times, increasing the time complexity. In dynamic programming, we maintain an array(data structure) containing numbers on different indexes of a fibonacci series. So, whenever a number at index x is needed we just need to check the array at indexes x-1 and x-2 without any need to call further indexes, reducing the time complexity quite effectively.

A. Generating Random Matrix

We can generate random elements in range 0 to 9 by using rand() function and then taking a mod with 10.
we can run the loops to store the generated random number in the 2-dimensional array. We then generate the diagonals by the following process:

```
int i = r-1;
while i>=0:
        int k = i, j=0;
        vector <int> v;
        while j<c AND k<r:
            v.push_back(arr[k][j]);
            j = j+1;
                k = k+1;
                i = i-1;
```

Each time we get out of the inner loop, we call the solve function to get the longest sorted subsequence for each diagonal. The algorithms are described as follows:

II. Approach 1

The first approach is based on dynamic programming[1] to find the longest increasing subsequence for a linear vector.

A. Algorithm

- We store each diagonal in a vector vec along primary and secondary diagonal. And each time we implement an algorithm to every stored diagonal.

- And the longest length of subsequence of vec is stored in ans and calculated as:
  We traverse vec starting from index 1 to the end of vector vec, we check for all the non greater elements present in the array on the left of the current index. For each satisfying element we update the dp at current index as:

```
Current index = curr
Satisfying element index = x
if(vec[x]<=vec[curr])
{
    dp[i]= max(dp[i],dp[j]+1);
    ans = max(ans,dp[i]);
}
```

- Finally, among all vectors of diagonals of matrix(along primary or secondary) the maximum length is stored in length variable and calculated as follows:
  length = max(length,solve(vec));

B. Time Complexity & Space Complexity Analysis

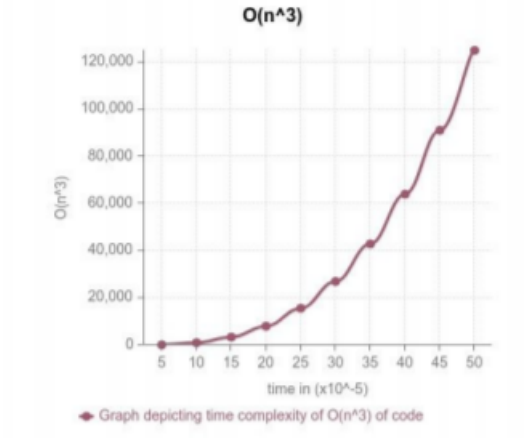
```
for(int i=c-1;i >= 0;i--){                    (c1)
    int k=i;
```

```
    vector <int> vec;
    for(int j=0;j<r && k<c;j++,k++)   (c2)
        vec.push_back(arr[k][j]);
    length = max(length,solve(vec));  (c3)
}
```

the outer loop runs in a linear manner. And the inner loop generates all diagonals in linear fashion as well.
After each diagonal is generated it is then put in the solve function whose complexity analysis is:

```
for i = 0 to L :                          (c1)
    for j = 0 to i :                      (c2)
        if arr[ j ] <= arr[ i ]           (c3)
            if dpr[ j ]+1 >=dpr[ i ]
                dpr[i]=dpr[j]+1;
                vecr[i]=vecr[j];
                vecr[i].push_back(arr[i]);
            if(dpr[i]>=ansr)
                ansr = dpr[i];
                vr =vecr[i];
```

Here c3 is a constant time for all the iterations. $T(n) = c1 * L + c2 * \sum_{j=0}^{L} t_j + c3 * \sum_{j=0}^{L} t_j$
$Here, t_j = j$
$\sum_{j=0}^{L} t_j = L(L+1)/2$
It runs for n^2 times. Where n is the length of diagonal.
Therefore,
Overall time complexity is O(n(n+n^2)) = O(n^3)
Overall space complexity is O(n^2) [O(r*c) if r!=c]



III. Approach 2

The second approach is based on a binary search $algorithm$[2] to find the longest increasing subsequence. for a linear array/vector.

A. Algorithm

STEP 1: We initialize a vector(say vans). Initially only the element at 0th index is pushed into this vector.
STEP 2: Traversing from index 1 to the end of the vector arr.

- If the element on the current index greater than or equal to the last element of the array we will push this element. It is done as:

```
if(vc[vc.size()-1] <= arr[i]){
    vc.push_back(arr[i]);
}
```

- Else, We will find the upper bound of the ith element in the vector vc using binary search and replace that element with the ith element of vector arr. It is done as:

```
int low=0, high=vc.size();
int ac=high;
while(low<=high){
    int mid = (low+high)/2;
    if(vc[mid]>arr[i]){
        ac=mid;
        high=mid-1;
    }
    else{
        low=mid+1;
    }
}
vc[ac]=arr[i];
```

B. Time Complexity & Space Complexity Analysis

```
for(int i=c-1;i>=0;i--){                  (c1)
    int k=i;
    vector <int> vec;
    for(int j=0;j<r && k<c;j++,k++)       (c2)
        vec.push_back(arr[k][j]);
    length = max(length,solve(vec));      (c3)
}
```

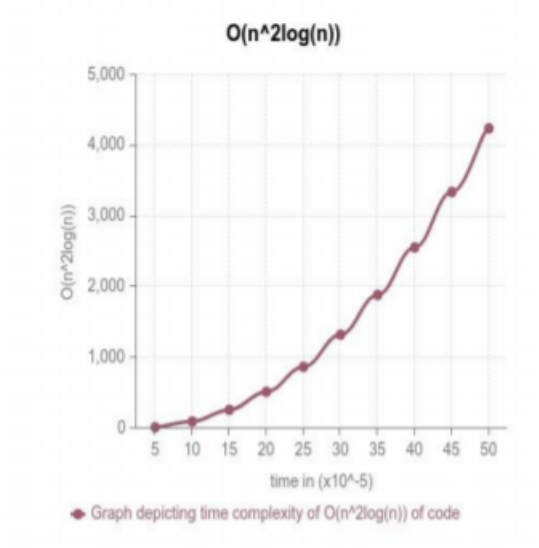
Here the outer loop runs in a linear manner . And the inner loop generates all diagonals in linear fashion as well.

After each diagonal is generated it is then put in the solve function whose complexity analysis is:

```
for(int i=1;i<l;i++)                      (c1)
    if(vc[vc.size()-1] <= arc[i])         (c2)
        vc.push_back(arc[i]);             (c3)
    else
        int low=0, high=vc.size();        (c4)
        int ac=high;                      (c5)
        while(low<=high)                  (c6)
            Int mid = (low+high)/2;       (c7)
            if(vc[mid]>arc[i])            (c8)
                ac=mid;                   (c9)
                high=mid-1;               (c10)
            else
                low=mid+1;                (c11)
        vc[ac]=arc[i];                    (c12)
```

$T(n) = c1.L + (c2 + .. + c5).(L-1) + (c6 + ..c13)(LlogL)$
$T(n) = L.log(L)$
It runs for n*log(n) times. Where n is the length of the diagonal.

Overall time complexity = O(n^2+n^2log(n))= O(n^2log(n))
Overall space complexity = O(n^2) [O(r*c) if r!=c]



IV. Conclusion

We can conclude that the best complexity to find the longest sorted component in a given (50x50) matrix diagonally is O(n^2log(n)).

References

[1] Longest Increasing Subsequence O(n^2).
[2] Longest Increasing Subsequence O(NlogN).
[3] Algorithm Design by J Kleinberg and E Tardos .