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

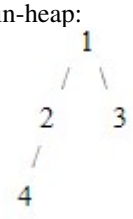
Naina Kumari
iit2019004@iiita.ac.in

***Abstract*—In this paper, we have devised an algorithm based on concepts of dynamic programming and recursion to find the no. of heaps that can be generated with a given set of numbers. Also, we have discussed and compared time and space complexity for the algorithm using Apriori Analysis as well as A Posteriori Analysis.**

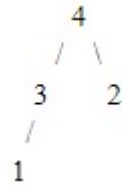
***Index Terms*—length; matrix; diagonally; sorted; increasing (key words)**

I. Introduction

A Special type of tree based data structure is called a heap. The tree used in the construction of a heap is a binary tree. Properties of the heap are written as follows: Max-Heap: The key present at the root node is the greatest of all and each parent node has a key greater than each of its child nodes. Min-Heap: The key present at the root node is the smallest of all and each parent node has a key smaller than each of its child nodes. For example: For a given set of 4 numbers the min-heap and max-heap are shown as follows:
S=1,2,3,4
Min-heap:



Max-heap:



Height, h of a heap is given by expression: h = log2(n), where n is the number of elements. In a heap: All levels must be filled, with an exception for the last level The nodes must be filled from left to right.

II. Approach

A. Explanation

We can devise the algorithm[1] in order to find the no. of max heaps and also for min heaps with minor modifications in the devises algorithm. We will be discussing the approach for the case of max-heaps only.

The root element of the heap will be the largest element of all the available elements of the given set. An important observation which must be made is that due to the heap properties, the structure of the heap will be the same in all the distinct heaps, the only difference being the values assigned to the node.

After the greatest element is assigned to the root node, let us assume there are l no. of nodes in the left subtree as well as r no. of nodes in the right subtree. And l + r = n-1. So, no. of ways of choosing l no. of elements out of the n-1 total elements is given by n-1Cl. Furthermore, no. of heaps for the subtrees are needed to be calculated. Let us divide this problem into smaller sub-problems. For n no. of total elements, the no. of heaps is given by:

H(n) = 1(the greatest element) * n-1Cl(no. Of ways of choosing l elements) * H(l)(no. Of heaps for the left subtree) * H(r)(no. Of heaps for the right subtree)
Mathematically, H(n) = n-1Cl * H(l) * H(r)

The function for calculating the no. of heaps for a given set of numbers comes out to be a recursive function. The complexity for a recursive function is exponential in nature. In order to optimize the devised approach, we can use the concept of recursion with dynamic programming. The need to calculate the value of the function for the same function repeatedly will be eliminated, drastically reducing the time required for the execution of the algorithm.

B. Algorithm

STEP 1: We initialize the dp(dp[n] stores the answer) and nck (It stores nCk for a given value of n and k) array as -1.

```
memset(dp,-1,sizeof(dp));
memset(cnk,-1,sizeof(nck));
```

STEP 2: For a given value of n (size of array under consideration) we calculate the number of elements in the left subtree as :

```
if (n == 1)
    return 0;
    height = log2(n);
        _2powh = (1 << height);
        last = n - (_2powh - 1);
        if (last >= (_2powh / 2))
        return (_2powh) - 1;
        else
        return
        (_2powh)-1-((_2powh/2)-last);
```

STEP 3: Let the value the above code return be l, then the Total no of max heap for n size array can be calculated as:

```
func numberOfHeaps(n):
    if(n<=1)
        return 1;
    if (dp[n] != -1)
        return dp[n];
    ans = (choose(n-1,left)
    *numberOfHeaps(left))
    *(numberOfHeaps(n-1-left));
    dp[n] = ans;
    return ans;
```

STEP 4: The above choose function find nCk as :

```
func choose(n, k):
    if (k > n)return 0;
        if (n<= 1)return 1;
        if (k== 0)return 1;
        if (nck[n][k] != -1)
        return nck[n][k];
        answer = choose(n - 1, k - 1)
        + choose(n - 1, k);
        nck[n][k] = answer;
            return answer;
```

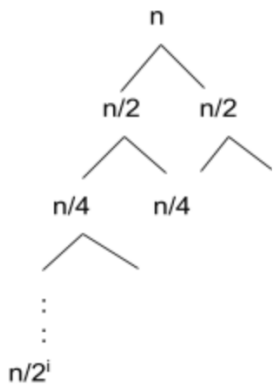
C. Time Complexity and Space Complexity

```
func choose(n, k):
          if (k > n)return 0;
          if (n<= 1)return 1;
                  if (k== 0)return 1;
          if (nck[n][k] != -1)
          return nck[n][k];
          answer = choose(n - 1, k - 1)
                 + choose(n - 1, k);
          nck[n][k] = answer;
                  return answer;
```

In the above, In order to find choose(n,k) each time choose(n-1,k-1) + choose(n-1,k) is called and value at (n,k) is being stored in nck[n][k]. So,this results to O(n).

```
func numberOfHeaps(n):
          if(n<=1) return 1;
          if (dp[n] != -1) return dp[n];
          ans = (choose(n-1,left)
                  *numberOfHeaps(left))
                *(numberOfHeaps(n-1-left));
         dp[n] = ans;
        return ans;
```

In the above, in order to calculate the numbersOfHeaps(n). at every state value is stored i.e. dp[n]. So this results O(n).



Since, $n=2^i$ // n is the given number

So, $i= \log_2 n$ // i is height

Hence, number of heaps

$\sum_{0}^{h} 2^i = [1 + 2 + \ldots.. + 2^i]$ //i is height or level

$\approx 2^h - 1$

$\approx 2^{\log_2 n} - 1 \approx n - 1 = O(n)$

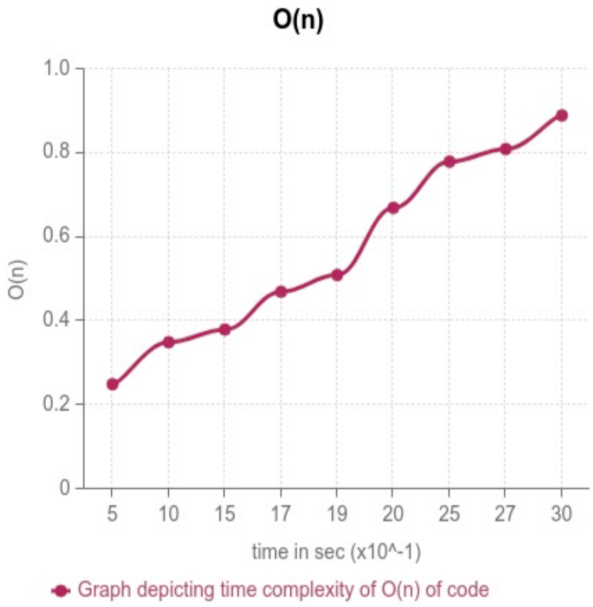
So, time complexity = O(n)

And, space complexity = O(n)

So, time complexity = O(n). And, space complexity = O(n)

Table 1: Time versus n.

Value of n	Time taken(sec)
5	0.025
10	0.035
15	0.038
17	0.047
19	0.051
20	0.067
25	0.078
27	0.081
30	0.091



III. Conclusion

We can conclude that the best complexity to number of ways to construct heaps with given number is in O(n) time complexity and space complexity O(n).

References

[1] Calculate the number of heaps possible.
[2] Algorithm Design by J Kleinberg and E Tardos .