Sorted Sequences

Medha Balani(IIT2019021) , Saksham Aggarwal(IIT2019022), Utkarsh Gangwar(IIT2019023)
4th Semester , Department of Information Technology
Indian Institute of Information Technology , Allahabad , India

Abstract:
This paper discusses the algorithm used to solve the task of generating all permutations of length 1 to 11 , detecting and printing sorted permutations.
Time Complexity of the algorithm and its analysis in both the best and worst case is also briefly discussed.

I. INTRODUCTION

Let's first formally define what a permutation is.
A Permutation of length n is a sequence in which every natural number from 1 to n occurs exactly once .(The total n elements of the sequence can be ordered in any way).

For example : [5,2,4,1,3] , [1,2,3] , [7,6,5,4,3,2,1] are permutations but [1,2,2] , [1,2,3,4,6] are not permutations.

Next we look at how many permutations of length n can exist . This can be analysed with basic combinatorics that the first element of the permutation has n possible ways of being arranged , second element has n-1 ways , similarly the last element has only 1 left position to occupy , so total number of permutations of length n comes out to be n *(n-1)*(n-2) ….3* 2*1 = Factorial(n).

II. ALGORITHM DESCRIPTION

Our proposed algorithm for this problem follows the following steps :

Step 1: Select a length from 1 to 11 which has not been selected previously .(Assume we selected length = **i**)
Step 2: Iterate over all possible permutations(factorial(i)) .
Step3 : For each permutation check if it is sorted or not , If it is sorted we print the permutation
Else keep iterating until we have iterated over all the Factorial(i) permutations for the current selected length i.
Step 4: Mark the current length as selected and go to Step 1.
Algorithm used for generating all permutations of a given length:
Basically we keep generating the next permutation from the current permutation.
Step 1: Iterate from the last of the sequence and find the first index for which sequence[i] < sequence[i+1]**.**(let's call this index p)
Step 2 : Iterate forward from p and find the first index at which sequence[i] > sequence[p] .(let's call this index q).
Step 3 : Swap sequence[p] with sequence[q].
Step 4: Reverse the sequence from (p+1)th index to the last index . (This gives us the next permutation).
Algorithm used to check if current permutation is sorted or not :
A single traversal from left to right in the sequence , if there exist no element such that
Sequence[i] > Sequence[i+1] ,then the permutation is sorted , else not.

III ALGORITHM AND ANALYSIS

Algorithm Print Sorted Sequences

Global Array *Visited[11] initialised to zero*

```
function PRINT_SORTED_SEQUENCES
 len <- GET_LENGTH()
 Array A <- RANDOM_PERMUTATION (len)
 For i = 0 to factorial(len)
   If ( IS_SORTED(A) = 1 ) then
        PRINT_ARRAY ( A)
        break
   Else A <- NEXT_PERMUTATION (A)
Return
```

```
function GET_LENGTH
 len<- RANDOM(1,11)
 If (Visited[len] = 0) then
       Visited[len]<-1
       Return len
 Else Return GET_LENGTH( )
```

```
function NEXT_PERMUTATION(A)
 len<- Length(A)
 p <- -1
   For i = len-2 to 0
       If (A[i] < A[i+1] ) then
         p <- i
         Break
   If (p = -1) then
      Reverse(A)
      Return A
   q <- -1
     For i = p+1 to len
        If (A[i] > A[p]) then
            q <- i , Break
if(q!=-1)
   swap(A[p],A[q])
if(p < len-1)
   Reverse(p+1,len)
   Return (A)
```

```
function IS_SORTED(A)
For i = 0 to len-1
   if(A[i+1]<A[i])
      Return 0
Return 1
```

In the above pseudo code the function RANDOM(1,11) returns a random integer between 1 to 11 , the function RANDOM_PERMUTATION(len) generates a random permutation of length equal to len .

III (a) Apriori Analysis

Since we are going to iterate through all lengths from 1 to n (n being 11 in our case), this contributes O(n) or Ω(n) to the overall time complexity. Then for each length we are iterating on all the permutations of that length , the maximum number of permutations can go upto factorial(n) for a given length n .
For each of these permutations we will check whether it is sorted or not .

Algorithmic Steps	**Worst case**	**Best case**
For each (length from 1 to n)	O(n)	Ω(n)
For each (permutation from 1 to n!)	O(n!)	Ω(1)

Check (if sorted or not)	O(n)	Ω(1)
Generating next permutation(If previous was not sorted)	O(n)	Ω(1)

So , the overall complexity comes out to be $O(n.(n!).(n+n)) \simeq O(n^2.n!)$ in the worst case and $\Omega(n\ .\ (1)(n)) = \Omega(n^2)$ in the best case.

Space Complexity = O(n)

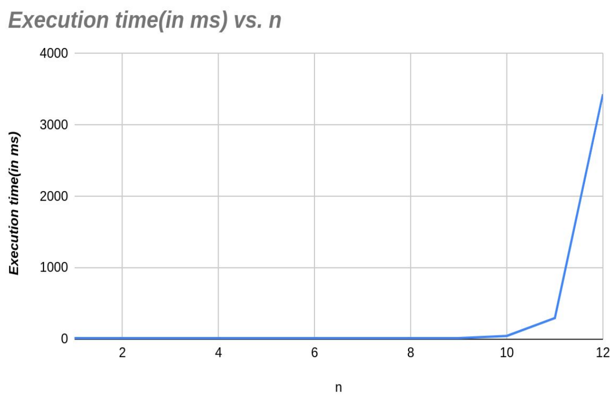

Figure 1:Time complexity graph for a posteriori analysis.

ILLUSTRATION

Suppose, we get length 4 from the GET_LENGTH function. Then, we calculate 4! , i.e 24. Now, while iterating through all permutations, we see 1,2,3,4 comes sorted so we break there printing it. Now, suppose we start with 4,3,1,2. So iterating through all permutations, we see 4,3,1,2 is not sorted so using NEXT_PERMUTATION function, we get 4,3,2,1 which is again not sorted so again using NEXT_PERMUTATION function, we get 1,2,3,4 which is sorted so printing it.

OTHER POSSIBLE SOLUTION

Next permutations can also be generated using heaps algorithm for generating permutations.
Idea in this is to generate each new permutation from the previous by choosing a pair of elements to interchange, without disturbing other (n-2) elements using recursion. Its time complexity is O(n!) , and then to check if sorted or not , O(n) . Selecting a length initially is also O(n) , so overall complexity remains the same as our proposed algorithm.

CONCLUSION

Our proposed algorithm works very well as of time and space complexity for the specified constraints in our problem statement.

REFERENCE

[1] Introduction to Algorithms (T.H Cormen)
[2] https://en.wikipedia.org/wiki/Permutation
[3] https://codeforces.com/problemset/customtest