.

MA5910: Data Structures in Scientific Computing

Home Assignment Number : 1

Name : **Sivaraman S** Roll.No: **oe19s012**

1. Write an algorithm for generating all the permutation of (1,2,3,...,n) exactly once.

Solution:

Permutation of n numbers can be generated by recursive function. Let's consider,
$n \longrightarrow$ sequence length,
$A \longrightarrow$ Array of n numbers in ascending order[1,2,$\cdots\cdots$,n].

Algorithm:

```
. generate(n, A):
.     if n = 1 then
.         output(A)
.     else
.         generate(n - 1, A)
.         for i = 0; i < n-1; i += 1 do
.             if n is even then
.                 swap(A[i], A[n-1])
.             else
.                 swap(A[0], A[n-1])
.             end if
.             generate(n - 1, A)
.         end for
.     end if
```

- The algorithm generates $(n-1)!$ permutations of the first $n-1$ elements, adjoining the last element to each of these. This will generate all of the permutations that end with the last element.
- If n is odd, swap the first and last element and if n is even, then swap the i^{th} element (i is the counter starting from 0) and the last element and repeat the above algorithm till i is less than n.
- In each iteration, the algorithm will produce all the permutations that end with the current last element.

This permutation for n objects was firstly proposed by B. R. Heap in 1963.so, it is called ***Heap's Algorithm***.

2. Let $P(x) = \sum_{k=0}^{n} a_k\, x^k$ be a polynomial of degree n≥0, where a_0,a_1,...,a_n are given constants. Write an algorithm that evaluates $P(x)$, for a given value x, that performs at most $n+1$ multiplications.

Solution:

For evaluating P(x) with less $(n+1)$ multiplications, we should write algorithm based on **Nested Multiplication**. let us consider example, when the order of polynomials are
$n = 2 : p(x) = a_0 + x(a_1 + a_2x)$
$n = 3 : p(x) = a_0 + x(a_1 + x(a_2 + a_3x))$
$n = 4 : p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + a_4x)))$
These contain, respectively, 2, 3, and 4 multiplications.for the general case, we can write,

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_nx)\cdots))$$

When we write an equation in nested multiplication format, it only requires n times multiplication only,
Algorithm:

. $poly = a_n$
. $for\ j = \text{n}-1 : -1 : 0$
. $\quad poly = a_j + x \cdot poly$
. end

There are other methods like recursive method, loose algorithmic method are having higher multiplication element.Even though number of addition is same for all methods, **Nested Multiplication** method only requires n multiplications and considered as optimal for computational cost.

3. Let A[1...n] be an array of n distinct integers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer. Also, give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n\log n)$ worst-case time.

Solution:
$A \longrightarrow$ Given Array with n elements $\longrightarrow [a_0, a_1, a_2, \cdots, a_n]$,
Condition for Inversion: $\longrightarrow i < j\ \&\ A[i] > A[j]$,
Number of Inversion:
When we have n unique Numbers and for the worst case, let us consider as, they are in **reversed sorted order**.When we run the algorithm, 1^{st} element in the sequence,make inversion pair with remaining all $(n-1)$ elements. consequently, 2^{nd} element make inversions with $(n-2)$ elements.so,

Total Number of inversions $S = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$

$$\sum_{n=i}^{n-1}(n-i) = \sum_{n=i}^{n-1} n - \sum_{n=i}^{n-1} i$$

$$= n(n-1) - \frac{n(n-1)}{2} \Longrightarrow \frac{n(n-1)}{2} = f(n) - inversions$$

Time Complexity:
Let us assume $f(n)$ denotes number of inversions in the input array of size n, we know that,

$$\boxed{T(n) = \Theta(f(n)) = \Theta(n^2)}$$

Algorithm for finding number of inversions:
We can write Algorithm based on Merge Sort(we may call **Modified-MS**) with slight modification for counting the inversions.the procedure is as follows:
Creating the Copy of array and Sub arrays $\longrightarrow$ Re-arranging, while counting the number $\longrightarrow$ inversion with Sub arrays $\longrightarrow$ Merging the Sub Arrays.

. **Count-Inversion**(A)
. $\quad$ *// create copy of Array*
. $\quad$ let Q[1,2, $\cdots$,n] be new array
. $\quad$ *// initialize the new arrays*
. $\quad$ for i = 1 to n
. $\quad\quad Q[i] = A[i]$
. $\quad$ **Return** Modified-MS(Q,0,Q.length)

. **Modified-MS**(A, p, r)
. $\quad$ *// if set is not null, proceed*
. $\quad$ **if** p < r:

```
    // The number of inversions are initially zero
    m = 0
    // Get a mid point
    q = (p+r)/2
    //Add the number of inversion in the left
    m = m + Modified-MS(A,p,q)
    //Add the number of inversion in the Right
    m = m + Modified-MS(A,q+1,r)
    //Add the number of inversion remained and in between of different set
    m = m + Modified-Merge(A,p,q,r)
    Return m

Modified-Merge(A, p, q, r)
    // The number of inversion is initially zero
    m = 0
    // Get the sizes
    S1 = q - p + 1
    S2 = r -q
    // Create new arrays for left ⟶ L, right ⟶ R
    Let L[1, 2, ··· , S1] and R[1, 2, ··· , S2] be new arrays
    //initialize the new arrays
    for i = 1 to S1
        L[i] = A[p + i − 1]
    for j = 1 to S2
        R[j] = A[q + j]
    //we use i,j to hold the current index of two arrays and initialize the indexes
    i =1
    j =1
    // Loop over the original Array and copy the items accordingly
    for k = p to r
    // Copy from the L, if i is within the range or either
    // the left has smaller value
    // or right index went out of the bounds
    if i ≤ S1 and (j > S2 or L[i] ≤ R[j])
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
    // when right is smaller, then left elements are there,which indicates inversion
    if i ≤ S1
        // each remained element in the left
        //makes an inversion pair
        m = m + (S1 − i) + 1
    Return m
```

while modifying the merge sort algorithm, we can count the number of inversion with this order.

$$T(n) = \underbrace{\Theta(n)}_{\text{copying}} + \underbrace{\Theta(nlogn)}_{\text{sorting}} = \Theta(n \log n)$$

4. Write C-programs for sorting n-given integers by using Bubble-Sort, Merge-Sort, Quick-Sort and Randomized Quick-Sort methods and compare their running time over a large set of inputs.

Solution:
Python programming language is used for encoding the different 4 types of algorithms. In order to check the running time over test input, there is 10000 random integers ranging from 0 to 10000 were generated by using random function. ***100 different trials*** were simulated.

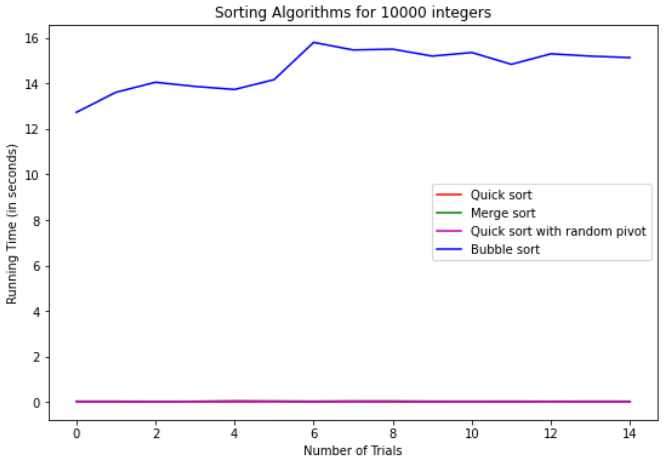
```
import random
import time

"Creating the test data"
Array list = list()
for _ in range(10000):
    temp = random.randint(0, 10000)
    Array_list.append(temp)

"Bubble Sort Algorithm"
def bubble sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
    return L

"Merge sort Algorithm"
def merge(left, right):
    result = []
    i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result

def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

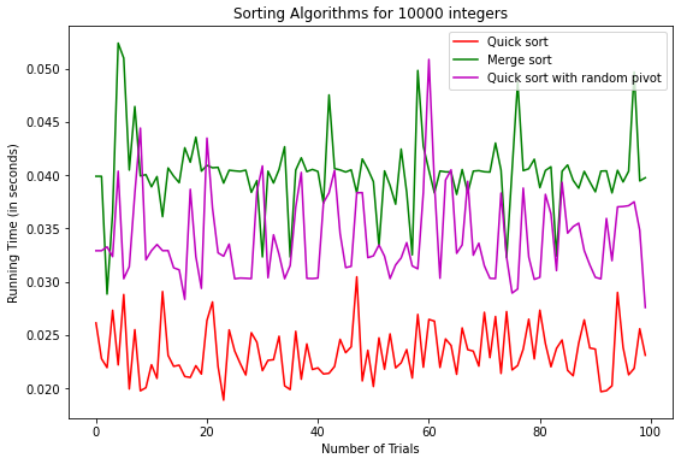
```
"""
 Quick sort Algorithm
 """
def partition(arr, low, high):
    i = (low-1)         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr, low, high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

```
"""
 Quick sort Algorithm with random pivoting
 """
 def quicksort(arr, start, stop):
    if(start < stop):
        pivotindex = partitionrand(arr, start, stop)
        quicksort(arr , start , pivotindex)
        quicksort(arr, pivotindex + 1, stop)

def partitionrand(arr , start, stop):
    randpivot = random.randrange(start, stop)
    arr[start], arr[randpivot] = arr[randpivot], arr[start]
    return partition(arr, start, stop)

def partition(arr,start,stop):
    pivot = start # pivot
    i = start - 1
    j = stop + 1
    while True:
        while True:
            i = i + 1
            if arr[i] >= arr[pivot]:
                break
        while True:
            j = j - 1
            if arr[j] <= arr[pivot]:
                break
        if i >= j:
            return j
        arr[i] , arr[j] = arr[j] , arr[i]
```

Average Time taken for 100 different trials:

Bubble Sort	$\longrightarrow$ 14.6600 *seconds*
Merge Sort	$\longrightarrow$ 0.04074 *seconds*
Quick Sort	$\longrightarrow$ 0.02371 *seconds*
Quick Sort with random pivot	$\longrightarrow$ 0.03613 *seconds*





Observation: $\longrightarrow$ Among all four algorithms Quick sort algorithm with high index pivot is optimal one.

5. Prove the following statements:i. For any two functionsf(n) andg(n), we havef(n) = (g(n)) if and only iff(n) =O(g(n)) andf(n) = (g(n)).ii.maxf(n),g(n)= (f(n)+g(n)), wheref(n) andg(n) are asymptotically nonnegative functions

Solution:

6. o(g(n)) =f(n)—for any constantc ¿0,a constantn0¿0,such that 0f(n)¡ g(n),nno.(g(n)) =f(n)—for any constantc ¿0,a constantn0¿0,such that 0cg(n)¡ f(n),nno.By using the above definitions prove thato(g(n))(g(n)) =.

Solution:

7. Find the solution of the following recurrence relations:i.T(n) =T(dn2e) + 1ii.T(n) = 2T(bn2c) +niii.T(n) = 2T(bn2c+ 37) +niv.T(n) = 2T(n) + 1v.T(n) = 3T(bn2c) +nvi.T(n) =T(n3) +T(2n3) +cn, wherecis a constant.vii.T(n) =T(na) +T(a) +cn, wherea1 andc ¿0.viii.T(n) =T(n) +T((1)n) +cn, whereis a constant in the range 0¡ ¡1 andc ¿0 is alsoa constant.

Solution:

8. Use master theorem to find the exact solution of the following recurrence relations:i.T(n) = 9T(n3) +n.ii.T(n) = 4T(n2) +n3.iii.T(n) =T(n2) + (1).iv.T(n) = 4T(n2) +n2logn.

Solution: