**Abdullah Tahir**

**21L-5419**

**Assignment # 1**

**Design and analysis with Divide and Conquer**

**Problem 1. Comparison of Sorting Algorithms**

It is claimed that Insertion sort is faster than any of its competing O(n2) sorts. This claim becomes important for small values of n. For large values of n, O(n2) is simply too slow and the differences of small constant multiples inside the big-O don't make a difference. In this exercise, your job is to test the claim for small and moderate values of n; specially, for n=10, n=25, n=100, n=1,000 and n=10,000. You will practically compare the execution times of Insertion Sort, Selection Sort and Bubble sort.

You must follow these steps:

**(1)** Write **C++** functions for Insertion Sort, Bubble Sort and Selection Sort.

**Solution:**

**Insertion Sort:**

void insertionSort(int arr[], int n)

{

for (int i = 1; i < n; i++)

{

int key = arr[i];

int j;

for ( j = i - 1; j >= 0 && arr[j] > key; j--)

{

arr[j + 1] = arr[j];

}

arr[j + 1] = key;

}

}

**Bubble Sort:**

void bubbleSort(int arr[], int n)

{

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < n - i - 1; j++)

{

if (arr[j] > arr[j + 1])

{

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

**Selection Sort:**

void selectionSort(int arr[], int n)

{

for (int i = 0; i < n - 1; i++)

{

int min\_ind = i;

for (int j = i + 1; j < n; j++)

{

if (arr[j] < arr[min\_ind])

{

min\_ind = j;

}

}

int temp = arr[i];

arr[i] = arr[min\_ind];

arr[min\_ind] = temp;

}

}

**(2)** For n=10, 25, 100, 1,000 and 10,000:

a. Generate a random array, A, of n integers.

b. Make three identical copies of A: A1, A2 and A3.

c. Execute Insertion Sort on A1, Bubble Sort on A2 and Selection Sort on A3, and compute their execution times T1, T2 and T3.

d. Present your results in a tabular format, as given below with rows for values of n and columns for each algorithm. Each entry should show the running time in **milliseconds**.

**Solution:**

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **Insertion Sort T1** | **Bubble Sort**  **T2** | **Selection Sort**  **T3** |
| 10 | 0.0007 ms | 0.0012 ms | 0.0001 ms |
| 25 | 0.0021 ms | 0.0039 ms | 0.0031 ms |
| 100 | 0.0254 ms | 0.2143 ms | 0.0248 ms |
| 1,000 | 23.6529 ms | 4.2678 ms | 2.5725 ms |
| 10,000 | 166.842 ms | 470.592 ms | 255.481 ms |

**(3)** Implement two types of Merge Sort

i. One with division in two parts each of size (n/2)

**Solution:**

**(i) Merge Sort with Division into Two Parts (n/2):**

void Merge(int arr[], int low, int mid, int high)

{

int n\_a = mid - low + 1;

int n\_b = high - mid;

int\* arr\_a = new int[n\_a];

int\* arr\_b = new int[n\_b];

for (int i = 0; i < n\_a; i++)

{

arr\_a[i] = arr[low + i];

}

for (int i = 0; i < n\_b; i++)

{

arr\_b[i] = arr[mid + 1 + i];

}

int i = 0;

int j = 0;

int k = low;

while (i < n\_a && j < n\_b)

{

if (arr\_a[i] <= arr\_b[j])

{

arr[k] = arr\_a[i];

i++;

}

else

{

arr[k] = arr\_b[j];

j++;

}

k++;

}

while (i < n\_a)

{

arr[k] = arr\_a[i];

i++;

k++;

}

while (j < n\_b)

{

arr[k] = arr\_b[j];

j++;

k++;

}

delete[] arr\_a;

delete[] arr\_b;

}

void mergeSort(int arr[], int low, int high)

{

if (low < high)

{

int mid = low + (high - low) / 2;

mergeSort(arr, low, mid);

mergeSort(arr, mid + 1, high);

Merge(arr, low, mid, high);

}

}

ii. Division in three parts each of size (n/3)

**Solution:**

**Merge Sort with Division into Three Parts (n/3):**

void merge(int arr[], int low, int mid1, int mid2, int high)

{

int n\_a = mid1 - low + 1;

int n\_m = mid2 - mid1;

int n\_b = high - mid2;

int\* arr\_a = new int[n\_a];

int\* arr\_m = new int[n\_m];

int\* arr\_b = new int[n\_b];

for (int i = 0; i < n\_a; i++)

{

arr\_a[i] = arr[low + i];

}

for (int i = 0; i < n\_m; i++)

{

arr\_m[i] = arr[mid1 + 1 + i];

}

for (int i = 0; i < n\_b; i++)

{

arr\_b[i] = arr[mid2 + 1 + i];

}

int i = 0;

int j = 0;

int k = 0;

int l = low;

while (i < n\_a && j < n\_m && k < n\_b)

{

if (arr\_a[i] <= arr\_m[j] && arr\_a[i] <= arr\_b[k])

{

arr[l] = arr\_a[i];

i++;

} else if (arr\_m[j] <= arr\_a[i] && arr\_m[j] <= arr\_b[k])

{

arr[l] = arr\_m[j];

j++;

} else

{

arr[l] = arr\_b[k];

k++;

}

l++;

}

while (i < n\_a)

{

arr[l] = arr\_a[i];

i++;

l++;

}

while (j < n\_m)

{

arr[l] = arr\_m[j];

j++;

l++;

}

while (k < n\_b)

{

arr[l] = arr\_b[k];

k++;

l++;

}

delete[] arr\_a;

delete[] arr\_m;

delete[] arr\_b;

}

void mergeSort(int arr[], int low, int high)

{

if (low < high)

{

int n = high - low + 1;

int mid1 = low + (n / 3);

int mid2 = low + 2 \* (n / 3);

mergeSort(arr, low, mid1);

mergeSort(arr, mid1 + 1, mid2);

mergeSort(arr, mid2 + 1, high);

merge(arr, low, mid1, mid2, high);

}

}

**(4)** Compare three algorithms for an array of size 100, to size 100,000 as given in following table. a. Generate a random array, A, of n integers.

b. Make three identical copies of A: A1, A2 and A3.

c. Execute Insertion Sort on A1, Merge Sort (n/2) on A2 and Merge Sort (n/3) on A3, and compute their execution times T1, T2 and T3. Each entry should show the running time in **milliseconds**. What time difference do you see in each case?

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **Insertion Sort T1** | **Merge Sort**  **(n/2)**  **T2** | **Merge Sort**  **(n/3)**  **T3** |
| 100 | 0.0234 ms | 0.1578 ms | 0.3478 ms |
| 1,000 | 2.601 ms | 1.6634 ms | 2.5463 ms |
| 10,000 | 54.8735 ms | 21.2506 ms | 14.2432 ms |
| 100,000 | 8499.345 ms | 130.232 ms | 123.2742 ms |

**Problem 2. Maximum Subarray Sum**

In this problem, you will practically compare the execution times of Maximum Subarray Sum problem.

**(1)** Write **C++** functions for the Maximum Subarray Sum problem.

i. Using Brute Force approach

ii. Divide and Conquer approach

**Solution:**

**(i) Brute Force Approach**

int maxSubarraySum(int arr[], int n)

{

int sum = 0;

for (int i = 0; i < n; i++)

{

int currsum = 0;

for (int j = i; j < n; j++)

{

currsum += arr[j];

if (currsum > sum)

{

sum = currsum;

}

}

}

return sum;

}

**(ii)**  **Divide and Conquer Approach**

int maxSubarraySum(int arr[], int low, int high)

{

if (low == high)

{

return arr[low];

}

int mid = (low + high) / 2;

int leftMax = maxSubarraySum(arr, low, mid);

int rightMax = maxSubarraySum(arr, mid + 1, high);

int totsum = Sum(arr, low, mid, high);

if(leftMax > rightMax && leftMax > totsum)

{

return leftMax;

}

else if(rightMax > leftMax && rightMax > totsum)

{

return rightMax;

}

else

{

return totsum;

}

}

int Sum(int arr[], int low, int mid, int high)

{

int leftSum = 0;

int rightSum = 0;

int sum = 0;

for (int i = mid; i >= low; i--)

{

sum += arr[i];

if (sum > leftSum)

{

leftSum = sum;

}

}

sum = 0;

for (int i = mid + 1; i <= high; i++)

{

sum += arr[i];

if (sum > rightSum)

{

rightSum = sum;

}

}

return leftSum + rightSum;

}

**(2)** Compare both algorithms for an array of size 1, to size 10,000 as given in following table. a. Generate a random array, A, of n integers positive and negative.

b. Make two identical copies of A: A1, and A2.

c. Execute Brute Force approach on A1, Divide and Conquer approach on A2, and compute their execution times T1 and T2. Each entry should show the running time in **milliseconds**. What time difference do you see in each case?

|  |  |  |
| --- | --- | --- |
| **n** | **Brute Force**  **approach**  **T1** | **Divide and Conquer approach**  **T2** |
| 1 | 0.0004 ms | 0.0002 ms |
| 10 | 0.0013 ms | 0.0016 ms |
| 100 | 0.0189 ms | 0.0056 ms |
| 1,000 | 2.0342 ms | 0.032 ms |
| 10,000 | 74.56 ms | 0.0123 ms |

**Problem 3. Design**

**(1)** Write an in-place version of the merge function that should works in **O(n2).**

**Solution:**

**In Place Merge Function:**

void Merge(int arr[], int low, int mid, int high)

{

int i = low;

int j = mid + 1;

while (i <= mid && j <= high)

{

if (arr[i] <= arr[j])

{

i++;

}

else

{

int temp = arr[j];

for (int k = j; k > i; k--)

{

arr[k] = arr[k - 1];

}

arr[i] = temp;

i++;

j++; }

}

}

**(2)** Given two sorted arrays A and B of sizes m and n respectively, where m ≥ n, devise an algorithm to merge them into a new sorted array C using only **O (n lg m)** comparison operations.

**Solution:**

void Merge(int A[], int m, int B[], int n, int C[])

{

int i = m - 1;

int j = n - 1;

int k = m + n - 1;

while (i >= 0 && j >= 0)

{

if (A[i] >= B[j])

{

C[k] = A[i];

i--;

} else

{

C[k] = B[j];

j--;

}

k--;

}

while (i >= 0)

{

C[k] = A[i];

i--;

k--;

}

while (j >= 0)

{

C[k] = B[j];

j--;

k--;

}

}

**(3)** Given an unsorted array of integers, A, its size n, and two numbers x and y both are elements of A, write an algorithm that returns the distance between x and y. The distance between two numbers of an array is the number of elements that lie between them in the sorted order. Achieve the asymptotically fastest time for this problem.

**Solution:**

int Distance\_Between\_XY(int arr[], int n, int x, int y)

{

int dis = 100000;

int ind\_x = -1;

int ind\_y = -1;

for (int i = 0; i < n; i++)

{

if (arr[i] == x)

{

if (ind\_y != -1)

{

if(dis > i - ind\_y)

{

dis = i - ind\_y;

}

}

ind\_x = i;

}

else if (arr[i] == y)

{

if (ind\_x != -1)

{

if(dis > i - ind\_x)

{

dis = i - ind\_x;

}

}

ind\_y = i;

}

}

if (ind\_x == -1 || ind\_y == -1)

{

return -1;

}

return dis - 1;

}

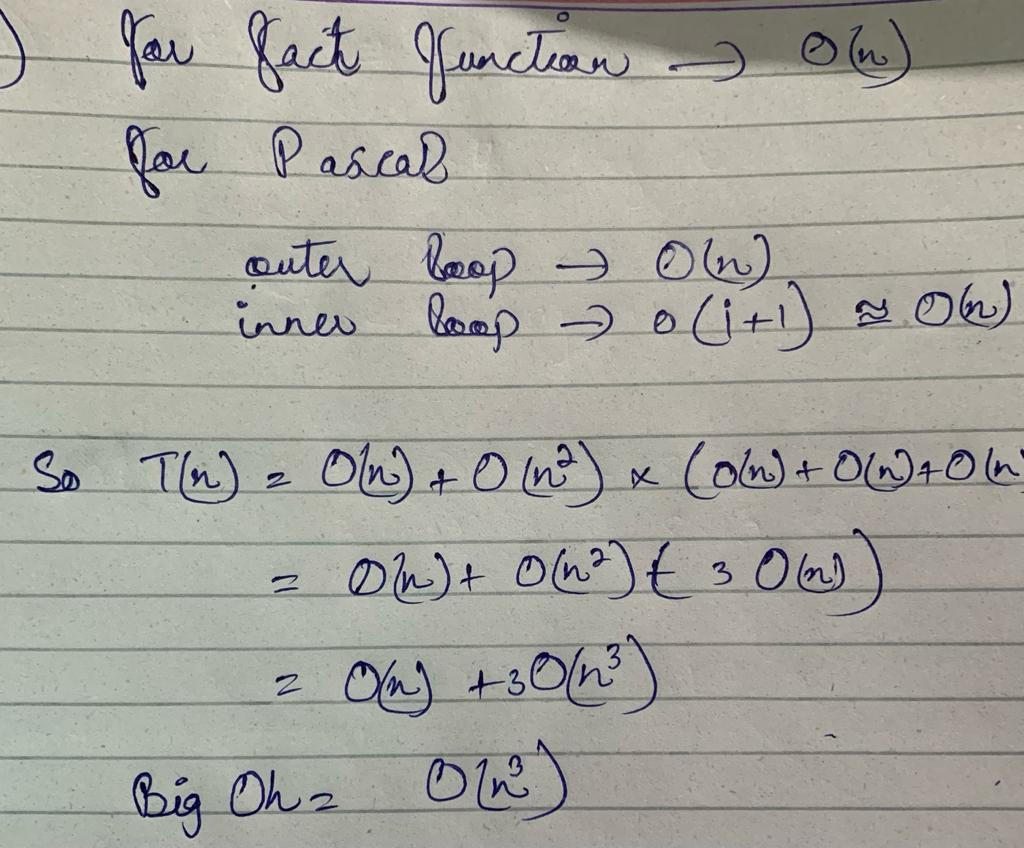
**Problem 4. Analysis**

Calculate the time complexity function T(n) and Big-O for the first two programs.

**(1)** Prints the Pascal Triangle

|  |  |
| --- | --- |
| void printPascal(int n){  long facti, factj, facti\_j;  for(int i=0;i<=n;i++){  for(int j=0;j<=i;j++){  facti=fact(i);  factj=fact(j);  facti\_j=fact(i-j);  cout<<facti/(factj\*facti\_j)<<'\t';  }  cout<<endl;  }  } | long fact(int m){  if (m==0 || m==1)  return 1;  long ans=1;  for(int i=2;i<=m;i++)  ans\*=i;  return ans;  } |

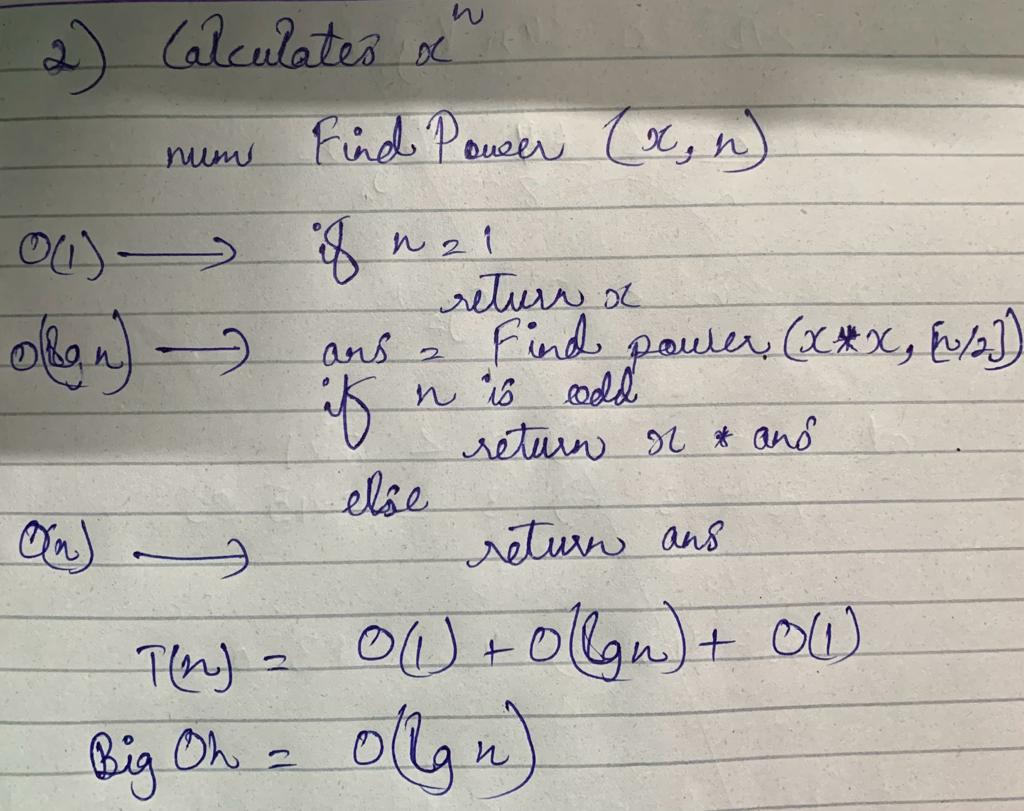
**Solution:**



**(2)** Calculates x*n*(where x and y are positive integers).

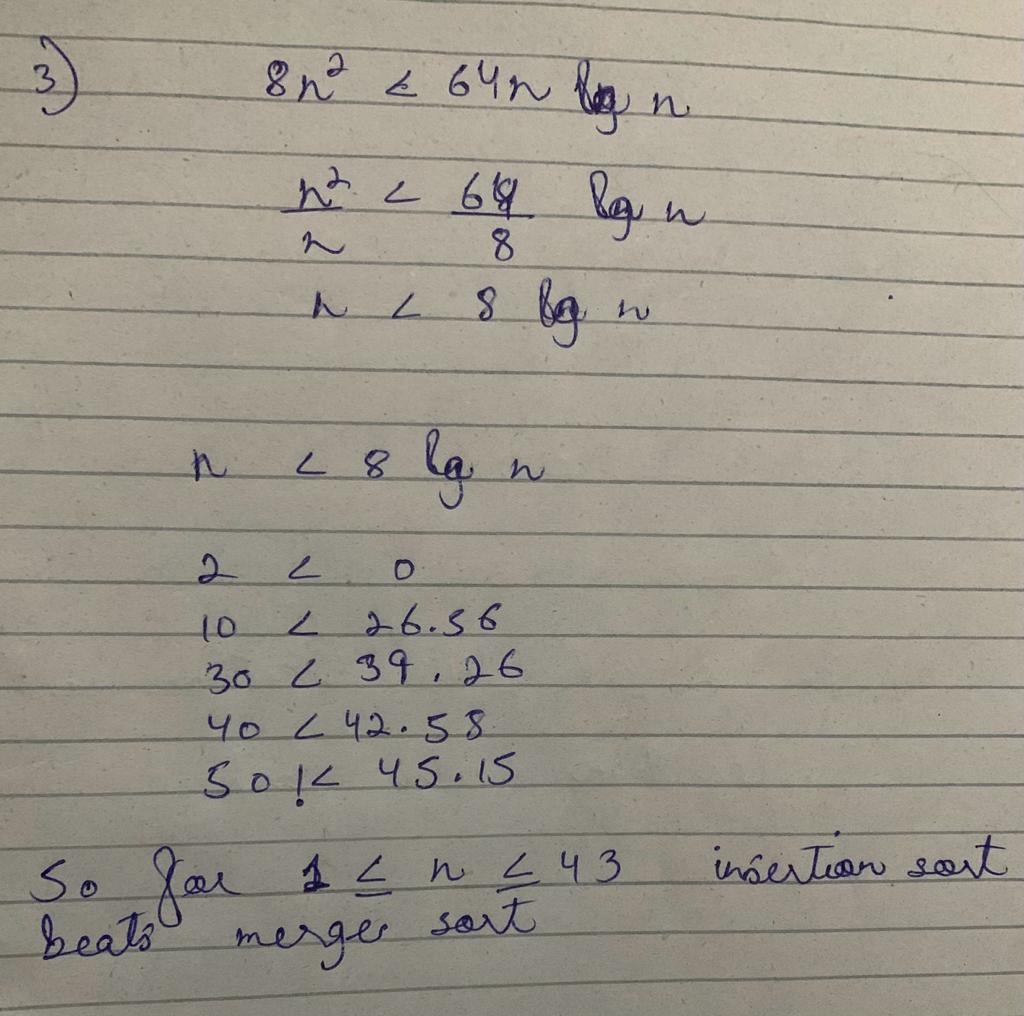
|  |
| --- |
| num FindPower(x,n)  if n = 1  return x  ans = FindPower(x\*x,[n/2])  if n is odd  return x\*ans  else  return ans |

**Solution:**

****

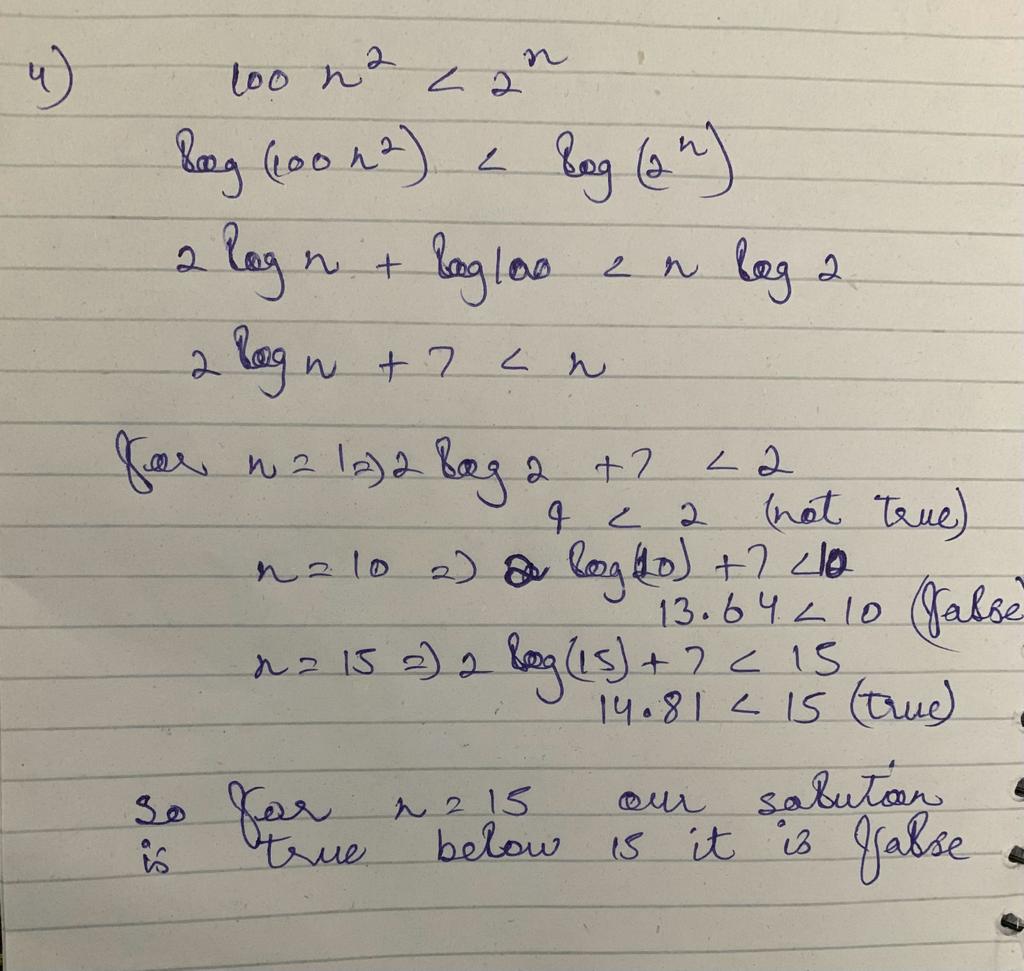
**(3)** Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size *n*, insertion sort runs in 8*n*2steps, while merge sort runs in 64*n* lg *n* steps. For which values of *n* does insertion sort beat the merge sort?

**Solution:**

****

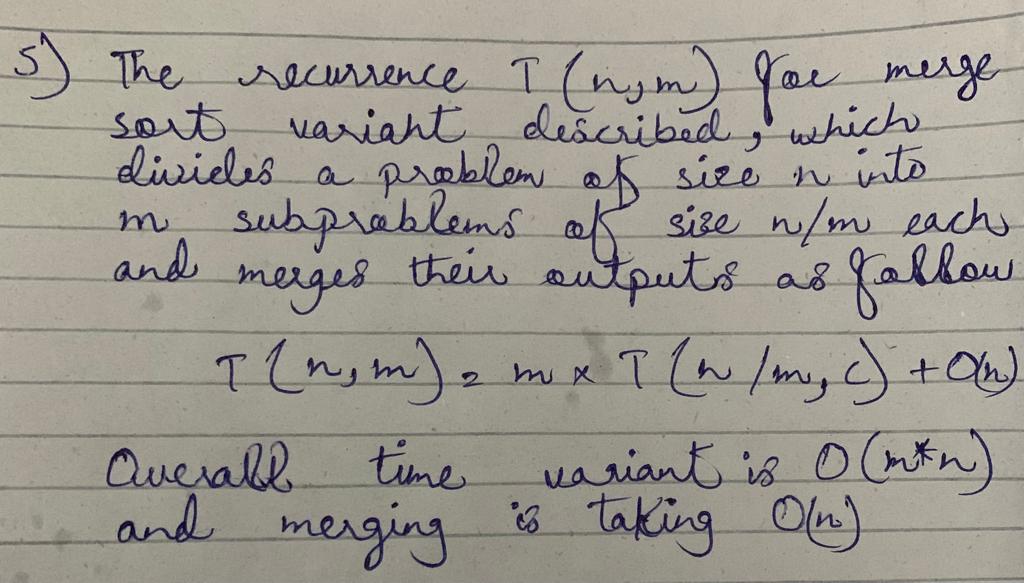
**(4)** What is the smallest value of *n* such that an algorithm whose running time is 100*n*2runs faster than an algorithm which running time is 2*n* on the same machine?

**Solution:**



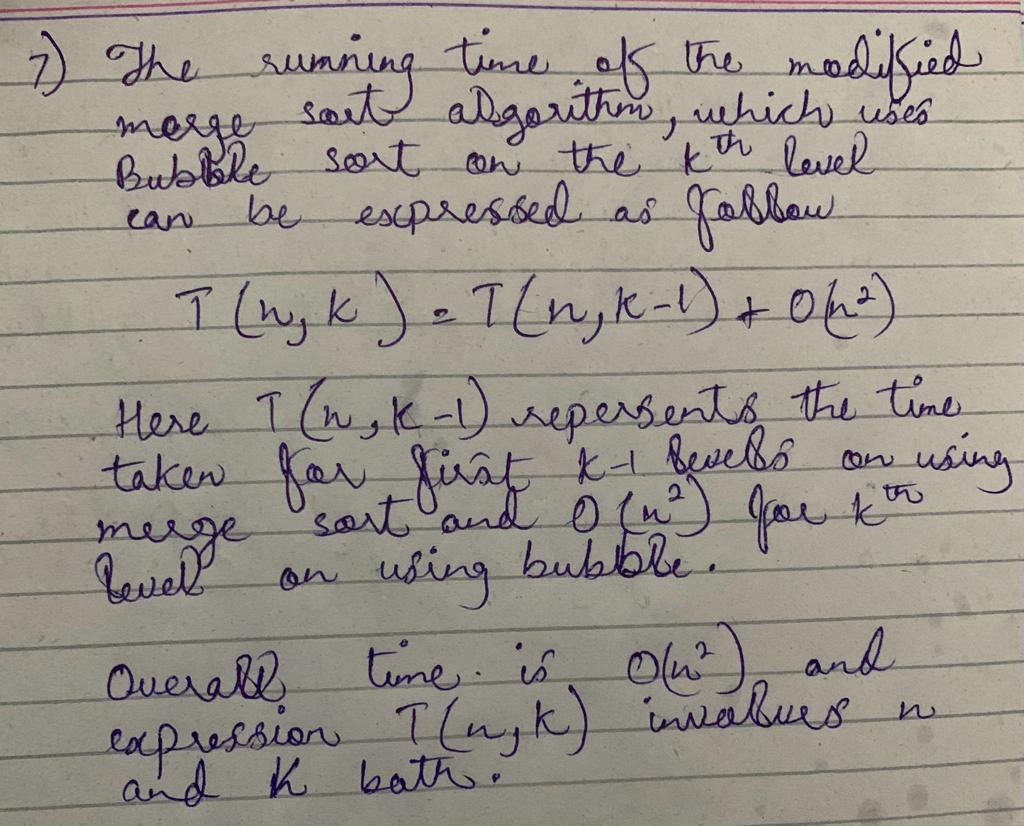
**(5)** Describe a recurrence T(n,m) of a Merge Sort variant which divides a problem of size n into m subproblems of size n/m each and merges their outputs. Provide an expression for T(n,m) in terms of n and m. Pay attention to how much time will be needed to merge the solutions of the m subproblems of size n/m each.

**Solution:**



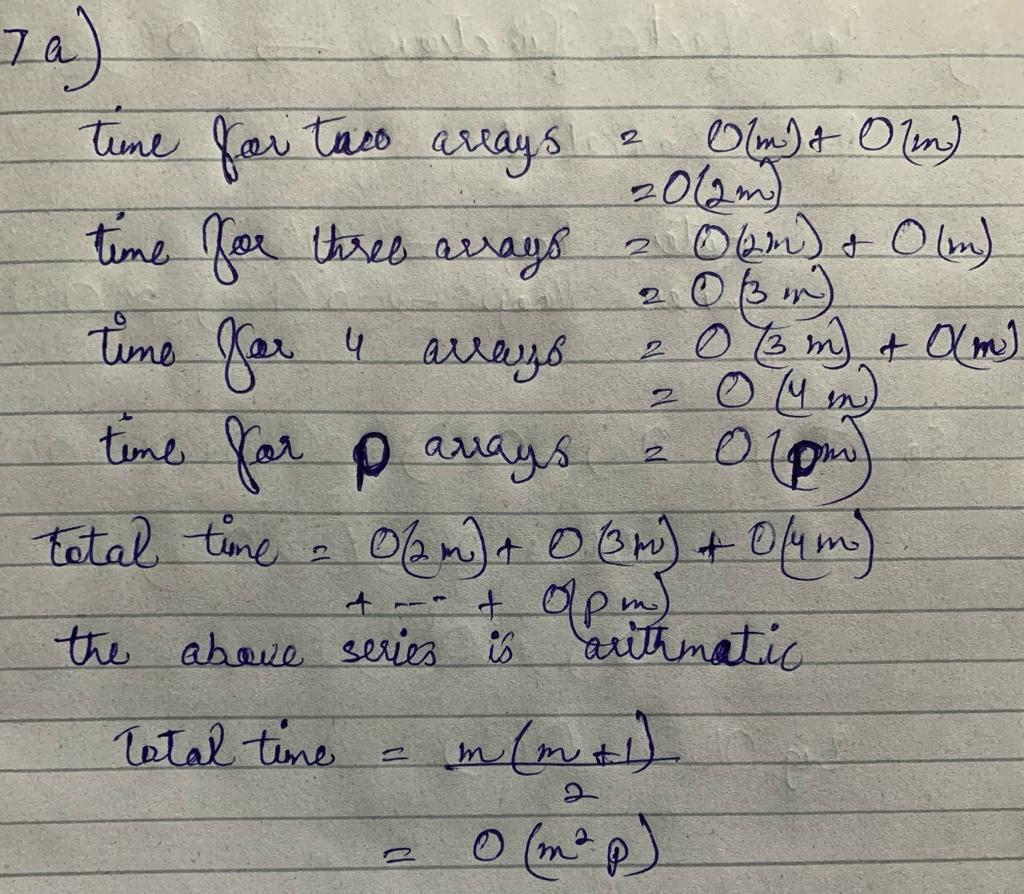
**(6)** In Merge Sort we divide the array as usual for the first k -1 levels of the recursion tree (where k is a parameter to the Merge Sort function). After that, we use Bubble Sort on the kth level to sort the subarrays at that level and go back up the tree merging as usual. What is the running time (in big-O terms) for this algorithm? The expression for T(n,k) should involve both n and k.

**Solution:**

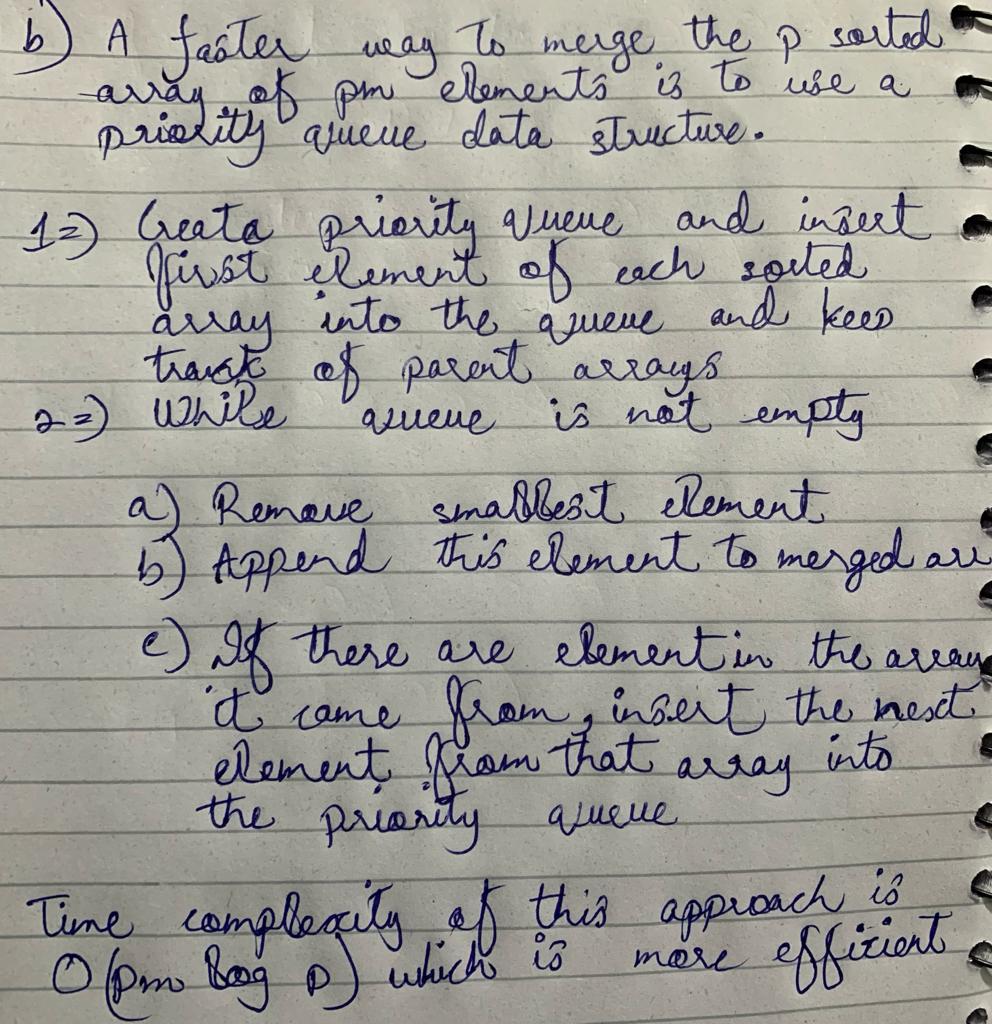


**(7)** Suppose you are given *p* sorted arrays, each with *m* elements, and you want to combine them into a single array of *pm* elements. Using the merge subroutine, you merge the first 2 arrays, then merge the 3*rd* given array with this merged version of the first two arrays, then merge the 4*th* given array with the merged version of the first three arrays, and so on until you merge the final (*pth*) input array.

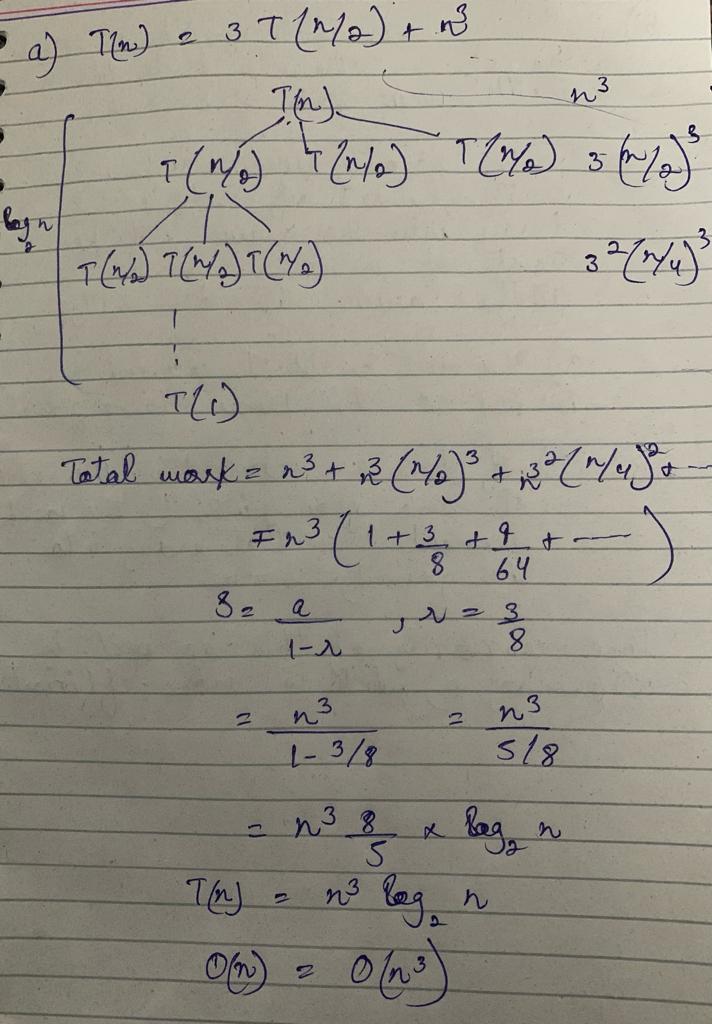
a. What is the running time taken by this successive merging algorithm, as a function of *p* and *m*?



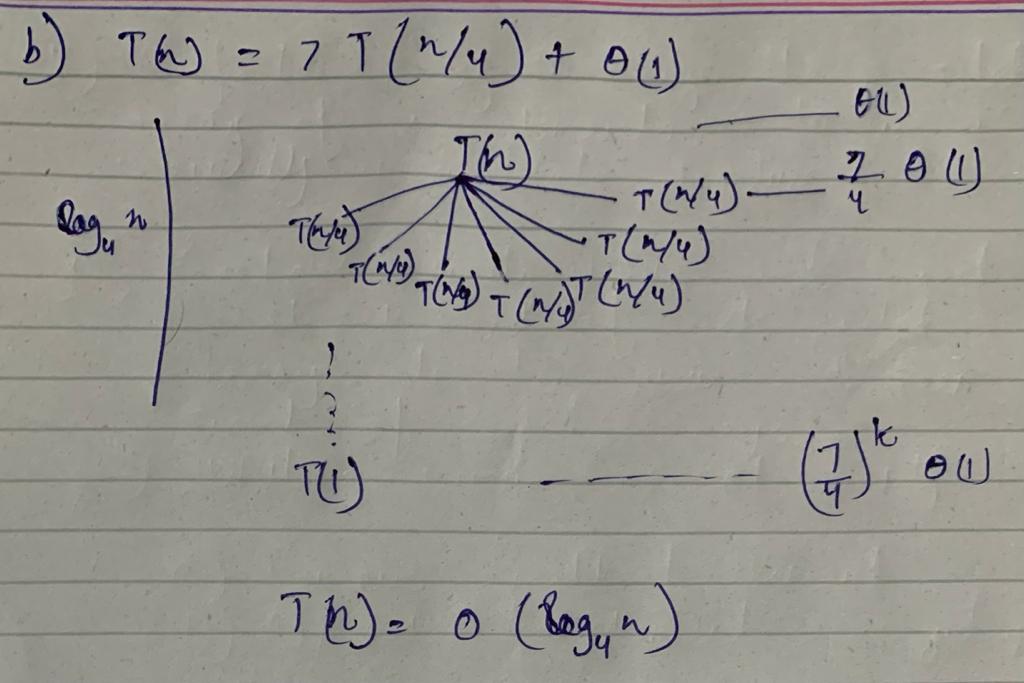
b. What is a faster way to do the above merge procedure?



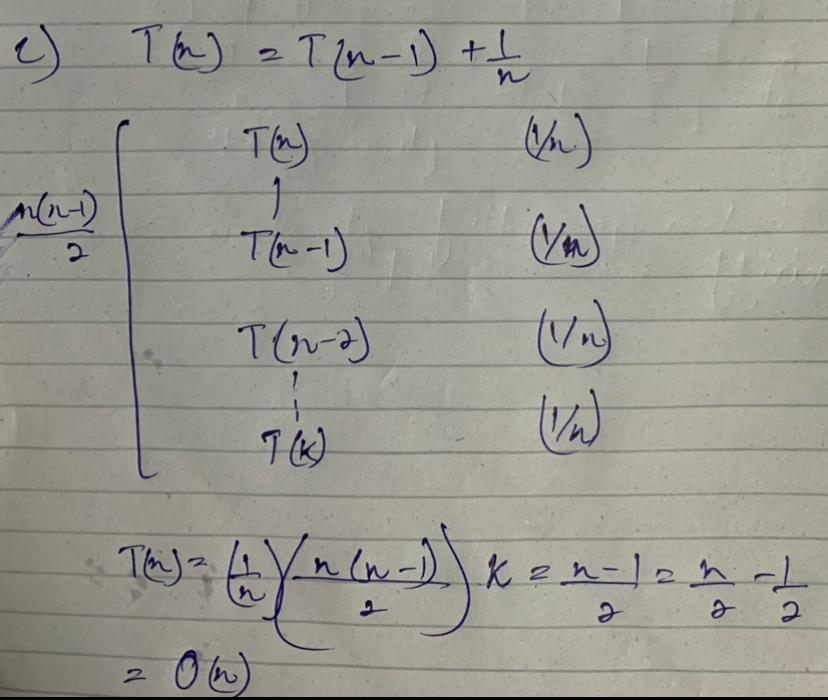
**(8)** Use recursion tree method to determine asymptotic upper bound on following recurrences. a. T (n) = 3T(n/2) + n3



b. T (n) = 7T(n/4) + Θ (1)



c. T (n) = T (n - 1) + 1/n



d. T (n) = 2T (n/2) + n/ lg n

