Ex-6	**Parallelizing recursive functions**
19/8/25	

AIM:

To

1. Write a C++ recursive function for the flood fill algorithm. Compare the OpenMP and serial versions of the code for different values of N.

2. Write a C++ program for merge sort with a cutoff array size for a serial nature. Experiment with different cutoff values and observe the corresponding running times.

CODE:

a)

```
#include<iostream>
#include<ctime>
#include<cstdlib>
#include<omp.h>

using namespace std;

const int MAXN=1000;
int image[MAXN][MAXN];
int backup[MAXN][MAXN];

void generateImage(int n){

  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
    image[i][j]=rand()%2;
   }
 }

}

void copyImage(int src[MAXN][MAXN],int dest[MAXN][MAXN],int n){
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
       dest[i][j]=src[i][j];
   }
  }
}

void floodFill(int x,int y,int targetColor,int replacementColor){
```

```
   if(x<0||x>=MAXN||y<0||y>=MAXN){
     return;
}
    if(image[x][y]!=targetColor){
   return;
  }
image[x][y]=replacementColor;
floodFill(x+1,y,targetColor,replacementColor);
floodFill(x-1,y,targetColor,replacementColor);
floodFill(x,y+1,targetColor,replacementColor);
floodFill(x,y-1,targetColor,replacementColor);

}

void parallelFloodFill(int x,int y,int targetColor,int replacementColor){
#pragma omp parallel
{
#pragma omp single
{
floodFill(x,y,targetColor,replacementColor);
}
}
}

int main(){
int sizes[]={100,200,300,400,500};

for(int k=0;k<5;k++){
  int n=sizes[k];

   generateImage(n);

  copyImage(image,backup,n);

  double start=omp_get_wtime();

  floodFill(0,0,image[0][0],3);

  double stop=omp_get_wtime();

  cout<<"serial n="<<n<<",time:"<<stop-start<<"s"<<endl;

  copyImage(backup,image,n);

  start=omp_get_wtime();

  parallelFloodFill(0,0,image[0][0],3);

  stop=omp_get_wtime();
  cout<<"parallel n="<<n<<",time:"<<stop-start<<"s"<<endl;
}
return 0;
```

```
}
```

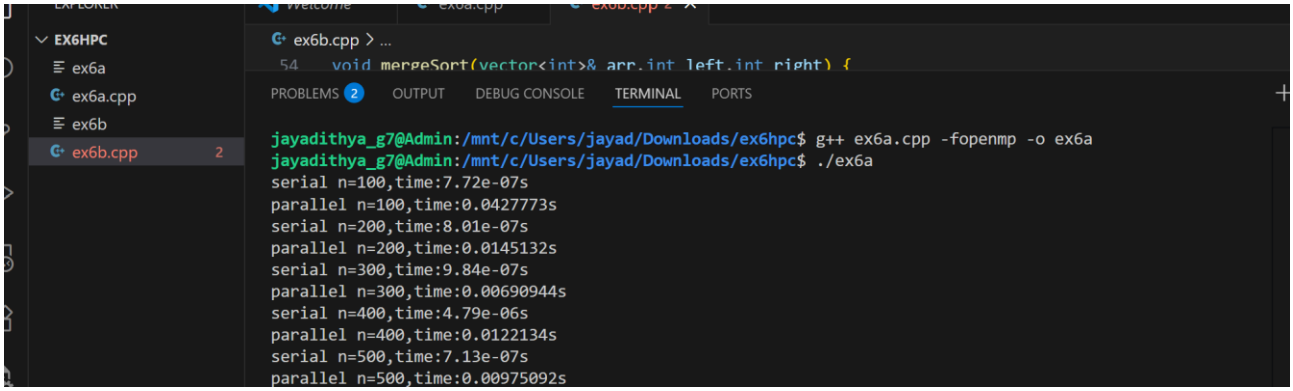
b)

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;
using namespace chrono;

const int MAXN=100000;
int cutoff=10;

void insertionSort(vector<int>& arr,int left,int right) {
    for (int i=left+1;i<=right;i++) {
        int key=arr[i];
        int j=i-1;
        while(j>=left && arr[j]>key) {
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=key;
    }
}

void merge(vector<int>& arr,int left,int mid,int right) {
    int n1=mid-left+1;
    int n2=right-mid;

    vector<int> L(n1),R(n2);
    for(int i=0;i<n1;i++){
    L[i]=arr[left+i];
    }
    for(int j=0;j<n2;j++){
    R[j]=arr[mid+1+j];
    }

    int i=0,j=0,k=left;
    while(i<n1 && j<n2){
        if(L[i]<=R[j]){
         arr[k++]=L[i++];
         }
        else{
         arr[k++]=R[j++];
         }
    }
    while(i<n1){
    arr[k++]=L[i++];
```

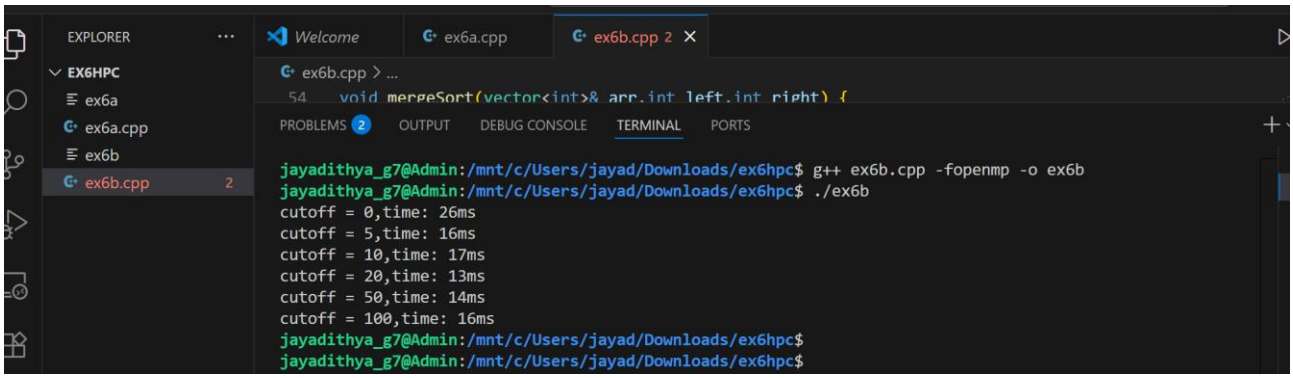
```
    }
    while(j<n2){
     arr[k++]=R[j++];
     }
}

void mergeSort(vector<int>& arr,int left,int right) {
    if(right-left+1<=cutoff){
        insertionSort(arr,left,right);
        return;
    }
    if(left<right){
        int mid=left+(right-left)/2;
        mergeSort(arr,left,mid);
        mergeSort(arr,mid+1,right);
        merge(arr,left,mid,right);
    }
}

void generateRandomArray(vector<int>& arr,int n) {
    arr.clear();
    arr.resize(n);
    for(int i=0;i<n;i++){
        arr[i]=rand()%100000;
    }
}

int main(){
    srand(time(0));
    int n=50000;
    vector<int> arr,temp;

    int cutoffs[]={0,5,10,20,50,100};

    for (int i=0;i<6;i++) {
        cutoff=cutoffs[i];
        generateRandomArray(arr,n);
        temp=arr;

        auto start=high_resolution_clock::now();
        mergeSort(arr,0,n-1);
        auto stop=high_resolution_clock::now();

        auto duration=duration_cast<milliseconds>(stop-start);
        cout<<"cutoff = "<<cutoff<<",time: "<<duration.count()<<"ms"<<endl;
    }

    return 0;
}
```

INPUT/OUTPUT:

a)



b)



RESULT:

Hence the code has been executed successfully and output has been verified.