//Sorting Algorithms.

//Selection Sort.

void selection(int \*A,int n)

{

int i,j,pmin;

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

{

pmin=i;

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

if(A[j]<A[pmin])

pmin-j;

A[i]=A[i]^A[pmin];

A[pmin]=A[i]^A[pmin];

A[i]=A[i]^A[pmin];

}

}

//----------------------------------------------------------

//Bubble Sort.

void traditional\_bubble(int \*A,int n)

{

int i,j;

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

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

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

{

A[j]=A[j]^A[j+1];

A[j+1]=A[j]^A[j+1];

A[j]=A[j]^A[j+1];

}

}

//----------------------------------------------------------

//Improved Bubble Sort.

void improved\_bubble(int \*A,int n)

{

int bound=n-1;

int j,temp,lastswap;

while(bound>0){

lastswap=-1;

for(j=0;j<bound;j++)

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

{

temp=A[j];

A[j]=A[j+1];

A[j+1]=temp;

lastswap=j;

}

bound=lastswap;

}

}

//----------------------------------------------------------

//Cocktail-Shaker Sort.

void cocktail\_shaker(int \*A,int n)

{

int boundleft=0;

int boundright=n-1;

int j,temp,lastswap;

while(boundleft<boundright){

lastswap=boundleft-1;

for(j=boundleft;j<boundright;j++)

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

{

temp=A[j];

A[j]=A[j+1];

A[j+1]=temp;

lastswap=j;

}

boundright=lastswap;

if(boundleft>=boundright)

break;

lastswap=boundright+1;

for(j=boundright;j>boundleft;j--)

if(A[j-1]>A[j])

{

temp=A[j-1];

A[j-1]=A[j];

A[j]=temp;

lastswap=j;

}

boundleft=lastswap;

}

}

//----------------------------------------------------------

//Merge Sort.

void merge(int \*A,int \*M,int sp1,int mid,int ep2)

{

int sp2,ep1,pos,temp,k;

temp=sp1;

sp2=mid;//starting pos(inclusive) of 2nd array.

ep1=mid;//ending pos(exclusive) of 1st array.

pos=sp1;//controls position of merged array.

while(sp1<ep1&&sp2<ep2)

{

if(A[sp1]<=A[sp2])//Maintaining Stability of initial array.

M[pos++]=A[sp1++];

else

M[pos++]=A[sp2++];

}

while(sp1<ep1)

M[pos++]=A[sp1++];

while(sp2<ep2)

M[pos++]=A[sp2++];

//Updating original array.

for(k=temp;k<pos;k++)

A[k]=M[k];

}

void divide\_and\_merge(int \*A,int \*M,int sp,int ep)//sp is inclusive while ep is exclusive.

{

if(sp+1<ep)//At least 2 elements in the range.

{

int mid=(sp+ep)/2;

divide\_and\_merge(A,M,sp,mid);

divide\_and\_merge(A,M,mid,ep);

merge(A,M,sp,mid,ep);

}

}

void merge\_sort(int \*A,int n)

{

int M[n];

divide\_and\_merge(A,M,0,n);

}

//----------------------------------------------------------

//Quick Sort.

int partition(int \*A,int sp,int ep)

{

int p=A[ep];//assuming pivot as last element.

int left,right;

left=sp;

right=ep;

while(left<right)

{

while(A[left]<p)

left++;

while(A[right]>=p)

right--;

if(left<right)

{

A[left]=A[left]^A[right];

A[right]=A[left]^A[right];

A[left]=A[left]^A[right];

}

}

A[ep]=A[left];

A[left]=p;

return left;

}

void quick(int \*A,int sp,int ep)//both sp and ep inclusive.

{

if(sp<ep)

{

int p=partition(A,sp,ep);

if(p<=(sp+ep)/2)

{

quick(A,sp,p-1);

quick(A,p+1,ep);

}

else

{

quick(A,p+1,ep);

quick(A,sp,p-1);

}

}

}

void quick\_sort(int \*A,int n)

{

quick(A,0,n-1);

}

//----------------------------------------------------------

//Randomized Quick Sort.

void randomized\_quick(int \*A,int sp,int ep)//both sp and ep inclusive.

{

if(sp<ep)

{

//Randomizing starting pivot element which is the last element here.

int r=sp+rand()%(ep-sp+1);

//r=ep will update elements at each of the positions as 0.

if(r!=ep)

{

A[ep]=A[ep]^A[r];

A[r]=A[ep]^A[r];

A[ep]=A[ep]^A[r];

}

int p=partition(A,sp,ep);

if(p<=(sp+ep)/2)

{

randomized\_quick(A,sp,p-1);

randomized\_quick(A,p+1,ep);

}

else

{

randomized\_quick(A,p+1,ep);

randomized\_quick(A,sp,p-1);

}

}

}

void randomized\_quick\_sort(int \*A,int n)

{

randomized\_quick(A,0,n-1);

}

//------------------------------------------------------------

//Insertion Sort.

void insertion\_sort(int \*A,int n)

{

int i,pos,key;

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

{

key=A[i];

pos=i-1;

while(pos>=0&&A[pos]>key)

{

A[pos+1]=A[pos];

pos--;

}

A[pos+1]=key;//Inserting key at correct position.

}

}

//------------------------------------------------------------

//Counting Sort.

int getmax(int \*A,int n)

{

int pmax=0,i;

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

if(A[i]>A[pmax])

pmax=i;

return A[pmax];

}

int getmin(int \*A,int n)

{

int pmin=0,i;

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

if(A[i]<A[pmin])

pmin=i;

return A[pmin];

}

void counting\_sort(int \*A,int n)

{

int max,min;

//Finding max and min in the array, O(n).

max=getmax(A,n);

min=getmin(A,n);

int K=max-min+1;//K is the size of the auxiliary array.

int C[K];//Cumulative array;

int S[n];//Sorted array.

int k,i;

for(k=0;k<K;k++)//O(K)

C[k]=0;

for(i=0;i<n;i++)//O(n)

C[A[i]-min]++;

for(k=1;k<K;k++)//O(K)

C[k]+=C[k-1];

for(i=n-1;i>=0;i--)//O(n)

S[--C[A[i]-min]]=A[i];

//Updating original array.

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

A[i]=S[i];

}

//------------------------------------------------------------

//Radix Sort.

void sort\_using\_bucket(int \*A,int n,int place\_value)//Bucket sort(Fixed set).

{

int K=10;//10 digits.

int C[K];//Cumulative array;

int S[n];//Sorted array.

int k,i;

for(k=0;k<K;k++)//O(K)

C[k]=0;

for(i=0;i<n;i++)//O(n)

C[(A[i]/place\_value)%10]++;//(A[i]/place\_value)%10 = the digits at current pass.

for(k=1;k<K;k++)//O(K)

C[k]+=C[k-1];

for(i=n-1;i>=0;i--)//O(n)

S[--C[(A[i]/place\_value)%10]]=A[i];

//Updating original array.

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

A[i]=S[i];

}

void radix\_sort(int \*A,int n)

{

int max=getmax(A,n);

int place\_value=1;

while(place\_value<=max)

{

sort\_using\_bucket(A,n,place\_value);

place\_value\*=10;

}

}

//Driver Functions to test the time taken by the sorting algorithms.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "sorting\_algorithms.c"

void write\_data\_to\_file(const char \*s,int \*A,int n)

{

FILE \*fp=fopen(s,"w");

int k;

for(k=0;k<n;k++)

fprintf(fp,"%d\n",A[k]);

fclose(fp);

}

void read\_data\_from\_file(const char \*s,int \*A,int n)

{

FILE\*fp=fopen(s,"r");

int k;

for(k=0;k<n;k++)

fscanf(fp,"%d\n",A+k);

fclose(fp);

}

//Generates data set from 1 to K in random order, K<=RAND\_MAX.

void generate1(int A[],int n,int K)

{

FILE \*fp=fopen("Data\_File.txt","w");

if(fp==NULL)

exit(0);

int k;

for(k=0;k<n;k++)

A[k]=rand()%K+1;

write\_data\_to\_file("Data\_File.txt",A,n);

}

//Generates data set from 1 to n in random order.

void generate2(int A[],int n)

{

FILE \*fp=fopen("Data\_File.txt","w");

if(fp==NULL)

exit(0);

int k,r;

for(k=0;k<n;k++)

A[k]=k+1;

//Shuffling.

for(k=0;k<n;k++)

{

r=rand()%n;

if(r!=k)

{

A[k]=A[k]^A[r];

A[r]=A[k]^A[r];

A[k]=A[k]^A[r];

}

}

write\_data\_to\_file("Data\_File.txt",A,n);

}

//Generates data set in ascending order.

void generate3(int A[],int n)

{

FILE \*fp=fopen("Data\_File.txt","w");

if(fp==NULL)

exit(0);

int k;

for(k=0;k<n;k++)

A[k]=k+1;

write\_data\_to\_file("Data\_File.txt",A,n);

}

//Generates data set in descending order.

void generate4(int A[],int n)

{

FILE \*fp=fopen("Data\_File.txt","w");

if(fp==NULL)

exit(0);

int k;

for(k=0;k<n;k++)

A[k]=n-k;

write\_data\_to\_file("Data\_File.txt",A,n);

}

//Main Generator.

void generate(int A[],int n)

{

int ch,K;

printf("1. Generate data set in random order with each data ranging from 1 to K.\n");

printf("2. Generate data set in random order from 1 to n.\n");

printf("3. Generate data set in ascending order.\n");

printf("4. Generate data set in descending order.\n");

printf("Enter your choice : ");

scanf("%d",&ch);

switch(ch)

{

case 1 :

printf("Enter the value of K (K<=%d) : ",RAND\_MAX);

scanf("%d",&K);

generate1(A,n,K);

break;

case 2 :

generate2(A,n);

break;

case 3 :

generate3(A,n);

break;

case 4 :

generate4(A,n);

break;

default :

printf("Wrong Choice.");

}

}

void driver\_function()

{

srand(time(NULL));//Seeding with current time.

int n;

printf("Enter the data set size : ");

scanf("%d",&n);

int A[n];

generate(A,n);

FILE \*result=fopen("Sorting\_Results.txt","w");

if(result==NULL)

return;

//Performing various sorting techniques on same data set and displaying time taken.

clock\_t t;

//Selection Sort.

t=clock();

selection(A,n);

t=clock()-t;

fprintf(result,"Time taken by Selection Sort = %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Selection Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Traditional Bubble Sort.

t=clock();

traditional\_bubble(A,n);

t=clock()-t;

fprintf(result,"Time taken by Traditional Bubble Sort = %.1f ms\n", (((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Traditional Bubble Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Improved Bubble Sort

t=clock();

improved\_bubble(A,n);

t=clock()-t;

fprintf(result,"Time taken by Improved Bubble Sort = %.1f ms\n", (((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Improved Bubble Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Cocktail Shaker Sort.

t=clock();

cocktail\_shaker(A,n);

t=clock()-t;

fprintf(result,"Time taken by Cocktail Shaker Sort= %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Cocktail Shaker Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Insertion Sort.

t=clock();

insertion\_sort(A,n);

t=clock()-t;

fprintf(result,"Time taken by Insertion Sort = %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Insertion Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Quick Sort.

t=clock();

quick\_sort(A,n);

t=clock()-t;

fprintf(result,"Time taken by Quick Sort = %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Quick Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Randomized Quick Sort.

t=clock();

randomized\_quick\_sort(A,n);

t=clock()-t;

fprintf(result,"Time taken by Randomized Quick Sort = %.1f ms\n", (((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Randomized Quick Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Merge Sort.

t=clock();

merge\_sort(A,n);

t=clock()-t;

fprintf(result,"Time taken by Merge Sort = %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Merge Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Counting Sort.

t=clock();

counting\_sort(A,n);

t=clock()-t;

fprintf(result,"Time taken by Counting Sort = %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Counting Sort Done.\n");

read\_data\_from\_file("Data\_File.txt",A,n);

//Radix Sort.

t=clock();

radix\_sort(A,n);

t=clock()-t;

fprintf(result,"Time taken by Radix Sort = %.1f ms\n",(((double)t)/CLOCKS\_PER\_SEC)\*1000);

printf("Radix Sort Done.\n");

write\_data\_to\_file("Sorted\_File.txt",A,n);

fclose(result);

}

void main()

{

driver\_function();

}