PESU University,Bengaluru.



DAA REPORT

**Title:**

**Mathematics and Statistics Library**

**Team Members:**

Anirudh Maiya PES1201700170

Pratheek Kamath M PES1201701595

**Semester**:IV

**Section:**H

**Program:**B.Tech

**Course code:**UE17CS251

To compute mode:

A ***mode*** is a value that occurs most often in a

given list of numbers. For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. First we sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the

longest run of adjacent equal values in the sorted array.This technique uses presorting idea.

**ALGORITHM** *PresortMode(A*[0*..n* − 1]*)*

//Computes the mode of an array by sorting it first

//Input: An array *A*[0*..n* − 1] of orderable elements

//Output: The array’s mode

sort the array *A*

*i* ←0 //current run begins at position *i*

*modef requency* ←0 //highest frequency seen so far

**while** *i* ≤ *n* − 1 **do**

*runlength*←1; *runvalue*←*A*[*i*]

**while** *i* + *runlength* ≤ *n* − 1 **and** *A*[*i* + *runlength*]= *runvalue*

*runlength*←*runlength* + 1

**if** *runlength > modef requency*

*modef requency* ←*runlength*; *modevalue*←*runvalue*

*i* ←*i* + *runlength*

**return** *modevalue*

The running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time. Consequently, with an *n* log *n* sort(where in the code we have implemented mergesort), this method’s worst-case efficiency will be in a better asymptotic class than the worstcase efficiency of the brute-force algorithm(Big theta(n2)).The following C code is:

void MergeSort(double\* A, int n)

{

int m;

if(n<=1)

return;

m = n/2;

MergeSort(A,m);

MergeSort(A+m,n-m);

Merge(A,n,m);

}

void Merge(double \*A,int n,int m)

{

int i,k,j = m;

double \*B = (double \*)malloc(sizeof(double) \* n);

i=k=0;

while(i<m && j<n)

{

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

B[k++] = A[i++];

else

B[k++] = A[j++];

}

if(j == n)

{

while(i<m && k<n)

B[k++] = A[i++];

}

else

{

while(j<n && k<n)

B[k++] = A[j++];

}

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

A[i] = B[i];

free(B);

}

double mode(double \*A,int n)

{

int i,mf,rl;

double rv,mv;

MergeSort(A,n);

i = mf = 0;

while(i<n)

{

rl = 1;

rv = A[i];

while(i+rl < n && A[i+rl] == rv)

rl+=1;

if(rl>mf)

{

mf = rl;

mv = rv;

}

i+=rl;

}

return mv;

}

**To compute median:**

Finding the median of k elements is same as finding the ceil(k/2)th smallest element in the array.This can be computed using Lomuto Partitioning and QuickSelect Algorithm.

To get the idea behind the Lomuto partitioning,it is helpful to think of an array—or, more generally, a subarray *A*[*l..r*] *(*0 ≤ *l* ≤ *r* ≤ *n* − 1*)*—under consideration as composed of three contiguous segments.Listed in the order they follow pivot *p*, they are as follows: a segment with

elements known to be smaller than *p,* the segment of elements known to be greater than or equal to *p,* and the segment of elements yet to be compared to *p.*. Note that the segments can be empty; for example, it is always the case for the first two segments before the algorithm starts.

**ALGORITHM** *LomutoPartition(A*[*l..r*]*)*

//Partitions subarray by Lomuto’s algorithm using first element as pivot

//Input: A subarray *A*[*l..r*] of array *A*[0*..n* − 1]*,* defined by its left and right

// indices *l* and *r (l* ≤ *r)*

//Output: Partition of *A*[*l..r*] and the new position of the pivot

*p*←*A*[*l*]

*s* ←*l*

**for** *i* ←*l* + 1 **to** *r* **do**

**if** *A*[*i*]*<p*

*s* ←*s* + 1; swap*(A*[*s*]*, A*[*i*]*)*

swap*(A*[*l*]*, A*[*s*]*)*

**return** *s*

Let us assume that the list is implemented as an array whose elements are indexed starting with a 0, and let *s* be the partition’s split position, i.e., the index of the array’s element occupied by the pivot after partitioning. If *s* = *k* − 1*,* pivot *p* itself is obviously the *k*th smallest element, which solves the problem. If *s > k* − 1*,* the *k*th smallest element in the entire array can be found as the *k*th smallest element in the left part of the partitioned array. And if *s < k* − 1*,* it can be found as the *(k* − *s)*th smallest element in its right part. Thus, if we do not solve the problem outright, we reduce its instance to a smaller one, which can be solved by the same approach, i.e., recursively. This algorithm is called ***quickselect***.

**ALGORITHM** *Quickselect(A*[*l..r*]*, k)*

//Solves the selection problem by recursive partition-based algorithm

//Input: Subarray *A*[*l..r*] of array *A*[0*..n* − 1] of orderable elements and

// integer *k (*1≤ *k* ≤ *r* − *l* + 1*)*

//Output: The value of the *k*th smallest element in *A*[*l..r*]

*s* ←*LomutoPartition(A*[*l..r*]*)* //or another partition algorithm

**if** *s* = l + *k* − 1 **return** *A*[*s*]

**if** *s > l* + *k* − 1 **return** *Quickselect(A*[*l..s* − 1]*, k)*

**return** *Quickselect(A*[*s* + 1*..r*]*, k* − 1− *s)*

The following C code :

int LomutoPartition(double \*A,int l,int r)

{

double p=A[l],t;

int s=l;

for(int i=l+1;i<=r;i++)

{

if(A[i]<p)

{

s += 1;

t = A[s];

A[s] = A[i];

A[i] = t;

}

}

t = A[l];

A[l] = A[s];

A[s] = t;

return s;

}

double QuickSelect(double \*A,int l,int r,int k)

{

int s = LomutoPartition(A,l,r);

if(s == l+k-1)

return A[s];

if(s > l+k-1)

return QuickSelect(A,l,s-1,k);

return QuickSelect(A,s+1,r,k-s-1);

}

double median(double \*A,int n)

{

if(n%2 != 0)

return QuickSelect(A,0,n-1,n/2 + 1);

double B[n];

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

B[i]=A[i];

return (QuickSelect(A,0,n-1,n/2) + QuickSelect(B,0,n-1,n/2 + 1))/2;

}

**To compute mean:**

Mean is computed as the sum of all the elements in the array divided by the count of the elements.The efficiency will be O(n).

**ALGORITHM** Mean(*A*[0*..n* – 1])

//Input:An array of elements

//Output:The mean of the input array

sum <- 0

for i <-0 to n - 1 do

sum <- sum + A[i]

return sum/n

The following C code is:

double mean(double \*A,int n)

{

int sum = 0;

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

sum+=A[i];

return sum/n;

}

**To compute harmonic mean:**

*Harmonic mean is used when average of rates is required, below is the formula.Harmonic mean of n numbers x1, x2, ,x3, . . ., xn can written as below.Harmonic mean = n****/****((1/x1) + (1/x2) + (1/x3) + . . . + (1/xn)).The efficiency of this algorithm is O(n).*

**ALGORITHM** Mean(*A*[0*..n* – 1])

//Input:An array of elements .

//Output:The harmonic mean of the input array.

sum <- 0

for i<-0 to n – 1 do

sum <- sum + (1/A[i])

return n/sum;

*The following C code is :*

double hm(double \*arr, int n)

{

double sum = 0;

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

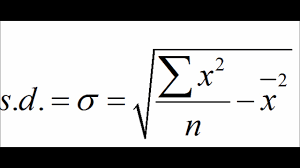
sum = sum + (double)1 / arr[i];

return (double)n/sum;

}

**To compute standard deviation:**

The standard deviation is calculated using the following formula



The efficiency of this algorithm O(n) ,because this algorithm involves calculating sum of squares of all numbers which is in O(n) and then finding mean which is O(n).

**ALGORITHM** Standard\_deviation(*A*[0*..n* – 1])

sumofsqr <- 0

for i<-0 to n-1 do

sumofsqr <- sumofsqr + A[i]\*A[i]

m <- Mean(A)

return

The following C code is:

double stddev(double \*A,int n)

{

double sumofsqr = 0;

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

sumofsqr+=(A[i]\*A[i]);

double m = mean(A,n);

return sqrt(sumofsqr/(double)n - m\*m);

}

**To compute least square line or plane fit:**

Here the input will be in the form of matrices where it signifies the set of equations of the form y = C + Dx for 2-D plane or z = C + Dy + Ex for 3-D plane and so on.The matrices will always have the order of the form ensuring the vacant entries in the matrices are filled with 0’s because there involves matrix multiplication using Strassen’s algorithm whose efficiency is of the order O(n2.8).In this calculation,it also involves finding inverse of a matrix which is given by

Inverse(Matrix) = Adjoint(Matrix)/Determinant(Matrix)

Adjoint is computed by placing cofactors of the elements in the place of the element itself in the matrix and thereby taking transpose of it.The efficiency of computing the inverse is O(n2).Hence the overall efficiency of the algorithm will be O(n2.8).

Here is the Strassens algorithm:

**ALGORITHM** Strassen(A[1...n,1...n], B[1...n,1...n], C[1...n,1...n], m, n)

if m<-2 do

P <- (A[0][0]+A[1][1])\*(B[0][0]+B[1][1])

Q <- (A[1][0]+A[1][1])\*B[0][0]

R <- A[0][0]\*(B[0][1]-B[1][1])

S <- A[1][1]\*(B[1][0]-B[0][0])

T <- (A[0][0]+A[0][1])\*B[1][1]

U <- (A[1][0]-A[0][0])\*(B[0][0]+B[0][1])

V <- (A[0][1]-A[1][1])\*(B[1][0]+B[1][1])

C[0][0]=P+S-T+V

C[0][1]=R+T

C[1][0]=Q+S

C[1][1]=P+R-Q+U

else

m <- m/2

Strassen(&A[0][0],&B[0][0],&C[0][0],m,n)

Strassen(&A[0][0],&B[0][m-1],&C[0][m-1],m,n)

Strassen(&A[0][m-1],&B[m-1][0],&C[0][0],m,n)

Strassen(&A[0][m-1],&B[m-1][m-1],&C[0][m-1],m,n)

Strassen(&A[m-1][0],&B[0][0],&C[0][m-1],m,n)

Strassen(&A[m-1][0],&B[0][m-1],&C[m-1][m-1],m,n)

Strassen(&A[m-1][m-1],&B[m-1][0],&C[m-1][0],m,n)

Strassen(&A[m-1][m-1],&B[m-1][m-1],&C[m-1][m-1],m,n)

The following is the C code for computing least square fit:

void getCofactor(float \*\*A, float \*\*temp, int p, int q, int n)

{

int i = 0, j = 0;

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

{

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

{

if (row != p && col != q)

{

temp[i][j++] = A[row][col];

if (j == n - 1)

{

j = 0;

i++;

}

}

}

}

}

float determinant(float \*\*A, int n,int m)

{

float D = 1,t,B[n][n];

if (n == 1)

return A[0][0];

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

{

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

B[i][j] = A[i][j];

}

int pivotrow,i,j,k,sgn=1;

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

{

pivotrow = i;

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

{

if(abs(B[j][i]) > abs(B[pivotrow][i]))

pivotrow = j;

}

if(pivotrow != i)

{

sgn = -sgn;

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

{

t = B[i][k];

B[i][k] = B[pivotrow][k];

B[pivotrow][k] = t;

}

}

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

{

t = B[j][i]/B[i][i];

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

B[j][k] -= B[i][k]\*t;

}

}

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

D \*= B[i][i];

D\*=sgn;

return D;

}

void adjoint(float \*\*A,float \*\*adj,int n)

{

if (n == 1)

{

adj[0][0] = 1;

return;

}

int sign = 1;

float \*\*temp = (float \*\*)malloc(sizeof(float \*)\*n);

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

temp[i] = (float \*)malloc(sizeof(float)\*n);

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

{

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

{

getCofactor(A, temp, i, j, n);

sign = ((i+j)%2==0)? 1: -1;

adj[j][i] = (sign)\*(determinant(temp,n,n-1));

}

}

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

free(temp[i]);

free(temp);

}

void inverse(float \*\*A, float \*\*inv,int n)

{

float det = determinant(A,n,n);

float \*\*adj = (float \*\*)malloc(sizeof(float \*)\*n);

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

adj[i] = (float \*)malloc(sizeof(float)\*n);

adjoint(A,adj,n);

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

{

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

inv[i][j] = adj[i][j]/det;

}

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

free(adj[i]);

free(adj);

}

void strassen(float \*A,float \*B,float \*C, int m, int n)

{

if(m==2)

{

float P=(\*A+\*(A+n+1))\*(\*B+\*(B+n+1));

float Q=(\*(A+n)+\*(A+n+1))\*(\*B);

float R=(\*A)\*(\*(B+1)-\*(B+n+1));

float S=(\*(A+n+1))\*(\*(B+n)-\*B);

float T=(\*A+\*(A+1))\*(\*(B+n+1));

float U=(\*(A+n)-\*A)\*(\*B+\*(B+1));

float V=(\*(A+1)-\*(A+n+1))\*(\*(B+n)+\*(B+n+1));

\*C=\*C+P+S-T+V;

\*(C+1)=\*(C+1)+R+T;

\*(C+n)=\*(C+n)+Q+S;

\*(C+n+1)=\*(C+n+1)+P+R-Q+U;

}

else

{

m=m/2;

strassen(A,B,C,m,n);

strassen(A,B+m,C+m,m,n);

strassen(A+m,B+m\*n,C,m,n);

strassen(A+m,B+m\*(n+1),C+m,m,n);

strassen(A+m\*n,B,C+m\*n,m,n);

strassen(A+m\*n,B+m,C+m\*(n+1),m,n);

strassen(A+m\*(n+1),B+m\*n,C+m\*n,m,n);

strassen(A+m\*(n+1),B+m\*(n+1),C+m\*(n+1),m,n);

}

}

void transpose(float \*\*A,float \*\*T,int n)

{

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

{

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

{

T[j][i] = A[i][j];

}

}

}

void two\_to\_one(float \*\*A,float \*b,int n)

{

int k=0;

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

{

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

b[k++] = A[i][j];

}

}

void one\_to\_two(float \*b,float \*\*A,int n)

{

int k=0;

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

{

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

A[i][j] = b[k++];

}

}

void leastsqrfit(float \*\*A,float \*\*B,int n,int m)

{

int i,j;

float \*\*At = (float\*\*)malloc(sizeof(float\*)\*n);

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

At[i] = (float\*)malloc(sizeof(float)\*n);

float \*at = (float\*)malloc(sizeof(float)\*(n\*n));

float \*a = (float\*)malloc(sizeof(float)\*(n\*n));

float \*b = (float\*)malloc(sizeof(float)\*(n\*n));

float \*t1 = (float\*)malloc(sizeof(float)\*(n\*n));

float \*t2 = (float \*)malloc(sizeof(float)\*(n\*n));

float \*t3 = (float \*)malloc(sizeof(float)\*(n\*n));

float \*\*T1 = (float \*\*)malloc(sizeof(float \*)\*n);

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

T1[i] = (float \*)malloc(sizeof(float)\*n);

float \*\*T3 = (float \*\*)malloc(sizeof(float \*)\*n);

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

T3[i] = (float \*)malloc(sizeof(float)\*n);

float \*inv = (float \*)malloc(sizeof(float)\*(n\*n));

transpose(A,At,n);

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

{

t1[i]=0;

t2[i]=0;

t3[i]=0;

}

two\_to\_one(At,at,n);

two\_to\_one(A,a,n);

strassen(at,a,t1,n,n);

one\_to\_two(t1,T1,n);

inverse(T1,T1,m);

two\_to\_one(T1,inv,n);

two\_to\_one(B,b,n);

strassen(at,b,t2,n,n);

strassen(inv,t2,t3,n,n);

one\_to\_two(t3,T3,n);

if(m==2)

printf("y = %f x + %f\n\n",T3[0][0],T3[1][0]);

else

printf("z = %f x + %f y + %f\n\n",T3[0][0],T3[1][0],T3[2][0]);

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

free(At[i]);

free(At);

free(a);

free(b);

free(t1);

free(t2);

free(t3);

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

free(T1[i]);

free(T1);

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

free(T3[i]);

free(T3);

free(inv);

}