PES University , Bengaluru.

DESIGN AND ANALYSIS OF ALGORITHMS PROJECT REPORT

Title:

Mathematics, Statistics, Vector Library

To:

Professor Shruti Kaivalya

From:

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
modefrequency ←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
```

```
ALGORITHM Mergesort(A[0..n − 1])
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0.._n/2_ − 1] to B[0.._n/2_ − 1]
    copy A[_n/2_..n − 1] to C[0.._n/2_ − 1]
    Mergesort(B[0.._n/2_ − 1])
    Mergesort(C[0.._n/2_ − 1])
    Merge(B, C, A)
```

```
ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ←0; j ←0; k←0
while i <p and j <q do
    if B[i] ≤ C[j ]
        A[k]←B[i]; i ←i + 1
else A[k]←C[j ]; j ←j + 1
    k←k + 1
```

```
if $i = p$
        copy $C[j..q - 1]$ to $A[k..p + q - 1]$
else copy $B[i..p - 1]$ to $A[k..p + q - 1]$
```

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(n^2)).The following C code is:

To compute median:

In this case,we use again the presorting idea just as computing mode in the previous section.It first sorts the array using a sorting technique with nlogn efficiency,and then if the number of elements in the array is odd,then the median is calculated as

$$\text{Median} = (n + 1)/2^{th} \text{ element}$$

If the number of elements is even, then the median is calculated as

$$\text{Median} = \text{Average}(n/2^{th} \text{ element} + (n/2 + 1)^{th} \text{ element})$$

```
ALGORITHM  PresortMedian($A[0..n - 1]$)
//Input:An array of elements
//Output:The median of the input array
sort the array
if n is even then
        return avg ($n/2^{th}$ element +$(n/2 + 1)^{th}$ element)
return $(n + 1)/2^{th}$ element  //end
```

The efficiency of this algorithm is t(n) € O(nlogn) + O(1) = O(nlogn).

The following C code is:

```
double median(double *A,int n) {
        MergeSort(A,n);
        if(n%2 == 1)
                return A[(n+1)/2];
        return (A[n/2]+A[(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    //end
```

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 x_1, x_2, ,x_3, . . ., x_n can written as below.Harmonic mean = n **/** ((1/x_1) + (1/x_2) + (1/x_3) + . . . + (1/x_n)).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

$$s.d. = \sigma = \sqrt{\frac{\sum x^2}{n} - \overline{x}^2}$$

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 $\sqrt{\frac{\text{sumofsqr}}{n} - m^2}$
```

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.The matrices will always have the order of the form 2^n 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 $O(n^{2.8})$.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 $O(n^2)$.Hence the overall efficiency of the algorithm will be $O(n^{2.8})$.

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 = 0;
    if (m == 1)
        return A[0][0];
    float **temp = (float **)malloc(sizeof(float *)*n);
    for(int i=0;i<n;i++)
      temp[i] = (float *)malloc(sizeof(float)*n);
    int sign = 1;
    for (int f = 0; f < m; f++)
    {
        getCofactor(A, temp, 0, f, m);
        D += sign * A[0][f] * determinant(temp,n,m - 1);
        sign = -sign;
    }
    for(int i=0;i<n;i++)
      free(temp[i]);
    free(temp);
    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);
}
```

VECTOR LIBRARY:

For all the vector functions that we have written, we tried to prove that we can get it in linear time.

To compute Dot Product:

Dot product is given by the product of norm of the two vectors and the cosine of the angle between them. The formula is

$$\boxed{\overline{a}\cdot\overline{b} = |\overline{a}||\overline{b}|\cos\theta}$$

```
ALGORITHM _DOTPRODUCT (A[0…n-1], B[0…n-1],theta)
    C <- 0   D <-0
    pi <- 3.142
    for i <- 0 to length(A) do
        C = C + A[i] * A[i]
        D = D + B[i] * B[i]
    C = sqrt(C)
    D = sqrt(D)
    radian = (pi * theta) / 180
    return  C * D * cos(radian)
```

Running time : O(n)

Similarly, we can compute the angle between them in Linear time if dot product is known.

To compute Dot product if components along axis is known:

```
ALGORITHM _DOTCOMPONENTS (A[0….n-1],B[0….n-1]):
        D = 0
        for i <- 0 to length(A) do
                D = D+ A[i] * B[i]
         return D
```

Running time : O(n)

To compute the projection of A on B:

The formula is:

$$(a\cos\theta)\frac{\overline{b}}{b} = \left(\overline{a}\cdot\frac{\overline{b}}{b}\right)\frac{\overline{b}}{b} = \left(\frac{\overline{a}\cdot\overline{b}}{b^2}\right)\overline{b}$$

```
ALGORITHM PROJECT_A_ON_B (A[0….n-1],B[0….n-1]):
        dotproduct  = dotcomponents(A,B)
        for i <- 0 to length(A) do
                D = D+ B[i] * B[i]
         D = sqrt(D)
         return (dotproduct/D)
```

Running time : O(n)

Similarly, we can compute projection of B on A in Linear time.

To compute Cross Product:

Cross product is given by the product of norm of the two vectors and the sine of the angle between them along unit vector in the direction of $\overline{a}\times\overline{b}$. The formula is

$$\boxed{\overline{c} = \overline{a}\times\overline{b} = |\overline{a}||\overline{b}|\sin\theta\,\overline{u}}, \quad 0 \le \theta \le \pi$$

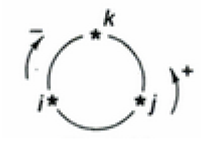
```
ALGORITHM _CROSSPRODUCT (A[0…n-1], B[0…n-1],theta)
        C <- 0   D <-0
        pi <- 3.142
        for i <- 0 to length(A) do
                C = C + A[i] * A[i]
                D = D + B[i] * B[i]
        C = sqrt(C)
        D = sqrt(D)
        radian = (pi * theta) / 180
        return  C * D * sin(radian)
```

Running time : O(n)

To compute Cross Product along unit vectors i, j, k:

The Cross product of two vectors along i, j, k is:

$$\begin{aligned}\bar{a} \times \bar{b} &= (a_1\bar{i} + a_2\bar{j} + a_3\bar{k}) \times (b_1\bar{i} + b_2\bar{j} + b_3\bar{k}) \\ &= (a_2b_3 - a_3b_2)\bar{i} + (a_3b_1 - a_1b_3)\bar{j} + (a_1b_2 - a_2b_1)\bar{k} \\ &= \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}\end{aligned}$$



Python code:

```
def crossProduct3(A, B, cross_p3):
   cross_p3.append(float(A[1]) * float(B[2]) - float(A[2]) * float(B[1]))
   cross_p3.append(float(A[0]) * float(B[2]) - float(A[2]) * float(B[0]))
   cross_p3.append(float(A[0]) * float(B[1]) - float(A[1]) * float(B[0]))
   return cross_p3
cross_p3 = []
crossProduct3(A, B, cross_p3)    #end of code
```

Constant running time: O(1)

To compute area of Parallelogram given two vectors a and b:

The area is given by the norm of $\overline{a} \times \overline{b}$.

Python code:

```
def areaOfParallelogram(A,B):
    if(len(A) == 3):
        cross_p3 = []
        x = crossProduct3(A,B,cross_p3)
        print("The area of parallelogram is : ",end = '')
        C = 0
        for i in range(len(x)):
            C = C + (x[i] * x[i])
        print(math.sqrt(C))
```

Running time: O(n)

To compute Vector Triple Product of Three Vectors:

The vector triple product is given by:

$$\boxed{\overline{a} \times (\overline{b} \times \overline{c}) = (\overline{a} \cdot c)\overline{b} - (\overline{a} \cdot \overline{b})\overline{c}} \qquad (\overline{a} \times \overline{b}) \times \overline{c} = (\overline{a} \cdot \overline{c})\overline{b} - (\overline{b} \cdot \overline{c})\overline{a}$$

Python code:

```
def vectortripleproduct1(A,B,C):
    ac = dotcomponents(A,C)
    ab = dotcomponents(A,B)
    answer = (ac * np.array(B,dtype = 'float')) - (ab * np.array(C,dtype = 'float'))
    return answer
```

Running time: O(n)

To compute the Volume of a Parallelepiped:

The volume is given by:

$$abs\left(\bar{a}\cdot(\bar{b}\times\bar{c})\right)$$

Python code:

```
def volumeofparallelepiped(A,B,edgeC):
    cross_p3 = []
    x = crossProduct3(B,edgeC,cross_p3)
    A[1] = -1 * float(A[1])
    answer = dotcomponents(A,x)
    print("The area of parallelepiped is : ",end = '')
    print(abs(answer))
```

Running time: O(n)

Thus we saw that most of the vector operations have linear running time.

Statistical Distributions:

Binomial Distribution:

The **binomial distribution** with parameters n and p is the discrete probability **distribution** of the number of successes in a sequence of n independent experiments, each asking a yes–no question, and each with its own boolean-valued outcome.

Probability of k out of n ways:

$$P(k \text{ out of } n) = \frac{n!}{k!(n-k)!}\, p^k(1-p)^{(n-k)}$$

```
def factorial(n):
  if(n == 1 or n == 0):
    return 1
  else:
    return n * factorial(n-1)
#finds factorial in O(n)
```

```
def fastpower(x,y):
   if(y == 0):
      return 1
   power  = fastpower(x,math.floor(y/2))
   power = power * power
   if(y%2 == 1):
      power = power * x
   return power #Finds power of number x y in O(log(n))   where y => 0
```

```
q = 1-p
probabilities= {}
factn = factorial(n)
for x in range(0,n+1):
   factx = factorial(x)
   factnx = factorial(n-x)
   i = (factn/(factx * factnx))*(fastpower(p,x)) * (fastpower(q,(n-x)))
   i  = f'{i:.10f}'
   probabilities[x] = i
```

Analysis of Fibonacci algorithms:

Dynamic Programming (slow):

If we have computed F(k-2) and F(k-1) then we can add them to compute F(k). This algorithm takes Θ(1) space and Θ(*n*) operations.

Fast Doubling (really fast):

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}.$$

Given $F(k)$ and $F(k+1)$, we can calculate these:

$F(2k) = =F(k)\ [2F(k+1) - F(k)]$

$F(2k+1) = F(k+1)^2 + F(k)^2$

These identities are extracted from the matrix exponentiation algorithm.

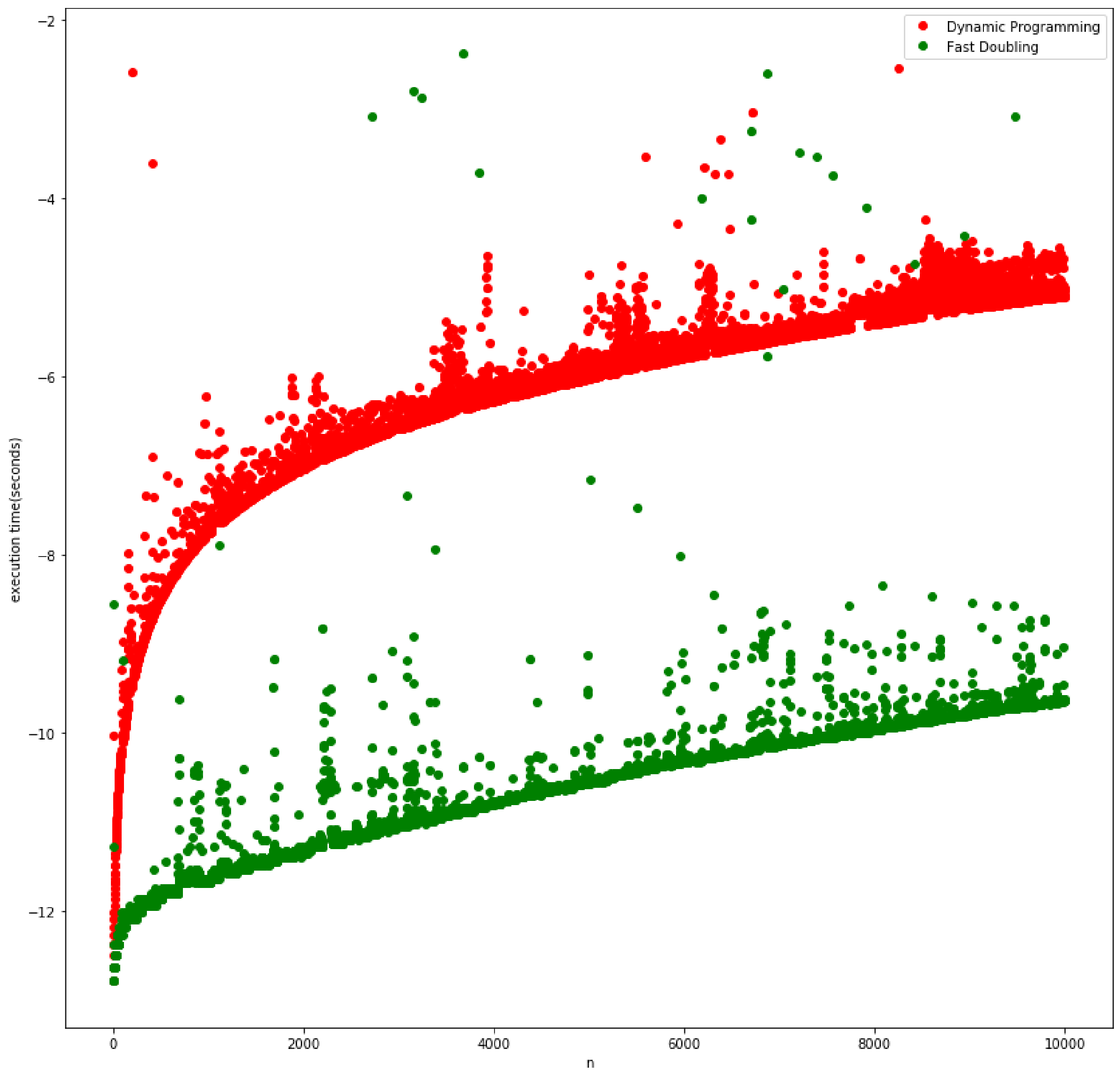
This algorithm has asymptotic complexity of $\Theta(\log(n))$.

Dynamic Programming:

```
a = [0,1]
for i in range(2,n):
    x  = a[i-1]+a[i-2]
    a.append(x)
```

Fast Doubling:

```
def fib(n):
  if n == 0:
    return (0, 1)
  else:
    a, b = fib(n // 2)
    c = a * (b * 2 - a)
    d = a*a + b*b
    if n % 2 == 0:
      return (c, d)
    else:
      return (d, c + d)
```



x-axis: number of Fibonacci numbers (till 10000)

y-axis: logarithm of time taken by each algorithm (time was calculated in seconds)

red label = time taken using Dynamic programming algorithm

green label = time take using Fast doubling algorithm.

An Excel sheet is attached with time taken to generate Fibonacci numbers till a billion, which was executed on Google Colab. (DP was executed on Jupyter notebook on local, Fast Doubling on Colab)

Matrix Library:

Normal Equation:

Normal Equation is an analytical approach to Linear Regression with a Least Square Cost Function. We can directly find out the value of θ without using Gradient Descent. Following this approach is an effective and a time-saving option when are working with a dataset with small features.

Normal equation

$$\theta = (X^T X)^{-1} X^T y$$

Implemented in normalEqnVsGradientDescent.ipynb(CODE)

Matrix multiplication has $O(n^3)$ complexity. So not desirable when dataset is huge.

An alternate iterative solution is finding the parameters using Gradient Descent which is greedy and has $O(kn^2)$ complexity.

The dataset is taken from Professor Andrew Ng's Machine Learning Course (COURSERA).

Some of the Linear Algebra / Matrix Functions that are implemented in Octave:

1. Gaussian Elimination
2. LU Decomposition of a Matrix
3. Finding inverse of Matrix using Gauss-Jordan Method
4. Finding Eigen Values and Eigen Vector of a matrix
5. Finding largest Eigen Value using Rayleigh's Power Method.