Department of Computer Science and Engineering

ASSIGNMENT SUBMISSION

Course Code : CSE-3608
Course Title : Numerical Methods Sessional

Assignment No : 01 & 02

Submitted By:

Name : Nayeem Mahmood
Matric No : C161026
Section : 6AM
Semester : 6th

Submitted To:

Mohammed Shamsul Alam
Asst. Professor,
Dept. of CSE, IIUC

Date of Submission : Feb 4, 2019

*** No copy has been done**

Problem

1-1: Write a program to count number of significant figures in a given number.

Theory

The digits that are used to express a number are called significant digits (or figures). A significant digit/figure is one of the digits 1, 2, ... , 9 and 0 is a significant figure except when it is used to fix the decimal point or to fill the places of unknown or disorder digits.

Algorithm

1. All non-zero digits are significant.
2. All zeros occurring between non-zero digits are significant digits.
3. Trailing zeros following a decimal point are significant. For example, 3.50, 65.0 and 0.230 have three significant digits each.
4. Zeros between the decimal point and preceding a non-zero digit are non-significant. For example, the following numbers have four significant digits: 0.0001234, 0.001234, 0.01234

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    string number, temp;
    int i, _count;
    bool decimalPoint;

    while (true) {
        cout << "Enter a number: ";
        cin >> number;

        if (!number.size()) {
            break;
        }

        temp = number;

        // removing all preceding 0s
        while (temp[0] == '0') {
            temp.erase(0, 1);
        }

        // looking for decimal point
        decimalPoint = false;
        for (i = 0; i < temp.size(); i++) {
            if (temp[i] == '.') {
                decimalPoint = true;
                break;
            }
        }

        if (!decimalPoint) {
            // if decimal point is not found,
            // remove all trailing 0s
            i = temp.size() - 1;
            while (temp[i] == '0') {
                temp.erase(i, 1);
                i--;
            }
            _count = temp.size();
        }
        else {
            if (temp[0] == '.') {
                // if there are no non-zero digits to the left of the
decimal point,
                // then remove all 0s between decimal point and the
next non-zero digit
                while (temp[1] == '0') {
                    temp.erase(1, 1);
                }
            }

            _count = temp.size() - 1; // decimal point is not a digit
        }

        cout << number << " has " << _count << " significant
digits.\n";
        number = "";
    }

    return 0;
}

```

Sample I/O

Enter a number: 120
120 has 2 significant digits.

Enter a number: 3.50
3.50 has 3 significant digits.

Enter a number: 65.0
65.0 has 3 significant digits.

Enter a number: 0.230
0.230 has 3 significant digits.

Enter a number: 0.001234
0.001234 has 4 significant digits.

Enter a number: 7.56
7.56 has 3 significant digits.

Problem

1-3: Write a program to evaluate a polynomial by using Horner's rule.

Theory

We can write the polynomial *f* (x) using *Horner's rule* (also known as *nested multiplication*) as follows:

$$f(x) = ((\ldots\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots\ldots\ldots\ldots + a_1)x + a_0)$$

Here the innermost expression $a_n x + a_{n-1}$ is evaluated first. The resulting value constitutes a multiplicand for the expression at the next level. The number of level equals n, the degree of polynomial. This approach needs a total of n additions and n multiplications.

Algorithm

$p_n = a_n$

$p_{n-1} = p_{n-x} + a_{n-1}$

. . .

. . .

. . .

$p_j = p_{j+1} x + a_j$

. . .

. . .

$p_1 = p_2 x + a_1$

$f(x) = p_0 = p_1 x + a_0$

```cpp
#include <bits/stdc++.h>

using namespace std;

#define MAX 100

int allA[MAX];
int allP[MAX];

int main()
{
    int n, i, x, a;

    while (true) {
        cout << "Degree of polynomial: ";
        cin >> n;

        if (!n) {
            break;
        }

        cout << "Enter the co-efficients: ";
        for (i = n; i >= 0; i--) {
            cin >> allA[i];
        }

        cout << "Enter value for x: ";
        cin >> x;

        allP[n] = allA[n];
        for (i = n - 1; i >= 0; i--) {
            allP[i] = allP[i + 1] * x + allA[i];
        }

        cout << "f(" << x << ") = " << allP[0] << "\n";
    }

    return 0;
}
```

Sample I/O

Degree of polynomial: 3
Enter the co-efficients: 1 -4 1 6
Enter value for x: 2
f(2) = 0

Degree of polynomial: 3
Enter the co-efficients: 1 -2 5 10
Enter value for x: 3
f(3) = 34

Problem

1-4: Write a program to find the root of the equation $x^3 - 9x + 1 = 0$, correct to 3 decimal places by using the Bisection method.

Theory

If f(x) is a continuous function is some interval [a, b] and f(a) and f(b) are of opposite signs, then the equation f(x) = 0 has at least one real root or an odd number of real roots in (a, b).

Algorithm

1. Decide initial values for x1 and x2 and stopping criterion E.
2. Compute f1 = f (x1) and f2 = f (x2).
3. If f1 * f2 > 0, x1 and x2 do not bracket any root and go to step 1.
4. Compute x0 = (x1 + x2) / 2 and compute f0 = f (x0).
5. If f1 * f0 < 0 then set x2 = x0 else set x1 = x0.
6. If absolute value of (x2-x1) is less then E, then root = (x1 + x2) / 2 and go to step 7
   Else go to step 4
7. Stop.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define EPS 1e-4
#define MAX 100

int degree;
int allA[MAX];

double fX(double x)
{
    double ans = 0;
    for (int i = degree; i >= 0; i--) {
        ans += allA[i] * pow(x, i);
    }
    return ans;
}

int main()
{
    int i;
    double x, x1, x2;

    while (true) {
        cout << "Degree of polynomial equation: ";
        cin >> degree;

        if (!degree) {
            break;
        }

        cout << "Enter the co-efficients: ";
        for (i = degree; i >= 0; i--) {
            cin >> allA[i];
        }

        do {
            cout << "Initial guess for x1: ";
            cin >> x1;
            cout << "Initial guess for x2: ";
            cin >> x2;
        } while (fX(x1) * fX(x2) > 0.0);

        while (fabs(x2 - x1) > EPS) {
            x = (x1 + x2) / 2.0;

            if (fX(x) * fX(x1) < 0.0) {
                x2 = x;
            }
            else {
                x1 = x;
            }
        }

        cout << "root = " << setprecision(3) << fixed << x1 << "\n";
    }

    return 0;
}
```

Sample I/O

Degree of polynomial equation: 3
Enter the co-efficients: 1 0 -1 -1
Initial guess for x1: 1
Initial guess for x2: 2
root = 1.325

Degree of polynomial equation: 3
Enter the co-efficients: 1 0 -9 1
Initial guess for x1: 1
Initial guess for x2: 3
root = 2.943

Problem

1-6: Write a program to find the root of the equation $x^4 - 12x + 7 = 0$, correct to 3 decimal places by using the Newton-Raphson method.

Theory

Let x_0 be an approximate root of $f(x) = 0$ and let $x_1 = x_0 + h$ be the correct root so that $f(x_1) = 0$.

Expanding $f(x_0 + h)$ by Taylor's series, we obtain

$$f(x_0) + h f'(x_0) + h^2/2!\ f''(x_0) + \ldots = 0$$

Neglecting the second and higher order derivatives, we have

$$f(x_0) + h f'(x_0) = 0$$

which gives $h = - f(x_0) / f'(x_0)$

A better approximation than x0 is therefore given by x1, where

$$x_1 = x_0 - f(x_0) / f'(x_0)$$

Successive approximations are given by $x_2, x_3, x_4, \ldots, x_{n+1}$, where

$$X_{n+1} = x_n - f(x_n) / f'(x_n)$$

which is called the *Newton-Raphson formula*.

The process will be continued till the absolute difference between two successive approximations is less than the specified error E i,e, $| x_{n+1} - x_n | < E$.

Algorithm

1. Assign an initial value for x, say x_0 and stopping criterion E.
2. Compute $f(x_0)$ and $f'(x_0)$.
3. Find the improved estimate of x_0
4. $x_1 = x_0 - f(x_0) / f'(x_0)$
5. Check for accuracy of the latest estimate.
6. If $| x_1 - x_0 | < E$ then stop; otherwise continue.
7. Replace x_0 by x_1 and repeat steps 3 and 4.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define EPS 1e-4
#define MAX 100

int degree;
int allA[MAX];
int allD[MAX]; // derivative

double fX(double x)
{
    double ans;
    for (int i = degree; i >= 0; i--) {
        ans += allA[i] * pow(x, i);
    }
    return ans;
}

double dX(double x)
{
    double ans;
    for (int i = degree - 1; i >= 0; i--) {
        ans += allD[i] * pow(x, i);
    }
    return ans;
}
int main()
{
    int i;
    double x, x0;

    while (true) {
        cout << "Degree of polynomial equation: ";
        cin >> degree;
        if (!degree) {
            break;
        }
        cout << "Co-efficients of the equation: ";
        for (i = degree; i >= 0; i--) {
            cin >> allA[i];
        }
        cout << "Co-efficients of the derivative equation: ";
        for (i = degree - 1; i >= 0; i--) {
            cin >> allD[i];
        }
        cout << "Initial guess for x: ";
        cin >> x0;

        while (true) {
            x = x0 - fX(x0) / dX(x0);
            if (fabs(x - x0) < EPS) {
                //cout << "# x  : " << x << "\n# x0  : " << x0 <<
"\n";
                break;
            }
            else {
                //cout << "x: " << x << "\n";
                x0 = x;
            }
        }
        cout << "root: " << setprecision(3) << fixed << x << "\n";
    }
    return 0;
}
```

Sample I/O

Degree of polynomial equation: 4
Co-efficients of the equation: 1 0 0 -12 7
Co-efficients of the derivative equation: 4 0 0 -12
Initial guess for x: 0
root: 0.594

Degree of polynomial equation: 3
Co-efficients of the equation: 1 0 -6 4
Co-efficients of the derivative equation: 3 0 -6
Initial guess for x: 0
root: 2.000

Problem

1-7: Write a program to find the root of the equation $x^3 + x - 1 = 0$, correct to 3 decimal places by using the False Position method.

Theory

The equation of the line joining the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is given by

$$(f(x_2) - f(x_1)) / (x_2 - x_1) = (y - f(x_1)) / (x - x_1)$$

Since the line intersects the x-axis at x_0, when $x = x_0$, $y = 0$, we have,

$$(f(x_2) - f(x_1)) / (x_2 - x_1) = (-f(x_1)) / (x_0 - x_1)$$

or $$x_0 - x_1 = - f(x_1) (x_2 - x_1) / (f(x_2) - f(x_1))$$

Then we have,

$$x_0 = x_1 - (f(x_1) (x_2 - x_1)) / (f(x_2) - f(x_1))$$

This equation is known as the *false position formula*.

Algorithm

1. Decide initial values for x_1 and x_2 and stopping criterion E.
2. Compute $x_0 = x_1 - (f(x_1) (x_2 - x_1)) / (f(x_2) - f(x_1))$
3. If $f(x_0) * f(x_1) < 0$ set $x_2 = x_0$ otherwise set $x_1 = x_0$
4. If the absolute difference of two successive x_0 is less then E, then root = x_0 and stop.
5. Else go to step 2.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define EPS 1e-4
#define MAX 100

int degree;
int allA[MAX];

double fX(double x)
{
    double ans = 0;
    for (int i = degree; i >= 0; i--) {
        ans += allA[i] * pow(x, i);
    }
    return ans;
}

int main()
{
    int i;
    double x, previousX, x1, x2;

    while (true) {

        cout << "Degree of polynomial equation: ";
        cin >> degree;

        if (!degree) {
            break;
        }

        cout << "Enter the co-efficients: ";
        for (i = degree; i >= 0; i--) {
            cin >> allA[i];
        }

        cout << "Initial guess for x1: ";
        cin >> x1;
        cout << "Initial guess for x2: ";
        cin >> x2;

        previousX = DBL_MIN;
        while (true) {
            x = x1 - (fX(x1) * (x2 - x1)) / (fX(x2) - fX(x1));

            if (fabs(x - previousX) < EPS) {
                break;
            }
            else {
                previousX = x;
            }

            if (fX(x) * fX(x1) < 0.0) {
                x2 = x;
            }
            else {
                x1 = x;
            }
        }
        cout << "root = " << setprecision(3) << fixed << x << "\n";
    }
    return 0;
}
```

Sample I/O

Degree of polynomial equation: 3
Enter the co-efficients: 1 0 1 -1
Initial guess for x1: 0
Initial guess for x2: 1
root = 0.682

Degree of polynomial equation: 2
Enter the co-efficients: 1 -1 -2
Initial guess for x1: 1
Initial guess for x2: 3
root = 2.000

Problem

1-8: Write a program to find the root of the equation $x^3 - 9x + 1 = 0$, correct to 3 decimal places by using the Secant method.

Theory

Secant method, like the False Position & Bisection methods, uses two initial estimates but does not require that they must bracket the root.
Let us consider two points x_1 and x_2 as starting values. Slope of the secant line passing through x_1 and x_2 is given by,

$$\frac{f(x_1)}{x_1 - x_3} = \frac{f(x_2)}{x_2 - x_3}$$

$$f(x_1)(x_2 - x_3) = f(x_2)(x_1 - x_3)$$

$$x_3[f(x_2) - f(x_1)] = f(x_2).x_1 - f(x_1).x_2$$

$$\therefore x_3 = \frac{f(x_2).x_1 - f(x_1).x_2}{f(x_2) - f(x_1)}$$

By adding and subtracting $f(x_2).x_2$ to the numerator and remaining the terms we have,

$$x_3 = x_2 - \frac{f(x_2)(x_2 - x_1)}{f(x_2) - f(x_1)}$$

which is called the *secant formula*.

Algorithm

1. Decide two initial points x_1 and x_2 and required accuracy level E.
2. Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$
3. Compute $x_3 = (f_2 x_1 - f_1 x_2) / (f_2 - f_1)$
4. If $|x_3 - x_2| > E$, then
   Set $x_1 = x_2$ and $f_1 = f_2$
   Set $x_2 = x_3$ and $f_2 = f(x_3)$
   Go to step 3
   Else
   Set root = x_3
   Print root
5. Stop.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define EPS 1e-4
#define MAX 100

int degree;
int allA[MAX];

double fX(double x)
{
    double ans = 0;
    for (int i = degree; i >= 0; i--) {
        ans += allA[i] * pow(x, i);
    }
    return ans;
}

int main()
{
    int i;
    double x, x1, x2;

    while (true) {
        cout << "Degree of polynomial equation: ";
        cin >> degree;

        if (!degree) {
            break;
        }

        cout << "Enter the co-efficients: ";
        for (i = degree; i >= 0; i--) {
            cin >> allA[i];
        }

        cout << "Initial guess for x1: ";
        cin >> x1;
        cout << "Initial guess for x2: ";
        cin >> x2;

        while (true) {
            x = x2 - (fX(x2) * (x2 - x1)) / (fX(x2) - fX(x1));

            if (fabs(x - x2) > EPS) {
                x1 = x2;
                x2 = x;
            }
            else {
                break;
            }
        }

        cout << "root = " << setprecision(3) << fixed << x << "\n";
    }

    return 0;
}
```

Sample I/O

Degree of polynomial equation: 3
Enter the co-efficients: 1 -5 0 -29
Initial guess for x1: 0
Initial guess for x2: 3
root = 5.848

Degree of polynomial equation: 3
Enter the co-efficients: 1 0 1 -1
Initial guess for x1: 0
Initial guess for x2: 1
root = 0.682

Degree of polynomial equation: 2
Enter the co-efficients: 1 -1 -2
Initial guess for x1: 1
Initial guess for x2: 3
root = 2.000

Problem

1-9: Write a program to find the Quotient Polynomial q(x) such that p(x) = (x – 2) q(x) where $p(x) = x^3 - 6x^2 + 11x - 6 = 0$ has a root at x = 2.

Theory

A polynomial of degree n can be expressed as

$$p(x) = (x - x_r)\, q(x)$$

where x_r is a root of the polynomial p(x) and q(x) is the *quotient. Polynomial* of degree n-1. Once a root is found, we can use this fact to obtain a lower degree polynomial q(x) by dividing p(x) by $(x - x_r)$ using a process known as *Synthetic division*. The name "Synthetic" is used because the quotient polynomial q(x) is obtained without actually performing the division. The activity of reducing the degree of a polynomial is referred to as *deflation*.

Algorithm

If $p(x) = (x - x_r)\ q(x)$, then

$a_n x^n + a_{n-1} x^{n-1} + \ldots \ldots ..+ a_1 x + a_0 = (x - x_r)\,(b_{n-1} x^{n-1} + b_{n-2} x^{n-2} + \ldots \ldots + b_1 x + b_0)$

By comparing the coefficients of like powers of x on both the sides of equation, we get the following relation between them:

$a_n = b_{n-1}$

$a_{n-1} = b_{n-2} - x_r b_{n-1}$

. . .

. . .

$a_1 = b_0 - x_r b_1$

$a_0 = - x_r b_0$

That is $a_i = b_{i-1} - x_r b_i$ where, $b_n = 0$ and i = n, n-1, ……0

Then $b_{i-1} = a_i + x_r b_i$ where, $b_n = 0$ and i = n, n-1, ……1

The above Equation suggests that we can determine the co-efficient of q(x) [i.e, b_{n-1} , b_{n-2} , …….. , b_0] from the co-efficient of p(x) [i.e a_n , a_{n-1} , …… , a_1] recursively. Thus we have obtained the polynomial q(x) without performing any division operation.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define MAX 100

int allP[MAX];
int allQ[MAX];

int main()
{
    int degree, i, x;

    while (true) {
        cout << "Degree of polynomial: ";
        cin >> degree;

        if (!degree) {
            break;
        }

        cout << "Co-efficients: ";
        for (i = degree; i >= 0; i--) {
            cin >> allP[i];
        }

        cout << "Root, x: ";
        cin >> x;

        allQ[degree] = 0;
        for (i = degree - 1; i >= 0; i--) {
            allQ[i] = allP[i + 1] + x * allQ[i + 1];
        }

        cout << "Co-efficients of q(x): ";
        for (i = degree - 1; i >= 0; i--) {
            cout << allQ[i] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Sample I/O

Degree of polynomial: 3
Co-efficients: 1 -6 11 -6
Root, x: 2
Co-efficients of q(x): 1 -4 3

Degree of polynomial: 3
Co-efficients: 1 -7 15 -9
Root, x: 3
Co-efficients of q(x): 1 -4 3

Problem

2-1: The following values of f(x) are given:

x	1	2	3	4	5
y = f(x)	1	8	27	64	125

Write a program to find difference table for above values.

Theory

If $y_0, y_1, y_2, \ldots \ldots, y_n$ denote a set of values of y, then $y_1 - y_0, y_2 - y_1, \ldots \ldots \ldots, y_n - y_{n-1}$ are called the *differences* of y. Denoting these differences by $\Delta y_0, \Delta y_1, \Delta y_2, \ldots \ldots \ldots, \Delta y_{n-1}$ respectively, we have,

$\Delta y_0 = y_1 - y_0, \quad \Delta y_1 = y_2 - y_1, \quad \ldots\ldots, \Delta y_{n-1} = y_n - y_{n-1}$

General form: $\Delta y_i = y_{i+1} - y_i$, where i = 0, 1, 2, 3, ..., n-1

where Δ is called the *forward difference operator* and $\Delta y_0, \Delta y_1, \Delta y_2, \ldots, \Delta y_{n-1}$ are called *first forward differences*.

The difference of the first forward differences are called *second forward differences* and denoted by $\Delta^2 y_0, \Delta^2 y_1, \Delta^2 y_2, \ldots \ldots$ Similarly, one can define *third forward differences*, *fourth forward differences*, etc.

$\Delta^2 y_0 = \Delta y_1 - \Delta y_0 = (y_2 - y_1) - (y_1 - y_0) = y_2 - 2y_1 + y_0$

$\Delta^3 y_0 = \Delta^2 y_1 - \Delta^2 y_0 = (y_3 - 2y_2 + y_1) - (y_2 - 2y_1 + y_0) = y_3 - 3y_2 + 3y_1 - y_0$

- Any higher order difference can easily be expressed in terms of the ordinates, since the coefficients occurring on the right side are the binomial coefficients.
- $\Delta^r y_i = \Delta^{r-1} y_{i+1} - \Delta^{r-1} y_i$
-

$$\Delta^n y_0 = \sum_{k=0}^{n} (-1)^k c_k y_{n-k}$$

$\Delta^n y_0 = y_n - {}^n c_1 y_{n-1} + {}^n c_2 y_{n-2} - {}^n c_3 y_{n-3} \ \ldots\ldots + (-1)^n y_o$

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, i, j;
    double x, y;
    vector<double>allX;
    vector<vector<double> >allY;

    while (true) {
        cout << "Number of points in the table: ";
        cin >> n;

        if (!n) {
            break;
        }

        allX.assign(n, 0);
        allY.assign(n, vector<double>());
        for (i = 0; i < n; i++) {
            cout << "Enter x" << i << " y" << i << "\n";
            cin >> allX[i] >> y;
            allY[0].push_back(y);
        }

        for (i = 1; i < n; i++) {
            for (j = 1; j < allY[i - 1].size(); j++) {
                y = allY[i - 1][j] - allY[i - 1][j - 1];
                allY[i].push_back(y);
            }
        }

        for (i = 1; i < n; i++) {
            cout << "del^" << i << "y:";
            for (j = 0; j < allY[i].size(); j++) {
                cout << " " << allY[i][j];
            }
            cout << "\n";
        }
    }

    return 0;
}
```

Sample I/O

Number of points in the table: 5
Enter x0 y0: 1 1
Enter x1 y1: 2 8
Enter x2 y2: 3 27
Enter x3 y3: 4 64
Enter x4 y4: 5 125
del^1y: 7 19 37 61
del^2y: 12 18 24
del^3y: 6 6
del^4y: 0

Number of points in the table: 5
Enter x0 y0: 1 11.022
Enter x1 y1: 3 12.023
Enter x2 y2: 5 15.134
Enter x3 y3: 7 16.723
Enter x4 y4: 9 22.442
del^1y: 1.001 3.111 1.589 5.719
del^2y: 2.11 -1.522 4.13
del^3y: -3.632 5.652
del^4y: 9.284

Problem

2-2: The following values of f(x) are given:

x	1	2	3	4	5
y = f(x)	1	8	27	64	125

Write a program to find f(1.7) by using Newton's Forward Interpolation.

Theory

Let $y = f(x)$ be a function which takes the values $y_0, y_1, y_2, \ldots, y_n$ corresponding to the $(n+1)$ values $x_0, x_1, x_2, \ldots, x_n$ of the independent variables. Let the values of x be equally spaced i.e. $x_i = x_0 + ih$ where $i = 0,1,2, \ldots \ldots n$ and h is the interval of differencing. Let $\varphi(x)$ be a polynomial of the n^{th} degree such that y and $\varphi(x)$ agree at the tabulated points i.e. $f(x_i) = \varphi(x_i)$, $i = 0,1,2 \ldots \ldots \ldots n$ and at all other points $f(x) = \varphi(x) + R(x)$ where R(x) is called *error term* of the interpolation formula. Ignoring the error term let us assume

$$f(x) \approx \varphi(x) = a_0 + a_1 (x-x_0) + a_2 (x-x_0) (x-x_1) + \ldots \ldots + a_n (x-x_0) (x-x_1) \ldots \ldots (x-x_{n-1})$$

The constants $a_0, a_1, \ldots \ldots, a_n$ can be determined as follows:

putting $x = x_0$ we get

$$f(x_0) \approx \varphi(x_0) = a_0$$

$$\therefore a_0 = y_0$$

putting $x = x_1$ we get

$$f(x_1) \approx \varphi(x_1) = a_0 + a_1 (x_1 - x_0) = y_0 + a_1 h$$

$$\therefore a_1 = (y_1 - y_0) / h = \Delta y_0 / h$$

putting $x = x_2$ we get

$$f(x_2) \approx \varphi(x_2) = a_0 + a_1 (x_2 - x_0) + a_1 (x_2 - x_0)(x_2 - x_1)$$

$$y_2 = y_0 + \Delta y_0/h(2h) + a_2 (2h)(h)$$

$$y_2 = y_0 + 2(y_1 - y_0) + a_2 (2h^2)$$

$$a_2 = \frac{y_2 - 2y_1 + y_0}{2h^2}$$

$$\therefore \quad a_2 = \frac{\Delta^2 y_0}{2!\, h^2}$$

Similarly by putting $x = x_3$, $x = x_4$, ... $x = x_n$ we get,

$$a_3 = \frac{\Delta^3 y_0}{3!\, h^3}, \quad a_4 = \frac{\Delta^4 y_0}{4!\, h^4}, \ \ldots\ldots, \quad a_n = \frac{\Delta^n y_0}{n!\, h^n}$$

putting the values of $a_0, a_1, \ldots \ldots, a_n$ we get

$$f(x) \approx \varphi(x) = y_0 + (\Delta y_0/h)(x-x_0) + (\Delta^2 y_0 / (2!\, h^2))\,(x-x_0)(x-x_1) + \ldots \ldots$$
$$\ldots \ldots + (\Delta^n y_0/ (n!\, h^n))\,(x-x_0)(x-x_1) \ldots \ldots (x-x_{n-1})$$

writing $u = ((x-x_0) / h)$, we get

$$x-x_0 = uh$$

$$x-x_1 = (x-x_0) + (x_0-x_1) = (x-x_0) - (x_1-x_0) = uh - h = (u-1)h$$

Similarly,

$$x-x_2 = (u-2)h$$

$$x-x_3 = (u-3)h$$

$$\ldots\ldots \ldots \ldots \ldots \ldots$$

$$x-x_{n-1} = (u-n+1)h$$

$$f(x) \approx \varphi(x) = y_0 + u(\Delta y_0/1!) + u(u-1)(\Delta^2 y_0)/ 2! + \ldots \ldots$$
$$\ldots \ldots + u(u-1)(u-2) \ldots \ldots (u-n+1)\,(\Delta^n y_0)/n!$$

The above formula is called *Newton's forward interpolation formula*.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, i, j, fact;
    double x, y, h, u, ans, tempU;
    vector<double>allX;
    vector<vector<double> >allY;

    while (true) {
        cout << "Number of points in the table: ";
        cin >> n;

        if (!n) {
            break;
        }

        allX.assign(n, 0);
        allY.assign(n, vector<double>());
        for (i = 0; i < n; i++) {
            cout << "Enter x" << i << " y" << i << "\n";
            cin >> allX[i] >> y;
            allY[0].push_back(y);
        }

        cout << "Value for x: ";
        cin >> x;

        for (i = 1; i < n; i++) {
            for (j = 1; j < allY[i - 1].size(); j++) {
                y = allY[i - 1][j] - allY[i - 1][j - 1];
                allY[i].push_back(y);
            }
        }

        h = allX[1] - allX[0];
        u = (x - allX[0]) / h;

        ans = allY[0][0];
        for (i = 1, fact = 1; i < n; i++) {
            tempU = u;
            for (j = 1; j < i; j++) {
                tempU *= (u - j);
            }
            fact *= i;
            ans += (tempU * allY[i][0]) / (fact * 1.0);
        }

        cout << "f(" << x << ") = " << ans << "\n";
    }

    return 0;
}
```

Sample I/O

```
Number of points in the table: 5
Enter x0 y0: 1 1
Enter x1 y1: 2 8
Enter x2 y2: 3 27
Enter x3 y3: 4 64
Enter x4 y4: 5 125
Value for x: 1.7
f(1.7) = 13.573

Number of points in the table: 5
Enter x0 y0: 1 11.022
Enter x1 y1: 3 12.023
Enter x2 y2: 5 15.134
Enter x3 y3: 7 16.723
Enter x4 y4: 9 22.442
Value for x: 7.2
f(7.2) = 10.0057
```

Problem

2-3: The following values of f(x) are given:

x	1	2	3	4	5
y = f(x)	1	8	27	64	125

Write a program to find f(4.7) by using Newton's Backward Interpolation.

Theory

Newton forward interpolation formula cannot be used for interpolating a value of y near the end of a table of values. For this purpose, we use another formula known as Newton-Gregory interpolation formula.

Let $y = f(x)$ be a function which takes the values $y_0, y_1, y_2, \ldots, y_n$ corresponding to the $(n+1)$ values $x_0, x_1, x_2, \ldots, x_n$ of the independent variables. Let the values of x be equally spaced i.e. $x_i = x_0 + ih$ where $i = 0,1,2, \ldots \ldots n$ and h is the interval of differencing. Let $\varphi(x)$ be a polynomial of the n^{th} degree such that y and $\varphi(x)$ agree at the tabulated points i.e. $f(x_i) = \varphi(x_i)$, $i = 0,1,2 \ldots \ldots \ldots n$ and at all other points $f(x) = \varphi(x) + R(x)$ where R(x) is called *error term* of the interpolation formula. Ignoring the error term let us assume

$$f(x) \approx \varphi(x) = a_0 + a_1 (x-x_n) + a_2 (x- x_n) (x- x_{n-1}) + \ldots \ldots + a_n(x- x_n)(x- x_{n-1}) \ldots \ldots (x-x_1)$$

putting $x = x_n$ we get

$$f(x_n) \approx \varphi(x_n) = a_0$$

$$\therefore a_0 = y_n$$

putting $x = x_{n-1}$ we get

$$f(x_{n-1}) \approx \varphi(x_{n-1}) = a_0 + a_1(x_{n-1} - x_n) = y_n + a_1(-h)$$

putting $x = x_{n-2}$ we get

$$f(x_{n-2}) \approx \varphi(x_{n-2}) = a_0 + a_1(x_{n-2} - x_n) + a_2 (x_{n-2} - x_n)(x_{n-2} - x_{n-1})$$

$$y_{n-2} = y_n + \nabla y_n/h\ (-2h) + a_2\ (-2h)(-h)$$

$$y_{n-2} = y_n - 2\ y_n + 2\ y_{n-1} + a_2\ (2h^2)$$

$$a_2 = \frac{y_n - 2y_{n-1} + y_{n-2}}{2h^2}$$

$$\therefore a_2 = \frac{\nabla^2 y_n}{2!\ h^2}$$

Similarly,

$$a_3 = \frac{\nabla^3 y_n}{3!\ h^3},\quad a_4 = \frac{\nabla^4 y_n}{4!\ h^4},\quad \ldots,\quad a_n = \frac{\nabla^n y_n}{n!\ h^n}$$

putting the values of $a_0, a_1, \ldots \ldots a_n$ we get

$$f(x) \approx \varphi(x) = y_n + (\nabla y_n/h)(x-x_n) + (\nabla^2 y_n / (2!\, h^2))\,(x-x_n)(x-x_{n-1}) + \ldots \ldots$$
$$\ldots \ldots + (\nabla^n y_n / (n!\, h^n))\,(x-x_n)(x-x_{n-1}) \ldots \ldots (x-x_1)$$

writing $u = ((x-x_n)/h)$, we get

$x-x_n = uh$

$x-x_{n-1} = (x-x_n) + (x_n - x_{n-1}) = (u+1)h$

Similarly, $x-x_{n-2} = (u+2)h$

$x-x_{n-3} = (u+3)h$

.....

$x-x_1 = (u+n-1)h$

$$f(x) \approx \varphi(x) = y_n + u\,(\nabla y_n /1!) + u\,(u+1)\,(\nabla^2 y_n) / 2! + \ldots \ldots$$
$$\ldots \ldots + u\,(u+1)\,(u+2) \ldots \ldots (u+n-1)\,(\nabla^n y_n) / n!$$

The above formula is called *Newton's backward interpolation formula*.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, i, j, fact;
    double x, y, h, u, ans, tempU;
    vector<double>allX;
    vector<vector<double> >allY;

    while (true) {
        cout << "Number of points in the table: ";
        cin >> n;

        if (!n) {
            break;
        }

        allX.assign(n, 0);
        allY.assign(n, vector<double>());
        for (i = 0; i < n; i++) {
            cout << "Enter x" << i << " y" << i << "\n";
            cin >> allX[i] >> y;
            allY[0].push_back(y);
        }

        cout << "Value for x: ";
        cin >> x;

        for (i = 1; i < n; i++) {
            for (j = 1; j < allY[i - 1].size(); j++) {
                y = allY[i - 1][j] - allY[i - 1][j - 1];
                allY[i].push_back(y);
            }
        }

        h = allX[1] - allX[0];
        u = (x - allX[n - 1]) / h;

        ans = allY[0][n - 1];
        for (i = 1, fact = 1; i < n; i++) {
            tempU = u;
            for (j = 1; j < i; j++) {
                tempU *= (u + j);
            }
            fact *= i;
            ans += (tempU * allY[i][allY[i].size() - 1]) / (fact * 1.0
);
        }

        cout << "f(" << x << ") = " << ans << "\n";
    }

    return 0;
}
```

Sample I/O

Number of points in the table: 5
Enter x0 y0: 1 1
Enter x1 y1: 2 8
Enter x2 y2: 3 27
Enter x3 y3: 4 64
Enter x4 y4: 5 125
Value for x: 4.7
f(4.7) = 103.823

Number of points in the table: 6
Enter x0 y0: 1961 20
Enter x1 y1: 1971 27
Enter x2 y2: 1981 39
Enter x3 y3: 1991 52
Enter x4 y4: 2001 70
Enter x5 y5: 2011 90
Value for x: 2006
f(2006) = 80.6211

Problem

2-4: The following values of f(x) are given:

x	1	2	3	4	5
y = f(x)	1	8	27	64	125

Write a program to find value of x for which f(x) = 85 by using Lagrange's Inverse Interpolation formula.

Theory

We know, the Lagrange interpolation formula is

$$y = f(x) = \frac{(x-x_1)(x-x_2)\dots\dots(x-x_n)}{(x_0-x_1)(x_0-x_2)\dots\dots(x_0-x_n)} f(x_0) + \frac{(x-x_0)(x-x_2)\dots\dots(x-x_n)}{(x_1-x_0)(x_1-x_2)\dots\dots(x_1-x_n)} f(x_1)$$

$$+ \dots\dots + \frac{(x-x_0)(x-x_2)\dots\dots(x-x_{n-1})}{(x_n-x_0)(x_n-x_2)\dots\dots(x_n-x_{n-1})} f(x_n)$$

By interchanging x and y we can express x as a function of y as follows:

$$x = \frac{(y-y_1)(y-y_2)\dots\dots(y-y_n)}{(y_0-y_1)(y_0-y_2)\dots\dots(y_0-y_n)} x_0 + \frac{(y-y_0)(y-y_2)\dots\dots(y-y_n)}{(y_1-y_0)(y_1-y_2)\dots\dots(y_1-y_n)} x_1$$

$$+ \dots\dots + \frac{(y-y_0)(y-y_2)\dots\dots(y-y_{n-1})}{(y_n-y_0)(y_n-y_2)\dots\dots(y_n-y_{n-1})} x_n$$

This is called *Lagrange's inverse interpolation formula*.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, i, j;
    double x, y, ans, numerator, denominator;
    vector<double>allX;
    vector<double>allY;

    while (true) {
        cout << "Number of points in the table: ";
        cin >> n;

        if (!n) {
            break;
        }

        allX.assign(n, 0);
        allY.assign(n, 0);
        for (i = 0; i < n; i++) {
            cout << "Enter x" << i << " y" << i << "\n";
            cin >> allX[i] >> allY[i];
        }

        cout << "Value for x: ";
        cin >> x;

        ans = 0.0;
        for (i = 0; i < n; i++) {
            numerator = denominator = 1.0;
            for (j = 0; j < n; j++) {
                if (j == i) {
                    continue;
                }

                numerator *= (x - allX[j]);
                denominator *= (allX[i] - allX[j]);
            }

            ans += (numerator * allY[i]) / denominator;
        }

        cout << "f(" << x << ") = " << ans << "\n";
    }

    return 0;
}
```

Sample I/O

```
Number of points in the table: 5
Enter x0 y0: 1 1
Enter x1 y1: 8 2
Enter x2 y2: 27 3
Enter x3 y3: 64 4
Enter x4 y4: 125 5
Value for x: 85
f(85) = 5.64691

Number of points in the table: 5
Enter x0 y0: 2 10.0123
Enter x1 y1: 4 13.0234
Enter x2 y2: 6 16.045
Enter x3 y3: 7 20.123
Enter x4 y4: 8 25.234
Value for x: 5
f(5) = 13.7645
```

Problem

2-5: The following values of f(x) are given:

x	1	3	4	6	10
y = f(x)	0	18	58	190	920

Write a program to find f(2.7) by using Newton's Divided Difference formula.

Theory

Let $y = f(x)$ be a function which assumes the values $f(x_0)$, $f(x_1)$, $f(x_2)$, …., $f(x_n)$ corresponding to the values $x = x_0, x_1, x_2, \ldots, x_n$ where the values of x are not necessarily equispaced. From the definition of divided difference

$$f[x,x_0] = \frac{f(x) - f(x_0)}{x - x_0}$$

$$f(x) = f(x_0) + (x-x_0) f[x,x_0] \ldots\ldots (1)$$

Again,

$$f[x,x_0,x_1] = \frac{f[x,x_0] - f[x_0,x_1]}{x - x_1}$$

$$f[x,x_0] = f[x_0,x_1] + (x-x_1) f[x,x_0,x_1] \ldots \ldots (2)$$

Substituting in (1) we get,

$$f(x) = f(x_0) + (x-x_0) f[x_0,x_1] + (x-x_0)(x-x_1) f[x,x_0,x_1]$$

Proceeding in this way we get,

$$f(x) = f(x_0) + (x-x_0) f[x_0,x_1] + (x-x_0)(x-x_1) f[x_0,x_1,x_2] + \ldots \ldots$$
$$+ (x-x_0)(x-x_1) \ldots (x-x_{n-1}) f[x_0,x_1,\ldots ..,x_n]$$
$$+ (x-x_0)(x-x_1) \ldots (x-x_n) f[x,x_0,x_1,\ldots ..,x_n]$$

If $f(x)$ is a polynomial of degree n then, the $(n+1)^{th}$ order divided differences of $f(x)$ will be zero.

$$\therefore f[x,x_0,x_1,\ldots \ldots x_n] = 0$$

Hence, $f(x) = f(x_0) + (x-x_0) f[x_0,x_1] + (x-x_0)(x-x_1) f[x_0,x_1,x_2] + \ldots \ldots$
$$+ (x-x_0)(x-x_1) \ldots (x-x_{n-1}) f[x_0,x_1,\ldots ..,x_n]$$

This is called *Newton's divided difference formula*.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n, i, j, k;
    double x, y, ans, temp;
    vector<vector<double> >allX, allY;

    while (true) {
        cout << "Number of points in the table: ";
        cin >> n;

        if (!n) {
            break;
        }

        allX.assign(n, vector<double>());
        allY.assign(n, vector<double>());
        for (i = 0; i < n; i++) {
            cout << "Enter x" << i << " y" << i << "\n";
            cin >> x >> y;
            allX[0].push_back(x);
            allY[0].push_back(y);
        }

        // denominator table
        for (i = 1; i < n; i++) {
            for (j = 0, k = j + i; j < n && k < n; j++, k++) {
                x = allX[0][k] - allX[0][j];
                allX[i].push_back(x);
            }
        }

        for (i = 1; i < n; i++) {
            for (j = 1; j < allY[i - 1].size(); j++) {
                y = (allY[i - 1][j] - allY[i - 1][j - 1]) / (allX[i][j
- 1]);
                allY[i].push_back(y);
            }
        }

        cout << "Enter value for x: ";
        cin >> x;

        ans = allY[0][0];
        for (i = 1; i < n; i++) {
            temp = 1.0;
            for (j = 0; j < i; j++) {
                temp *= (x - allX[0][j]);
            }
            ans += (temp * allY[i][0]);
        }

        cout << "f(" << x << ") = " << ans << "\n";
    }

    return 0;
}

```

Sample I/O

Number of points in the table: 5
Enter x0 y0: 1 0
Enter x1 y1: 3 18
Enter x2 y2: 4 58
Enter x3 y3: 6 190
Enter x4 y4: 10 920
Value for x: 2.7
f(2.7) = 9.35463

Number of points in the table: 5
Enter x0 y0: 2 10.012
Enter x1 y1: 4 12.123
Enter x2 y2: 5 14.234
Enter x3 y3: 7 18.123
Enter x4 y4: 9 22.342
Value for x: 4.2
f(4.2) = 12.5318