Chapter 1
Binary Search

a: element list

x: the element to be searched.

left, *right*: the left and right ends of the list

Iterative Binary Search (對應課本程式 Program 1.7)

```
initialize left and right;
while(there are more integers to check)        // ____ ≦ _____
{
    let a[middle] be the middle element;       // middle = ________________
    switch(COMPARE(x, a[middle])
    {
        case '>':   left = middle + 1;
                    break;
        case '=':   return middle;
        case '<':   right = middle – 1;
    }
}
return –1;
```

Recursive Binary Search (對應課本程式 Program 1.8)

n: the number of elements in the list

```
int BinarySearch(list, search_num, left, right)
{
    int middle;
    if(left <= right)    // terminate condition
    {
        middle = (left + right) / 2;
        switch(COMPARE(search_num, list[middle])
        {
            case '>':   return BinarySearch(list, search_num, middle + 1, right);
            case '=':   return middle;
            case '<':   return BinarySearch(list, search_num, left, middle – 1);
        }
    }
    return –1;
}
```

change of parameters

```
BinarySearch(a, x, 0, n – 1);
```

Chapter 2
Polynomial

Polynomial Addition (Array)

```
polynomial add_p(polynomial a, polynomial b)
{
        polynimial c;
        int m_degree;

        m_degree = MAX(a.degree, b.degree);
        for(i = 0; i <= m_degree; i++)
                c.coef[i] = a.coef[i] + b.coef[i];

        c.degree = m_degree;
}
```

Polynomial Addition (Linked List) (對應課本程式 Program 2.6)

p : points to the first element of polynomail *a*

q : points to the first element of polynomail *b*

```
while(!IS_ZERO(p) && !IS_ZERO(q))
{
        switch(COMPARE(EXP_OF(p), EXP_OF(q)))
        {
                case '>':    attach p to c;
                             advance p to the next element in a;
                             break;

                case '<':    attach q to c;
                             advance q to the next element in b;
                             break;

                case '=':    add the coefficients of p and q;
                             attach to c;
                             advance p to the next element in a;
                             advance q to the next element in b;
        }
}

if(there are remaining terms in a)
        attach all of them to c;

if(there are remaining terms in b)
        attach all of them to c;
```

Matrix (2D-Array)

Transpose

a: m × n matrix

c: n × m matrix

c ← transpose(*a*)

```
for(i = 0; i < m; i++)                    // O(m)
    for(j = 0; j < n; j++)                // O(n)
        c[j][i] = a[i][j];
                                          O(m • n)
```

Addition

a, *b*, *c*: m × n matrices

c ← add(*a*, *b*)

```
for(i = 0; i < m; i++)                    // O(m)
    for(j = 0; j < n; j++)                // O(n)
        c[i][j] = a[i][j] + b[i][j];
                                          O(m • n)
```

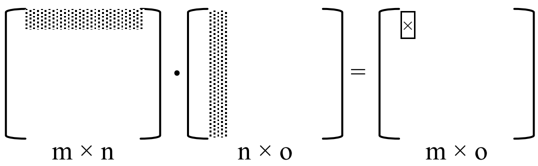
Multiplication

a: m × n matrix

b: n × o matrix

c: m × o matrix

c ← multiply(*a*, *b*)




```
for(i = 0; i < m; i++)                    // O(m)
    for(j = 0; j < o; j++)                // O(o)
    {
        sum = 0;
        for(k = 0; k < n; k++)            // O(n)
            sum = sum + a[i][k] * b[k][j];
        c[i][j] = sum;
    }
                                          O(m • n • o)
```

Sparse Matrix (1D Array)

<u>Fast Transpose</u> (對應課本程式 Program 2.9)

b ← transpose(*a*)

```
FastTranspose(term a[], term b[])
{
        initialize b[0];
        initialize row_terms[] to Ø;

        for(i = 0; i <= #terms; i++)
        {
                // compute row size
                row_terms[a[i].col] = row_terms[a[i].col] + 1;
        }

        compute row_start[];

        for(i = 1; i <= #terms; i++)
        {
                assign data in a[i] to b[row_start[a[i].col]];
                row_start[a[i].col = row_start[a[i].col + 1;
        }
}
```

O(#terms + #col)