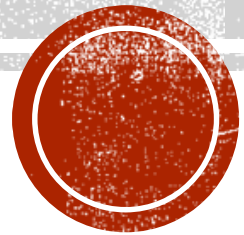
ARRAY & MATRIX



Kasım 2013

ARRAY

int list[5];

Variable	Memory Address
list[0]	base address = x
list[1]	x + sizeof(int)
list[2]	x + 2*sizeof(int)
list[3]	x + 3*sizeof(int)
list[4]	x + 4*sizeof(int)

list2 → list2[0]

list2 +i → list2[i]

list2+i = &(list2[i])

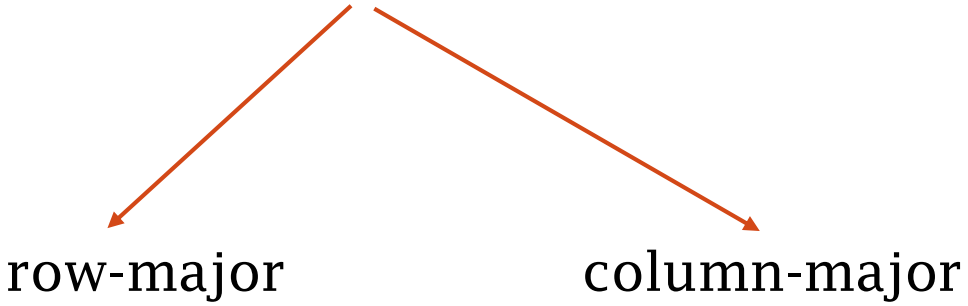
*(list2+i) = list2[i]

$A[limit_0][limit_1]....[limit_n]$

size of the array = $\prod limit_i$

ARRAY LAYOUT

- critical for correctly passing arrays between programs written in different languages.
- array elements that are contiguous in memory is usually faster than accessing elements.
- important for performance when traversing an array.



ROW-MAJOR ORDERING

A multidimensional array in linear memory is organized such that

- rows are stored one after the other.
- C programming language.
- offset = row*NUMCOLS + column

y[2][2][3] ($limit_0 = 2$, $limit_1 = 2$, $limit_2 = 3$)

y	[0][0][0]	[0][0][1]	[0][0][2]	[0][1][0]	[0][1][1]	[0][1][2]	[1][0][0]	[1][0][1]	[1][0][2]

y[0][0][0] → x

y[i][0][0] → x + i*$limit_1$*$limit_2$

y[i][l][0] → x + i*$limit_1$*$limit_2$ + l*$limit_2$

COLUMN-MAJOR ORDERING

- the columns are listed in sequence.
- The scientific programming lang : Fortran
- The matrix-oriented lang : Matlab, Octave
- The statistical lang : R
- offset = row + column*NUMROWS

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

if stored contiguously in memory with column-major order:

1 4 2 5 3 6

```
#define MAX_SIZE 15
float sum(float [], int);
float input[MAX_SIZE], answer;

void main()
{
        for(i = 0; i < MAX_SIZE; i++)
                input[i] = i;
        answer = sum(input, MAX_SIZE);
        printf("The sum is: %f ", answer);
}

float sum(float list[], int n)
{
        int i;
        float tempSum = 0;

        for(i = 0; i < n; i++)
                tempsum += list[i];
        return tempSum;
}
```

STRUCTURES

```
struct{
      char name[10];
       int age;
       float salary;
} employee;
```

```
Typedef struct {…} employee;
employee emp1, emp2;
```

UNIONS

```
typedef struct sex_type {
      enum tag_field {female, male} sex;
       union{
            int children;
            int beard;
       } u;
};

employee emp1, emp2;
emp1.sex_info.sex = female;
Emp1.sex_info.children =  3;
```

```
typedef struct employee{
            char name[10];
            int age;
            float salary;
            sex_type sex_info;
            }
```

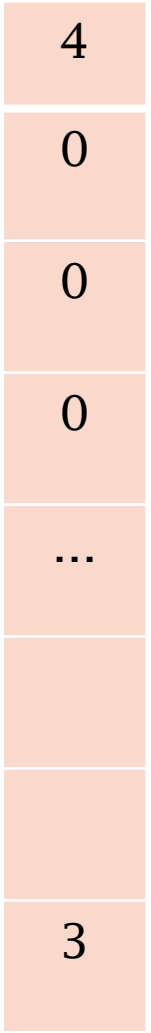
THE POLYNOMIAL ADT

$A(x) = 3x^{20} + 2x^5 + 4$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$

$A(x) + B(x) = \sum(a_i+b_i)x^i$

$A(x) * B(x) = \sum(a_i x^i * \sum(b_j x^j))$



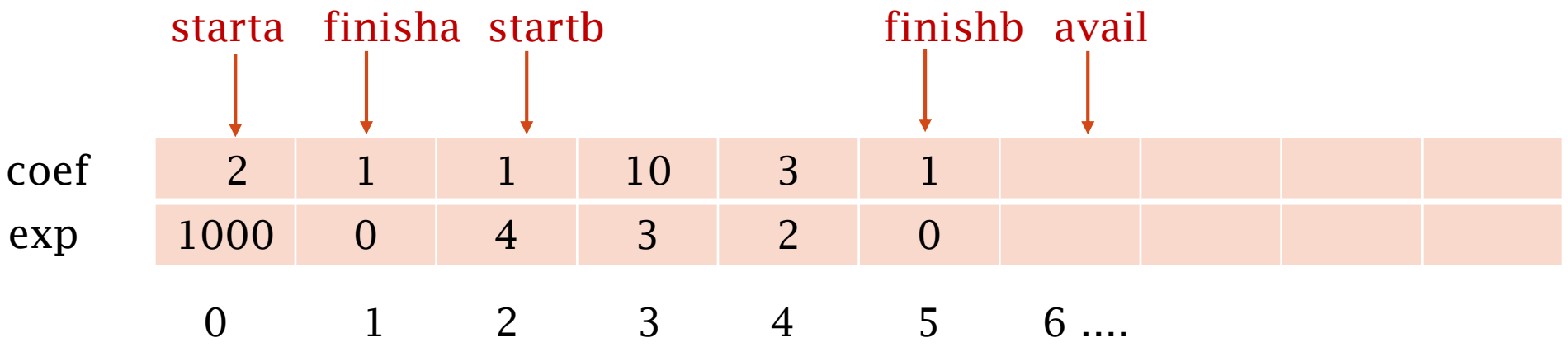

```
typedef struct{
        int degree;
        float coef[MAX_DEGREE];
}poylnomial;
```

THE POLYNOMIAL ADT

```
typedef struct{
        float coef;
        int expon;
}polynomial;
polynomial terms[MAX_TERMS];
```

$A(x) = 2x^{1000} + 1$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$



```
void padd(int starta, int finisha, int startb, int finishb, int *startd, int *finishd){
    float coefficient;
   *startd = avail;
   while(starta <= finisha && startb <=finishb){
          switch(COMPARE(terms[starta].expon, terms[startb].expon)){
             case -1:  //a expon < b expon
                attach(terms[startb].coef, terms[startb].expon);
                startb++; break;
             case 0 :
                coefficient = ters[starta].coef + terms[startb].coef;
                if(coefficient)
                   attach (coefficient, terms[starta].expon);
                starta++;
                startb++; break;
             case 1:  //a expon > b expon
                attach(terms[starta].coef, terms[starta].expon);
                starta;
          }}
```

```
        for(; starta <= finisha; starta++)
            attach(terms[starta]. coef, terms[starta].expon);
        for(; startb <= finishb; starta++)
            attach(terms[startb]. coef, terms[startb].expon);
        *finish = avail-1;
}
```

```
void attach(float coefficient, int exponent)
{/*add a new term to the polynomial*/
    if(avail >= MAX_TERMS)
        exit(1);
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

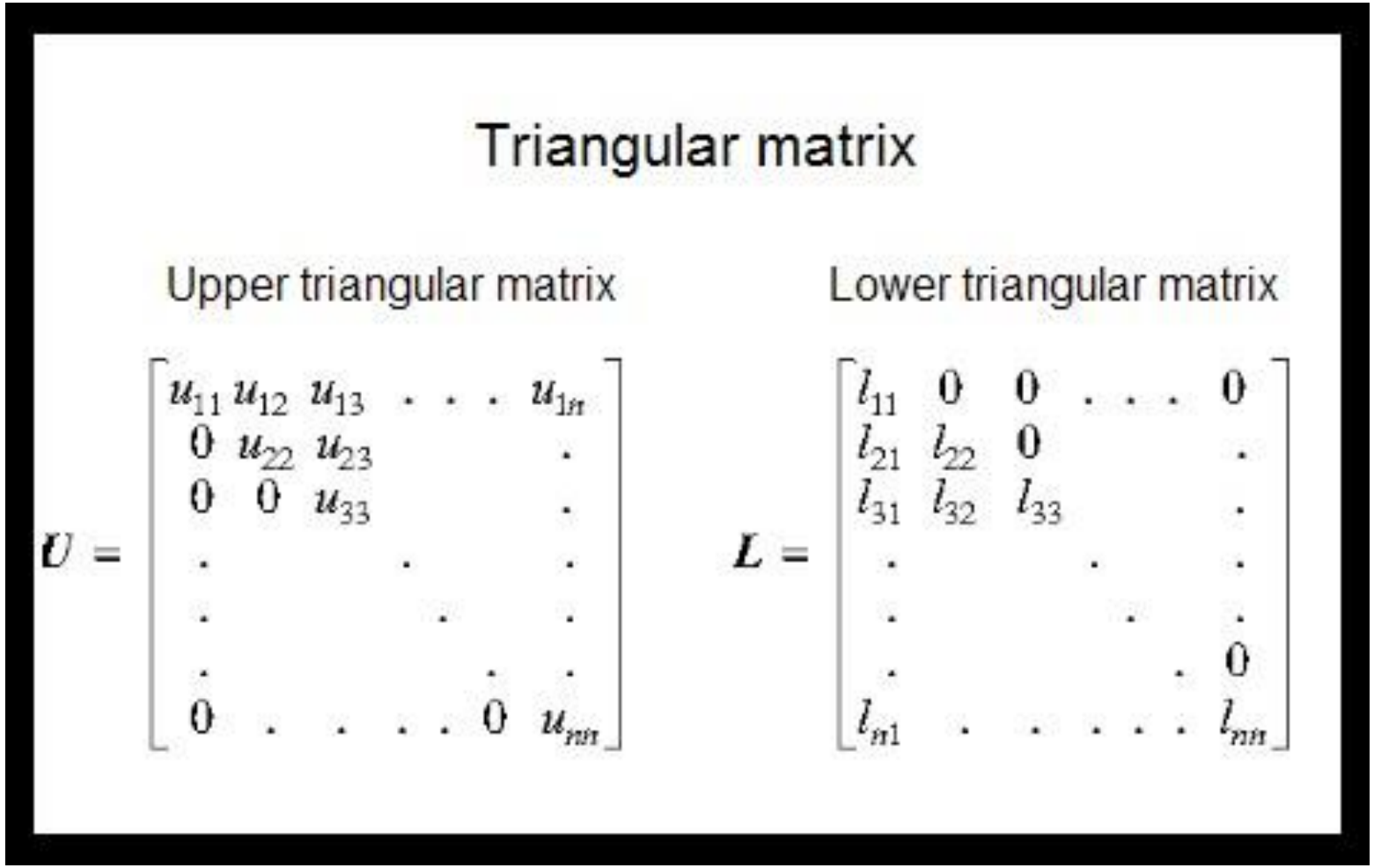
ANALYSIS OF *PADD*

m : the number of non-zero elements in A

n : the number of non-zero elements in A

O(m+n)

TRIANGULAR MATRIX



LOWER TRIANGULAR MATRIX

$$a \begin{bmatrix} a_{oo} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

1 element in the first row
2 elements in the second row
...
...
n elements in the n^{th} row

size of the array = n(n+1)/2

ALT	a_{oo}	a_{1o}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	a_{33}

a[i][j] → ALT[i*(i+1)/2 + j]

```c
#define MAX_SIZE 100

void altUcgenOku(int [], int);
int  altUcgenEris(int, int, int);

int main()
{
	int alt[MAX_SIZE];
	int indis = 0, i, n;
	//boyut oku
	scanf("%d", &n);
	altUcgenOku(alt, n);
	for(i = 0 ; i < n*(n+1)/2; i++)
		printf("%d", alt[i]);
		printf("\n");

	indis = altUcgenEris(2, 0 , n);
	if(indis == -2)
		printf("gecersiz indis\n");
	else if (indis == -1)
		printf("ust ucgene erisim var\n");
	else
		printf("konumu : %d, deger: %d", indis, alt[indis]);

	return 0;
}
```

```c
void altUcgenOku(int alt[], int n)
{
	int i, j,k;

	if(n*(n+1)/2 > MAX_SIZE)
		printf("Alt dizi boyu yetersiz");
	else{
		for(i = 0; i <=n-1; i++){
			k= i*(i+1)/2;
			for(j = 0; j <=i; j++)
				scanf("%d", &alt[k+j]);
		}
	}
}
```

```c
int altUcgenEris(int i, int j, int n)
{
	printf("%d %d", i, j);
	if(i < 0 || i >= n || j < 0 || j>=n){
		printf("gecersiz indis\n");
		return -2;
	}
	else if(i >= j)
		return (i+1)*i/2 + j;
	else
		return -1;
}
```

BAND MATRIX (KUŞAK MATRIS)

- a sparse matrix.
- comprising the main diagonal and zero or more diagonals on either side.

$$\begin{bmatrix} B_{11} & B_{12} & 0 & \cdots & \cdots & 0 \\ B_{21} & B_{22} & B_{23} & \ddots & \ddots & \vdots \\ 0 & B_{32} & B_{33} & B_{34} & \ddots & \vdots \\ \vdots & \ddots & B_{43} & B_{44} & B_{45} & 0 \\ \vdots & \ddots & \ddots & B_{54} & B_{55} & B_{56} \\ 0 & \cdots & \cdots & 0 & B_{65} & B_{66} \end{bmatrix}$$

BAND MATRIX (KUŞAK MATRIS)

$$\begin{bmatrix} 6 & 7 & 0 & 0 \\ 8 & 0 & 4 & 0 \\ 9 & 3 & 2 & 9 \\ 0 & 7 & 6 & 8 \end{bmatrix}$$

9	7	8	3	6	6	0	2	8	7	4	9

$$\begin{matrix} -2 & 0 \\ -1 & 1 \\ 0 & 2 \\ 1 & 3 \end{matrix} \begin{bmatrix} 0 \\ 2 \\ 5 \\ 9 \end{bmatrix}$$

```c
#include <stdio.h>
#define MAX_SIZE 100

void kusakMKur(int [], int [], int, int, int);
int  kusakMEris(int, int, int, int, int, int []);

int main()
{
    int kusak[MAX_SIZE];
    int ara[MAX_SIZE];

    int indis, row, col, n, i;
    printf("n : "); scanf("%d", &n);
    printf("row : "); scanf("%d", &row);
    printf("column : "); scanf("%d", &col);
    kusakMKur(kusak, ara, n, row, col);

    printf("\n");
    for(i = 0; i <= row + col -2 ; i++)
            printf("%d ", ara[i]);
    printf("\n");
    indis = kusakMEris(3, 1, 4, 3, 2, ara);

    if(indis == -2)
        printf("indis gecersiz \n");
    else if(indis == -1)
        printf("\n aranan eleman:0");
    else
        printf("\n aranan eleman: %d ---> %d",
indis, kusak[indis]);

    return 0;
}
```

```
void kusakMKur(int kusak[], int ara[], int n, int row, int col)
{
    int kosegen, k, ogeSay;

    ogeSay = 0;

    for(kosegen = -row + 1; kosegen <= col-1; kosegen++)
    {
        ara[kosegen + row - 1] = ogeSay;
        for(k = 0; k <= n - abs(kosegen)-1 ; k++)
            scanf("%d", &kusak[ara[kosegen + row - 1]+k]);
        ogeSay += (n - abs(kosegen));
    }
}
```

```
int  kusakMEris(int i, int j, int n, int row, int col, int ara[])
{
    if(i >= n || i < 0 || j >= n || j < 0)
        return -2;
    else{
        if(j > i){       //kosegen ustunde mi?
            if(j-i < col)  //ust kusakta mİ
                return (ara[row-1+j-i]+i);
            else         //aranan eleman 0
                return -1;
        }
        else if(i - j < row)    //alt kusakta mi? (kosegen dahil)
            return (ara[j-i+row-1]+j);
        else
            return -1;
    }
}
```

SPARSE MATRIX (SEYREK MATRIS)

- Primarily with zeros.
- Wastes space.

Sparsity (**density**) : the fraction of zero elements.

Basic matrix operations

Creation

Addition

Multiplication

Transpose

SPARSE MATRIX

A	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

	Row	Column	Value
A[0]	6	6	8
A[1]	0	0	15
A[2]	0	3	22
A[3]	0	5	-15
A[4]	1	1	11
A[5]	1	2	3
…			
A[8]	5	2	28

SPARSE MATRIX - ADT

```
typedef struct{
        int row;
        int col;
        int value;
} term;

term A[MAX_TERMS];
```

For efficient matrix operations

- The row indices are in ascending order.
- The column indices are in ascending order (for any row).
- The number of rows, columns and non-zero elements are known.

```
int  transpose(term A[], term newA[])
{
    int n, i, j, current;
    n = A[0].value;
    newA[0].row = A[0].col;
    newA[0].col = A[0].row;
    newA[0].value = n;

    if(n > 0){
        current = 1;
       for(i=0; i < A[0].col; i++)
            for(j=1; j <=n; j++)
                /*A icinde kolonu i olanlari bul*/
                if(A[j].col == i){
                    newA[current].row = A[j].col;
                    newA[current].col = A[j].row;
                    newA[current].value = A[j].value;
                    current++;
                }
    }
}
```

O(columns * elements)

O(columns * columns*rows)
O(columns^2 * rows)

If we represent the matrices as two dimensional arrays of size rowx*column →

O(rows*columns)

```
int  fastTranspose(term A[], term newA[])
{
    int row_terms[MAX_COL], startingPos[MAX_COL];
    int i, j, num_cols = A[0].col, num_terms = A[0].value;
    newA[0].row = num_cols;
    newA[0].col = A[0].row;
    newA[0].value = num_terms;

    if(num_terms > 0){
      for(i = 0; i < num_cols; i++)
            row_terms[i] = 0;
      for(i = 1; i <= num_terms; i++)
            row_terms[A[i].col]++;
      starting_pos[0] = 1;
      for(i = 1; i < num_cols; i++)
            starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
      for(i=1; i <= num_terms; i++){
            j = starting_pos[A[i].col]++;
            newA[j].row = A[i].col;
            newA[i].col = A[i].row;
            newA[i].value = A[i].value;
       }

    }
}
```

O(columns + elements)