TÀI LIỆU THUYẾT MINH

I. Hướng dẫn sử dụng:

- Khi bắt đầu chạy, phần mềm sẽ yêu cầu người dùng nhập vào một ma trận. Người dùng tiến hành nhập ma trận vào. Các trường hợp ma trận nhập vào không hợp lệ thì phần mềm sẽ báo lỗi.

- Tiếp theo, phần mềm sẽ yêu cầu người dùng nhập vào chức năng cần thực hiện (tương ứng với các số từ 0 đến 11). Phần mềm sẽ báo lỗi nếu chức năng nhập vào không tồn tại.

- Các chức năng của phần mềm:

0. Hiện menu các chức năng của phần mềm.

1. Nhập một ma trận khác.

2. Xuất ma trận.

3. Tìm phần tử X trong ma trận.

4. Tìm phần tử lớn nhất trong ma trận và trên biên ma trận.

5. Tính tổng các phần tử trong ma trận và xác định các phần tử có giá trị lớn hơn tổng này.

6. Tính tổng các dòng & các cột trong ma trận, cho biết gia trị lớn nhất thuộc dòng /cột nào.

7. Kiểm tra xem ma trận có phải là ma phương hay không.

8. Chỉ ra các vị trí “yên ngựa” trên ma trận. (lớn nhất trên dòng và nhỏ nhất trên cột).

9. Đếm số “hoàng hậu” trên ma trận. (lớn nhất trên dòng, cột và 2 đường chéo đi qua nó).

10. Sắp xếp các giá trị nằm trên biên ma trận tăng dần theo chiều kim đồng hồ.

11. Exit.

II. Giải thích một số thuật toán trong phần mềm:

1. Hàm kiểm tra xem một ma trận có phải là ma phương hay không:

- Theo Wikipedia, một ma phương là một cách sắp xếp n^2 số, thường là các số nguyên phân biệt, trong một bảng vuông sao cho tổng n số trên mỗi hàng, cột, và đường chéo đều bằng nhau. Ma trận kì ảo chuẩn chứa các số nguyên từ 1 đến n^2.

- Phần mềm được viết dựa trên các khái niệm trên (ma trận nhập vào có số dòng, số cột bằng nhau, các số nhập vào là số nguyên không trùng nhau, ma trận chứa các số nguyên từ 1 đến n^2). Trường hợp không thỏa các điều kiện sẽ không được coi là ma phương.

- Thuật toán:

- Hàm chính:

```
// Hàm kiểm tra xem ma trận có phải là ma phương hay không:
bool MaPhuong(int a[][10], int m, int n)
{
	int sum1 = 0, sum2 = 0;
	if (m != n || Trung(a, m, n))
		return false; // Ma trận hình chữ nhật hoặc ma trận có phần tử trùng nhau
không phải là ma phương.
	else {
		for (int i = 0; i < m; i++) {
			for (int j = 0; j < n; j++) {
				if (a[i][j] < 1 || a[i][j] > n * n) // Ma phương chỉ chứa các
số nguyên từ 1 đến n^2.
					return false;
			}
		}

		// Kiểm tra trên đường chéo:
		for (int i = 0; i < m; i++) {
			sum1 += a[i][i]; // sum1 bằng tổng phần tử đường chéo thứ nhất.
			sum2 += a[i][m - i - 1]; // sum2 bằng tổng phần tử đường chéo thứ
hai.
		}

		if (sum1 != sum2)
			return false;
		else
```

```
                        sum2 = 0; // Trả sum2 về 0 để sử dụng cho lần kiểm tra tiếp theo.

                // Kiểm tra trên hàng:
                for (int i = 0; i < m; i++) {
                        for (int j = 0; j < n; j++) {
                                sum2 += a[i][j];
                        }
                        if (sum1 != sum2) {
                                return false;
                                break;
                        }
                        else
                                sum2 = 0;
                }

                // Kiểm tra trên cột:
                for (int j = 0; j < n; j++) {
                        for (int i = 0; i < m; i++) {
                                sum2 += a[i][j];
                        }
                        if (sum1 != sum2) {
                                return false;
                                break;
                        }
                        else
                                sum2 = 0;
                }
        }
        return true;
}
```

- Hàm phụ:

```
// Hàm kiểm tra xem trong ma trận có các phần tử trùng nhau hay không:
bool Trung(int a[][10], int m, int n)
{
        int t[100], nt = 0; // t[100] là mảng trung gian, nt là số phần tử của mảng t.
        // Xuất tất cả các phần tử của ma trận vào mảng trung gian t:
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                        t[nt] = a[i][j];
                        nt++;
                }
        }

        // Kiểm tra xem trên mảng t có phần tử trùng nhau hay không:
        for (int i = 0; i < nt - 1; i++) {
                for (int j = i + 1; j < nt; j++) {
                        if (t[i] == t[j]) {
                                return true;
                                break;
                        }
                }
```

```
        }
        return false;
}
```

- Giải thích:

+ Khởi tạo 2 biến sum1 = 0 và sum2 = 0. sum1 sẽ lưu một giá trị cố định (ở đây là tổng các phần tử trên đường chéo thứ nhất). sum2 sẽ lần lượt lưu các giá trị: Tổng các phần tử trên đường chéo thứ hai, dòng thứ nhất, thứ hai…,cột thứ nhất, thứ hai…. Sau đó lần lượt so sánh sum1 với sum2.

+ Đầu tiên, phần mềm sẽ kiểm tra các điều kiện: Ma trận có số dòng và số cột bằng nhau hay không, có phần tử trùng nhau hay không (thông qua hàm Trung), có chứa các số nguyên từ 1 đến n^2 hay không.

+ Nếu thỏa 3 điều kiện trên, phần mềm sẽ tính sum1, rồi so sánh sum1 với từng trường hợp của sum2 và cho ra kết quả.

2. Tìm vị trí và đếm số yên ngựa trên ma trân:

- Vị trí yên ngựa trên ma trận là vị trí mà phần tử tại đó là lớn nhất trên dòng và nhỏ nhất trên cột đi qua nó.

- Thuật toán:

+ Hàm chính:

```
// Hàm tìm vị trí và đếm số yên ngựa trên ma trận:
void YenNgua(int a[][10], int m, int n, int row[], int col[], int &so_yen_ngua, int t[],
int nt)
{
        int k = 0;
        for (int i = 0; i < m; i++) {
                ViTriMaxdong(a, n, t, nt, i);
                for (int j = 0; j < nt; j++) {
                        if (MinCot(a, m, i, t[j])) {
                                row[k] = i;
                                col[k] = t[j];
                                k++;
                        }
                }
        }
        so_yen_ngua = k;
```

```
}
```

+ Các hàm phụ:

```
// Hàm tìm vị trí của phần tử lớn nhất trên một dòng:
void ViTriMaxdong(int a[][10], int n, int t[], int &nt, int rowpos) // nt là số phần tử
của t[].
// t[] là mảng lưu các vị trí cột của các phần tử tìm được, nt là số phần tử của t[].
{
	// Giả sử phần tử đầu tiên trên dòng là max:
	int max = a[rowpos][0], k = 0;

	// So sánh max với các phần tử còn lại, nếu có phần tử lớn hơn max thì max sẽ là
phần tử đó:
	for (int j = 0; j < n; j++) {
		if (max < a[rowpos][j])
			max = a[rowpos][j];
	}

	// Kiểm tra xem trên dòng có bao nhiêu phần tử bằng max, lưu vị trí của các phần
tử đó vào mảng t:
	for (int j = 0; j < n; j++) {
		if (max == a[rowpos][j]) {
			t[k] = j;
			k++;
		}
	}
	nt = k;
}



// Hàm kiểm tra xem một phần tử trong ma trận có phải là phần tử nhỏ nhất trên cột hay
không:
bool MinCot(int a[][10], int m, int rowpos, int colpos)
{
	int min = a[rowpos][colpos];
	for (int i = 0; i < m; i++) {
		if (a[i][colpos] < min) {
			return false;
			break;
		}
	}
	return true;
}
```

- Giải thích:

+ Vì có thể có nhiều yên ngựa trên một ma trận nên ta sử dụng các mảng row[], co[] để lưu lại vị trí của yên ngựa, biến so_yen_ngua để lưu số yên ngựa. Mảng t[] dùng để chứa các vị trí cột của phần tử lớn nhất trên một dòng, nt là số phần tử của t[].

+ Hàm phụ ViTriMaxDong sẽ lần lượt tìm các vị trí max trên dòng thứ i (i sẽ chạy từ 0 đến m – 1) rồi lưu vào t[], hàm sẽ trả số phần tử của t[] vào nt.

+ Tiếp theo hàm phụ MinCot sẽ kiểm tra xem phần tử t[j] thuộc dòng i có là phần tử nhỏ nhất trên cột hay không (j sẽ chạy từ 0 đến nt – 1). Nếu có thì vị trí i, j của phần tử sẽ lần lượt được lưu trong row[], col[], số yên ngựa sẽ tăng lên 1.

3. Hàm tìm vị trí và đếm số hoàng hậu trên ma trận:

- Vị trí hoàng hậu là vị trí mà phần tử tại đó có giá trị lớn nhất trên dòng, cột và hai đường chéo đi qua nó.

- Thuật toán:

+ Hàm chính:

```
// Hàm tìm vị trí và đếm số hoàng hậu trên ma trận:
void HoangHau(int a[][10], int m, int n, int row[], int col[], int &so_hoang_hau, int
t[], int nt)
{
	int k = 0;
	for (int i = 0; i < m; i++) {
		ViTriMaxdong(a, n, t, nt, i);
		for (int j = 0; j < nt; j++) {
			if (MaxCot(a, m, i, t[j]) && Max2cheo(a, m, n, i, t[j])) {
				row[k] = i;
				col[k] = t[j];
				k++;
			}
		}
	}
	so_hoang_hau = k;
}
```

+ Các hàm phụ:

```
// Hàm tìm vị trí và đếm số yên ngựa trên ma trận:
void YenNgua(int a[][10], int m, int n, int row[], int col[], int &so_yen_ngua, int t[],
int nt)
{
	int k = 0;
	for (int i = 0; i < m; i++) {
		ViTriMaxdong(a, n, t, nt, i);
		for (int j = 0; j < nt; j++) {
			if (MinCot(a, m, i, t[j])) {
				row[k] = i;
```

```
                        col[k] = t[j];
                        k++;
                    }
                }
            }
            so_yen_ngua = k;
}


// Hàm kiểm tra xem một phần tử trong ma trận có phải là phần tử lớn nhất trên cột hay không:
bool MaxCot(int a[][10], int m, int rowpos, int colpos)
{
            int max = a[rowpos][colpos];
            for (int i = 0; i < m; i++) {
                        if (a[i][colpos] > max) {
                                    return false;
                                    break;
                        }
            }
            return true;
}


// Hàm kiểm tra xem một phần tử trong ma trận có phải là phần tử lớn nhất trên 2 đường chéo đi qua nó hay không:
bool Max2cheo(int a[][10], int m, int n, int rowpos, int colpos)
{
            int t = (rowpos <= colpos ? colpos : rowpos);
            // t là số lần lặp, t bằng giá trị lớn nhất giữa rowpos và colpos.

            // Kiểm tra đường chéo thứ nhất:
            for (int i = 1; i <= t; i++) {
                        if (rowpos + i < m - 1 && colpos + i < n - 1) { // Kiểm tra đến các phần tử nằm trên biên thì dừng lại.
                                    if (a[rowpos][colpos] < a[rowpos + i][colpos + i]) {
                                                return false;
                                                break;
                                    }
                        }
                        else
                                    break; // Nếu phần tử cần kiểm tra nằm trên biên thì break.
            }
            for (int i = 1; i <= t; i++) {
                        if (rowpos - i > 0 && colpos - i > 0) {
                                    if (a[rowpos][colpos] < a[rowpos - i][colpos - i]) {
                                                return false;
                                                break;
                                    }
                        }
                        else
                                    break;
            }

            // Kiểm tra đường chéo thứ hai:
            for (int i = 1; i <= t; i++) {
```

```
			if (rowpos - i > 0 && colpos + i < n - 1) {
				if (a[rowpos][colpos] < a[rowpos - i][colpos + i]) {
					return false;
					break;
				}
			}
			else
				break;
		}
		for (int i = 1; i <= t; i++) {
			if (rowpos + i < m - 1 && colpos - i > 0) {
				if (a[rowpos][colpos] < a[rowpos + i][colpos - i]) {
					return false;
					break;
				}
			}
			else
				break;
		}

		return true;
}
```

- Giải thích:

+ Các bước thực hiện: Tìm các vị trí lớn nhất trên dòng, sau đó kiểm tra xem phần tử tại các vị trí đó có là max trên cột và hai đường chéo đi qua nó hay không.

+ Vì cách hoạt động của hàm này tương tự như hàm tìm vị trí và đếm số yên ngựa trên ma trận nên ở đây chỉ giải thích hàm phụ Max2cheo.

+ Giải thích hàm phụ Max2cheo:

- Hàm sẽ kiểm tra phần tử tại dòng rowpos, cột colpos (a[rowpos][colpos].)
- Đầu tiên khai báo biến lưu tạm thời t, t sẽ bằng số dòng nếu số dòng nhỏ hơn số cột và ngược lại (ví dụ ma trận 3 dòng 4 cột thì t = 3). Lý do là vì dù có xét bất cứ vị trí nào trên bất kỳ ma trận nào thì số bước di chuyển trên đường chéo đều không thể lớn hơn con số này (ví dụ xét ma trận 3x3, vị trí đang xét a[0][0], vị trí kết thúc của đường chéo là a[2][2], số bước di chuyển trên đường chéo từ a[0][0] đến a[2][2] cũng chỉ là 2.
- Xét phần tử bất kỳ trên ma trận, vì có hai đường chéo đi qua nó nên ta xét 4 hướng khác nhau:
    a. a[rowpos + i][colpos + i] (dòng tăng, cột tăng).
    b. a[rowpos - i][colpos - i] (dòng giảm, cột giảm).

c. a[rowpos - i][colpos + i] (dòng giảm, cột tăng).
d. a[rowpos + i][colpos - i] (dòng tăng, cột giảm).

- i sẽ tăng dần từ 1 đến t, và các trường hợp trên phải đảm bảo các điều kiện sau để vị trí cần kiểm tra không ra ngoài phạm vi ma trận (tương ứng với các trường hợp a, b, c, d ở trên):
  a. rowpos + i <= m - 1 && colpos + i <= n – 1.
  b. rowpos - i >= 0 && colpos - i >= 0.
  c. rowpos - i >= 0 && colpos + i <= n – 1.
  d. rowpos + i <= m - 1 && colpos - i >= 0.

- Vấn đề, hạn chế còn tồn tại: Hàm này sẽ kiểm tra từ đầu đến cuối, bất kể có sai ở trường hợp a, b, c hay d.

4. Hàm sắp xếp các giá trị nằm trên biên ma trận tăng dần theo chiều kim đồng hồ:

- Thuật toán:

+ Hàm chính:

```
// Hàm sắp xếp các giá trị nằm trên biên ma trận tăng dần theo chiều kim đồng hồ:
void SapBienTang(int a[][10], int m, int n, int t[])
{
	int k = 0;
	// Lưu tất cả các giá trị trên biên ma trận vào mảng trung gian t:
	for (int j = 0; j < n; j++) {
		t[k] = a[0][j];
		k++;
	}
	for (int i = 1; i < m; i++) {
		t[k] = a[i][n - 1];
		k++;
	}
	for (int j = n - 2; j > 0; j--) {
		t[k] = a[m - 1][j];
		k++;
	}
	for (int i = m - 1; i > 0; i--) {
		t[k] = a[i][0];
		k++;
	}

	// Sắp xếp các phần tử của t theo chiều tăng dần:
	SapTang(t, k);
	k = 0;
```

```
	// Trả các phần tử đã sắp xếp trong t vào các vị trí ban đầu trong ma trận:
	for (int j = 0; j < n; j++) {
		a[0][j] = t[k];
		k++;
	}
	for (int i = 1; i < m; i++) {
		a[i][n - 1] = t[k];
		k++;
	}
	for (int j = n - 2; j > 0; j--) {
		a[m - 1][j] = t[k];
		k++;
	}
	for (int i = m - 1; i > 0; i--) {
		a[i][0] = t[k];
		k++;
	}
}
```

+ Hàm phụ:

```
// Hàm sắp xếp các phần tử theo thứ tự tăng dần:
void SapTang(int a[], int n)
{
	for (int i = 0; i < n - 1; i++)
		for (int j = i + 1; j < n; j++)
			if (a[i] > a[j])
				swap(a[i], a[j]);
}
```

- Giải thích:

+ Khai bao mảng trung gian t[] với k phần tử, ban đầu k = 0.

+ Lần lượt xuất các phần tử trên biên ma trận vào t[k], k sẽ tăng dần.

+ Sắp xếp các phần tử trong t[] theo chiều tăng dần thông qua hàm phụ SapTang (chứa thuật toán swap hoán vị các phần tử).

+ Cho k = 0. Sau đó lần lượt trả các phần tử đã sắp xếp trong t[] vào lại vị trí ban đầu trong ma trận, k sẽ tăng dần.

III. CÁC CẢI TIẾN, NÂNG CẤP CỦA PHẦN MỀM:

- Phần mềm có bảng menu liệt kê tất cả các chức năng. Người dùng chỉ cần nhập chức năng cần thực hiện.

- Người dùng chỉ cần nhập ma trận một lần duy nhất để sử dụng cho nhiều chức năng khác nhau, thuận tiện cho việc sử dụng. Nếu muốn, người dùng vẫn có thể nhập một ma trận khác bằng cách sử dụng chức năng số 1.