**1.** داخل آرایه ای از کاراکتر به نام ArrayA به طول n الگو ای نوشته شده است. همچنین آرایه ی دیگری به نام ArrayB به طول m داریم که حروفی در آن قرار دارد. که m از n بزرگ تر است.

**کد** الگوریتمی را بنویسید که در صورت وجود الگو ArrayA در آرایه ی ArrayB 1 و در غیر این صورت 0 را باز گرداند.
مثال:

ArrayA = {A, B, A, A}
ArrayB = {C,D,A,F,A,B,A,A,P,R}
result: 1

چون داخل ArrayB دقیقا رشته ی داخل ArrayA تکرار شده است.

```
int containsPattern(char ArrayB[], int m, char ArrayA[], int n){

}
```

```
int containsPattern(char ArrayB[], int m, char ArrayA[], int n) {
    for (int i = 0; i <= m - n; i++) {
        int found = 1;
        for (int j = 0; j < n; j++) {
            if (ArrayB[i + j] != ArrayA[j]) {
                found = 0;
                break;
            }
        }
        if (found) {
            return 1;  // الگو پیدا شد
        }
    }
    return 0;  // الگو پیدا نشد
}
```

**2.** میخواهیم خانه های یک شهر را پلاک گذاری کنیم به صورتی که هیچ دو خانه ای نباشد که پلاک یکسانی داشته باشند. برای این کار از یک linked list کمک میگیرم که در هر node آن، یک اشاره گر به خانه

ی بعدی و یک عدد int که بیانگر پلاک است وجود دارد.

**الف**- کد struct گره را بنویسید. (5 نمره)

```
typedef struct Node{

} Node;
```

```
typedef struct Node {
    int plate;
    struct Node* next;
} Node;
```

**ب**- تابعی بنویسید که یک اشاره گر به اول لیست پیوندی و یک عدد که پلاک جدید است را دریافت میکند و در صورتی که آن پلاک تا به حال ثبت نشده بود، آن را به آخر لیست اضافه کند و در خروجی اشاره گر به سر لیست پیوندی را بازگرداند.. (5 نمره)

```
Node* appendNode(Node* headRef, int plate){

}
```

```
Node* appendNode(Node* head, int plate) {
    Node* current = head;

    while (current != NULL) {
        if (current->next == NULL) {
            break;
        }
        current = current->next;
    }

    Node* newNode = (Node*)malloc(sizeof(Node));

    newNode->plate = plate;
    newNode->next = NULL;

    if (current == NULL) {
        head = newNode;
    } else {
        current->next = newNode;
    }

    return head;
}
```

**ج**- حال میخواهیم پلاک خانه هایی که خراب شده اند را از لیست حذف کنیم. تابعی بنویسید که یک اشاره گر به اول لیست پیوندی و یک عدد هم پلاکی است که باید حذف بشود را در ورودی دریافت کند و در صورت وجود آن عدد در لیست، آن node را حذف کند. (10 نمره)

Node* deleteNode(Node* headRef, int plate){

}

```
Node* deleteNode(Node* head, int plate) {
    Node* currentNode = head;
    Node* previousNode = NULL;

    while (currentNode != NULL) {
        if (currentNode->plate == plate) {
            if (previousNode == NULL) {
                head = currentNode->next;
            } else {
                previousNode->next = currentNode->next;
            }
            free(currentNode);
            break;
        }
        previousNode = currentNode;
        currentNode = currentNode->next;
    }
    return head;
}
```

**د**- حال تابعی بنویسید که در ورودی اشاره گر به اول لیست پیوندی را گرفته و لیست را به شکلی مرتب سازی کند که پلاک ها به صورت صعودی قرار بگیرند. (10 نمره)

void sortList(Node* headRef){

}

```
void SortList(Node* head) {
    if (head == NULL) return;

    Node* Iterator = head;
    while (Iterator != NULL) {
        Node* Iterator2 = head;
        while (Iterator2 != NULL && Iterator2->next != NULL) {
            if (Iterator2->plate > Iterator2->next->plate) {
                int tempPlate = Iterator2->plate;
                Iterator2->plate = Iterator2->next->plate;
                Iterator2->next->plate = tempPlate;
            }
            Iterator2 = Iterator2->next;
        }
        Iterator = Iterator->next;
    }
}
```

ه- حال میخواهیم لیست را به شکلی پیاده سازی کنیم که همواره پلاک ها به صورت صعودی داخل لیست مرتب باشند و نیازی نباشد که تابعی برای مرتب سازی آن بنویسیم. برای این کار کافی است که تابع اضافه کردن پلاک را تغییر دهیم. تابع appendNode را به شکلی بنویسید که خواسته ی ما برآورده شود. (10 نمره)

```c
Node* appendNode(Node* head, int plate) {
    Node* newNode = (Node*)malloc(sizeof(Node));

    newNode->plate = plate;
    newNode->next = NULL;

    if (head == NULL || head->plate >= plate) {
        newNode->next = head;
        head = newNode;
        return head;
    }

    Node* temp = head;
    while (temp->next != NULL && temp->next->plate < plate) {
        temp = temp->next;
    }

    newNode->next = temp->next;
    temp->next = newNode;

    return head;
}
```

**3.** در یک خیابان تعدادی تاکسی و مسافر وجود دارد. مسافری را سوار کند که فاصله آن با او کمتر مساوی از d باشد. به دنبال آن هستیم حداکثر مسافر را بتوانیم سوار تاکسی کنیم. آرایه ای از int به طول n داریم که نشان دهنده ی خیابان است. تاکسی را با 2 و مسافر را با 1 در آرایه مشخص می کنیم و همچنین اگر مقدار خانه ای 0 باشد یعنی هیچ چیز در آن وجود ندارد. همچنین فاصله ی تاکسی با مسافر برابر است با تفاضل index آن ها در آرایه. **تابع** زیر را به شکلی کامل کنید که در آن بیشترین تعداد مسافری که میتوانند سوار تاکسی شوند را به دست آورد.

```
int maxPassenger(int avenue [], int n, int d){

}
```

مثال:

D = 2

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 0 | 0 | 2 | 0 | 1 | 1 | 1 | 1 | 2 | 0 | 0 | 2 | 0 | 1 | 2 |

در این حالت بیشترین تعداد مسافری که میتوان سوار بر تاکسی شوند برابر با 4 است.

الگوریتم:
ابتدا از سمت چپ‌ترین تاکسی و مسافر شروع می‌کنیم.
فاصله بین تاکسی و مسافر را بررسی می‌کنیم:
اگر فاصله کمتر از d باشد، مسافر می‌تواند سوار تاکسی شود و هر دو اندیس (تاکسی و مسافر) به جلو حرکت می‌کنند.
اگر فاصله برابر یا بزرگ‌تر از d باشد:
اگر تاکسی قبل از مسافر است، تاکسی بعدی را پیدا می‌کنیم.
اگر مسافر قبل از تاکسی است، مسافر بعدی را پیدا می‌کنیم.
این فرآیند را تا زمانی که یکی از مسافر یا تاکسی به آخر آرایه برسد ادامه میدهیم.

```
int maxPassenger(int avenue[], int n, int d) {
    int count = 0;
    int taxiIndex = 0;
    int passengerIndex = 0;

    while (taxiIndex < n && passengerIndex < n) {
        while (taxiIndex < n && avenue[taxiIndex] != 2) {
            taxiIndex++;
        }

        while (passengerIndex < n && avenue[passengerIndex] != 1) {
            passengerIndex++;
        }

        if (taxiIndex >= n || passengerIndex >= n) {
            break;
        }

        if (abs(taxiIndex - passengerIndex) < d) {
            count++;
            taxiIndex++;
            passengerIndex++;
        } else if (taxiIndex < passengerIndex) {
            taxiIndex++;
        } else {
            passengerIndex++;
        }
    }

    return count;
}
```