Aufgabe 4: Nandu

Team-ID: 00064

Team-Name: SpaceX

Bearbeiter/-innen dieser Aufgabe:
Fabian Lehmann

20. November 2023

Inhaltsverzeichnis

1 Lösungsidee

Das Ziel meines Programms ist es, für jede mögliche Startkombinationen der Tachenlampen (an oder aus) die dazu passende Kombination der LEDs auszugeben. Meine Idee ist wie folgt: man geht jede mögliche Startkombination durch. Für jede Startkombination iteriert man Zeile für Zeile durch bis man bei den LEDs angekommen ist. Dabei speichert man den Zustand für die vorherige Zeile und geht die nächste Zeile durch. Wenn man dabei auf ein Baustein trifft, schaut man sich den Zustand der vorherigen Zeile für die Position auf der sich der Baustein befindet an und nimmt diesen als Input für die jeweilige Funktion der verschiedenen Bestandteile. Die Funktionen überprüfen dann für den passenden Baustein, wo zum Beispiel für einen blauen Baustein das Licht auf die Sensoren fallen und geben sie in diesem Fall einfach weiter. Dabei gibt es drei mögliche Bausteine:

1. weißer Baustein: leuchtet nur nicht, wenn beide Sensoren bestrahlt werden. Also kann man dies mit einer AND Überprüfung machen. Das heißt wenn beide Sensoren true sind, leuchten sie nicht, ansonsten schon.

2. roter Baustein: leuchten nur nicht, wenn Sensor bestrahlt wird. Also überprüft man für diesen Fall, ob die LED vor dem Sensor True oder False bwz. leuchten oder nicht leuchten. Wenn die LED vor dem Sensor nicht leuchtet, strahlen beide LEDs.

3. blauer Baustein: gibt das Licht einfach weiter. Also muss man überprüfen, welche Lichtsensoren bestrahlt werden und von den LEDs, bei dem die passenden Lichtsensoren bestrahlt werden, geben das Licht weiter.

2 Umsetzung

2.1 Input

Ich speichere während des Eingabevorgangs mehrere Informationen. Ich nutze einen zweidimensionalen Vektor, um alle 'X' und Bausteine zu speichern. Gleichzeitig erfasse ich die Positionen der Taschenlampen und somit auch ihre Anzahl. Diese Information wird später benötigt, um die verschiedenen Möglichkeiten und die Positionen der Taschenlampen in der Bitmaske einzusetzen. Die Bitmaske wird im weiteren Verlauf genauer erläutert. Zudem halte ich die Positionen der LEDs in der letzten Zeile fest. Dies ermöglicht am Ende eine Überprüfung der LED-Positionen und somit eine Ausgabe an den entsprechenden Stellen.

2.2 Funktion für die Lichter

Es gibt insgesamt 4 mögliche Bausteine für die es jeweils eine Funktion gibt. Jede dieser Funktionen gibt ein Pair aus Integer zurück, die entweder 1 oder 0 sind. Ein Pair passt hier besonders gut, da jeder Baustein 2 LEDs hat. In dem Pair speicher ich den Zustand, der das jeweilige LED hat. Für den Input aller Funktionen nimmt man den Zustand der vorherigen Zeile.

1. weißer Baustein:

```
//FUNKTION FUER "WW"
std::pair<int, int> white(int l1, int l2){
    if(l1 && l2){
        return std::make_pair(0,0);
    }
    else{
        return std::make_pair(1,1);
    }
}
```

   Man überprüft, ob beide vorherigen LEDs 1 oder 0 sind.

2. roter Baustein(Sensor ist an erster Position):

```
//FUNKTION FUER "Rr"
std::pair<int,int> redR(int l1, int l2){
    int b1 = 0;
    int b2 = 0;
    if(!l1){
        b1 = 1;
        b2 = 1;
    }
    return std::make_pair(b1,b2);
}

```

Da es bei den roten Bausteinen zwei Möglichkeiten von der Anordnung des Sensors (Sensor an erster Stelle oder Sensor an zweiter Stelle) gibt, benutzte ich auch zwei Funktionen jeweils. Hier ist die Anordnung 'R' 'r' und $l1$ ist die Position vor dem Sensor.

3. roter Baustein(Sensor ist an zweiter Position):

```
//FUNKTION FUER "rR"
std::pair<int,int> redr(int l1, int l2){
    int b1 = 0;
    int b2 = 0;
    if(!l2){
        b1 = 1;
        b2 = 1;
    }
    return std::make_pair(b1,b2);
}
```

Hier ist die Anordnung 'r' 'R' und $l2$ ist die Position vor dem Sensor.

4. blauer Baustein:

```
//FUNKTION FUER "BB"
std::pair<int,int> blue(int l1, int l2){
    int b1 = 0;
    int b2 = 0;
    if(l1){
        b1 = 1;
    }
    if(l2){
        b2 = 1;
    }
    return std::make_pair(b1,b2);
}
```

Bei einem blauen Baustein wird das Licht weitergegeben, wenn der Sensor des LEDs angestrahlt wird.

Als Speicher für die jeweiligen Zeilen benutzte ich zwei Bitmasks. Die erste Bitmask speichert den Zustand der vorherigen Zeile und die zweite Bitmask speichert, den durch die Bausteine und durch das 'X' veränderten Zustand. Die Bitmasks habe ich mit einem Vektor implementiert, sodass ich mehr Freiheit habe in der Veränderung einzelner Bits.

```
vector<int> currentBitmask(width, 0);
vector<int> previusBitmask(width, 0);
```

Mein Programm ist so geschrieben, dass ein 'X' kein lichtdurchlässiger Bestandteil ist bzw. wenn der Sensor nicht durch eine direkte Verbindung mit einem LED angestrahlt wird, wird kein Licht auf den Sensor gestrahlt.

2.3 Berechnung aller Zustände für Startlichter

Es gibt 2^n Möglichkeiten, die n Startlichter annehmen können, denn ein Licht kann entweder 1 oder 0 sein. Da die binäre Darstellung von Zahlen nur aus Einsen und Nullen besteht, müssen wir nur alle Zahlen von 0 bis 2^n in Binär umwandeln. Dabei ist zu beachten, dass wenn es zum Beispiel 4 Startlichter gibt, 0 in Binär nicht '0' sein soll, sondern '0000'. Das bedeutet, dass alle vier Startlichter aus sind. Um jede Möglichkeit (insgesamt 2^n) auszurechnen, speicher ich je den Zustand in ein Vektor aus Strings. Der Algorithmus sieht wie folgt aus:

```
//FUNKTION FUER DIE VERSCHIEDENEN ZUSTAENDE DER LAMPEN
vector<string> binary(int n){   //n ist die Anzhal der Q's

    vector<string> possibilities; //Vektor zur Speicherung aller binaeren Zeichenketten
    vector<bool> s(n, 0); //Hilfsvektor fuer die Erzeugung der Binaerrepraesentation
    string a; //Variable zur Speicherung der Binaerzeichenkette

    for(int i = 0;i < (1 << n); i++){
    //Schleife von 0 bis exklusive 2^n
        for(int j = n - 1; j>=0;j--){
            s[j] = (( i >> (n-1-j)) & 1);
        }
        a = "";
        for(bool bit: s){
            a += (bit ? '1' : '0');
        }
        possibilities.push_back(a);
    }
    return possibilities;
}
```

Die äußere For-Schleife startet von 0 bis 2^n, was alle möglichen Kombinationen von n Bits repräsentiert. Die innere For-Schleife dient erstens dazu, dass die Länge der binären Zeichenketten n Zeichen lang ist. Die Zeile s[j] = ((i » (n-1-j)) AND 1); kann in mehrere Schritte aufgeteilt werden.

1. i wird um $(n-1-j)$ Positionen nach rechts verschoben. Dies wird erreicht durch $i >> (n-1-j)$.
2. Das Ergebnis dieser Verschiebung ist eine Zahl, bei der das $(n-1-j)$-te Bit das am wenigsten signifikante Bit (LSB) ist.
3. Das 'AND 1' bewirkt eine bitweise AND-Operation mit 1. Da das LSB von i das (n-1-j)-te Bit ist, wird durch die AND-Operation mit 1 sichergestellt, dass nur dieses Bit extrahiert wird und in dem Hilfsvekotr 's' geschrieben wird.

Diese Operation ermöglicht es, die Binärrepäsentation der Zahl i schrittweise zu extrahieren und im Hilfsvektor 's' zu speichern.

2.4 Iterieren durch die Konstruktion mit den Bausteinen

Für jede 2^n Möglichkeiten mache ich einen Durchlauf durch alle Bausteine, aber mit unterschiedlich leuchteten Startlampen. Für die Speicherung der Zustände der LEDs verwende ich zwei Bitmasks wie oben schon genannt.

```
for(int i = 0;i<pos_q.size();i++){
    currentBitmask[pos_q[i]] = condition[i]-48;
    previusBitmask[pos_q[i]] = condition[i]-48;
}
```

currentBitmask speichert die Zustände auf der jetztigen Zeile in der man sich befindet und previusBitmask speichert den Zustand der LEDs in der vorherigen Lampen. Da man mit condition[i] auf ein Character greift, aber die Bitmasks aus Integern besteht, muss man -48 rechnen, da man die Zeichen von ASCII zu den Integern decodieren muss. Als Input wird previusBitmask genommen, wenn man auf blaue, rote oder weiße Bausteine trifft. Die veränderten Zustände der LEDs werden dann in der currentBitmask gespeichert. Dafür iteriere ich durch jede Zeile beginnend in der zweiten Zeile und aufhörend in der vorletzten Zeile, da in der letzten Zeile nur noch die LEDs zur Überprüfung liegen.

```
int currentPosWidth;
for(int currentPosHeight = 1;currentPosHeight<height-1;currentPosHeight++){
    currentPosWidth = 0;
    while(currentPosWidth<width){
        switch (nandu[currentPosHeight][currentPosWidth]){
            case 'X':
                currentBitmask[currentPosWidth] = 0;
                currentPosWidth++;
                break;
            case 'W':
                w = {white(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                currentBitmask[currentPosWidth] = w.first;
                currentBitmask[currentPosWidth+1] = w.second;
                currentPosWidth+=BAUSTEIN_BREITE;
                break;
            case 'B':
                b = {blue(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                currentBitmask[currentPosWidth] = b.first;
                currentBitmask[currentPosWidth+1] = b.second;
                currentPosWidth+=BAUSTEIN_BREITE;
                break;
            case 'R':
                R = {redR(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                currentBitmask[currentPosWidth] = R.first;
                currentBitmask[currentPosWidth+1] = R.second;
                currentPosWidth+=BAUSTEIN_BREITE;
                break;
            case 'r':
                r = {redr(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                currentBitmask[currentPosWidth] = r.first;
                currentBitmask[currentPosWidth+1] = r.second;
                currentPosWidth+=BAUSTEIN_BREITE;
                break;
        }
        previusBitmask = currentBitmask;
    }
}
```

Da die weißen, roten und blauen Bausteinen immer in Paare vorkommen, verwende ich hier eine Konstante namens BAUSTEIN'Unterstrich'BREITE dafür, da ich den zweiten Teil von dem Paar mitberechnet habe.

2.5 Laufzeitanalyse

Die Laufzeit des Programms hängt entscheidend von der Anzahl der möglichen Zustände der Taschenlampen (Q's) ab. Die Funktion 'binary(int n)' erzeugt alle möglichen Zustandskombinationen für die Taschenlampen in exponentieller Zeit. Daher beträgt die Laufzeit dieser Funktion $O(2^n)$.
Für jeden Zustand wird die Funktion 'solve' aufgerufen, die wiederum durch jede Zeile der Konstruktion durch iteriert, Die Konstruktion aus Bausteinen und 'X' hat eine genau vorgegebene Anzahl an einer Höhe und Breite. Der Worst-Case wäre hierbei jetzt, dass alle Zeichen in der Konstrukion 'X' wären, denn bei einem 'X' geht man nur eins nach vorne. Zum Worst Case gehört dazu, dass die letzte Zeile nur aus LEDs besteht, sodass das Programm durch die komplette letzte Zeile durch iterieren müsste. Dasselbe gilt auch für die anfänglichen Taschenlampen. Der Worst-Case wäre ,wenn die erste Zeile nur aus Taschenlampen bestehen würde. Das würde bedeuten, das n die Anzahl der Taschenlampen als auch der Breite der Konstruktion wäre, somit würde die Solve-Funktion in einer Worst-Case Laufzeit von $O(n*m)$, wobei n die Anzahl der Taschenlampen als auch die Breite ist und m die Höhe ist. Die Solve-Funktion wird jetzt 2^n mal aufgerufen. Daraus entsteht eine Gesamtlaufzeit von $O(2^n * n * m)$, wobei n die Breite als auch die Anzahl der Taschenlampen ist und m die Höhe der Konstruktion.

3 Beispiele

Wichtig: Beim Ausführen müssen die Input-Dateien im gleichen Ordner sein und es muss Windows benutzt werden.

3.1 Beispiel 1

```
00 => 11
01 => 11
10 => 11
11 => 00
```

3.2 Beispiel 2

```
00 => 01
01 => 01
10 => 01
11 => 10
```

3.3 Beispiel 3

```
000 => 1001
001 => 1000
010 => 1011
011 => 1010
100 => 0101
101 => 0100
110 => 0111
111 => 0110
```

3.4 Beispiel 4

```
0000 => 00
0001 => 00
0010 => 01
0011 => 00
0100 => 10
0101 => 10
0110 => 11
0111 => 10
1000 => 00
1001 => 00
1010 => 01
1011 => 00
1100 => 00
1101 => 00
1110 => 01
1111 => 00
```

3.5 Beispiel 5

```
000000 => 00010
000001 => 00010
000010 => 00011
000011 => 00011
000100 => 00100
000101 => 00100
```

```
000110 => 00011
000111 => 00011
001000 => 00010
001001 => 00010
001010 => 00011
001011 => 00011
001100 => 00100
001101 => 00100
001110 => 00011
001111 => 00011
010000 => 00010
010001 => 00010
010010 => 00011
010011 => 00011
010100 => 00100
010101 => 00100
010110 => 00011
010111 => 00011
011000 => 00010
011001 => 00010
011010 => 00011
011011 => 00011
011100 => 00100
011101 => 00100
011110 => 00011
011111 => 00011
100000 => 10010
100001 => 10010
100010 => 10011
100011 => 10011
100100 => 10100
100101 => 10100
100110 => 10011
100111 => 10011
101000 => 10010
101001 => 10010
101010 => 10011
101011 => 10011
101100 => 10100
101101 => 10100
101110 => 10011
101111 => 10011
110000 => 10010
110001 => 10010
110010 => 10011
110011 => 10011
110100 => 10100
110101 => 10100
110110 => 10011
110111 => 10011
111000 => 10010
111001 => 10010
111010 => 10011
111011 => 10011
111100 => 10100
111101 => 10100
111110 => 10011
111111 => 10011
```

4 Quellcode

4.1 Input

```
std::fstream sample("nandu" + txt + ".txt");

int width, height;
sample >> width >> height;

vector<vector<char>> nandu(height, vector<char>(width));

vector<int> pos_q;
vector<int> pos_l;
char temp;

//INPUT + INFORMATION UEBER Q UND L
for(int i = 0; i<height;i++){
    for(int j = 0;j<width;j++){
        sample >> nandu[i][j];

        if(nandu[i][j] == 'Q'){
            pos_q.push_back(j);
            sample >> temp;
        }
        if(nandu[i][j] == 'L'){
            pos_l.push_back(j);
            sample >> temp;
        }
    }
}
```

4.2 Funktionen für Bausteine

```
//FUNKTION FUER "BB"
std::pair<int,int> blue(int l1, int l2){
    int b1 = 0;
    int b2 = 0;
    if(l1){
        b1 = 1;
    }
    if(l2){
        b2 = 1;
    }
    return std::make_pair(b1,b2);
}

//FUNKTION FUER "Rr"
std::pair<int,int> redR(int l1, int l2){
    int b1 = 0;
    int b2 = 0;
    if(!l1){
        b1 = 1;
        b2 = 1;
    }
    return std::make_pair(b1,b2);
}

//FUNKTION FUER "rR"
std::pair<int,int> redr(int l1, int l2){
    int b1 = 0;
    int b2 = 0;
    if(!l2){
        b1 = 1;
        b2 = 1;
    }
    return std::make_pair(b1,b2);
}

//FUNKTION FUER "WW"
```

```
std::pair<int, int> white(int l1, int l2){
    if(l1 && l2){
        return std::make_pair(0,0);
    }
    else{
        return std::make_pair(1,1);
    }
```

4.3 Binärrepräsentation

```
//FUNKTION FUER DIE VERSCHIEDENEN ZUSTAENDE DER LAMPEN
vector<string> binary(int n){   //n ist die Anzhal der Q's
    vector<string> possibilities;
    vector<bool> s(n, 0);
    string a;
    for(int i = 0;i < (1 << n); i++){
        for(int j = n - 1; j>=0;j--){
            s[j] = (( i >> (n-1-j)) & 1);
        }
        a = "";
        for(bool bit: s){
            a += (bit ? '1' : '0');
        }
        possibilities.push_back(a);
    }
    return possibilities;
}
```

4.4 Solve-Funktion

```
//PRINTET FUER JEDEN ZUSTAND DEN ZUGEHOERIGEN ZUSTAND DER L's AUS
void solve(string condition, vector<int> pos_q, vector<int> pos_l, int width, int height,vector<vector<
    std::pair<int,int> w;
    std::pair<int,int> b;
    std::pair<int,int> r;
    std::pair<int,int> R;

    vector<int> currentBitmask(width, 0);
    vector<int> previusBitmask(width, 0);

    for(int i = 0;i<pos_q.size();i++){
        currentBitmask[pos_q[i]] = condition[i]-48;
        previusBitmask[pos_q[i]] = condition[i]-48;
    }

    int currentPosWidth;
    for(int currentPosHeight = 1;currentPosHeight<height-1;currentPosHeight++){
        currentPosWidth = 0;
        while(currentPosWidth<width){
            switch (nandu[currentPosHeight][currentPosWidth]){
                case 'X':
                    currentBitmask[currentPosWidth] = 0;
                    currentPosWidth++;
                    break;
                case 'W':
                    w = {white(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                    currentBitmask[currentPosWidth] = w.first;
                    currentBitmask[currentPosWidth+1] = w.second;
                    currentPosWidth+=BAUSTEIN_BREITE;
                    break;
                case 'B':
                    b = {blue(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                    currentBitmask[currentPosWidth] = b.first;
                    currentBitmask[currentPosWidth+1] = b.second;
                    currentPosWidth+=BAUSTEIN_BREITE;
                    break;
```

```
                case 'R':
                    R = {redR(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                    currentBitmask[currentPosWidth] = R.first;
                    currentBitmask[currentPosWidth+1] = R.second;
                    currentPosWidth+=BAUSTEIN_BREITE;
                    break;
                case 'r':
                    r = {redr(previusBitmask[currentPosWidth], previusBitmask[currentPosWidth+1])};
                    currentBitmask[currentPosWidth] = r.first;
                    currentBitmask[currentPosWidth+1] = r.second;
                    currentPosWidth+=BAUSTEIN_BREITE;
                    break;
            }
            previusBitmask = currentBitmask;
        }
    }
    string end = "";
    for(int i = 0;i<pos_l.size();i++){
        end += std::to_string(previusBitmask[pos_l[i]]);
    }
    std::cout << condition << "␣=>␣" << end <<'\n';
    return;
}
```

4.5 Aufruf der Solve-Funktion

```
vector<string> possibilities = binary(pos_q.size());

for(string s:possibilities){
    solve(s, pos_q, pos_l, width, height, nandu);
}
```