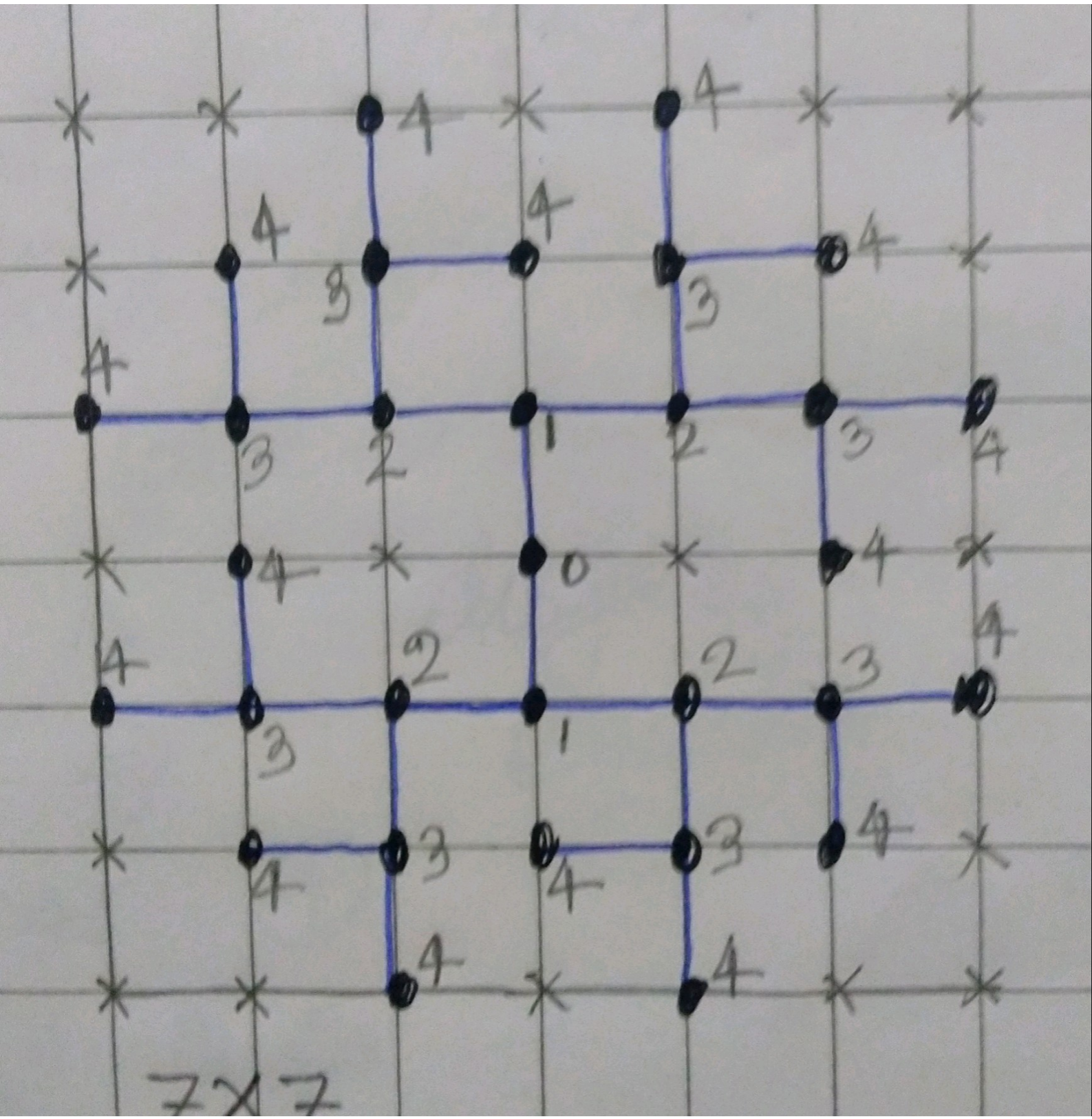
7 X 7

let array B[L] .. be B[0, 1, -- L-1] s.t.

$B[l] = N_l$ for $L = l+1$

From observation

$B[L=0] = 1$, $B[L=1] = 1$, $B[L=2] = 2$

$B[L=3] = 4$, ...

$B[l] = B[l-1] + l-1$

$\Rightarrow N_l = N_{l-1} + l-1$

$\Rightarrow N_L = N_{L-1} + L-1$

$\Rightarrow N_L = N_1 + 1+2+3+ \ldots (L-1)$

$= 1 + \frac{L(L-1)}{2}$

for $L=5$; $l=4$

$N_4 = 1 + \frac{4(3)}{2} = 7$

$\Rightarrow$ For $L=5$ we need 7×7 grid ($N=7$)

In given grid. Black dot : activated processors.

cross : disabled processors.

For large L % (fraction of activated processors)

$$\lim_{L \to large} \frac{2^L - 1}{1 + \frac{(L-1)(L-2)}{2}}$$

Here for $L=5$

$\frac{31}{49} \simeq \frac{30}{50} \simeq 60\%$

Q-2

```
1. A[1.... n]
2. B[0,1, .... n]
3. B[i] = A[1] + .... + A[i]
4. B[0] = 0
5. MSA(B[0,1 .... n]){        // returns sum of max sub-array
                              //according to A using B.
6.  msa_so_far = 0            // stores MSA so far
7.  msa_till   = 0            // stores MSA till index
                                 encountered.
8.  Prev_neg   = 0            // last negative element
                                 encountered in B
9.  For i=0 to n {
10.     If B[i] is negative
            update Prev_neg to B[i]
11.     msa_till = B[i] - Prev_neg
12.     msa_till = max(msa_till, 0)
                   //msa should not be negative
13.     msa_so_far = max(msa_till, msa_so_far)
    }
14. return msa_so_far
  }
```

Analysis:

statements 1,2,3 execution time depends upon "n"

statements 4,5,6,7,8 executated once, so it is some fixed constant for system and language. let C_1.

statement 9 depends on "n"

statements 10,11,12,13 depends upon "1*n"

Statement 14 takes some constant time.

Overall execution time T(n) have format

~~an^2~~. $an+b$

$\Rightarrow T(n) = O(n)$

Earlier problem was similar

Both problem have similar algorithms to solve. and execution time depends upon "n" having order $O(n)$.

Q-3

Let

$S(n, D)$ = ways of changing n paise using set of denominations $D[0, 1, \ldots d-1]$

$S(n, D)$ = non-decreasing sequence of elements of D each adding to n.

$S_0(n, D) = S(n, D - D[0])$

$S_1(n, D) = S(n - D[0], D - D[0])$

$S_2(n, D) = S(n - 2D[0], D - D[0])$

$\vdots$

$S_i(n, D) = S(n - iD[0], D - D[0])$; $0 \le i < n \% D[0]$

(A) $|S(n, D)|$ = for $i = 0$ to $n \% D[0]$

// $n \% D[0]$ is quotient when "n" is divided by $D[0]$.

$|S(n - iD[0], D - D[0])|$

Corner Cases:

(B) $|S(n, \{\})| = 0$ if $n > 0$

(C) $|S(0, D)| = 1$

(D) $|S(n, D)| = 0$ if $n < 0$

Analogy

$N(r, i)$ = number of ways changing r using $D[i] \ldots D[d-1]$

$N(r, i) = 0$ $r < 0$ (D)

$N(0, i) = 1$ $0 \le i < d$ (C)

$N(r, d) = 0$ $r > 0$ (B)

$N(r, i) =$ for $j = 0$ to $n \% D[i]$

$N(r - j D[i], i+1)$

```
int N[n+1][d+1]
    For (int r=1; r<=n; r++)
            N[r][d] = 0;                    // B
    For (int i=0; i<=d; i++)                // C
            N[0][i] = 1;
    For (int r=1; r<=n; r++) {
        For (int i=d-1; i>=0; i--) {
            For j=0 to n%D[i] {
                N[r][i] = N(r-jD[i], i+1)
            } If (r-jD[i]>=0)
                N[r][i] += N(r-jD[i], i+1)
        }                                   // A and D
    }
}
cout << N[n][0] << endl;
```

Time Analysis (B) : $O(n)$

(C) : $O(n)$

(D) : $(nd * d) = (nd^2)$

Total : $O(nd^2)$

: $O(nd^2)$

This is slower than previous approach discussed in class. But faster than exponential time for some n, d.