Data Structures and Algorithms

(ESO207)

Lecture 12:

- **Queue** : a new data Structure :
- Finding shortest route in a grid in presence of obstacles

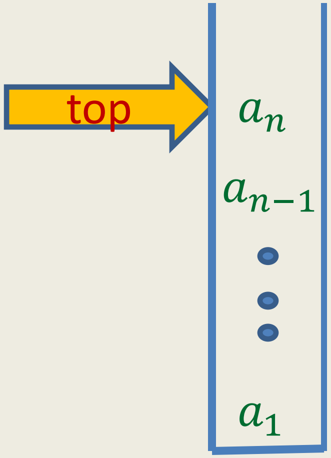
Queue: a new data structure

Data Structure Queue:

- Mathematical Modeling of Queue
- Implementation of Queue using arrays

Stack

A special kind of list where all operations (insertion, deletion, query) take place at one end only, called the **top**.



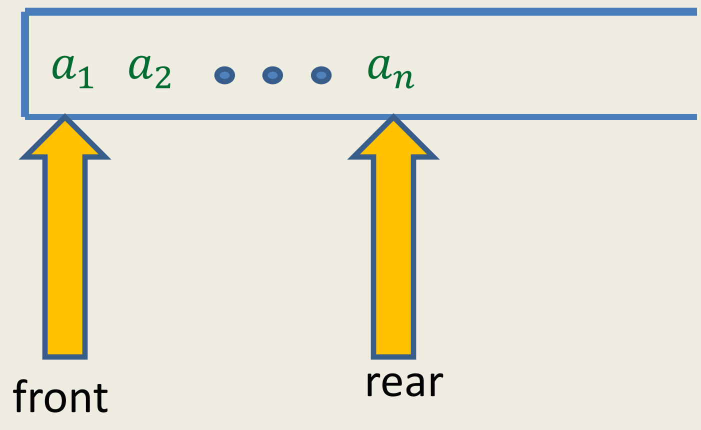
Behavior of Stack:

Last in **First out** **(LIFO)**

Queue: a new data structure

A <u>special kind</u> of list based on **(FIFO)**

First in **First Out**



Operations on a Queue

Query Operations

- **IsEmpty(Q)**: determine if **Q** is an empty queue.
- **Front(Q)**: returns the element at the **front** position of the queue.

  Example: If **Q** is $a_1, a_2, \ldots, a_n$, **then Front(Q)** returns $\boxed{a_1}$.

Update Operations

- **CreateEmptyQueue(Q)**: Create an empty queue
- **Enqueue(x,Q)**: insert **x** at the **end** of the queue **Q**

  Example: If **Q** is $a_1, a_2, \ldots, a_n$, then after **Enqueue(x,Q)**, queue **Q** becomes

  $$\boxed{a_1, a_2, \ldots, a_n, \mathbf{x}}$$

- **Dequeue(Q)**: return element from the **front** of the queue **Q** and <u>delete</u> it

  Example: If **Q** is $a_1, a_2, \ldots, a_n$, then after **Dequeue(Q)**, queue **Q** becomes

  $$\boxed{a_2, \ldots, a_n}$$

How to access ith element from the front ?

$$a_1 \bullet \bullet \bullet\ a_{i-1}\ \ a_i \bullet \bullet \bullet\ a_n$$

- To access ith element, we **must** perform **dequeue** (hence delete) the first $i-1$ elements from the queue.

An Important point you must remember for every data structure

You can define any **new** operation only in terms of the primitive operations of the data structures defined during its modeling.

Implementation of Queue using array

Assumption: At any moment of time, the number of elements in queue is n.

Keep an array of **Q** size n, and two variables front and rear.

- front: the position of the **first** element of the queue in the array.
- rear: the position of the **last** element of the queue in the array.

Enqueue(x,**Q**)

{ rear ← rear+1;

Q[rear]←x

}

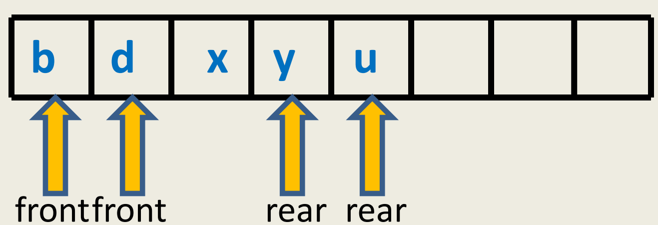
Deqeue(**Q**)

{ x← **Q**[front];

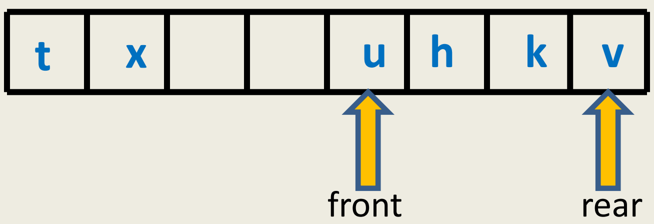
front← front+1;

return x;}

There is a serious problem !



Implementation of Queue using array



How to perform
Enqueue(x,Q) ?

Implementation of Queue using array

```
Enqueue(x,Q)
{    rear ← (rear+1) mod n ;
  Q[rear]←x
}

Dequeue(Q)
{        x← Q[front];
    front← (front+1) mod n ;
    return x;
}
IsEmpty(Q)
{    Do it as an exercise    }
```

Shortest route in a grid with obstacles

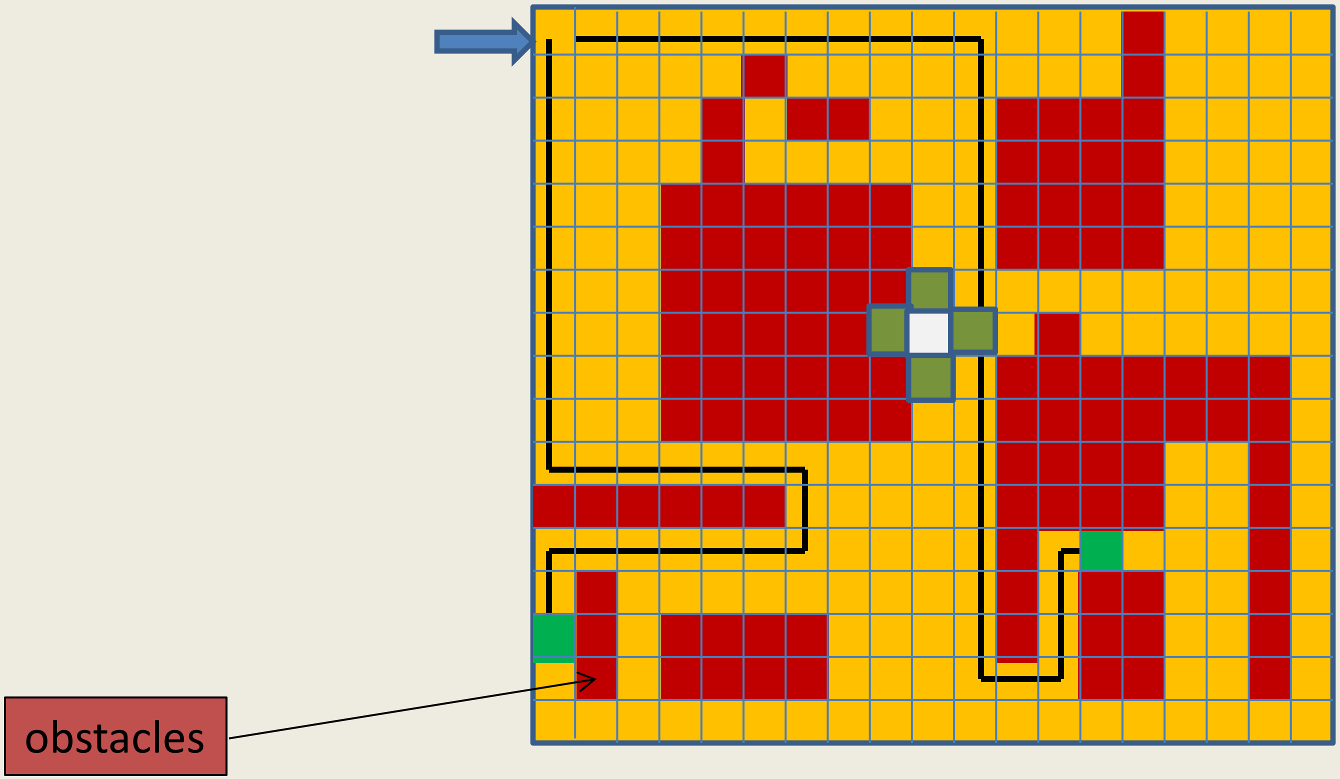
Shortest route in a grid

From a cell in the grid, we can move to any of its neighboring cell in one step.

Problem: From top left corner, find shortest route to each cell avoiding obstacles.

Input : a Boolean matrix G representing the grid such that

$G[i, j]$ = 0 if (i, j) is an obstacle, and 1 otherwise.



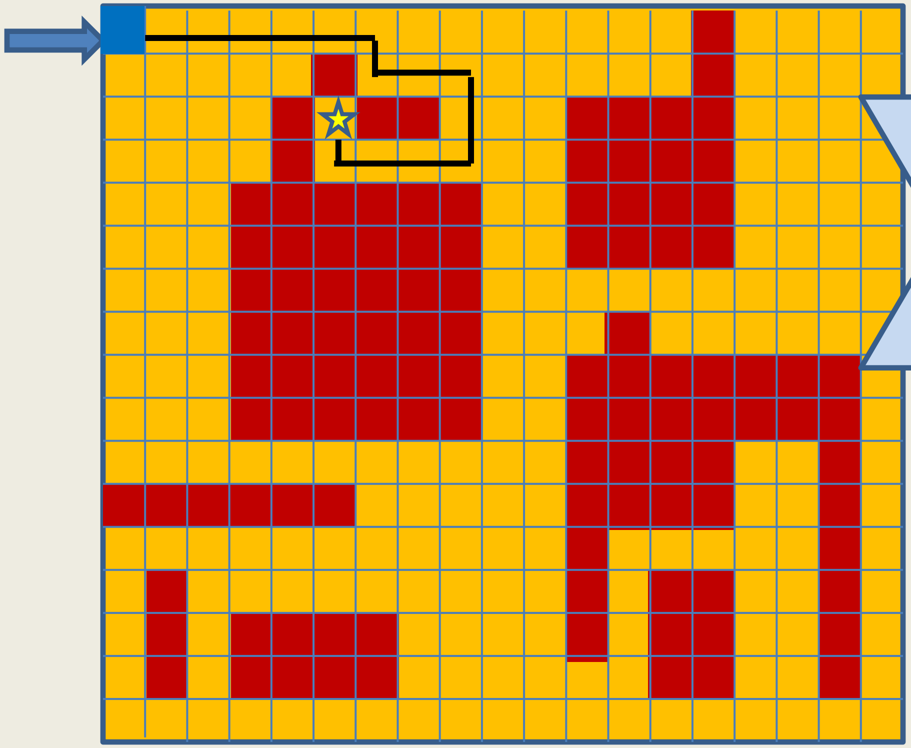
Step 1:

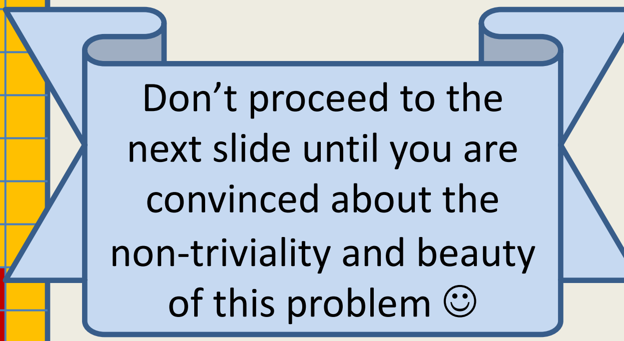
Realizing

the nontriviality of the problem

Shortest route in a grid

nontriviality of the problem





Definition: Distance of a cell **c** from another cell **c'**

is the length (number of steps) of the shortest route between **c** and **c'**.

We shall design algorithm for computing distance of each cell from the start-cell.

As an exercise, you should extend it to a data structure for retrieving shortest route.

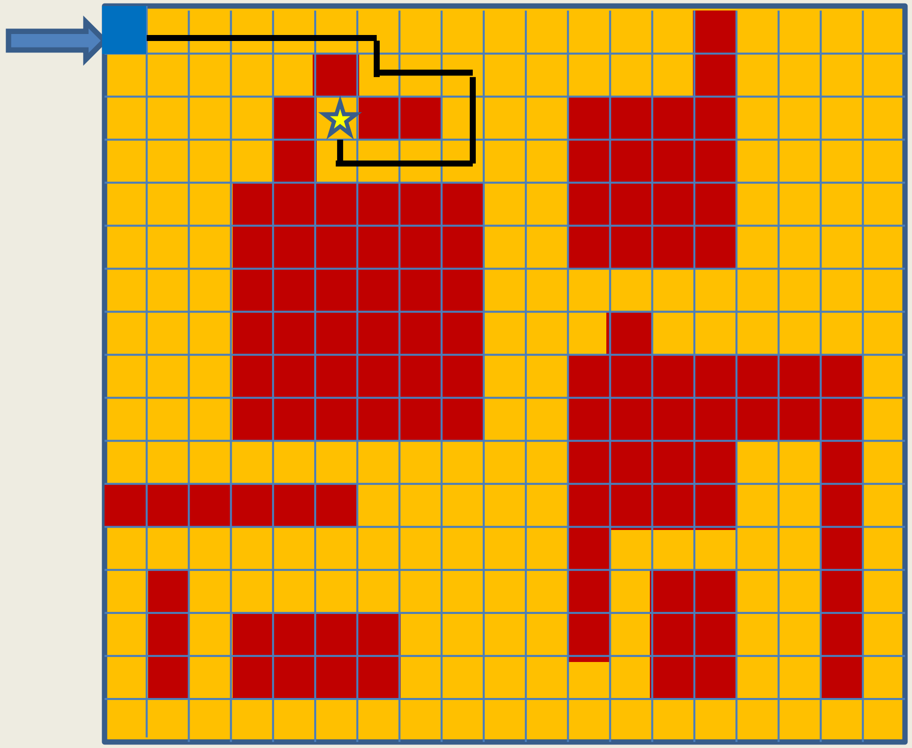
Get inspiration from nature

The ripples travels along the shortest route ?

Shortest route in a grid

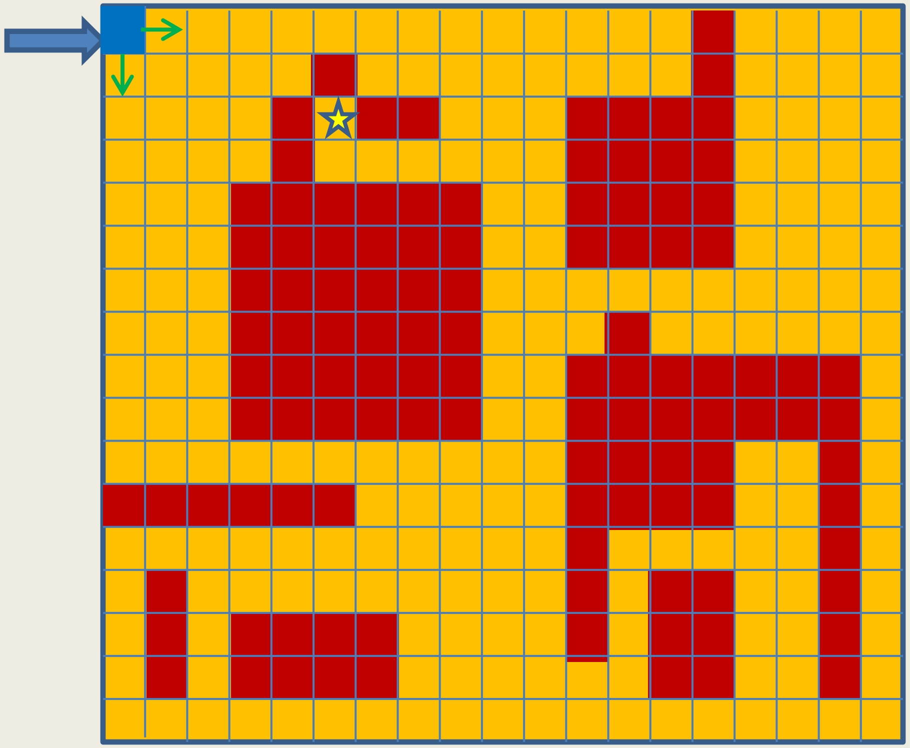
nontriviality of the problem

How to find the shortest route to ☆ in the grid **?**

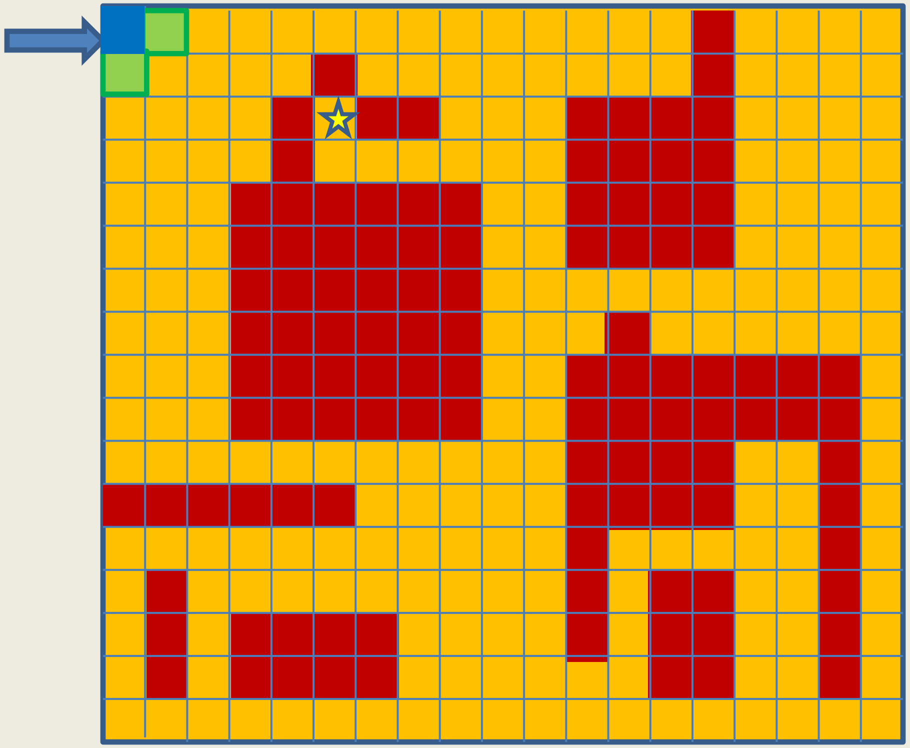


Create a ripple at the start cell and trace the path it takes to ☆

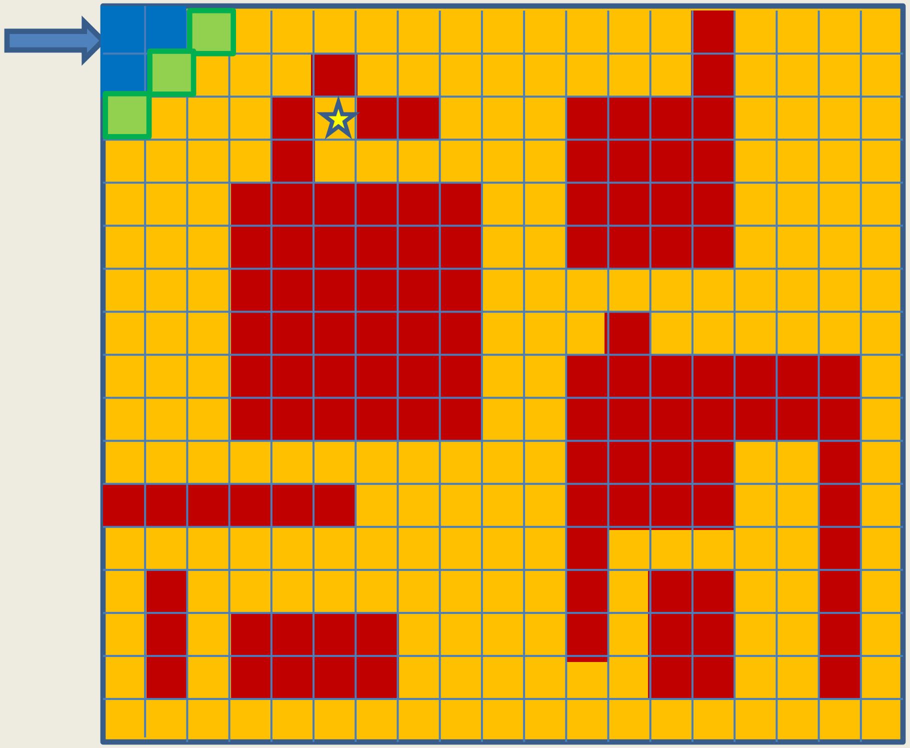
propagation of a ripple from the start cell



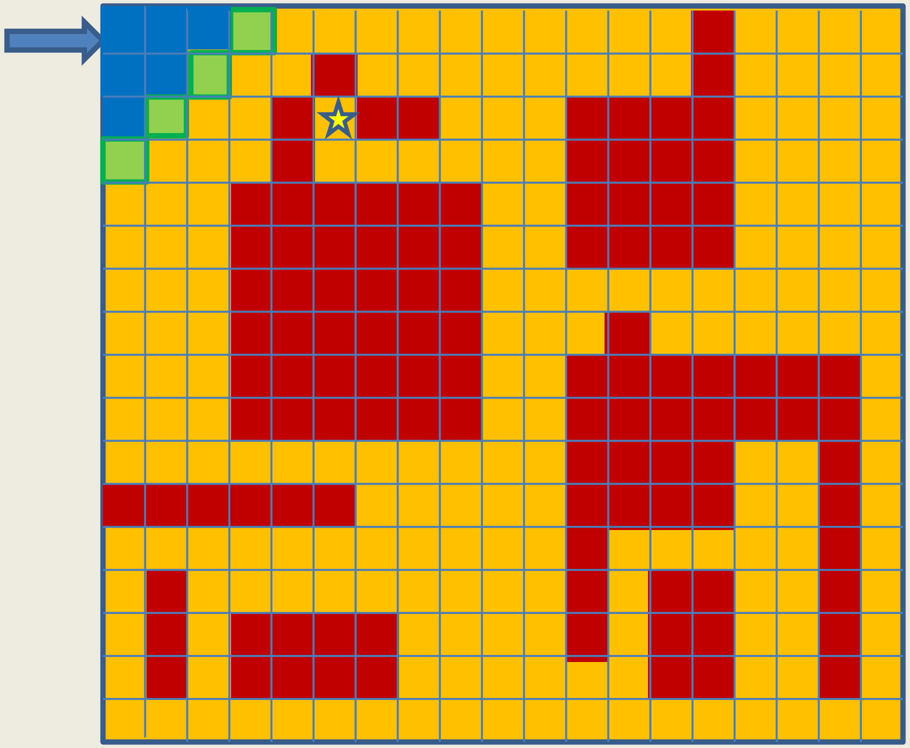
ripple reaches cells at distance 1 in step 1



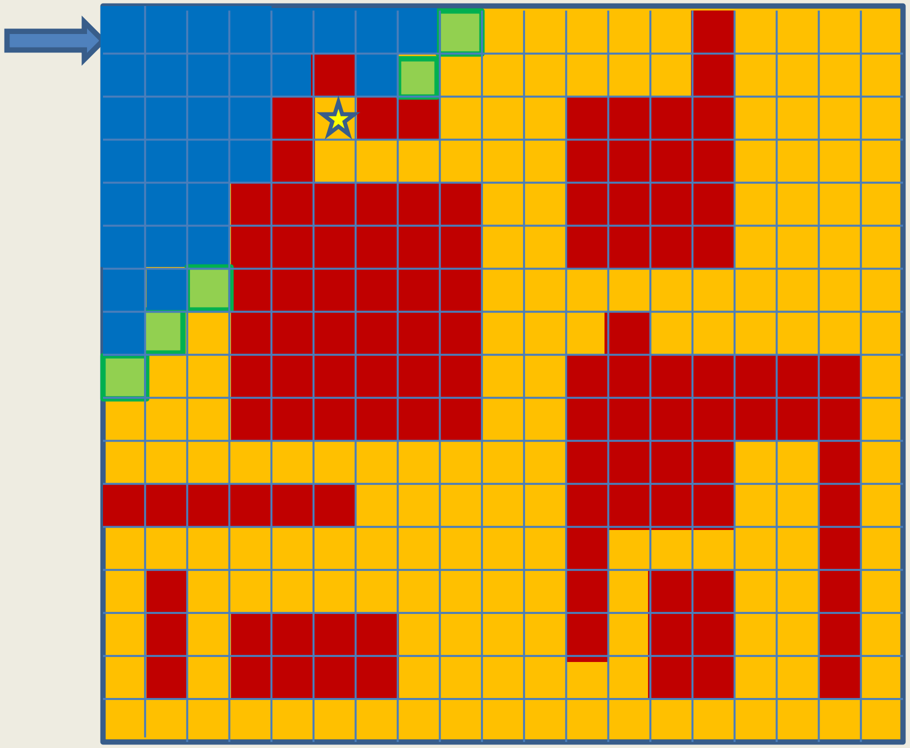
ripple reaches cells at distance 2 in step 2



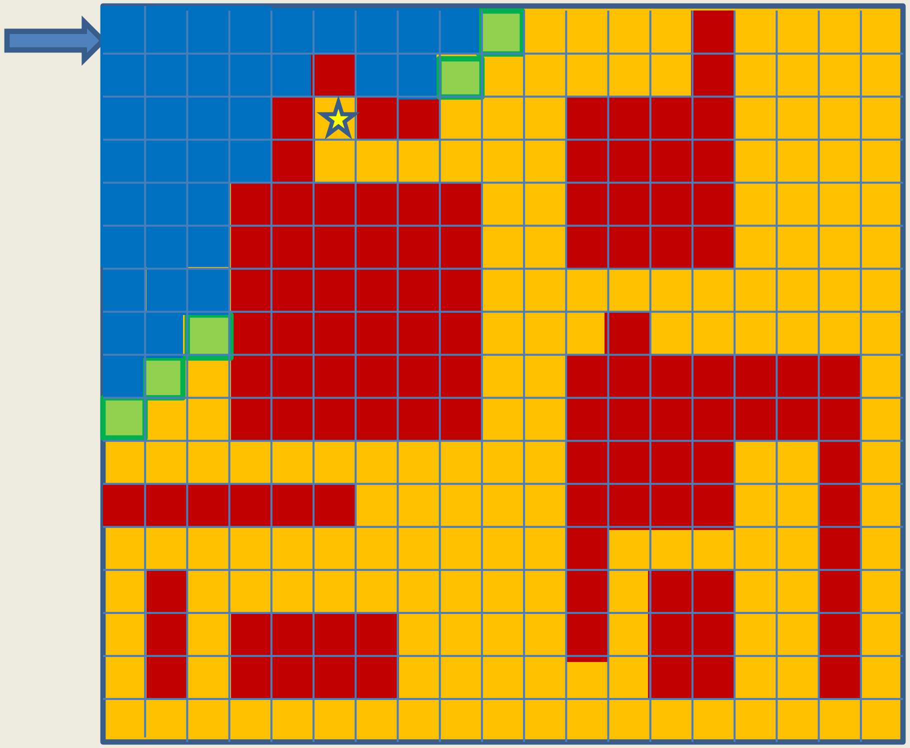
ripple reaches cells at distance 3 in step 3



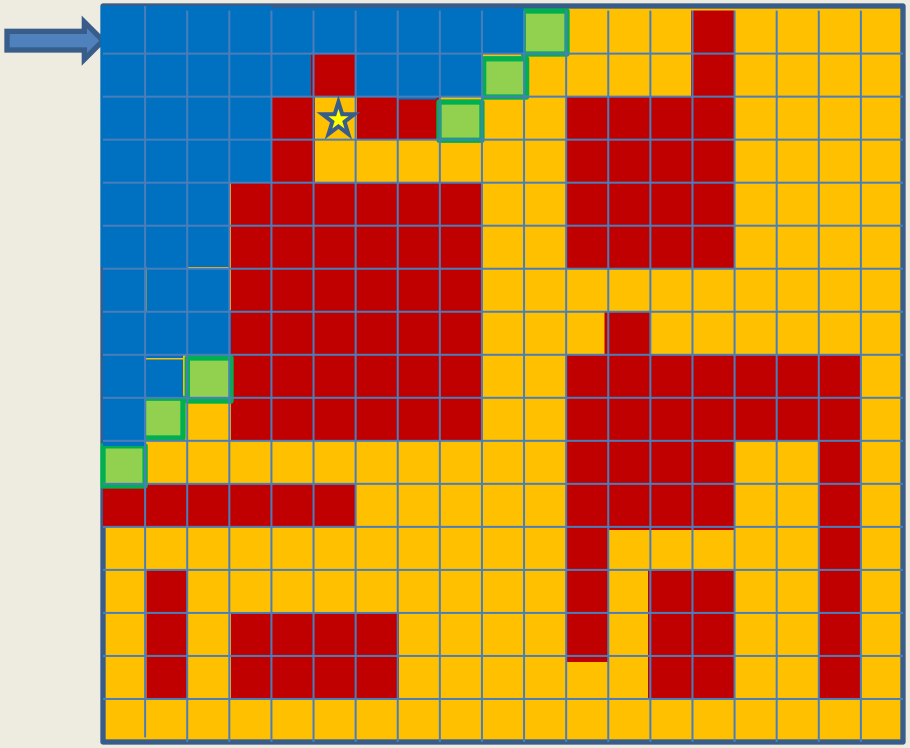
ripple reaches cells at distance 8 in step 8



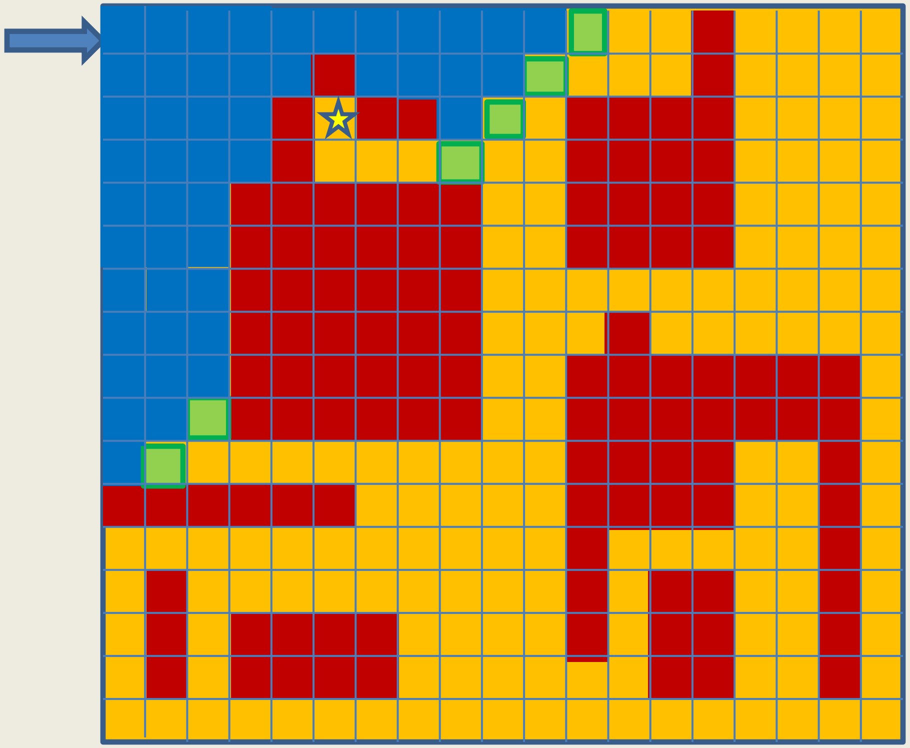
ripple reaches cells at distance 9 in step 9



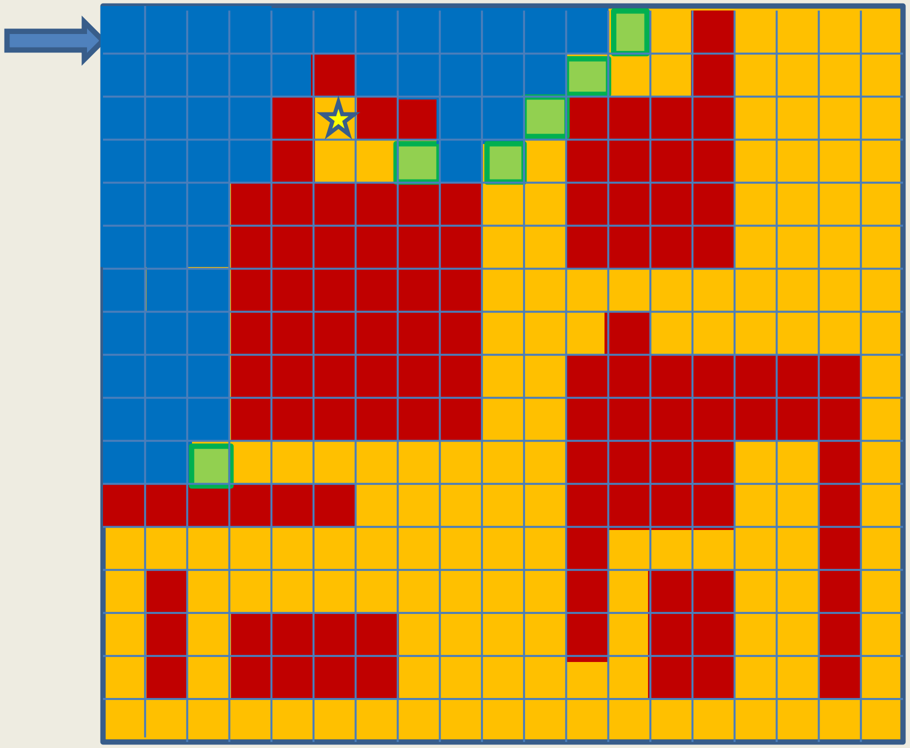
ripple reaches cells at distance 10 in step 10



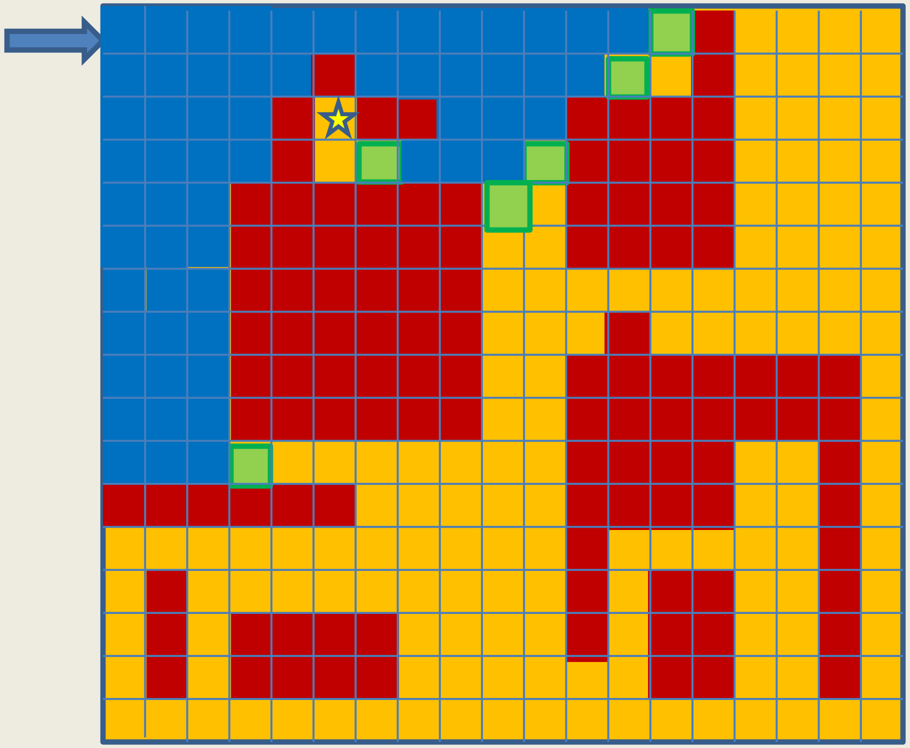
ripple reaches cells at distance 11 in step 11



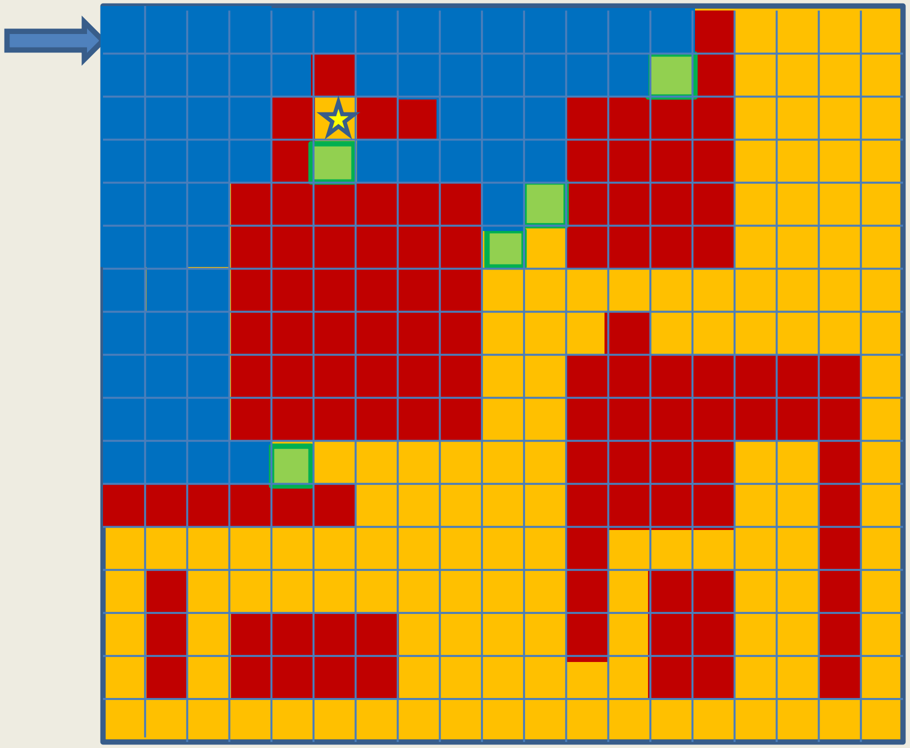
ripple reaches cells at distance 12 in step 12



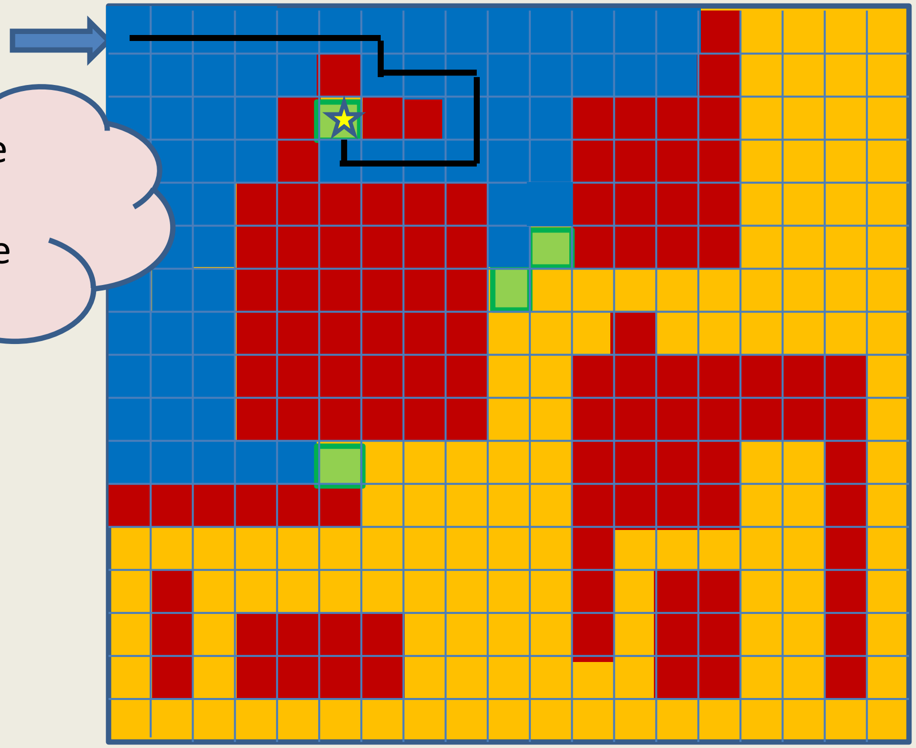
ripple reaches cells at distance 13 in step 13

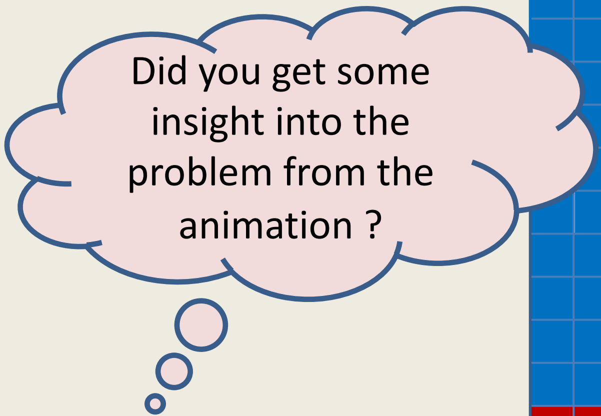


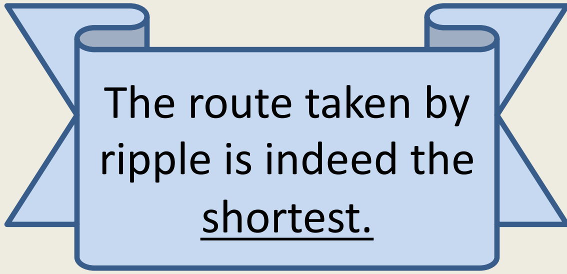
ripple reaches cells at distance 14 in step 14



ripple reaches cells at distance 15 in step 15







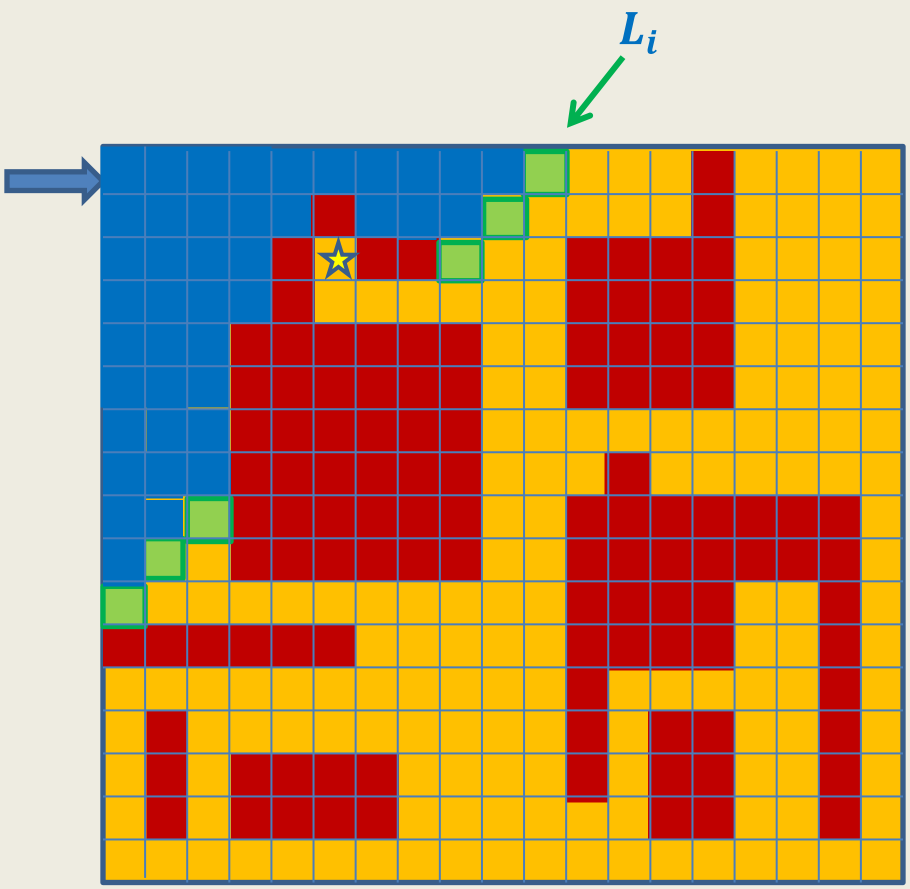
Think for a few more minutes with a free mind ☺.

Step 2:
Designing algorithm for distances in grid

(using an insight into propagation of ripple)

A snapshot of ripple after i steps

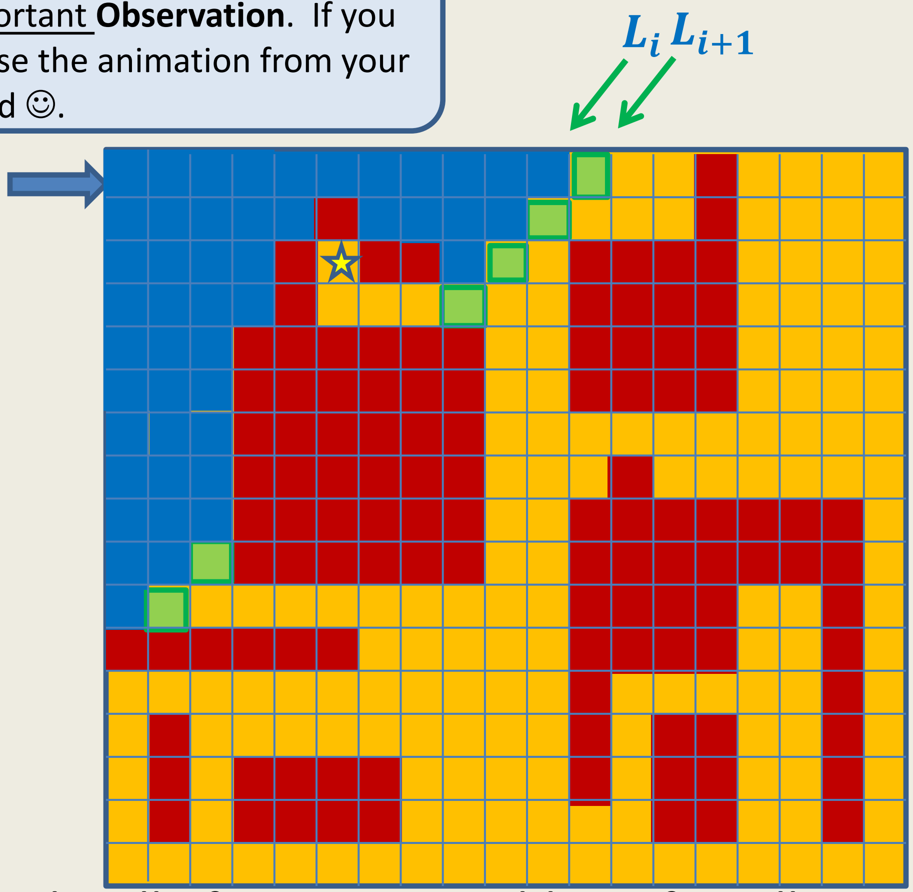
A snapshot of ripple after i steps



L_i : the cells of the grid at distance i from the starting cell.

A snapshot of the ripple after $i + 1$ steps

All the hardwork on the animation was done just to make you realize <u>this important </u>**Observation**. If you have got it, feel free to erase the animation from your mind ☺.



Observation: Each cell of L_{i+1} is a neighbor of a cell in L_i.

Distance from the start cell

It is worth spending some time on this matrix.
Does the matrix give some idea to answer the question ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13		27	28	29	30
1	2	3	4	5		7	8	9	10	11	12	13	14		26	27	28	29
2	3	4	5		15			10	11	12					25	26	27	28
3	4	5	6		14	13	12	11	12	13					24	25	26	27
4	5	6							13	14					23	24	25	26
5	6	7							14	15					22	23	24	25
6	7	8							15	16	17	18	19	20	21	22	23	24
7	8	9							16	17	18		20	21	22	23	24	25
8	9	10							17	18								26
9	10	11							18	19								27
10	11	12	13	14	15	16	17	18	19	20					35	36		28
						17	18	19	20	21					34	35		29
24	23	22	21	20	19	18	19	20	21	22		30	31	32	33	34		30
25		23	22	21	20	19	20	21	22	23		29			34	35		31
26		24					21	22	23	24		28			33	34		32
27		25					22	23	24	25	26	27			32	33		33
28	27	26	27	26	25	24	23	24	25	26	27	28	29	30	31	32	33	34

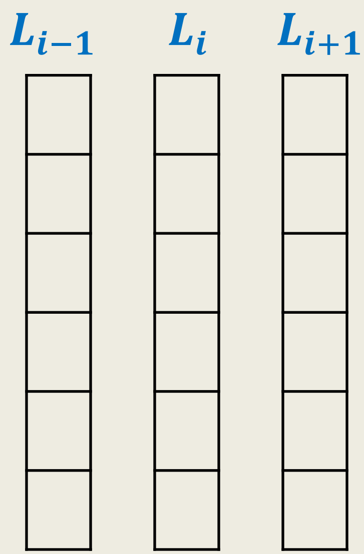
How can we generate L_{i+1} from L_i ?

Observation: Each cell of L_{i+1} is a neighbor of a cell in L_i.

But every neighbor of L_i may be a cell of L_{i-1} or L_{i+1}.

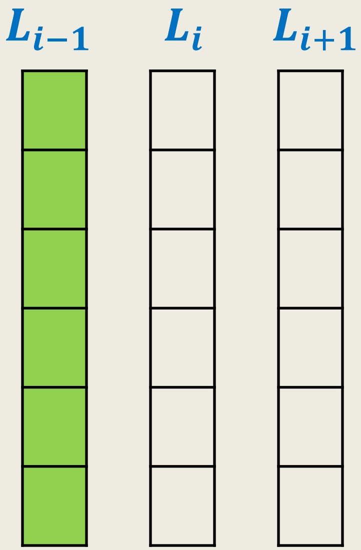
How can we generate L_{i+1} from L_i ?

How can we generate L_{i+1} from L_i ?



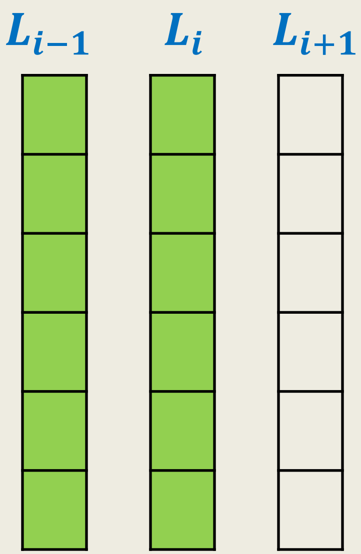
How can we **generate** L_{i+1} from L_i ?

Suppose all cells of L_{i-1} get visited first.



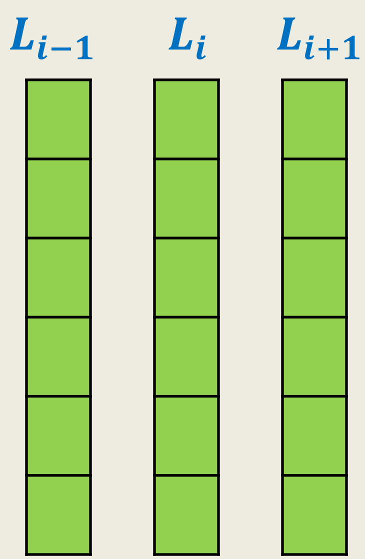
How can we **generate** L_{i+1} from L_i ?

Suppose all cells of L_{i-1} get visited first.
Then all cells of L_i are visited, and



How can we **generate** L_{i+1} from L_i ?

Suppose all cells of L_{i-1} get visited first.
Then all cells of L_i are visited, and
then all cells of L_{i+1} are visited.



So by the time all cells of L_i are visited, if a cell neighboring to a cell of L_i is unvisited, it must be a cell of L_{i+1}.
☺

How can we generate L_{i+1} from L_i ?

So the algorithm should be:

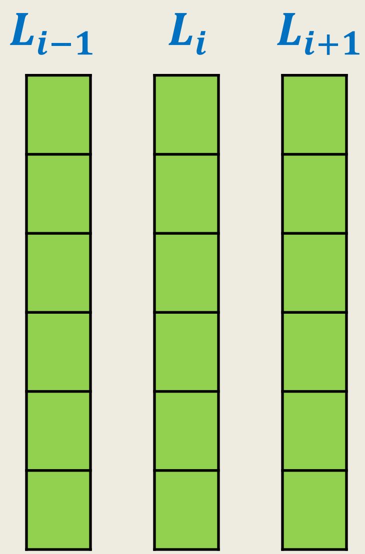
Initialize the distance of all cells except start cell as ∞

First compute L_1.

Then using L_1 compute L_2

Then using L_2 compute L_3

…



Algorithm to compute L_{i+1} if we know L_i

```
Compute-next-layer(G, L_i)
{
   CreateEmptyList(L_{i+1});
   For each cell c in L_i
        For each neighbor b of c which is not an obstacle
        {      if (Distance[b] = ∞)
               {      Insert(b, L_{i+1});
                      Distance[b] ← i + 1 ;
               }
        }
   return L_{i+1};
}
```

The first (not so elegant) algorithm

(to compute distance to all cells in the grid)

It can be as high as O(n^2)

Distance-to-all-cells(G, $\mathbf{c}_0$)

{ $\boldsymbol{L_0}$ ← {$\mathbf{c}_0$};

For($\boldsymbol{i}$ = 0 to ??)

$\boldsymbol{L_{i+1}}$ ← **Compute-next-layer**(**G**, $\boldsymbol{L_i}$);

}

The algorithm is not elegant because of

- So many temporary lists that get created.

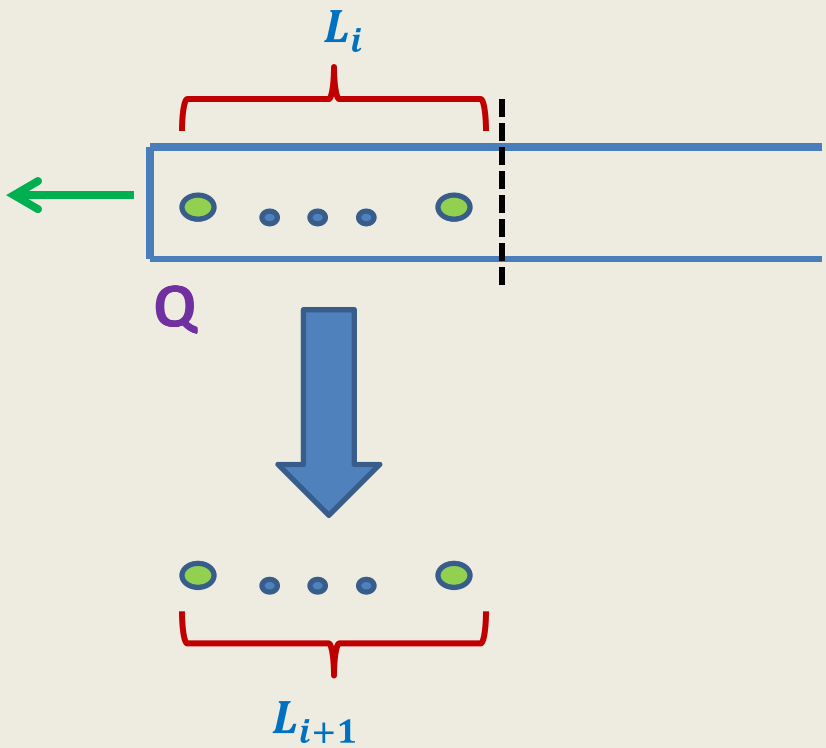
Towards an elegant algorithm …

Key points we observed:

- We can compute cells at distance $i + 1$ if we know all cells up to distance i.
- Therefore, we need a mechanism
  to enumerate the cells in **non-decreasing** order of **distances** from the start cell.

How to design such
a **mechanism** ?

Keep a queue Q



Spend some time to see how seamlessly the queue ensured
the requirement of visiting cells of the grid in non-decreasing order of distance.

An elegant algorithm

(to compute distance to all cells in the grid)

```
Distance-to-all-cells(G, c0)
   CreateEmptyQueue(Q);
   Distance(c0) ← 0;
   Enqueue(c0,Q);
   While( Not IsEmptyQueue(Q) )
   {        c ←Dequeue(Q);
            For each neighbor b of c which is not an obstacle
            {       if (Distance(b) = ∞)
                    {  Distance(b) ←  Distance(c) +1 ;
                       Enqueue(b, Q); ;
                    }
            }
   }
```

Proof of correctness of algorithm

Question: What is to be proved ?

Answer: At the end of the algorithm,

Distance[c]= the distance of cell c from the starting cell in the grid.

Question: How to prove ?

Answer: By the principle of mathematical induction on

the distance from the starting cell**.**

Inductive assertion:

P(i):

The algorithm correctly computes distance to all cells at distance i from the starting cell.

As an exercise, try to prove **P(i)** by induction on i.